THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

On Supervisor Synthesis via Active Automata Learning

Ashfaq Farooqui

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2021

On Supervisor Synthesis via Active Automata Learning

Ashfaq Farooqui

ISBN: 978-91-7905-510-3
Series Number. 4977
in the series Doktorsavhandlingar vid Chalmers tekniska högskola. Ny serie
(ISSN0346-718X)
This thesis has been prepared using LaTeX.

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

Printed by Chalmers Reproservice
Gothenburg, Sweden, June 2021

To my parents,
Kauser and Ather Farooqui,
for their steadfast love and support.

Abstract

Our society's reliance on computer-controlled systems is rapidly growing. Such systems are found in various devices, ranging from simple light switches to safety-critical systems like autonomous vehicles. In the context of safety-critical systems, safety and correctness are of utmost importance. Faults and errors could have catastrophic consequences. Thus, there is a need for rigorous methodologies that help provide guarantees of safety and correctness. Supervisor synthesis, the concept of being able to mathematically *synthesize* a supervisor that ensures that the closed-loop system behaves in accordance with known requirements, can indeed help.

This thesis introduces *supervisor learning*, an approach to help automate the learning of supervisors in the absence of plant models. Traditionally, supervisor synthesis makes use of *plant models* and *specification models* to obtain a supervisor. Industrial adoption of this method is limited due to, among other things, the difficulty in obtaining usable plant models. Manually creating these plant models is an error-prone and time-consuming process. Thus, supervisor learning intends to improve the industrial adoption of supervisory control by automating the process of generating supervisors in the absence of plant models.

The idea here is to learn a supervisor for the *system under learning* (SUL) by active interaction and experimentation. To this end, we present two algorithms, $SupL^*$, and MSL, that directly learn supervisors when provided with a simulator of the SUL and its corresponding specifications. $SupL^*$ is a language-based learner that learns one supervisor for the entire system. MSL, on the other hand, learns a *modular* supervisor, that is, several smaller supervisors, one for each specification. Additionally, a third algorithm, MPL, is introduced for learning a modular plant model.

The approach is realized in the tool MIDES and has been used to learn supervisors in a virtual manufacturing setting for the *Machine Buffer Machine* example, as well as learning a model of the *Lateral State Manager*, a subcomponent of a self-driving car. These case studies show the feasibility and applicability of the proposed approach, in addition to helping identify future directions for research.

Keywords: Model learning, Automata learning, Active learning, Supervisory control theory, Discrete-event systems, Finite-state machines.

List of Publications

This thesis is based on the following publications:

[A] **Ashfaq Farooqui**, Ramon Tijsse Claase, Martin Fabian, "On Plant-Free Active Learning of Supervisors ". *Submitted to IEEE Transactions on Automation Science and Engineering (TASE)* .

[B] **Ashfaq Farooqui**, Fredrik Hagebring, Martin Fabian, "Active Learning of Modular Plant Models". *15th IFAC Workshop on Discrete Event Systems.*

[C] Fredrik Hagebring, **Ashfaq Farooqui**, Martin Fabian, "Modular Supervisory Synthesis for Unknown Plant Models Using Active Learning". *In proceeding of the 15th IFAC Workshop on Discrete Event Systems.*

[D] Yuvaraj Selvaraj, **Ashfaq Farooqui**, Ghazala Panahandeh, Wolfgang Ahrendt, Martin Fabian, "Automatically Learning Formal Models from Autonomous Driving Software". *Submitted to the Special Issue on Recent Trends in Model-based Engineering of Automotive Systems, JASE.*

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

[E] **Ashfaq Farooqui**, Fredrik Hagebring, Martin Fabian, "MIDES: A Tool for Supervisory Synthesis via Model Inference". *Submitted to CASE.*

[F] Yuvaraj Selvaraj, **Ashfaq Farooqui**, Ghazala Panahandeh, Martin Fabian, "Automatically Learning Formal Model: An Industrial Case from Autonomous Driving Development". *In Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, October 2020.

[G] **Ashfaq Farooqui**, Kristofer Bengtsson, Petter Falkman, Martin Fabian, "Towards Data-driven Approaches in Manufacturing; An Architecture to Collect Sequences of Operations". *International Journal of Production Research*, March 2020.

[H] **Ashfaq Farooqui**, Kristofer Bengtsson, Petter Falkman, Martin Fabian, "From factory floor to process models: a data gathering approach to generate, transform, and visualize manufacturing processes". *CIRP-Joural of manufacturing Science and Technology*, January 2019.

[I] **Ashfaq Farooqui**, Martin Fabian, "Synthesis of Supervisors for Unknown Plant Models Using Active Learning". *In Proceedings of the 15th IEEE Conference on Automation, Science and Engineering*, August 2019.

[J] **Ashfaq Farooqui**, Petter Falkman, Martin Fabian, "Towards Automatic Learning of Discrete-Event Models from Simulations". *In Proceedings of the 14th IEEE Conference on Automation, Science and Engineering*, August 2018.

[K] **Ashfaq Farooqui**, Kristofer Bengtsson, Petter Falkman, Martin Fabian, "Real-time Visualization of Robot Operation Sequences". *In Proceedings of the 16th IFAC Symposium on Information Control Problems in Manufacturing*, June 2018.

[L] **Ashfaq Farooqui**, Patrik Bergagård, Petter Falkman, Martin Fabian, "Error handling within highly automated automotive industry: Current practice and research needs". *In Proceedings of the 21st IEEE International Conference on Emerging Technologies and Factory Automation*, September 2016.

Acknowledgments

Seeing my work come together in this thesis and the defense getting closer has made me aware of the significance of this moment. My odyssey as a doctoral student draws to a close, a major milestone in the journey of life. Feels like it was a couple of months ago that I first stepped into the EDIT building. Like in all journeys, the people encountered have left an impact, some more than others, but each profound and beautiful in their own ways. And to each, I owe a debt of gratitude. This odyssey would certainly not have been possible without the guidance, support, and patience of a lot of people who have made this milestone possible for me.

Foremost is my mentor and supervisor, Martin Fabian. I am forever grateful for the time and effort you have invested in me and this work. You have been an impeccable example of what a supervisor must be, not just for the guidance and support you provide, but for teaching me the what, why, and how, of supervision. You have proofread almost every line of text I have written in these last few years, and provided meticulous invaluable feedback. It has been an exhilarating experience working with, and learning, from you. Something happened to me during these past years that evolved me from a hacker to a beginner researcher, and I can't put my finger on any specific events or actions that explain the transformation; it was just the magic that you conjured up and bestowed upon me somehow. I can only hope I will be able to emulate everything I have learned from you. Thank you!

Petter Falkman, my co-supervisor, I have learned from you the difficult art of always trying to look at the bigger picture. Thank you for helping me keep my work anchored in reality by helping me find use cases to apply my ideas. But more importantly, for inspiring and offering me the opportunity to embark on what has been a rewarding odyssey.

Fredrik and Yuvaraj, collaborating with you guys has been lots of fun and a great learning experience. I really appreciate all of the insightful and stimulating discussions that have helped change my perspectives at times I needed to most. Thank you for having the patience to listen to my (often) incoherent ramblings.

A big thank you to all the students I have had the opportunity to work with, especially Ramon and Marco.

Thanks to the administrative team, especially Christine, Madeleine, and Natasha, who make day-to-day work run smoothly.

Besides the work, I have had a great time with colleagues at SysCon. To all the present and former members, thanks for being there, as considerate friends. Special shout out to Sarmad for the warm welcome on my first day at the department! Sabino and Endre, for all the impromptu discussions on science and philosophy, we should have them more often! Adnan and Martin D., thanks for all the times you kept my sanity in check! Kristofer, thanks for introducing me to Scala; also, developing the IA course along with you has been a fun experience.

On a personal note, this journey would have been impossible without the constant support of loved ones. My family, though far away, are a source of strength. I am forever grateful to my parents and sister for being ever so supportive and letting me do what I find interesting.

To my wonderful wife Tania–the love of my life, thank you for being so loving, patient, and understanding. Especially these last few months, I have spent way too little time with you; I promise to make up for it!

Ashfaq Farooqui
Gothenburg, June 2021

This work was supported by VR SyTeC (2016-06204), the Chalmers Production Area of Advance, FFI-VINNOVA Auto-CAV (2017-05519), and the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

Acronyms

DES:	Discrete Event Systems
DFA:	Deterministic Finite Automaton
EFSM:	Extended Finite State Machine
EQ:	Equivalence Query
LSM:	Lateral State Manager
MAT:	Minimally Adequate Teacher
MIDES:	Model Inference for Discrete Event Systems
MPL:	Modular Plant Learner
MQ:	Membership Query
MSL:	Modular Supervisor Learner
PLC:	Programmable Logic Controller
PSH:	Plant Structure Hypothesis
SCT:	Supervisory Control Theory
SUL:	System Under Learning

CONTENTS

Part I

Overview

CHAPTER 1

INTRODUCTION

In 2018 two students working on their master's thesis found a critical bug in a sub-module of a self-driving vehicle [1]. The students were translating the program of the sub-module into a *state machine* model. Using this model, they were able to produce an error trace that allowed the engineers to simulate the incorrect behavior and then provide a fix.

The above is undoubtedly not a one-off case. History is replete with examples of unintended programming mistakes that have led to fatal accidents. Due to faulty code, the Ariane-5 exploded mid-air 37 seconds after its launch [2]. Malfunction of the Therac-25 caused radiation overdoses, killing three patients [3]. Knight Capital Group lost $450 million after a software update [4]. Another famous instance is Intel's "Pentium FDIV bug", which was a hardware bug found in 1994, leading Intel to replace defective chips, costing the company some half-billion dollars [5]. Subsequently, Intel began expanding their staff with formal methods experts [6].

Today, safety-critical and security-critical systems are being integrated into our daily lives. For example, autonomous vehicles are soon the default and no longer a luxury. Robots are being actively used for medical surgeries [7]. Software and computers are used in medical implants [8], [9]. Many of us rely on security for online banking, shopping, and so on. The inclusion of such systems into our daily lives is coupled with their increased complexity and functionality.

Formal methods are techniques used to model complex systems as mathematical entities. By building such a model, designers can then verify the system's properties. Additionally, mathematical proofs can be offered as a complement to the system tests to show correctness. For a long time, formal methods have been proclaimed to be the best means available for developing safe and reliable systems. To many researchers, the necessity of formal methods is now a given. However, from an industrial perspective, this has not been the case.

There has been some level of industrial acceptance of formal methods in the last few years. However, it is still a long way before formal methods are part of the industrial development process. There have been several studies that focus on understanding the reasons for the lack of adoption of formal methods by the industry [10]–[12]. Several reasons have been suggested for this situation, including lack of accessible tools, high costs, incompatibility with existing development techniques, and that these methods require a certain level of mathematical sophistication.

This thesis aims to provide tools and techniques to help enable the adoption of formal methods. More specifically, the focus is on formal methods that are employed for model-based verification of systems designs. The aim is to provide a tool that helps integrate formal methods into the existing developmental life-cycle for model-based development. It does so by helping engineers obtain of the target system a logical model that can be used for formal analysis.

In this context, *models* are mathematical descriptions of a system. These mathematical descriptions are abstractions that define a particular aspect or property of the system. Hence, there exist several valid models, each describing a particular aspect, for any given system.

Consider a coffee machine. One model can be built from the knowledge of the system's physical components using differential equations to represent the brewing process. On the other hand, another model can be created to define the machine's interactions with the user. For example, the model can show that the machine can be switched *on* or *off* using the power button. When it is powered on, the machine is in a *Standby* state until the user requests either *tea* or *coffee* using the appropriate button. Once the choice of beverage is dispensed, the machine goes back into its *Standby* state. These models can be visualized as seen in Figure 1.1. Such systems are referred to as *discrete event systems* (DES) and are the main focus of this thesis. Discrete event systems are modeled using a formalism such as *finite-state machines* [13].

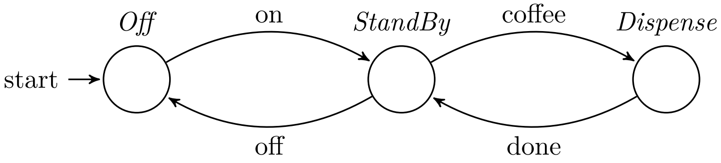


Figure 1.1: Simple model of a coffee machine

These models are then used for formal analysis. *Model checking* [14] and *Supervisory Control Theory* (SCT) [15], [16] are two common formal methods. In model checking, given a model of a system and some specifications that the model must satisfy, the model checker answers with a *yes* if the model satisfies the specifications; else, the model checker provides a counterexample with a *no* response. The model (and correspondingly the original system) is manually updated, in an iterative process, till it satisfies all of its specifications. SCT, on the other hand, performs *synthesis* by taking a model of the system and its specifications to automatically produce a *supervisor*. This supervisor is used in combination with the original system to ensure the closed-loop system does not violate any of the the specifications. Manually building such models is an error-prone and time-consuming process. Providing tools to enable engineers in the process of building models is essential. This thesis looks into the possibilities of automatically learning a discrete model.

1.1 Problem Description

SCT has gained a lot of traction within the academic community. Unfortunately, it has not been fully adopted into industrial practice. This is quite unfortunate, since SCT provides a promising approach to design and develop industrial control systems. There are several reasons for why SCT has not been able to gain a footing in the industry. We believe the most crucial reason is related to the access of useful models. Here, the problem is two-fold. Constructing models manually is challenging. Maintaining these models such that they always conform to the system requires additional effort.

As mentioned, SCT relies on the existence of a logical model describing the system. Defining such a model is a creative task requiring the engineers to make crucial decisions on what abstraction level to adopt. These abstractions necessarily focus on capturing some aspects at a certain level of detail of the system. If the aspects and their detail are adequate for what the model is used for, the model is useful. Else it is not. Furthermore, the level of abstraction determines the complexity of the model. It is not uncommon to develop several models each with a different level of abstraction. Consequently, constructing these models is an expensive undertaking that not many industries can justify. Hence, access to logical models is limited.

A much more significant problem pertains to the maintenance of these models. The target system is not fixed; it keeps evolving. When the system is updated, the model of the system needs to be updated along with its implementation. Additionally, the updated system and its model must be verified and corrected in case of model checking. In the case of SCT, the updated model is used to synthesize a new supervisor, which then has to be implemented to interact with the target system to ensure that the system behaves according to the specifications. Repeating the verification or synthesis procedure every time the system is updated is time-consuming. Hence, a common practice is only to update the system implementation, neglecting the model. The challenge of maintaining the usefulness of the model as the system is updated impacts the adoption of formal methods. The ability to have bi-directional updating of the model and the corresponding system and an automated workflow to obtain the supervisors would significantly improve the adoption of formal methods.

Therefore, to reap the benefits of formal methods, and SCT in particular, there is a need for tools and techniques to assist engineers in building models. Furthermore, automated workflows that integrate model building and supervisor synthesis will go a long way in popularizing formal methods in general. Such methods will help find and avoid potential errors as they can automatically generate models and apply formal techniques at regular intervals. Automatically constructing formal models can also help understand and reason about ill-documented legacy systems, something that is crucial for the quality assurance of large-scale and complex systems.

The work in this thesis is grounded in the belief that it is possible to learn a system's logical model by interacting with the target system. Humans interact with and learn unknown systems by repeatedly forming conjectures about the system and falsifying these conjectures until they cannot be falsified. Doing the same algorithmically has been studied under the field commonly known as *active automata learning* [17]–[20]. Active automata learning is a field of research that addresses the problem of automatic model construction. These approaches constitute a class of machine learning algorithms that aim to deduce, by actively interacting with the target system, a finite-state machine that describes the system's logical behavior.

Interacting with the physical system to learn a model, though possible, is impractical. Physical systems are expensive to build, maintain and are prone to wear and tear. Furthermore, using physical systems to learn a model is potentially unsafe as it can lead to unforeseen collisions. Interacting with a simulation of the system rather than the physical system is a more feasible approach. Simulation-based design is being adopted in the industry [21]. These techniques include building a virtual replica of the target system, usually a digital twin [22], that behaves just like its physical counterpart. Since such simulation models behave like their physical counterparts, they can be used instead of the physical systems to interact with to learn the system's logical model.

1.2 Research Questions

In order to enable industrial adoption of SCT, we want to, in the best of scenarios, use a simulation to directly learn a supervisor that can control the simulation or its physical counterpart, such that the controlled system behaves correctly according to some specifications. In case it is not possible to directly learn a supervisor, the work in this thesis aims to learn a model describing the behavior of the system, which can then be used to synthesize a supervisor. Hence, we pose the following questions:

RQ1 *How can we integrate automata learning techniques and SCT to help design systems that are correct-by-construction?*

As mentioned previously, obtaining models to apply SCT is a tedious task. Active automata learning presents an approach to learn a model by interacting with the target system. Supervisor synthesis can then be applied to remove the unwanted behaviors from this model and obtain the supervisor to ensure that the controlled system behaves according to the specifications. Integrating the learning and the synthesis into a single step will help, in the best of cases, to learn a smaller model where the unwanted behaviors are absent, avoiding the computationally expensive synthesis step.

RQ2 *What techniques can help learn models for larger and complex systems?*

SCT is known to be an NP-hard problem [23]. Hence, it becomes impossible to obtain a supervisor for large and complex systems. Therefore, there is a need to investigate innovative methods that can solve the problem under special cases.

RQ3 *What are the challenges faced when applying these methods to real-world scenarios?*

We intend to provide tools and techniques to help industries develop systems that are correct-by-construction by applying formal methods, SCT in particular, in their day-to-day development. A natural step as part of this thesis is to investigate challenges faced when applying the research in this thesis on practical, real-world problems.

1.3 Main Contributions

Attempting to answer the above research questions this thesis results in the following contributions:

- The *SupL** (see Paper A) algorithm is presented that is an extension of the well-known L^* algorithm. The presented algorithm can learn a supervisor of a target system when the specification is available; else, it learns a model describing the system's logical behavior.

- A case study applying *SupL** to learn a supervisor of the *MBM*, a well-known example in the SCT community, is presented in Paper A. The system is simulated in a virtual environment and controlled using a PLC program to emulate the virtual commissioning set up to create a more realistic scenario. The learning algorithm then interacts with the PLC to learn a supervisor.

- Two novel algorithms, the Modular Plant Learner in Paper B to learn a model, and the Modular Supervisor Learner in Paper C to learn a supervisor are introduced. The resulting automata obtained from these algorithms are composed of smaller modules that, taken together, define the system's behavior. Hence, the algorithms can be used to learn models for large and complex systems.

- Another case study where a model of a sub-component of a self-driving vehicle is learned (see Paper D) is presented. This is done by interacting with the MATLAB code of this component. Algorithms from papers A and B are evaluated.

- All the algorithms discussed in this thesis are implemented in MIDES [24], a tool for automatic learning of models and supervisors for discrete event systems. Apart from the algorithms, the tool provides interfaces for MATLAB code and OPC-UA for PLC programs to be used as simulators, from which models can be learned. The tool is designed in a modular manner to enable easy integration of custom simulators and algorithms for rapid prototyping purposes.

1.4 Research Methods

The work done in this thesis is applied in nature. Most of the activities are related to conceptual analysis and implementation [25]. That is to say, a significant portion of the research focuses on prototyping and validating the ideas. From a high level, the core research method is to form a hypothesis and develop a proof of concept to test this hypothesis. It is natural, in this process, to continuously update and refine the hypothesis from the insights gained during the process of developing the proof of concept.

Before jumping into the implementations, a first step is to engage with the literature to discover promising methodologies of interest. In this thesis, the literature study revealed several existing algorithms to learn models. However, most, if not all, automata learning algorithms build upon the L^* algorithm, which is foundational within the research field commonly known as "active automata learning".

The next step is to form a hypothesis to integrate automata learning and SCT. In our first hypothesis, we are interested in using L^* to synthesize a supervisor. Doing so provides an initial proof of validity about the integration and also leverages upon the known foundations. The work thus far is built upon incrementally updating and refining the hypothesis to tackle the problems faced at each stage.

To test and evaluate our proof of concept hypothesis, we use existing toy examples known in the supervisory community. One example is the Machine Buffer Machine (*MBM*) introduced in Chapter 2. *MBM* is a simple example whose supervisor has all the properties we are interested in studying, such as *controllability* and *non-blocking* defined in Chapter 3. Furthermore, the *MBM* has certain additional properties that make it interesting to study, such as being extendable either by cascading several other *MBM*s or varying the buffer's size to obtain larger models. Apart from the *MBM*, the Cat and Mouse example seen in Paper C is used to evaluate the algorithms. Validating the obtained results is a relatively easy task in the case of the toy examples as their models are known.

In addition to the toy examples, a sub-component of a self-driving car is learned. Validating the results, in this case, is a challenging task since the complete behavior is unknown. Hence, we rely on simulation-based methods and visual inspection (see Section D7).

1.5 Outline

This thesis consists of two parts. Part I is a general introduction that puts the appended papers into context. Part II contains the appended papers. Part I is organized as follows: Chapter 2 gives a background to discrete event systems and introduces the modeling formalism used in this thesis. Chapter 3 provides some definitions and introduces supervisory control. Chapter 4 introduces the field of active automata learning. In Chapter 5 the main contributions of this thesis are summarized. In Chapter 6 arguments for the correctness of the modular approach are discussed. Chapter 7 contains a summary of the appended papers. Part I ends with some closing remarks and directions for future work in Chapter 8.

CHAPTER 2 PRELIMINARIES

The systems dealt with in this thesis relate to a class of systems called *discrete event systems* (DESs) [13]. These systems can be modeled using *states* and *transitions*, where the system occupies a state at any given time and can transit between the states as defined by the transitions. Common examples include traffic control systems, automated manufacturing systems, and communication protocols. In this chapter, a formalism to model and study DESs is introduced.

2.1 Modeling Formalism

There are several different formalisms to model a discrete event system, for instance Petri nets [26], Transition Graphs [27], and Formal Languages [13], [15]. This thesis we will use *deterministic finite-state automata* [28] as the modeling formalism.

Deterministic Finite-State Automata

Deterministic finite-state automata (DFA) a.k.a *deterministic finite state machines* or simply *automata* [28] are commonly used to model discrete-event systems. In this formalism the system is abstracted into states, where a state

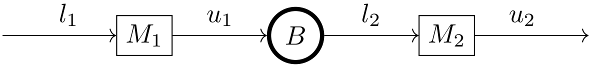


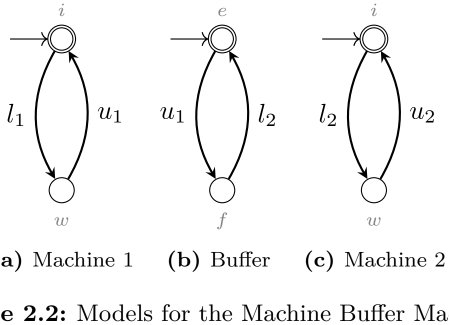
Figure 2.1: Machine Buffer Machine

defines a particular configuration of the system. The system transits from one state to another according to some defined transition. Each transition is labeled with an *event*. The state changes, in the DFA, are considered to be atomic, hence are instantaneous and so take zero time. Before any transition, the system is in its *initial state*. A particular string of events can lead the system, from its initial state, to some desired state; such a desired state is called a *marked state*, and signifies a state of particular importance. The set of all events of the automaton is known as the *alphabet*.

Definition 1 (DFA)**:** *A DFA is defined as a 5-tuple $\langle Q, \Sigma, \delta, q_i, Q_m \rangle$, where:*

- *Q is the finite set of* states*;*
- *Σ is the* alphabet *containing the finite set of events;*
- *$\delta : Q \times \Sigma \rightharpoonup Q$ is the partial* transition function*;*
- *$q_i \in Q$ is the* initial state*;*
- *$Q_m \subseteq Q$ is the set of* marked states.

A DFA can be visualized using a directed graph with nodes that represent the states, and edges that represented the transitions.



(a) Machine 1 **(b)** Buffer **(c)** Machine 2

Figure 2.2: Models for the Machine Buffer Machine

Example 1. *Consider the example with two machines and a buffer, called MBM, shown in Figure 2.1 [15]. Two identical machines M_1 and M_2 are connected with a buffer B in-between. Each machine can load a part and then unload it, represented by the events l_1 and u_1, respectively, for M_1, and l_2 and u_2 for M_2. When M_1 unloads a part, the part moves to the buffer; when M_2 loads a part it does so from the buffer. The models representing the behaviors of M_1, M_2, and B are shown in Figure 2.2.*

Modular Model

Modeling all possible behaviors as a single DFA, referred to as a *monolithic model*, is not an easy task. Even small practical systems end up with several thousand states. Thus, making the task of modeling error-prone and time-consuming. Instead, it is beneficial to model smaller interacting modules. Each of these modules defines the behavior of one part of the system. In Example 1 discusses the behavior of the *MBM* that is modeled using three automata seen in Figure 2.2. The automaton in figures 2.2a and 2.2c correspond to the machines M_1 and M_2, and Figure 2.2b models the buffer B. The complete behavior can then be obtained by performing *synchronous composition* over all the small models.

Definition 2 (Synchronous Composition)**:** *Let $G_1 = \langle Q_1, \Sigma_1, \delta_1, q_{i_1}, Q_{m_1}\rangle$ and $G_2 = \langle Q_2, \Sigma_2, \delta_2, q_{i_2}, Q_{m_2}\rangle$ be two automata. The synchronous composition of G_1 and G_2 is given by:*

$$G_1 \| G_2 = \langle Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (q_{i_1}, q_{i_2}), Q_{m_1} \times Q_{m_2}\rangle,$$

where:

$$\delta(\langle q_1, q_2\rangle, e) = \begin{cases} \{\delta_1(q_1, e)\} \times \{\delta_2(q_2, e)\} & \textit{if } e \in (\Sigma_1 \cap \Sigma_2), \\ \{\delta_1(q_1, e)\} \times \{q_2\}, & \textit{if } e \in (\Sigma_1 \backslash \Sigma_2), \\ \{q_1\} \times \{\delta_2(q_2, e)\}, & \textit{if } e \in (\Sigma_2 \backslash \Sigma_1), \\ \textit{undefined}, & \textit{otherwise.} \end{cases}$$

The worst case number of states for the synchronized result of two automata G_1 and G_2 is $|Q_1| \times |Q_2|$. Hence, the number of states increases exponentially with the number of modules, and very quickly becomes unmanageable. This is known as the *state-space explosion* problem.

Alphabets, Strings, and Languages

Recall that the alphabet Σ is a finite set of events. Let Σ^2 be defined as $\Sigma\,\Sigma$, i.e the set of sequences of events, called *strings*, of length 2 formed by *concatenation*. Similarly, $\Sigma^{(n+1)} = \Sigma^n\,\Sigma$, and Σ^* denotes the set of all strings of finite length including $\Sigma^0 = \{\varepsilon\}$, the empty string.

A string s is a *prefix* of a string u, if there exists a string t such that $u = st$; t is then a *suffix* of u. For a string $s \in \Sigma^*$, its *prefix-closure* $\bar{s}$ is the set of all prefixes of s, including s itself and ε. A set Pr is said to be *prefix-closed* if the prefix-closures of all strings are also in Pr, that is $\overline{Pr} = Pr$. Similarly, for a string $s \in \Sigma^*$, its *suffix-closure* $\underline{s}$ is the set of all suffixes of s, including s itself and ε. A set Su is said to be *suffix-closed* if the suffix-closures of all strings are also in Su, that is $\underline{Su} = Su$.

The transition function $\delta(q,\sigma)$ denotes the state reached by the transition labeled with σ from the state q, if $\delta(q,\sigma)$ is defined. This notation can be extended to strings, and defined recursively as follows, with $q \in Q$, $\sigma \in \Sigma$, $s \in \Sigma^*$:

$$q = \delta(q,\varepsilon),$$

$$\delta(q,\sigma\, s) = \delta(\delta(q,\sigma), s).$$

A *language* $\mathcal{L} \subseteq \Sigma^*$ is a set of strings over Σ. This gives rise to the notion of languages defined by a DFA. Consider the set of all deterministic finite automata denoted by $\mathcal{A}$. Given $A \in \mathcal{A}$, the language *generated* by A, $\mathcal{L}(A) = \{s \in \Sigma^* \mid \delta(q_0, s) \text{ is defined}\}$, is the set of all strings defined from A's initial state, q_0; the *marked* language, $\mathcal{L}_m(A) = \{\, s \in \mathcal{L}(A) \mid \delta(q_0, s) \in Q_m \,\}$ is the set of all strings that lead to a marked state [13]. While the generated language denotes behavior that is possible but not necessarily accepted, the marked language denotes possible behavior that is accepted.

Converting Regular Languages to Deterministic Automata

In this thesis, we focus on *regular languages* [28]. These are a class of languages that can be represented using DFA. There are several automata that represent a given regular language. However, for every regular language there always exists a unique automaton that has the least number of states and this is called the *minimal automaton* [13]. Furthermore, this minimal automaton can be calculated efficiently [29].

The *Myhill-Nerode theorem* [28], [30], introduced in the 1958 by Anil Nerode and James Myhill, provides a way to construct the minimal automaton. The theorem introduces an *equivalence relation* on the strings, known as the *Nerode Equivalence.* This equivalence relation partitions the set of strings into *equivalence classes.* Each of these equivalence classes corresponds to a state in the automaton.

Definition 3 (Nerode Equivalence)**:** *Given a language $\mathcal{L}$ over the alphabet Σ, two strings $u, v \in \Sigma^*$ are Nerode equivalent, denoted by $u \equiv_{\mathcal{L}} v$, if for all $w \in \Sigma^*$, $u\,w \in \mathcal{L}$ if and only if $v\,w \in \mathcal{L}$.*

According to this definition, two strings u and v in a language are equivalent, if when extended with a given suffix w the extended strings $u\,w$ and $v\,w$ are either both in the language or not. Conversely, if two strings are not equivalent, there exists a *distinguishing suffix* that, when appended to both the strings, will result in one of the extended strings existing in the language and the other not being in the language.

The Nerode equivalence for a regular language results in a finite set of equivalence classes. Each equivalence class maps to one state in the minimal automaton; this mapping is one-to-one. Thus, the number of equivalence classes are equivalent to the number of states in the minimal automaton. This is given by the Myhill-Nerode Theorem.

Theorem 1 (Myhill-Nerode Theorem)**:** *The language $\mathcal{L} \subseteq \Sigma^*$ is regular if and only if the equivalence relation $\equiv_{\mathcal{L}}$ represents a finite number of equivalence classes. Furthermore, there exists a DFA M that represents the language $\mathcal{L}$ and has exactly one state for each equivalence class of $\equiv_{\mathcal{L}}$, and M is the minimal automaton.*

The Myhill-Nerode theorem shows whether a language is regular (or not), but more importantly it is leveraged in Paper A to learn a minimal automaton of a system under learning.

A Note on the Use of DFAs in this Thesis

In this thesis, the above presented definition of DFA is used. However, it is important to point out a subtle difference in how the notion of states is used in Paper A vs Paper B and Paper C. Paper A uses the definition of states as defined above. Here, the states do not hold any information about the system. They are locations used to represent the state reached by a string. Two states can be differentiated according to the Nerode equivalence. Hence, two different states can correspond to the same configuration of the system. On the other hand, in papers B and C the state explicitly holds the values of some variables of the system. That is, the state is defined as the valuation of the variables. Hence two different states cannot represent the same configuration. This is presented formally in Section B2. This subtle but important difference forms the bedrock for the two algorithms presented in papers B and C.

CHAPTER 3

SUPERVISORY CONTROL

Supervisory Control Theory (SCT) introduced in [15] by Ramadge and Wonham provides a mathematical framework to synthesize supervisors that ensure a DES satisfies certain specifications. The assumption here is that the *plant* – that needs to be controlled – does not meet certain *specifications*. Hence, using SCT, a *supervisor* can be calculated. This supervisor interacting with the plant forms a closed-loop system. Here, the supervisor can dynamically disables certain events that the plant could have otherwise generated. Thus, ensuring that the controlled system behaves in accordance with the specifications.

The DES model of the plant (denoted by G) captures everything the system is capable of doing. The specification (denoted by K) expresses the system's required behavior, including the aspects that ensure the system operates safely and ultimately can reach its intended goal. Given complete freedom, the plant would violate the specification. The objective is to calculate a supervisor (denoted by S) – a device to restrict the behavior of the plant. This supervisor ensures that the plant does not violate the specification and fulfills its intended goal. It is worth noting that the supervisor can only restrict the behavior by disabling certain events. It is the plant (or an external system) that decides which transition to take.

To restrict the behavior of the plant, the supervisor needs to be connected

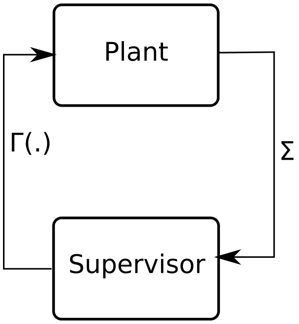


Figure 3.1: Feedback loop of supervisory control.

to the plant in a feedback loop as seen in Figure 3.1. The supervisor continuously listens to the events generated by the plant. It can disable a subset of controllable events based on past behavior (according to some function) to ensure that the specifications are not violated. The closed-loop behavior of the plant under control is given by the synchronous composition $G \,||\, S$.

Note that in the above explanation, and the rest of the thesis, we assume that all the events are observable by the supervisor.

Originally, SCT was introduced based on a language formulation [15]. In [31] it was shown that, when modeled as DFA, the synchronous composition can describe the behavior of the plant under the control of the supervisor.

3.1 Properties of the Supervisor

The supervisor calculated using the SCT framework satisfies certain properties. Here, we look at the properties important for this thesis.

Controllability

Recall that the DES consists of transitions that take the system from one state to the next. Each transition is associated with an event. The event set is partitioned into into two disjoint subsets, Σ_c containing the controllable events and Σ_u containing the uncontrollable events. The plant is allowed to take a transition labeled with an uncontrollable event anytime such an event is enabled in a state. Thus, the supervisor is allowed to disable only the controllable events and not the uncontrollable ones. Hence, we can say that a supervisor is controllable if the synchronous composition of the supervisor and the plant never disables a transition labeled with an uncontrollable event.

Formally, this is expressed as:

Definition 4 (Controllability)**:** *For a plant G and uncontrollable events Σ_u, a supervisor S is said to be* controllable *if* $\forall s \in \mathcal{L}(G \,||\, S), \sigma \in \Sigma_u : s\sigma \in \mathcal{L}(G) \implies s\sigma \in \mathcal{L}(G \,||\, S)$

To distinguish between controllable and uncontrollable events, uncontrollable events are marked with an exclamation mark (!) in the figures.

Non-blocking

A controllable supervisor need not always be useful. The controllable supervisor guarantees that the plant does not violate the specification; however, the case may be that the supervisor does not let the plant do what it is supposed to. For example, the supervisor might allow the plant to reach a state from which the plant cannot continue; or, the plant enters into an endless loop. To avoid such a scenario, the desired goal of the plant needs to be defined by the specification. This is achieved using the notion of *marked states*. The supervisor calculated should then ensure that the plant can always reach one or more of these marked states. That is, the supervisor should be *non-blocking*. In other words, non-blocking is used to guarantee that the supervisor does not restrict the plant from doing what it is supposed to do.

Definition 5 (Non-blocking)**:** *For a plant G, a supervisor S is said to be* non-blocking *if the closed-loop system $G \,||\, S$ is non-blocking, that is $\mathcal{L}(G \,||\, S) = \overline{\mathcal{L}_m(G \,||\, S)}$.*

In other words, the closed-loop system is non-blocking if every reachable state can continue to reach some marked state.

Maximally Permissive

Given a plant and its specifications, several controllable and non-blocking supervisors exist that satisfy the specifications. The difference between these supervisors is in how aggressive they are in restricting the plant's behavior. In an aggressive approach, the supervisor restricts as much behavior of the plant as possible, resulting in a minimal behavior of the controlled system. On the other hand, the least restrictive (aka maximally permissive) approach produces a supervisor that restricts the plant only when needed. Thus, the resulting controlled system contains all possible behaviors satisfying the specifications.

In this thesis, the goal is to calculate the maximally permissive, controllable, and non-blocking supervisor. The conditions under which such a supervisor exists are well established in literature [32]. Furthermore, it is known that such a supervisor is unique for a given plant and specification [16]. It is also known that it is computable (that the computation of it will terminate). However, it does not always exist, which is when the computation terminates with the degenerate null supervisor.

3.2 Synthesis

One of the many ways to calculate a supervisor is by converting the initial controllability problem into one of non-blocking using the *plantification* [33] step. In this step, each specification automaton K, in the set of specifications, is transformed into a Σ_u-*saturated* automaton $K^\perp$, by adding to every state, for every uncontrollable event not enabled in that state, a transition to a new blocking state $\perp$.

The synthesis algorithm first calculates the synchronous composition of the plant G and the plantified specification $K^\perp$. The result is then iteratively pruned to remove all blocking and uncontrollable states. By removing all states (and their corresponding transitions) that violate the properties controllability and non-blocking, and keeping all the other states results in the maximally permissive controllable and non-blocking supervisor [34].

Definition 6 (Plantification)**:** *For $K = \langle Q, \Sigma, \delta, q_0, Q_m \rangle$, the Σ_u-saturated automaton $K^\perp$ is given by $K^\perp = \langle Q \cup \{\perp\}, \Sigma, \delta^\perp, q_0, Q_m \rangle$, where $\perp$ is a new state, and*

$$\delta^\perp(q,u) = \begin{cases} \perp & q \in Q, u \in \Sigma_u, \delta(q,u) \textit{is undefined} \\ \delta(q,u), & \textit{otherwise.} \end{cases}$$

Example 2. *To illustrate the different properties and the synthesis steps consider MBM. The events corresponding to unloading a part from the machines are uncontrollable and hence written as $!u_1$ and $!u_2$. Machine M_1 and M_2 form the plant, and B is treated as the specification. The model obtained by synchronizing the three components consists of eight states and twelve transi-*

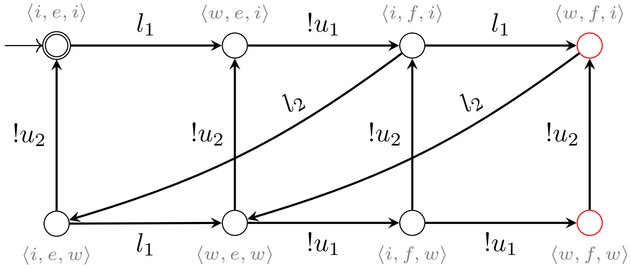


Figure 3.2: The synchronous composition of M_1, B, and M_2

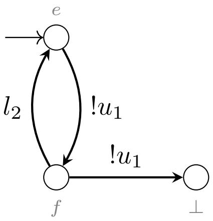


Figure 3.3: Plantified B

tions as seen in Figure 3.2.

It is evident that the obtained model is non-blocking, i.e., it is possible to reach the marked state from every state. However, this model is uncontrollable as the two states to the right, $\langle w, f, i\rangle$ and $\langle w, f, w\rangle$, have the $!u_1$ event disabled. According to the plant model, M_1 can, by the $!u_1$ event, unload a part into the buffer. However, since the buffer is already full, it cannot take an additional part. Hence, a supervisor needs to be synthesized.

Figure 3.3 shows the plantified specification for $B^{\perp}$. This specification reaches a blocked state $\perp$ if M_1 unloads two parts without M_2 loading a part in between. The synchronous composition $M_1 \,||\, M_2 \,||\, B^{\perp}$ is seen in Figure 3.4. It can be seen here that states $\langle w, \perp, w\rangle$ and $\langle i, \perp, i\rangle$ are blocking and no state is uncontrollable.

The synthesis algorithm disables event l_1 from the states $\langle i, f, i\rangle$ and $\langle i, f, w\rangle$ avoiding the possibility of reaching the blocked states. The resulting automaton is the maximally permissive controllable and non-blocking supervisor is seen in Figure 3.5.

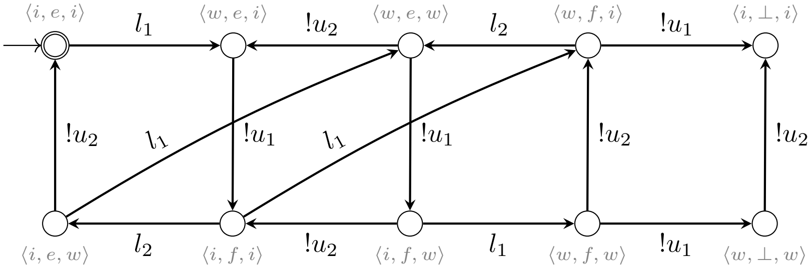

$\langle i, e, i \rangle$
l_1
$\langle w, e, i \rangle$
$!u_2$
$\langle w, e, w \rangle$
l_2
$\langle w, f, i \rangle$
$!u_1$
$\langle i, \bot, i \rangle$
$!u_2$
l_1
$!u_1$
l_1
$!u_1$
$!u_2$
$!u_2$
$\langle i, e, w \rangle$
l_2
$\langle i, f, i \rangle$
$!u_2$
$\langle i, f, w \rangle$
l_1
$\langle w, f, w \rangle$
$!u_1$
$\langle w, \bot, w \rangle$

Figure 3.4: The result of synchronization $M_1 B^{\perp} \,||\, M_2$

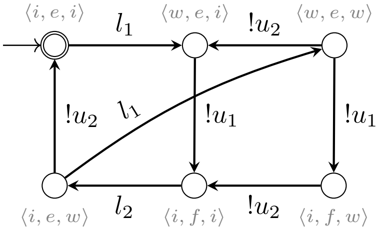

$\langle i, e, i \rangle$
l_1
$\langle w, e, i \rangle$
$!u_2$
$\langle w, e, w \rangle$
$!u_2$
l_1
$!u_1$
$!u_1$
$\langle i, e, w \rangle$
l_2
$\langle i, f, i \rangle$
$!u_2$
$\langle i, f, w \rangle$

Figure 3.5: The maximally permissive controllable and non-blocking supervisor

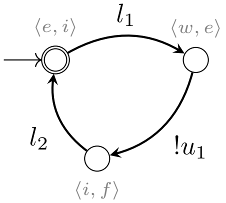


Figure 3.6: Modular supervisor for *MBM*

Modular Synthesis

Given a modular plant $\mathcal{G} = \{G_1, G_2, \ldots, G_n\}$ and its corresponding specification $\mathcal{K} = \{K_1, K_2, \ldots, K_n\}$, it is known [35] that a controllable modular supervisor can be calculated by selecting for each specification $K_i \in \mathcal{K}$, all plant components $G_j \in \mathcal{G}$ such that $\Sigma_{K_i} \cap \Sigma_{G_j} \cap \Sigma_u \neq \varnothing$ and performing monolithic synthesis on this sub-system. To guarantee a maximally permissive modular supervisor, also all plant components $G_k \in \mathcal{G}$ such that $\Sigma_{G_k} \cap \Sigma_{G_j} \cap \Sigma_u \neq \varnothing$ for each G_j previously selected have to be included. This selection of new plant components sharing uncontrollable events with the already selected ones has to be iterated until a fix-point. However, these latter G_k plant components can be included incrementally, as needed [35], to lessen the risk of including the monolithic plant. However, it must be noted that the modular supervisor does not guarantee non-blocking behavior.

Example 3. *The specification in MBM shares uncontrollable events with only M_1, and M_1 does not share uncontrollable events with any other components. Hence, it is sufficient to calculate a supervisor for specification B taking only M_1 into consideration. The resulting supervisor is much smaller as seen in Figure 3.6.*

3.3 Note about Marked States in the Plant

A point to highlight is that the marked states can be defined in the plant, the specification, or both plant and specification. The premise of this thesis is that the plant models do not exist. When the model does not exist, defining the marked states would require extensive knowledge of the system, akin to manually modeling the system. However, the author's view is that it is the specification's role to define these marked states. The plant must merely define all possible behaviors of the system modeled. Hence, the work in this thesis assumes all the states in the plant to be marked.

CHAPTER 4

ACTIVE AUTOMATA LEARNING

The behavior of many technical systems can be described using regular languages. Such systems include traffic control systems, manufacturing systems etc [13]. Manually describing the behavior of these systems is not easy. There exist techniques known as *automata learning* to automatically learn the behavior and represent it using automata.

Automata learning has been studied both as a theoretical problem, where its goal is to uncover some hidden function, and as a practical problem of attempting to represent some knowledge using a mathematical formalism such as an automaton [20]. Automata learning finds its origin in various fields of study: computational linguistics, machine learning, formal learning theory, pattern recognition, and computational biology. Hence, it is also known by different names depending on the field: Model learning, Model inference, Grammar induction, Grammar inference, etc. Though the different names can have different connotations, they all refer to similar ideas and processes. This chapter provides a broad overview of the field of automata learning and its application.

An automata learning algorithm, referred to as a *learner*, aims to learn a model that captures the behavior of the *system under learning* (SUL). These algorithms are broadly classified as *passive learning* when the learner is provided with data logs of the SUL, or *active learning* when the learner can directly interact with the SUL. Passive learning is a setting in which labeled data is provided to the learner. The learner is tasked to find a model that represents this data [36]–[38]. The data is usually an observation log from the SUL. The observations are labeled as accepted (or rejected) if they belong (or do not belong) to the marked language. Thus, the models learned using passive learning methods are in-exact; they represent only those behaviors that were provided in the observation logs. Behaviors possible by the SUL but not observed in the log (for example, observations of errors that rarely manifest) are not included in the model. However, we are interested in learning a model that contains all possible behaviors of the SUL, with the intention to synthesize a controllable and non-blocking supervisor. Doing so is possible using active learning, thus making it the focus of this thesis.

4.1 Active Learning

Active learning, in contrast to passive learning, learns a model by experimentation. Given a SUL, the goal is to find an automaton model that represents the behavior of the SUL. To this end, the learner observes the behavior of the SUL for some set of inputs. The term *active* implies that the inputs are chosen by the learner.

The active learner can pose queries to a *teacher*. Based on the responses from the teacher, the learner updates its knowledge about the SUL. It is assumed that this teacher is an abstract entity that knows the language of the SUL and can correctly answer certain types of queries.

There are several types of queries depending on the application and field of study [20]. For the purposes of this thesis, we are interested in the following two types of queries (interested readers are directed to [20] for a more extensive list of queries):

- **Membership Queries (MQs):** are made by asking the teacher if a given string is allowed by the SUL. The teacher responds with a `YES` or `NO`.

- **Equivalence Queries (EQs):** are made by the learner presenting a *hypothesis* automaton that the learner believes to represent the SUL. The teacher replies positively if this hypothesis correctly represents the SUL. Else, a counterexample needs to be provided that differentiates between the SUL and the submitted hypothesis. This counterexample is a string that is allowed in the SUL but not in the hypothesis, or the other way around.

Learning a DFA that correctly represents the language of the SUL cannot be done using only MQs or only EQs [20], [39], [40]. However, [17] shows that it is possible to learn such a DFA by using both MQs and EQs, and presents L^* an algorithm for doing this. Moreover, [17] introduces the concept of *minimal adequate teacher*, a teacher that can answer both MQs and EQs.

The Archetype Learner

Algorithm 1: The learning loop

```
Input: Access to a MAT answering MQs and EQs w.r.t a target DFA $\mathcal{A}$
Build initial hypothesis $\mathcal{H}$ using MQs
while EQ for the $\mathcal{H}$ does not succeed do
    Let $c \in \Sigma^*$ be the counterexample returned by the MAT
    Update $\mathcal{H}$ using MQs taking $c$ into account
end
return Final hypothesis $\mathcal{H}$ that is equivalent to $\mathcal{A}$
```

Nearly all algorithms that are developed using the MAT model follow the same structure. The pseudo-code for this is provided in Algorithm 1 which we refer to as the learning loop. The learner starts by creating an initial hypothesis using only MQs. This hypothesis is then used to pose an EQ. If a counterexample is found, it is taken into consideration to refine the hypothesis (using MQs). The refined hypothesis is again used to pose an EQ, and this process continues until no counterexample can be found. Since this loop terminates only when no counterexample is found, it inherently provides some guarantee for the correctness of the algorithm.

L^* was the first algorithm that showed the possibility of learning a DFA for some unknown language. The learner, L^*, makes MQs and stores the responses in an *observation table*. This observation table is a data structure from which a hypothesis automaton is obtained. The MQs are made until the observation table is *closed* and *consistent*. A closed table ensures that all transitions in the corresponding automaton lead to valid states, while a consistent table ensures there exists no non-deterministic behavior in the automaton. When a closed and consistent table is obtained, L^* generates a hypothesis to pose an EQ. A counterexample, if obtained, is used to extend the observation table, and the learning continues repeating the process in accordance with Algorithm 1.

The theoretical worst-case size of the observation table, in L^*, is calculated to be $(k+1)(n+m(n-1))n$, where $k = |\Sigma|$ is the size of the alphabet, $n = |Q|$ is the number of states, and m is the maximum length of any counterexample presented by the MAT [17]. This is also the upper bound for the number of MQs, and thus the complexity is $\mathcal{O}(m|\Sigma||Q|^2)$. Assuming that $m = |Q|$ in the worst case, the number of MQs (and the size of the observation table) is $\mathcal{O}(|\Sigma||Q|^3)$.

It must also be noted that learning a prefix-closed language results in a larger number of MQs and EQs compared to learning non-prefix-closed languages. For instance, [41] observes that the required number of MQs grows quadratically in the number of transitions when learning prefix-closed languages. Further empirical studies in [42] show that in general more queries are required (i.e., it is harder) to learn a DFA with more marked states ($\approx |Q|$) and that the number of MQs per EQ grows linearly as a function of $|\Sigma|$ and $|Q|$.

The work in this thesis shows a way to extend L^* to learn a supervisor of the SUL. The modifications to L^* presented in Paper A can be ported (with minimal effort) to other algorithms developed around L^*. The following section provides a glimpse into learning algorithms that build upon L^*.

Algorithms that build upon L^*

There have been a handful of algorithms that build upon L^*. Some of these algorithms improve upon the complexity of the learning algorithms. This is mainly achieved by using more efficient data structures. The Kearns-Vazirani algorithm [43] builds upon L^* but stores the gained knowledge of the SUL in a tree data structure – *discrimination trees* – rather than an observation table. In principle, the algorithm uses MQs and EQs to refine the discrimination tree. The update is, in essence, sifting strings into the tree, thus increasing the size of the tree to include an additional node and leaf. Such growth in size is much better in comparison to the observation table. The Rivest and Schapire algorithm [44] improves on L^* by handling counterexamples that include a homing sequence when it is not possible to reset the SUL. Here, [44] introduce binary search to find a suffix of a counterexample that refines the hypothesis. Using only the interesting information in the counterexample leads to better performance of the algorithm. The idea of discrimination trees and homing sequence is combined in [45] to introduce Observation Packs. These are trees of the information collected called *components*. Each component corresponds to one state in the hypothesis. A component consists of suffix, prefix, and an access sequence, as well as a local observation table derived from the suffix and prefix associated with that corresponding state. The idea of Observation Packs is further explored by Malte et al. [46] who suggest the TTT[1] algorithm. The idea behind TTT is that it continuously maintains the counterexamples in trie structures, efficiently eliminating the evaluation of redundant queries.

In general, L^* performs the largest number of MQs, compared to the other algorithms, before the final model is created. But because L^* collects more information before presenting a hypothesis, it is also more likely to produce fewer false hypotheses and fewer EQs compared to the other algorithms [47]. However, the upper bound on the number of EQs is the same for all algorithms [41].

Apart from improving the complexity, another class of algorithms aims to learn different formalisms depending upon the application. For example, the NL^* algorithm learns a non-deterministic automaton [48]; in [49], [50] an approach to learn symbolic register automata is presented; learning of weighted automata is presented in [51]; automata learning from a categorical perspective is done in [52], [53].

Automata Learning Integrated into Other Model-Based Techniques

Model checking [14] is a technique to automatically verify if a given model satisfies certain properties. Combining automata learning and model checking has been introduced in [54], [55] and is called *adaptive model checking*.

The basic idea in adaptive model checking is to use automata learning techniques to learn a system model. Then, use model checking algorithms to verify if the learned model satisfies certain properties needed by the system. If the properties hold, the learning algorithm makes EQs and continues according

[1] The name is derived from *Spanning Tree*, *Discrimination Tree*, and *Discriminator Trie*; the three concepts fundamental to the algorithm.

to the learning loop.

If the properties do not hold, an error trace is provided by the model checking algorithm. This error trace is a string that violates one of the properties when traversed by the system. The adaptive model checking algorithm executes the string on the real system and observes if it indeed violates some property. If it does, the adaptive learning algorithm concludes that the system does not satisfy one or more properties. On the other hand, if the system correctly executes the error trace without violating the property, the error trace is used as a counterexample, and the learning algorithm updates its knowledge and continues learning.

Learning-based testing [56], [57] is another approach where automata learning (both active and passive) approaches are integrated into, and used for, testing whether the system satisfies its specifications. In this approach, a hypothesis model is first created using the inputs to a system and their corresponding outputs. A model is synthesized based on the input/output pairs. The testing algorithm tries to find a counterexample where the model violates the specifications. This counterexample is then tested in the actual system. If the test fails, the testing algorithm terminates, producing a failure report. If the test passes, the input/output pair is taken as input to the learning algorithm to create a new model, and the testing continues. The algorithm terminates when a bound on the maximum testing time is reached or a bound on the maximum number of test cases is fulfilled.

4.2 Applications of Active Model Learning

This thesis contains one real-world example in Paper D, it is good to mention a few others that use active learning. This is by no means a complete list:

- **Verifying communication protocols:** Active automata learning has been applied to verify communication protocols using Mealy machines [18], [58]. In [59] automata learning is used to find bugs in the closed-source implementation of the TCP protocol in Windows. Furthermore, in [60] the authors found a bug in the sliding window TCP implementation in Linux. Other protocols like the MQTT [61], TLS [62], and SSH [63] have been learned using active automata learning. In [64] active automata learning is applied to SSL/TLS for hostname verification. Several different implementations are learned and analyzed, discovering eight unique vulnerabilities.

- **Verifying smart cards and passports:** In [65] a model of bank cards is learned. This is a really good example of a real black-box system since it is not possible to access the code or perform any other type of monitoring for these cards. In [66] a lego machine to operate the bank card machine is constructed to automatically test inputs. Using this machine as an interface for active learning, the authors could find a security flaw in the card reader. In [67] an abstracted model of the biometric passport protocol is learned.

- **Learning models of software systems:** In [68] active automata learning is applied to learning register automata models of software programs. [69] focus on learning embedded software programs for industrial printers. [70] apply automata learning to learn the control system of legacy software systems at Philips. They learn both an old version and a new version of the same component. Comparing these models presents an insight into the differences between them, giving developers the opportunity to solve problems before replacing the old components. There have been some attempts to apply automata learning to automotive systems in [56], [71].

4.3 Available Tools

There exists a handful of automata learning tools, each with its own specialization, requirements, and benefits. The following is a brief overview of the most well-known tools:

- *LearnLib:* LearnLib [72] is a free and open source library for automata learning, written in Java and being actively developed. It provides a framework for conducting research on learning algorithms and their application. LearnLib supports a number of learning algorithms for active as well as passive learning. Though most of the work in LearnLib focuses on Mealy machines, it does support other modeling formalism such as non-deterministic automata, deterministic automata, and visibly pushdown automata. Additionally, it provides several equivalence approximation strategies including variations of the W-method [73], Wp-method [74], and random walks. It also provides logging facilities to analyze the learning statistics including number of MQs and EQs, running time, and memory consumption.

Recently, there has been a number of new tools that use LearnLib as the learning engine, RALib [75], ALEX [76] and Tomte [77] are notable mentions. RALib is used to learn register automata. ALEX (Automata Learning Experience) provides an easy to use mechanism to learn models of web applications. And Tomte makes it possible to learn models that have a large or even infinite alphabet.

- *LibAlf:* The libalf [78] library is an open-source library for learning finite-state automata written in C++. It provides well-known learning techniques, such as Angluin's L^* [17] for active learning, and Biermann's learning [79] approach for passive learning. However, it has not been in active development in recent times.

- *RALT:* RALT [80] (Rich Automata Learning and Testing) is a closed-source tool developed for France Telecom. This tool is focused on integration testing in the absence of a model. It first learns a hypothesis model of the system, and then generates test strings to test it. RALT supports algorithms for deterministic finite automata, mealy machines, and simple parameterized machines.

CHAPTER 5

LEARNING SUPERVISORS

Chapter 3 introduced supervisor synthesis, an approach to automatically compute supervisors for a given plant and specification. These methods assume access to usable plant and specification models. However, manually constructing these models is difficult, time-consuming, and prone to errors. Automata learning, introduced in Chapter 4, provides an approach to interact with a SUL and learn a model that represents its behavior. Using automata learning to replace the manual step of constructing the supervisor would indeed benefit the SCT community. This chapter discusses an approach to integrate the two fields of study. First, some related work is presented.

Within the supervisory control community, there have been a handful of works that incorporate active automata learning, specifically L^*, to calculate supervisors. A synthesis approach for concurrent systems where the specification is not explicitly defined but known to the designer is presented in [81]. In [82] an algorithm S^* is presented that extends upon L^* to synthesize a maximally permissive supervisor when the specification is not available in a known format. An example is studied to show the synthesis of a supervisor when the specification is not a regular language. The main difference between [81] and [82] is that the latter presents a way to implement the teacher queries, whereas [81] treats the designer as the teacher.

Both [83] and [84] use L^* to learn a supervisor without having access to the plant. The former modifies L^* to learn an N-step controllable supervisor, and the latter learns a maximally permissive controllable supervisor. In [85] automata learning is used to learn a decentralized controller for multi-robot coordination.

5.1 Missing Pieces

Recall that automata learning methods assume access to a teacher responsible for answering certain queries. The question arises, *how does the teacher answer these queries?*

In the original L^* the teacher is assumed to be an abstract entity that has complete knowledge of the unknown language. However, to practically apply active learning, we require concrete techniques to answer the two types of queries, membership queries and equivalence queries. To this end, we use a simulator that can, by simulating the SUL, answer MQs. Furthermore, we adopt ideas from the *testing community* to use the simulation to test the hypotheses and find counterexamples.

Simulation

Simulations provide several advantages in comparison to using a physical system. Unlike the physical system, the simulation can run faster than real-time, even multiple instances in parallel, thereby speeding up the learning process. Event sequences that might result in dangerous collisions and unforeseen errors are confined to the simulation, providing a safe learning environment. Additionally, once a simulation is available, the necessary financial investment relates to obtaining powerful computers, which in today's world is relatively cheap.

Simulation-based design and development is well adopted in most industries [21]. For example, the manufacturing industry has seen an interest in virtual commissioning [86] where a virtual model of the system is first created and tested along with its actual control code before physically building the system. Hence, there already exists the practice of developing simulation models, these can be used to learn logical models of the systems.

It is important to highlight the requirements of the simulation in light of this thesis. This work aims to learn a discrete model; hence, we assume the simulator to be a discrete system as well. In most cases, these are not discrete, neither in time nor variable values. However, we assume that the simulation can be abstracted to behave like a discretized system.

Discrete systems work with the assumption of instantaneous events. That is, a transition from one state to another takes no time. At each state, the simulation has enabled events that can be executed, via an interface, to perform specific actions. When such an action has been performed, the state of the simulation is updated, resulting in a new set of enabled events. Thus a string of events can be executed, taking the simulation from one state to another. If an event is requested to be executed that is not enabled by the simulator at a particular state, the simulator replies with an error message.

Though theoretically useful, the idea of instantaneous events does not translate well to real-world systems where the transition from one state to another takes time. For such systems, an abstraction of *operations* can be used, where an operation in the real system relates to an event. In this work we will define two types of operations, *two-state operations* and *three-state operations* [87].

Two-state operations are similar to instantaneous events and are usually used for a simulator where state changes are a matter of updating variables, and hence instantaneous. Three-state operations have the three states `initial`, `executing`, and `finished`. A three-state operation is triggered from its `initial` state. The operation is in the `executing` state while the simulator executes the operation. Only when the operation has completed and is in its `finished` state, is the learning algorithm notified to continue. Thus, of interest to the learning algorithm are the `initial` and `finished` states. Three-state operations are used particularly in the PLC simulator using OPC-UA of Paper A.

Furthermore, it should be possible to observe a subset of the variables used in the simulation. Specifically, those variables that relate to the logical behavior of the system. This specific requirement applies mainly to the algorithms contained in papers B and C.

Finding Counterexamples

An important step during active learning is equivalence checking: given a hypothesis, does this hypothesis describe the behavior of the SUL? An obvious way is to test every string in the hypothesis against the SUL. However, this is an arduous task as the error can be hidden deep within the state-space. The presence of loops in the hypothesis adds to the complexity. Also, there is a need to run a possibly unbounded number of test cases to find counterexamples. Under the condition that we aim to learn a model that has a minimal number of states, there exist testing methods [88], [89] that can be leveraged to find counterexamples for the purposes of active learning.

Some of the commonly used methods are random walk [90], the W-method [73], the Wp-method [74], and the HSI-method [91]. Using the random walk algorithm does not guarantee full coverage of the state-space. Hence, they are usually used for learning models that are not required to be fully correct [17]. The W-method provides a way to generate test cases for a given number of states. This idea is further developed in the Wp-method and the HSI-method to reduce the number of test cases generated. For this thesis, the W-method sufficed and was used in Paper A, where details are found in Section A2.3.

To apply the W-method, we assume a known bound on the number of states in the SUL. However, in practice, this information is not always known in advance. Setting a bound lower than the actual will terminate the learning algorithm prematurely, resulting in an incomplete model. On the other hand, a higher value could result in the non-termination of the algorithm. One option to avoid setting the upper bound is to look for counterexamples for only N-step ahead. Such an option works, reducing the burden on validating if the learned model completely represents the language of the SUL or not. In this thesis, the learned automaton represents the generated language. Thus, making it feasible to use a 1-step or a 2-step look ahead counterexample generator.

An unwanted consequence of using testing methods to find counterexamples is the exponential growth of MQs. The test strings generated are used to perform MQs in the SUL. The responses obtained are compared to the expected value from the hypothesis.

5.2 Learning Supervisors

To learn a supervisor, a new algorithm $SupL^*$, based on L^*, is presented in Paper A. Unlike the original L^* the membership queries include querying, not just the simulated plant but also the specifications. The details on what these queries contain are found in Section 2.2. It suffices to here say that these queries can return values $\{0, 1, 2\}$ instead of only $\{0, 1\}$. Here, the value 2 corresponds to strings that belong to the marked language. 1 corresponds to strings in the generated language; invalid strings have a value 0.

It is now guaranteed that the above modifications will always result in a maximally permissive controllable supervisor. For this, the controllability problem must be converted into one of non-blocking by plantification of the specification, as introduced in Chapter 3. As a consequence of plantification, the total number of states in the learned automaton increases. The resulting learned automaton is the maximally permissive controllable supervisor. The blocking states added due to plantification need to be deleted, along with any uncontrollable states, to obtain the maximally permissive controllable and non-blocking supervisor. However, the practical cases to which this algorithm can be applied are limited as the resulting model is monolithic and suffers from the state-space explosion [14] problem. Also, as observed in Paper D, the counterexample generation suffers due to the large number of states and transitions resulting in non-termination. Hence, we turn our attention to learning modular models instead of large monolithic ones.

Taming the State-Space Explosion Problem

The *SupL** takes a language-based approach. That is, the SUL is treated as a black-box entity to which inputs can be provided and outputs can be observed. More specifically, the inputs are strings of events, and the output indicates if the provided input is executable by the SUL or not. These event sequences are what give meaning to the states (in the automaton). Thus, it is not possible to directly learn a modular model just by observing these event sequences. Indirectly learning a modular model would still require exploration of the complete state-space.

There is the possibility of defining states based on the system's internal observations when it comes to cyber-physical systems. These observations include internal variables, sensor reading, and actuator values of the SUL. In such a setting, it is possible to explore the system's behavior and identify individual states based on the valuation of the internal variables, sensor reading, and actuator values. Starting from the initial state, the next states can be discovered by attempting to run all possible events in the simulator. The new states observed are further explored in a breath-first search manner until no new states can be discovered. Such an approach is adopted in papers B and C.

By default, such an exploration will result in traversing the entire state-space resulting in the state-space explosion problem. Additional information about the system is required to perform the search in a smart manner, such that searching the monolithic state-space is avoided. Section B3.3 introduces the Plant Structure Hypothesis (PSH) that provides information defining the different interacting components in the SUL. The learning algorithm can then exploit this knowledge to divide the learned information into separate modules and reduce the search space, thereby mitigating the state-space explosion problem.

Paper B introduces the Modular Plant Learner (MPL) algorithm that uses a PSH and a simulation to smartly explore the state-space to learn a modular model describing the SUL. Defining the PSH is not an easy task; it requires skill and in-depth knowledge of the SUL. Chapter 6 introduces the properties a PSH must adhere to and provides arguments for the correctness of modular learning using the PSH.

MPL interacts with the simulation to learn a modular plant model of the SUL. This model can then be used to synthesize a modular supervisor using well-known algorithms [34]. Instead of this two-step process, Modular Supervisor Learner (MSL), an algorithm for directly learning modular supervisors, is introduced in Paper C. Given a set of specification, MSL learns one supervisor for each specification. The results are promising as it was possible to learn a modular supervisor for the Cat and Mouse example [16] and the AGV example [92]. The latter is well known for its computational complexity in the monolithic setting.

5.3 MIDES – A Tool for Model Learning of Supervisors

This section presents MIDES (Model Inference for Discrete-Event Systems) a tool to learn finite-state models and supervisors. MIDES is a tool that is constantly updated, and the source can be found on Github [24].

MIDES is aimed to be used within the supervisory control [16] context. Hence, it is built to be used alongside Supremica [93] and uses the data structures available in Supremica, namely that of DFA described in Chapter 2.

MIDES contains implementations of the algorithms discussed in this thesis. Additionally, it contains interfaces to interact with external systems to act as simulators.

Tool Structure

MIDES is focused on a simulation-based approach to learning DFA models. It caters to both a *minimally adequate teacher* [17] and exploration-based learning. Its high-level structure is shown in Figure 5.1.

The user input consists of the simulation and the meta-model that make up the SUL. These are interfaced with MIDES using the appropriate interfaces. The learning algorithms in MIDES interact with the SUL through these interfaces. There is a possibility to learn either a supervisor when the meta-model provides specifications, or a plant model in the absence of specifications. As it is built upon the data-structures present in Supremica, the learning algorithms can use algorithms provided by Supremica.

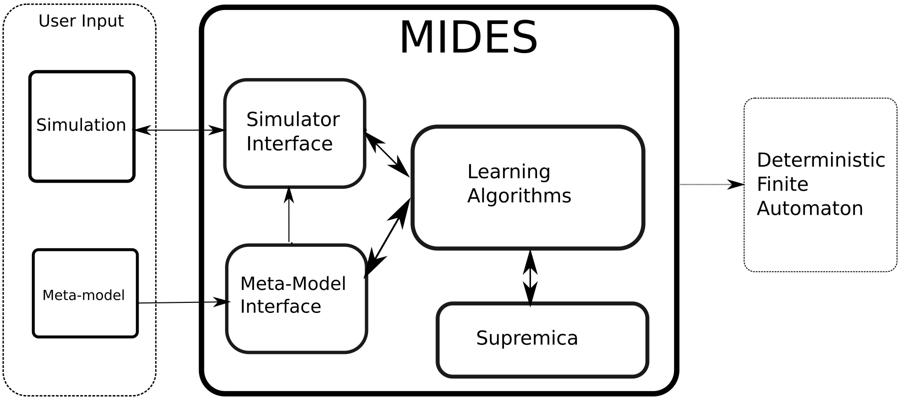


Figure 5.1: High-level structure of MIDES

System Under Learning (SUL)

The SUL is made up of two parts, the simulation and its corresponding meta-model. These need to be interfaced with MIDES using appropriate interfaces.

Simulation

Currently, MIDES supports PLC simulators, using the OPC-UA [94] protocol, a MATLAB [95] interface to communicate with MATLAB code, and an internal *code* simulator that simulates the behavior of a system defined using variables and logical predicates. Furthermore, an interface is provided to allow other simulation tools to be integrated.

Meta-Models

The learning algorithm also requires some additional information – a meta-model – to help interface with the simulation. This includes the alphabet of the SUL, the type of model (modular or monolithic) to learn, the state variables to monitor, and the system's initial and marked states. Additionally, in the case of learning supervisors, the specifications of the system are needed. In the case of learning a modular model, a Plant Structure Hypothesis (PSH) is required as described in Paper B.

5.4 Insights from the Case Studies

The usability of the work in this thesis is demonstrated by applying the implemented algorithms on two use-cases presented in Paper A and Paper D. The motivation for the case studies was to identify the bottlenecks in applying supervisor learning techniques in real-world applications. This section briefly highlights the experience and learning outcomes from these two methods.

Paper A studies the feasibility of these methods in a virtual commissioning setting. Here, *MBM* introduced in Example 1 is recreated in a simulation environment using the 3D simulation and virtual commissioning tool Xcelgo Experior [96], which is then controlled using a PLC. To reduce the complexity of the simulation model, the simulation is made using pushers and sensors. The pushers can push an object from one platform to another. The sensors are used to check if an object exists on a platform. We then use the $SupL^*$ algorithm to learn a model describing the system, as well as its supervisor.

Paper D looks at learning a model for the *Lateral State Manager* (*LSM*) of an autonomous car. The *LSM*, implemented in MATLAB, is a sub-component of the planning module that keeps track of the process of switching between lanes during autonomous driving. In this case study, a model of the *LSM* is learned by actively interacting with its implementation. Here, we use *SupL**– without specifications, hence learning a model of the system – and MPL. The paper discusses the reasons for the failure of *SupL** to learn a model of the *LSM*. MPL, though used to learn a monolithic model in this case, had no problems learning a model of the *LSM*.

Both these studies highlighted certain issues that need to be addressed when using supervisor learning techniques in practical applications.

Choosing the Abstraction

Both cases show the importance of deciding on a relevant level of abstraction to learn the model. In both cases, the level of abstraction needs to conform with the implementation of the system. If not, it becomes difficult to establish an interface between the learner and the SUL. From the perspective of the algorithms, the decision on the abstraction level plays a crucial role in the modular learning algorithms. Both MPL and MSL require a PSH that defines the system's modular structure. Moreover, this modular structure ties into the intended level of abstraction, which in turn impacts the search strategy.

The level of abstraction also plays a key role in determining the time needed to learn a model. For example, in learning from the virtual commissioning example, it was noted that having more complex simulation models resulted in a long time for learning the model. Changing the simulation, without any change to the PLC logic, to reduce the simulation time helped reduce learning time from almost 50 hours to about 20 hours. Since the change does not impact the system's logical behavior, the resulting models in both cases are the same.

The decisions made regarding the abstraction levels also determine the size of the resulting model, and ultimately the applicability of these methods. The learning can be applied at the level of sensors and actuators, which can result in a huge model that contains the detailed behaviors of these components. Alternatively, it can be applied at a higher level, as it is done at the abstraction level of operations in Paper A. In the case of *LSM*, the abstraction decided the size of the alphabet resulting in a trade-off between the events and states discussed in Section D8.2.

Access to Specifications

This thesis aims to calculate supervisors that ensure that the SUL, when controlled, behaves according to certain specifications. However, these specifications do not always exist in a usable format. When learning a model of *LSM*, it was noted that though there existed natural language specifications, these were not useful. The specifications had to be converted from natural language to something more useful, such as an automaton that has the same level of abstraction as the model. However, since no information about the model (for example, the alphabet) existed from the start, no usable specification could be created. This is partially a modeling problem but is also connected to the previously mentioned issue with defining abstractions.

Model Validation

Evaluating if the learned model describes the behavior of the SUL correctly is not always easy. If some behavior does not exist in the model or the other way around, it is essential to establish where the error originates. The incorrect behavior could result from bad specification, wrong simulation model, nonconforming meta-model, etc. In Paper D, manually simulating the model and the implemented code side-by-side was one way to validate the model. However, it is infeasible to scale such an approach. Thus, there is a need for approaches to determine the validity of the resulting model.

The existence of a software bug in the *LSM* was known in advance. Hence, the manifestation of this bug in the learned model provided a way to validate the learned model in Paper D.

CHAPTER 6

ON THE CORRECTNESS OF MODULAR LEARNING

Paper B presents MPL, an algorithm to learn a modular model of a SUL by actively interacting with a simulation of the SUL. Additionally, an algorithm to learn a modular supervisor, MSL, is presented in Paper C. Both these algorithms require a PSH that provides some assumptions about the modular structure of the SUL. The algorithms use these assumptions to avoid exploring the monolithic state-space and instead learn a modular model. This chapter provides some insights into the properties that the PSH must satisfy, and also reasons about the correctness of the MPL. MSL extends the MPL and hence the same arguments follow.

6.1 Prerequisites

Let I be a totally ordered index set. Let $V = \{\, v_i \mid i \in I \,\}$ be a set of variables such that each variable is indexed by one element of the indexing set, that is $|V| = |I|$. Let $V' = \{\, v_{i'} \in V \mid i' \in I' \subseteq I \,\}$ be a subset of variables of V respecting the indexing order, with $|V'| = |I'|$. Each variable v_i has a (finite discrete) domain D_i, and let the domain of V be $D_V = D_{i_1} \times D_{i_2} \times \cdots \times D_{i_{|I|}}$, where the indices $i_j \in I$ (for $j \in 1..|I|$) respect the indexing order. In the same way, the domain of V' is given as the Cartesian product over the domains of the variables of V' in the order defined by the indexing subset $I' \subseteq I$.

Let a *state* q be defined as an element of the domain of V, that is, $q \in D_V$. Thus, a state q is a valuation of the variables of V. Likewise, let a *sub-state* q' be a valuation over V'. Define the *projection* of a state $q \in D_V$ onto a sub-state $q' \in D_{V'}$ as $P_{V'}(q) = q'$, such that all $v_i \in q'$ have the same valuation as in q. For a set of states Q, let $P_{V'}(Q) = Q'$ denote the projection of each element in Q on V', that is $Q' = \bigcup\limits_{q \in Q} P_{V'}(q)$.

Let Σ, called an *alphabet*, be a finite set of *events*. Denote by τ the *silent event*, not part of Σ.

Definition 7 (DFA)**:** *A (*deterministic finite*)* automaton *is defined as a 4-tuple* $\langle Q, \Sigma, T, q_0 \rangle$*,where:*

- Q *is the set of* states*;*
- Σ *is the* alphabet *containing the events;*
- $T \subseteq Q \times \Sigma \times Q$*, is the* transition relation*;*
- $q_0 \in Q$ *is the* initial state.

Plant modules

Let $G_i = \langle Q_i, \Sigma_i, \delta_i, q_{0i} \rangle$ be a DFA that describes the behavior of a plant module. The full behavior of the plant is then described by a set of all its individual modules given by $G = \{G_1, G_2, ...G_n\}$, known as the *modular model.* The synchronous composition [13] of all the modules $G_1 \parallel G_2 \parallel ... \parallel G_n$ results in the complete behavior of the plant referred to as the *monolithic model.*

6.2 On Conformance Between the Simulation and PSH

Consider a discrete-event system whose simulation is represented by a set of variables V. The valuation of all the variables at any given point in time determines the state of the system. All the possible events that the system can perform are represented by the alphabet Σ. The system can change its state on the occurrence of an event $e \in \Sigma$. The occurrence of event e updates a subset of variables $\hat{V} \subseteq V$, resulting in a new valuation V'. Let *DepV* map events to the corresponding variables they update.

$$DepV : \Sigma \to 2^V$$

Let $\mathcal{F}_e$ be a function that defines the update performed by each event e on the set of variables $V_e \subseteq V$, such that $\hat{V}_e' = \mathcal{F}_e(\hat{V}_e)$, where $\hat{V}_e$ is the current valuation of set V_e, and $\hat{V}_e'$ is the updated valuation when applying function $\mathcal{F}_e$ to the current valuation.

A DFA modeling the system can in principle be obtained by exploring the system in a breath-first search manner to discover all the possible states and transitions. However, doing so is expensive and requires exploration of the complete state-space. Instead, it might be possible to explore a subset of the state-space and learn several smaller DFA's that together define the system's complete behavior by synchronous composition.

To do so, some knowledge about the different interacting components (in the simulation) is needed. Firstly, the subset of the alphabet for each of these components is required. Secondly, a mapping between the events and the variables that they affect is needed. The former is usually easier to obtain/guess based on, say, the physical partitioning of the system. The latter, however, requires creativity and deeper knowledge about the system.

In essence, we assume that the simulation is a gray-box that defines the behavior of several automata under synchronous composition. We can only execute an event and observe its behavior. The task is then to identify the different components. Since this is a simulation, we have fine-grained control over how we choose to execute these events. All the events are assumed to be atomic and instantaneous. Thus, we can assume that this hidden DFA's behavior consists of $n \in \mathbb{N}$ automata. Let $G_s = \{G_1, G_2, \ldots, G_n\}$ be the set of all automata in the simulation, each of them having an alphabet defined by Σ_n. Furthermore, the function L_v maps an automaton to its corresponding set of variables in the simulation. This would then mean there are at least n variables, each corresponding to one automaton having a domain defined by the corresponding automaton's state labels. The set of variables defined by $L_v(g)$, for $g \in G_s$ is called the *local variables* for that module.

MPL uses such a simulation to learn a modular model. To do so it requires a Plant Structure Hypothesis (PSH). The PSH is defined using three pieces of information. Firstly, a set M provides a unique name for each module that is to be learned. The cardinality of M defines the number of modules that will be learned. Secondly, a mapping E, called *event mapping*, defines which events of the global alphabet Σ belong to which module $m \in M$. Thus, $E(m) \subseteq \Sigma$ is the local alphabet of the module m. That an event is part of the event mapping implies that the corresponding module is involved in executing the event and, furthermore, that it requires this event to be represented as transitions in the automaton of the module. Finally, a mapping S, called *state mapping*, defines the relation between the modules and the set of variables in the simulation. That is, for all $m \in M$, $S(m) \subseteq V$ contains those variables that either affect or are affected by events in the module. Variables that are not part of a specific state mapping can be ignored by that module. Thus, for a given module $m \in M$, two global states $q_i, q_j \in D_V$ are equal within the module if their projections onto $S(m)$ are equal, that is, if $P_{S(m)}(q_i) = P_{S(m)}(q_j)$. Hereinafter the projection of a state q onto a state mapping $S(m)$ is denoted $P_m(q)$.

Definition 8 (PSH)**:** *Formally, the PSH is a 3-tuple* $H = \langle M, E, S \rangle$, *where:*

- M *is a set of identifiers for the modules;*
- $E : M \to 2^{\Sigma}$ *is the* event mapping*;*
- $S : M \to 2^{V}$ *is the* state mapping*;*

To guarantee that the MPL explores the full plant, the union of all event mappings should encompass the whole alphabet Σ and the union of all state mappings should encompass the whole of V. That is, each event $\sigma \in \Sigma$ and variable $v \in V$ must be included in the event and state mapping of at least one module, respectively. For any given system there may exist multiple PSH of various coarseness; the coarsest one being a PSH defining only a single module m with $E(m) = \Sigma$ and $S(m) = V$. This does satisfy the criteria and will ensure that a full plant model is learned but the learning will result in exploring the monolithic plant, since there is no modular information to exploit. In many cases a PSH can be refined by considering the physical structure of the plant, defining separate modules for subsystems, such as machines, robots, or vehicles. In other cases it may be more efficient to combine multiple strongly connected subsystems into a single module, since their shared behavior otherwise needs to be represented redundantly in each module, or to define modules that capture specific operations or actions regardless of the subsystems involved.

In the best of scenarios a PSH would consist of n modules named $m_1, m_2, \ldots m_n$ where $E(m_i) = \Sigma_i$. Note, however, that the design engineer does not always know the simulation's internal logic and structure. It is, of course, possible to know or infer some aspects of the system. For example, in the case of physical systems, the engineer can guess how many components exist and the event set for each of these components. The greater knowledge that the engineer has, the more accurate is the PSH. Yet, defining the PSH is a modeling problem requiring creativity and skill.

The PSH should fulfill the following properties:

Property 1. *Each module defined in M must correspond to one or more components in the simulation.*

Let there be a mapping $\mathcal{M}_o : M \longrightarrow 2^{G_s} \setminus \{ \varnothing \}$

Property 2. *All events that update one or more local variables of the simulation component must be included in the alphabet of the module.*

For each module $m \in M$*,*

$$E(m) = \{ e \in \Sigma \mid \exists g \in \mathcal{M}_o(m) \ s.t. \ L_v(g) \cap DepV(e) \neq \varnothing \}$$

Property 3. *All variables updated by the events in the module are present in the state mapping*

For each module $m \in M$*,*

$$\bigcup_{e \in E(m)} DepV(e) \subseteq S(m)$$

Property 4. *All the events in all of the modules constitute the alphabet* Σ

$$\bigcup_{m \in M} E(m) = \Sigma$$

Example 4. *Consider MBM introduced in Example 1. A simulation containing the three automata has the variables* $V = \{v_1, v_B, v_2\}$*, internal automata* $G_s = \{G_{M_1}, G_B, G_{M_2}\}$ *and the alphabet* $\Sigma = \{l_1, u_1, l_2, u_2\}$*.*

A potential PSH, $H = \langle M, E, S \rangle$*, could have* $M = \{M_1, B, M_2\}$*. The simulation has local variables as:*

- $L_v(G_{M_1}) = \{v_1\}$
- $L_v(G_B) = \{v_B\}$
- $L_v(G_{M_2}) = \{v_2\}$

and dependencies:

- $DepV(l_1) = \{v_1\}$
- $DepV(u_1) = \{v_1, v_B\}$
- $DepV(l_2) = \{v_B, v_2\}$
- $DepV(u_2) = \{v_2\}$

Based on the above we can define the event mapping:

- $E(M_1) = \{l_1, u_1\}$*, as* $DepV(l_1)$ *and* $DepV(u_1)$ *both contain* v_1
- $E(B) = \{u_1, l_2\}$*, as* $DepV(u_1)$ *and* $DepV(l_2)$ *both contain* v_B
- $E(M_2) = \{l_2, u_2\}$*, as* $DepV(l_2)$ *and* $DepV(u_2)$ *both contain* v_2

6.3 Modular Plant Learner

The Modular Plant Learner (MPL) is a state-based active learning algorithm developed to learn a *modular model*, that is, one composed of a set of interacting automata. These *modules* together define the behavior of the SUL. MPL does so by actively exploring the state-space of a program in a breadth-first search manner. It exploits structural knowledge of the SUL to search smartly. Hence, it requires access to the SUL's variables and a PSH defining the structure of the SUL. MPL consists of an *Explorer* and one *ModuleBuilder* for each module to learn. The *Explorer* is responsible for exploring new states, while each *ModuleBuilder* keeps track of its module as it is learned. The *Explorer* and the *ModuleBuilder*s are initialized and managed by the *Main* routine. Pseudo-code for the MPL is given in Algorithm 2.

The *Explorer* maintains a queue of states that need to be explored, terminating the algorithm when the queue is empty. The learning is initialized with the SUL's initial state in the queue, which becomes the search's starting state. For each state in the queue, the *Explorer* checks if an event from the alphabet Σ can be executed (lines 15-18). If a transition is possible, the *Explorer* sends the current state (q), the event (σ), and the state reached (q') to all the *ModuleBuilder*s.

The *ModuleBuilder* tracks the learning of each module as an automaton. This is done by maintaining a set Q_m containing the module's states and a transition function $T_m : Q_m \times \Sigma_m \rightharpoonup Q_m$, for each module $m \in M$. Once the transition is processed, the *ModuleBuilder* waits for the *Explorer* to send the next transition. Each of the *ModuleBuilder*s evaluates if the received transition is relevant to its particular module. If it is, the transition $\langle q, \sigma, q' \rangle$ is added to the module; else the transition $\langle q, \tau, q' \rangle$ is added to the module (lines 30-31). These τ-transitions are placeholders for transitions that impact some variable in the module but do not directly contribute to the behavior of the module. The algorithm terminates when all *ModuleBuilder*s are waiting, the exploration queue is empty, and there are no other transitions to process. Each *ModuleBuilder* can now construct and return an automaton based on Q_m and T_m (line 44).

The automata obtained could possibly contain τ-transitions; these need to be removed as they correspond to events not part of the module. Algorithm 3 provides the pseudo-code for the removal of τ-transitions. After the initialization step, the algorithm starts by identifying the set of variables, V', that have been updated as part of any τ-transition (lines 4-6). A new set of transitions can be created containing all non-τ-transitions. Additionally, all the states in the automaton are projected such that the states do not contain the set of variables V' (lines 9 and 12).

In addition to all that is described above, the implemented MPL requires some way to establish communication between the *Explorer*, *ModuleBuilder*s, and the *Main*. Additionally, the *Main* routine needs to coordinate between the *Explorer*, and all the *ModuleBuilder*s to ensure the algorithm terminates only when it has completed. Further optimizations, such as maintaining a set of visited states; using multiple instances of the simulation to parallelize the exploration, could be made to improve the algorithm's efficiency.

Termination of MPL

The algorithm works by exploring the state-space of the system. Since the state-space is finite and the *ModuleBuilder* only adds previously unseen states to the *Explorer* queue (lines 37-39 in Algorithm 2), the *Explorer* cannot explore more states than what exist. Furthermore, there is nothing in the *ModuleBuilder* that might prevent the processing of transitions. Hence, the MPL must terminate.

Algorithm 2: The Modular Plant Learner

```
Input: The simulation of the SUL, the initial state, $q_0$, and a PSH $H = \{M, E, S\}$.
Result: A set $G$ of modular plant components, $G_m \in G, \forall m \in M$.
begin
    Procedure Main
        $Q_G \longleftarrow \{q_0\}$ and Run $\longleftarrow$ $true$
        foreach $m \in M$ do
            - run ModuleBuilder($m$)
        end
        - run Explorer()
        - Wait until $Q_G$ is empty, and ModuleBuilders are waiting.
        - Run $\longleftarrow$ $false$
        - Call Algorithm 3 for each learned automaton $G_m$ returned by ModuleBuilders, and store the results in set $G$.
        return $G$
    end
    Procedure Explorer
        while Run do
            for $q \in Q_G$ do
                for $\sigma \in \Sigma$ do
                    - Broadcast the transition $\langle q, \sigma, q' \rangle$, if $q'$ is reached by executing $\sigma$ from state $q$ in the simulator.
                end
                - Remove $q$ from $Q_G$
            end
        end
    end
    Procedure ModuleBuilder(m)
        $Q(m) \longleftarrow \{P_m(q_0)\}$, $T(m) \longleftarrow \varnothing$
        while Run do
            if $\langle q, \sigma, q' \rangle$ received then
                $V' \longleftarrow \{v \in V \mid q(v) \neq q'(v)\}$
                $newState \longleftarrow False$;
                if $\sigma \in E(m)$ or $S(m) \cap V' \neq \varnothing$ then
                    $\sigma' \longleftarrow$ if $\sigma \in E(m)$ then $\sigma$ else $\tau$
                    $T(m) \longleftarrow T(m) \cup \{\langle P_m(q), \sigma', P_m(q') \rangle\}$
                    if $P_m(q') \notin Q(m)$ then
                        $Q(m) \longleftarrow Q(m) \cup \{P_m(q')\}$
                        $newState \longleftarrow True$
                    end
                end
                if $newState$ then
                    $Q_G \longleftarrow Q_G \cup \{q'\}$
                end
            else
                - wait for broadcast
            end
        end
        return $G_m = \langle Q(m), E(m), T(m), P_m(q_0) \rangle$
    end
end
```

Algorithm 3: Algorithm for the removal of τ events.

```
Input: An automaton $A = \langle Q, \Sigma, \delta, q_0 \rangle$
Result: An updated automaton $A'$ with that contains no $\tau$-transitions
begin
    Procedure RemoveTau(A)
        Initialize $V' \longleftarrow \varnothing, \delta' \longleftarrow \varnothing$
        foreach $\langle q, \tau, q' \rangle \in \delta$ do
            $V' \longleftarrow V' \cup \{ v \in V \mid q(v) \neq q'(v) \}$
        end
        foreach $\langle q, \sigma, q' \rangle \in \delta$ do
            if $\sigma \neq \tau$ then
                $\delta' \longleftarrow \delta' \cup \{ \langle P_{V'}(q), \sigma, P_{V'}(q') \rangle \}$
            end
        end
        return $\langle P_{V'}(Q), \Sigma, \delta', P_{V'}(q_0) \rangle$
    end
end
```

Correctness of MPL

Theorem 2: *When given a PSH that defines just one module m, with $E(m) = \Sigma$, and $S(m) = V$; where Σ is the set of events, and V is the set of variables in the simulation, MPL explores the reachable state-space, resulting in a single automaton.*

Proof. Since there is just one module m, MPL creates only one instance of the *ModuleBuilder*. Lines 29 and 30 always evaluate to `True` since $E(m) = \Sigma$. Thus, line 31 adds every transition received by the *ModuleBuilder*. If the state reached by the transition has not been seen previously, it is added to the sets $Q(m)$ and Q_G. Note that the projection operator P_m does not have any effect since $S(m) = V$. Thus, every state reachable (from the initial state) in the SUL is added to Q_G and is explored. The obtained set of transitions and their corresponding states are used to create the final automaton according to line 44. □

Theorem 3: *Given a PSH that satisfies properties 1-4, and a simulation as described above, MPL learns several models, one for each module defined in the PSH, that together define the behavior of the simulation.*

The theorem can be proved using the following lemmas.

Lemma 1: *Consider a simulation with the set of variables V whose valuations represent the behavior of n automata, each having an alphabet Σ_1, Σ_2, ..., Σ_n. A PSH with n modules, $M = \{ m_1, m_2, \ldots, m_n \}$, that correctly maps the events for each module to the corresponding alphabet in the simulation (i.e. $E(m_i) = \Sigma_i$), and where each state mapping includes all variables, $S(m_i) = V$. In such a case, each of the n* ModuleBuilder*s will result in a similar structured monolithic automaton. The difference between them being, those transitions that are labeled with $\sigma \notin E(m)$, during the exploration, will instead have the label τ during the learning.*

Proof. Consider a module $m \in M$, the *ModuleBuilder* for this module enters the if-condition on line 26 when a transition $\langle q, \sigma, q' \rangle$ is received from the *Explorer*. The condition on line 29 will always evaluate to `True` since $S(m) = V$ and $V' \subseteq V$, ensuring that $T(m)$ is updated with $\langle q, \sigma, q' \rangle$, if $\sigma \in E(m)$, or $\langle q, \tau, q' \rangle$, if $\sigma \notin E(m)$ according to lines 30 and 31. Hence, every transition explored is represented in the output of the *ModuleBuilder*.

Since, $S(m) = V$, the projection operator does not have any impact, i.e. $P_m(q') = q'$. Thus, if q' has not been seen before, it is added to $Q(m)$ and Q_G, ensuring every reachable state is explored.

The *ModuleBuilder* results in an automaton that contains all reachable states of the monolithic model. The transitions between these states keep the same label according to what was explored if $\sigma \in E(m)$ else they are labeled with τ. Hence, all *ModuleBuilder*s learn the similarly structured automata with differently labeled transitions.

□

Lemma 2: *Removal of τ-transitions in Algorithm 3 does not result in non-determinism.*

Proof. Non-determinism could occur due to one of three reasons.

1. *Multiple initial states:* Algorithm 3 does not alter the initial state, and deletion of a transition cannot create new initial states.
2. *Existence of τ-transitions:* Since all transitions labeled with τ are removed the resulting automaton cannot have any τ-transitions remaining.

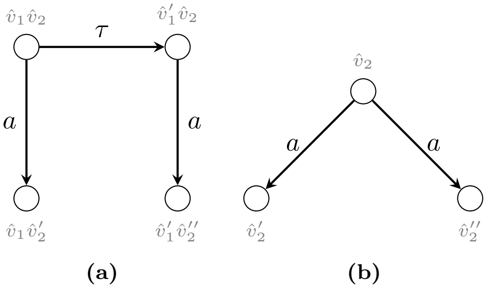


Figure 6.1: An example of potential non-determinism.

3. *Multiple same-labeled transitions with the same source state but different target states:* Deleting τ-transitions could potentially lead to states having multiple same-labeled transitions.

It is clear that points 1 and 2 cannot occur. Hence, to prove this theorem we need to show that same-labeled transitions from the same source state will not occur.

Consider the following three transitions: $\langle\hat{v}_1\hat{v}_2, b, \hat{v}'_1\hat{v}_2\rangle$, $\langle\hat{v}_1\hat{v}_2, a, \hat{v}_1\hat{v}'_2\rangle$, $\langle\hat{v}'_1\hat{v}_2, a, \hat{v}_1\hat{v}''_2\rangle$.

Let these three transitions be processed by a *ModuleBuilder* for module $m \in M$, with $a \in E(m)$ and $b \notin E(m)$. According to line 30, the transition labeled with b has its labeled changed to τ. Figure 6.1a illustrates this scenario. Removal of τ would merge state $\hat{v}_1\hat{v}_2$ and $\hat{v}'_1\hat{v}_2$ causing non-determinism.

According to Algorithm 3, the first step (lines 4-6) is to identify all variables that are updated by the τ-transition. In this case, the τ-transition updates v_1 (from $\hat{v}_1$ to $\hat{v}'_1$ in Figure 6.1a). Hence, according to lines 7-11, non-τ-transition are selected and the projection operator P_{v_1} is applied to the states. This results in two transitions $\langle\hat{v}_2, a, \hat{v}'_2\rangle$ and $\langle\hat{v}_2, a, \hat{v}''_2\rangle$ as in Figure 6.1b. The source and target state of the τ-transition are merged into a single state as they are represented by the same valuation of the variables. This would seem to result in non-determinism. However, the update function $\mathcal{F}_a$ for an event a will result in *the same* output value for a given input. Therefore, $\hat{v}'_2$ must be the same as $\hat{v}''_2$, as they are the result of the function $\mathcal{F}_a(\hat{v}_2)$. Thus, deletion of τ events will not result in non-determinism. □

Lemma 3: *Consider a simulation with a set of variables V whose valua-*

tions represent the behavior of n automata, each having an alphabet Σ_1*,* Σ_2*, …,* Σ_n*, respectively. A PSH with n modules,* $M = \{ m_1, m_2, \ldots, m_n \}$*, that correctly maps the events for each module to the corresponding alphabet in the simulation (i.e.* $E(m_i) = \Sigma_i$*), and where each state mapping includes all variables,* $S(m_i) = V$*. Removal of* τ*-transitions preserves the behavior of the module.*

Proof. It is known from 1 such a PSH results in the monolithic model for each of the modules. Recall that the behavior of a component (in the simulation) is defined by its local variables $L_v(g)$, for $g \in G_s$. Since Property 2 holds, $\forall m \in M$, any event that updates $L_v(g)$, for $g \in \mathcal{M}_o(m)$, is present in $E(m)$. Thus, no transition where a variable in $L_v(g)$ is updated will be re-labeled to a τ-transition (according to line 30). Therefore, removal of τ-transitions will not impact the variables in $L(g)$, thus the behavior of the module is preserved. □

Lemma 4: *Consider a SUL with a set of variables V whose valuations represent the behavior of n automata, each having an alphabet* $\Sigma_1, \Sigma_2, \ldots, \Sigma_n$*. A PSH with n modules,* $M = \{ m_1, m_2, \ldots, m_n \}$*, that correctly maps the events for each module to the corresponding alphabet in the simulation (i.e.* $E(m_i) = \Sigma_i$*), then the state mapping* $\forall m \in M$*,* $S(m) = \bigcup_{e \in E(m)} DepV(e)$ *is sufficient to learn a set of automata that interacting through synchronous composition result in a model with the same language as the SUL.*

Proof. Assume for now that $\forall m \in M$, $S(m) = V$.

Let the *Explorer* broadcast the transition $t = \langle q, \sigma, q' \rangle$, with v being the variable updated in q'.

For a module $m \in M$, with $\sigma \notin E(m)$, meaning $v \notin \bigcup_{e \in E(m)} DepV(e)$ (according to Property 2), the *ModuleBuilder* for m stores the transition as $\langle q, \tau, q' \rangle$ based on lines 30-31.

Since, Property 4 holds, $\exists m' \in M$, where $\sigma \in E(m')$, implying $v \in \bigcup_{e \in E(m')} DepV(e)$ (according to Property 2). Hence, the label of transition t is not changed into a τ-transition for m', as $S(m') = V$.

Since, $v \in S(m)$ and $v \in S(m')$, the *ModuleBuilder*s for both these modules will evaluate the change to v and add the state reached to the exploration queue if the updated value of v is a previously unseen. It is however, sufficient if one module tracks the changes of this variable to identify the new state that arise due to the valuation change of v.

Thus, it is possible to remove v from $S(m)$ while still preserving Property 3. In doing so, module m will ignore updates to v, as line 29 will always evaluate to `False` (since, $e \notin E(m)$ and $v \notin S(m)$); therefore *ModuleBuilder* m will not contribute to finding new, previously unexplored, states that are a consequence of a valuation of v. Since, Property 4 holds, there will always be another module that will keep track of exploring v.

For $m \in M$, the above can be iteratively applied for all variables $v \notin \bigcup\limits_{e \in E(m)} \mathit{DepV}(e)$, but $v \in S(m)$, resulting in $S(m) = \bigcup\limits_{e \in E(m)} \mathit{DepV}(e)$.

□

Now we are ready for the proof of Theorem 3.

Proof. Lemma 1 shows that if all modules $m \in M$ of the PSH, and their corresponding event mappings $E(m)$, are in conformance with the simulation, and the state mappings contain all variables of the simulation $S(m) = V$, MPL learns a monolithic model for each of the modules. The transitions in the obtained model have an event label τ if the corresponding exploration transition had an event label $s \notin E(m), m \in M$. Lemmas 3 and 2 show that the removal of these τ-transitions will preserve the behavior of the module and not introduce any non-determinism. Thus, MPL learns a modular model that corresponds to one or more internal automata in the simulation. □

Example 5. *Based on Lemma 4, the state mapping in example 4 can be defined as follows:*

- $S(M_1) = \{v_1, v_B\}$, *as* $\mathit{DepV}(l_1) = \{v_1\}$ *and* $\mathit{DepV}(u_1) = \{v_1, v_B\}$.
- $S(B) = \{v_1, v_B, v_2\}$, *as* $\mathit{DepV}(u_1) = \{v_1, v_B\}$ *and* $\mathit{DepV}(l_2) = \{v_B, v_2\}$.
- $S(M_2) = \{v_B, v_2\}$, *as* $\mathit{DepV}(l_2) = \{v_B, v_2\}$ *and* $\mathit{DepV}(u_2) = \{v_2\}$.

Satisfying Property 3 ensures, if possible, to avoid the monolithic search. However, depending on the simulation it might be possible to further reduce the size of $S(m)$. A possible example of this has been shown in Section B4.

CHAPTER 7

SUMMARY OF INCLUDED PAPERS

This chapter provides a summary of the included papers.

7.1 Paper A

Ashfaq Farooqui, Ramon Tijsse Claase, Martin Fabian
On Plant-Free Active Learning of Supervisors
Submitted to IEEE Transactions on Automation Science and Engineering (TASE).

Paper A suggests a plant-free approach to obtain the maximally permissive controllable and non-blocking supervisor. This is done in two stages, first, *SupL**, an algorithm to learn supervisors, is introduced to learn a maximally permissive controllable supervisor. Second, if this supervisor is blocking, existing synthesis techniques can be used to obtain the non-blocking supervisor.

Identifying uncontrollable strings during learning presents a challenge to using *SupL**. Hence, a method to convert the controllability problem into one of non-blocking is presented. Here, the specification is converted to a Σ_u-saturated specification before running the learning algorithm. This results in a controllable but blocking supervisor from which the maximally permissive controllable and non-blocking supervisor can be synthesized.

Furthermore, the paper presents a case study where *SupL** is used to learn a supervisor in a virtual commissioning setting. Here, a simplified version of *MBM* is created in the simulation software Experior, and is controlled using a PLC. We then interface to the PLC through the OPC-UA protocol to interact with and learn a model of the system.

My contributions: I am the main responsible for developing, implementing, and proving the algorithm. Also, I was involved in implementing the communication interface between MIDES and OPC-UA. I supervised the Master's project that involved developing the simulation model and its control software to finally learn a model. I am the primary author of the paper.

7.2 Paper B

Ashfaq Farooqui, Fredrik Hagebring, Martin Fabian
Active Learning of Modular Plant Models
15th IFAC Workshop on Discrete Event Systems, November 2020.

Paper B tackles the state-space explosion for learning models. This paper introduces the Modular Plant Learner (MPL) algorithm. The MPL is a state-based active learning algorithm specifically developed to learn a modular model, composed of a set of interacting automata. These modules together define the behavior of a system. The MPL learns by actively exploring the state-space in a breadth-first search manner. To do so, MPL requires a Plant Structure Hypothesis (PSH) that defines the structure of the SUL. By using the PSH, the MPL explores, in a smart way, only a subset of the state-space to learn a modular model of the SUL.

My contributions: I was involved in developing the idea of PSH apart from implementing MPL for modular learning. Also, I was involved in authoring the paper.

7.3 Paper C

Fredrik Hagebring, **Ashfaq Farooqui**, Martin Fabian
Modular Supervisory Synthesis for Unknown Plant Models Using Active Learning

In proceeding of the 15th IFAC Workshop on Discrete Event Systems, November 2020.

Modular Supervisor Learner (MSL), an algorithm for learning modular supervisors, is introduced in Paper C. MSL extends the MPL (from Paper B) to include knowledge of the specifications and thus learn a modular maximally permissive controllable supervisor. Here, a supervisor is learned for each specification in the set of specifications.

To this end, a new mapping is calculated where each specification is matched with a subset of modules in the PSH that directly or indirectly share uncontrollable events—the learning results in the maximally permissive controllable supervisor. Unfortunately, in some cases, the obtained new mapping could potentially result in searching the monolithic state-space. To avoid this, the paper suggests learning a modular supervisor and only those modules that directly share uncontrollable events with the specifications. Additionally, modules not included in any supervisor need to be learned as plant models. Then, the obtained supervisors and plant models can be used to check for uncontrollability issues. If any exist, traditional synthesis techniques can be used to compute the modular maximally permissive controllable supervisor. It must be noted that the obtained modular supervisors need not be non-blocking. Furthermore, the paper demonstrates the use of MSL to the well-known Cat and Mouse example.

My contributions: I developed and implemented the idea to extend the PSH and generate the supervisor mapping, apart from implementing the MSL, running experiments, and authoring the paper.

7.4 Paper D

Yuvaraj Selvaraj, **Ashfaq Farooqui**, Ghazala Panahandeh, Wolfgang Ahrendt, Martin Fabian
Automatically Learning Formal Models from Autonomous Driving Software
Submitted to the Special Issue on Recent Trends in Model-based Engineering of Automotive Systems, JASE.

Paper D presents a case study in which the MPL algorithm from Paper B and the *SupL** algorithm presented in Paper A are applied to learn a model of the *Lateral State Manager (LSM)*, a sub-component of a self-driving car. The paper first gives an illustrative example of the working of the two algorithms. It then presents the setup made to interface MIDES, the learning tool, and the *LSM*. Here, the paper shows how the *LSM* is first abstracted and then interfaced with the learning tool. Finally, the paper discusses the resulting models and the insights gained from these experiments.

My contributions: I developed and implemented the algorithms in addition to the communication interface to MATLAB. I contributed to the discussions about abstractions and finally implemented a semi-automated way to create the abstraction interface in MIDES. I ran the experiments and collected the results using the implementations. I also contributed to authoring the paper, specifically the parts that relate to learning.

CHAPTER 8

CONCLUDING REMARKS AND FUTURE WORK

Supervisory Control Theory (SCT) provides a promising approach to design and develop industrial control systems. Using SCT, a supervisor is synthesized using a plant model describing the behavior of the system and its relevant specifications. The obtained supervisor is used in conjunction with the plant, where the supervisor is allowed to disable certain plant events, thus ensuring that the closed-loop system behaves according to the specifications. The usefulness of the supervisor depends on how well the plant model describes the behavior of the system. Manually creating plant models is error-prone and time-consuming. Thus, the cost of developing and maintaining models is, among other things, an obstacle for the industrial adoption of SCT.

The objective of this thesis has been to develop tools and techniques to obtain a supervisor in the unavailability of plant models, thereby improving the industrial adoption of SCT. To this end, this thesis introduces algorithms to learn supervisors for a given system and its specifications automatically. In the absence of usable specifications, it is possible to learn a plant model that can then be used appropriately to obtain a supervisor. Two case studies are performed, first to apply supervisor learning in a virtual commissioning setting. In the second case, automata learning is used to learn a model of the *lateral state manager* of a self-driving car.

More specifically the contributions of the thesis are grouped around the research questions:

RQ1 *How can we integrate automata learning techniques and SCT to help design systems that are correct-by-construction?*

Automata learning techniques (introduced in Chapter 4) show a promising approach to obtaining a system model. SCT (from Chapter 3) are formal techniques to generate supervisors that can ensure the controlled system behaves according to given specifications, and hence the controlled system is correct-by-construction. This thesis shows a way to integrate the two. To this end, two algorithms are presented that interact with the system under learning (SUL) to learn a maximally permissive controllable supervisor. The $SupL^*$, from Paper A, extends upon and modifies L^* algorithm by providing a practical way to interact with the SUL and learn a supervisor. The MSL, introduced in Paper C, is the other algorithm. The MSL uses knowledge of the interacting modules in the SUL to explore the state-space in an intelligent way to obtain a modular supervisor.

Papers A and C introduce their respective algorithms and demonstrate using an example how automata learning can be used to obtain supervisors. The main requirement in both the algorithms is an interface to a simulation model of the system. Furthermore, the $SupL^*$ requires a way to falsify its hypotheses, and the MSL requires some meta-level knowledge about the SUL. Given this, we show how it is possible to learn a maximally permissive controllable supervisor in the absence of plant models. Furthermore, a tool MIDES [24] is presented in Paper E that implements the algorithms presented in this thesis.

RQ2 *What techniques can help learn models for larger and complex systems?*

The state-space explosion is a well-known problem within SCT. In brief, the problem arises due to the combinatorial growth of the number of states as the models are built. The computation required to learn supervisors for large systems may fail due to time and memory. One technique to learn supervisors for large systems is to learn many smaller supervisors that, when taken together, result in the maximally permissive controllable supervisor. Paper B introduces an approach to smartly explore the system thus learning a modular model. This approach is extended in Paper C to compute the maximally permissive controllable supervisor in the absence of a plant model. The modular learning method, being work-in-progress, needs more development before its efficiency can be evaluated. However, preliminary results show that the number of states that need to be explored can be reduced considerably, making it possible to obtain supervisors for larger systems.

RQ3 *What are the challenges faced when applying these methods to real-world scenarios?*

This thesis presents two case studies. In the first, the well-known Machine Buffer Machine was simulated in Experior and controlled using a PLC to emulate the virtual commissioning process. The maximally permissive controllable supervisor of the *MBM* is learned using the *SupL** algorithm by interacting with the PLC. In the second study, the learning algorithms, *SupL** and MPL, were applied to the *lateral state manager* module of a self-driving car.

Several challenges were identified during these case studies. Firstly, choosing the right abstraction to learn is crucial. However, several factors need to be considered while choosing such an abstraction. Learning a particular abstraction might require making modifications to the SUL, which might not always be possible. Furthermore, the specifications might not be available on the same abstraction level, and hence the specifications cannot be used directly. Also, validation of the obtained supervisors is tricky. Incorrect behavior could arise for several reasons, including inadequate specifications, incorrect simulation models, or non-conformant meta-models. Identifying incorrect behaviors is a challenge. Additionally, in the case of modular learning, obtaining a useful PSH requires in-depth knowledge of the internals of the SUL. Hence, defining a PSH for a black-box system is a challenging task.

The overall work in this thesis is but one step towards introducing supervisory control theory techniques into day-to-day industrial practice. Hence, there are several open avenues for future improvement and extensions.

Future Work

Models are not perfect; they represent only a part of reality. Determining the validity of the learned models is no trivial task. A major challenge lies in reasoning about the quality and usability of the obtained models. Thus, to be able to reason and make decisions by using these models, there needs to be some metric to help classify them. These metrics can provide the engineer with a degree of confidence when performing analysis on the model and help reason about the meaning of these results in relation to the system. Investigation into methods and tools that can validate the models, and including these tools as part of the learning process can possibly help in the adoption of automatic model learning.

Now that we know a supervisor can be learned in the absence of plant models, the next step is to use more efficient data structures and algorithms [43], [45], [46] to apply supervisor learning in practice. Research into more efficient data structures like BDDs [97] and the ability to learn richer formalisms, notably Extended Finite State Machines [98], is needed. These techniques may improve the efficiency of learning and allow the application os supervisor learning on larger, more complex systems. Furthermore, it would be interesting to study the possibilities to learn supervisors as symbolic automata [50].

The MPL and MSL algorithms put some strict requirements on the simulation. Further research into relaxing these requirements may broaden the application domains for these algorithms. Furthermore, a pain point in this method is the creation of the PSH. Methods to automate and help designers define the PSH will significantly improve the algorithms usability.

The $SupL^*$ algorithm in its current state uses plantified specifications that convert initial controllability problems into non-blocking problems. This, however, results in additional states that will eventually be deleted. Further work is needed to improve the $SupL^*$ to learn the maximally permissive controllable and non-blocking sub-automaton directly.

As pointed out in Chapter 5, performing equivalence queries is a significant bottleneck. Research into leveraging domain-specific knowledge to find counterexamples might provide valuable insights.

REFERENCES

[1] 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)*, Xian, China: IEEE, 2017, pp. 932–937.

[2] M. Dowson, "The Ariane 5 software failure", *ACM SIGSOFT Software Engineering Notes*, vol. 22, no. 2, p. 84, 1997.

[3] N. G. Leveson and C. S. Turner, "An investigation of the Therac-25 accidents", *Computer*, vol. 26, no. 7, pp. 18–41, 1993.

[4] R. Seyfert, "Bugs, predations or manipulations? Incompatible epistemic regimes of high-frequency trading", *Economy and Society*, vol. 45, no. 2, pp. 251–277, 2016.

[5] V. Pratt, "Anatomy of the Pentium bug", in *Colloquium on Trees in Algebra and Programming*, Springer, 1995, pp. 97–107.

[6] D. L. Dill and J. Rushby, "Acceptance of formal methods: Lessons from hardware design", *IEEE Computer*, vol. 29, no. 4, pp. 23–24, 1996.

[7] N. Sanchez-Tamayo and J. P. Wachs, "Collaborative robots in surgical research: A low-cost adaptation", in *Companion of the 2018 ACM/IEEE International Conference on Human-Robot Interaction*, ser. HRI '18, Chicago, IL, USA: Association for Computing Machinery, 2018, pp. 231–232, ISBN: 9781450356152. [Online]. Available: `https://doi.org/10.1145/3173386.3176978`.

[8] M. Almasi, “Designing a non-contact sensor for capturing pacemaker”, *International Journal of Computer Trends and Technology*, vol. 68, no. 7, pp. 43–48, 2020.

[9] S. Dosen, C. Prahm, S. Amsüss, I. Vujaklija, and D. Farina, “Prosthetic feedback systems”, in *Bionic Limb Reconstruction*, O. C. Aszmann and D. Farina, Eds., Cham: Springer International Publishing, 2021, pp. 147–167, ISBN: 978-3-030-60746-3.

[10] J. C. Knight, “Challenges in the utilization of formal methods”, en, in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, A. P. Ravn and H. Rischel, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 1998, pp. 1–17, ISBN: 978-3-540-49792-9.

[11] J. C. Knight, C. L. DeJong, M. S. Gibble, and L. G. Nakano, “Why Are Formal Methods Not Used More Widely?”, in *Fourth NASA Formal Methods Workshop*, 1997, pp. 1–12.

[12] H. C. Michael, “Why Engineers Should Consider Formal Methods”, NASA Langley Technical Report Server, Technical Report, 1997.

[13] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems.* New York, NY: Springer Science & Business Media, 2009.

[14] C. Baier and J.-P. Katoen, *Principles of Model Checking.* Cambridge, Massachusetts: MIT press, 2008.

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

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

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

[18] B. Steffen, F. Howar, and M. Merten, “Introduction to active automata learning from a practical perspective”, in *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, 2011.

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

[20] C. de la Higuera, *Grammatical Inference: Learning Automata and Grammars.* Cambridge University Press, 2010.

[21] W. Mefteh, "Simulation-based design: Overview about related works", *Mathematics and Computers in Simulation*, vol. 152, pp. 81–97, 2018, ISSN: 0378-4754. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0378475418300739`.

[22] F. Tao, J. Cheng, Q. Qi, M. Zhang, H. Zhang, and F. Sui, "Digital twin-driven product design, manufacturing and service with big data", *The International Journal of Advanced Manufacturing Technology*, vol. 94, no. 9, pp. 3563–3576, Feb. 2018.

[23] P. Gohari and W. M. Wonham, "On the complexity of supervisory control design in the RW framework", *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 30, no. 5, pp. 643–652, 2000.

[24] *MIDES- Model Inference for Discrete-Event Systems.* [Online]. Available: `https://github.com/ashfaqfarooqui/MIDES`.

[25] 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 2013; Las Vegas, Nevada, USA, 22-25 July*, CSREA Press USA, 2013, pp. 67–73.

[26] J. L. Peterson, "Petri nets", *ACM Comput. Surv.*, vol. 9, no. 3, pp. 223–252, Sep. 1977, ISSN: 0360-0300. [Online]. Available: `http://doi.acm.org/10.1145/356698.356702`.

[27] M. Yoeli, "The cascade decomposition of sequential machines", *IRE Transactions on Electronic Computers*, vol. EC-10, no. 4, pp. 587–592, Dec. 1961, ISSN: 0367-9950.

[28] J. E. Hopcroft, R. Motwani, Rotwani, and J. D. Ullman, *Introduction to Automata Theory, Languages and Computability*, 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000, ISBN: 0201441241.

[29] J. Hopcroft, "An n log n algorithm for minimizing states in a finite automaton", in *Theory of machines and computations*, Elsevier, 1971, pp. 189–196.

[30] A. Nerode, “Linear automaton transformations”, *Proceedings of the American Mathematical Society*, vol. 9, no. 4, pp. 541–544, 1958, ISSN: 00029939, 10886826. [Online]. Available: `http://www.jstor.org/stable/2033204`.

[31] R. Kumar, V. Garg, and S. I. Marcus, “On controllability and normality of DEDS”, in *American Control Conference*, 1991, pp. 2905–2910.

[32] W. M. Wonham and P. J. Ramadge, “On the Supremal Controllable Sublanguage of a Given Language”, *SIAM J. Control Optim.*, vol. 25, no. 3, pp. 637–659, May 1987, ISSN: 0363-0129.

[33] H. Flordal and R. Malik, “Supervision equivalence”, in *2006 8th International Workshop on Discrete Event Systems*, 2006, pp. 155–160.

[34] H. Flordal, *Compositional Approaches in Supervisory Control with Application to Automatic Generation of Robot Interlocking Policies.* Chalmers University of Technology, 2006.

[35] K. Åkesson, H. Flordal, and M. Fabian, “Exploiting modularity for synthesis and verification of supervisors”, *IFAC Proceedings Volumes*, vol. 35, no. 1, pp. 175–180, 2002, 15th IFAC World Congress, ISSN: 1474-6670.

[36] Colin de la Higuera, “A bibliographical study of grammatical inference”, *Pattern Recognition*, vol. 38, no. 9, 2005.

[37] M. Bugalho and A. L. Oliveira, “Inference of regular languages using state merging algorithms with search”, *Pattern Recogn.*, vol. 38, no. 9, 2005.

[38] R. Parekh and V. Honavar, “Grammar inference, automata induction, and language acquisition”, *Handbook of natural language processing*, pp. 727–764, 2000.

[39] E. M. Gold, “Complexity of automaton identification from given data”, *Information and Control*, vol. 37, no. 3, pp. 302–320, 1978.

[40] ——, “Language identification in the limit”, *Information and Control*, vol. 10, no. 5, pp. 447–474, 1967, ISSN: 0019-9958.

[41] T. Berg, *Regular inference for reactive systems*, Lic. Thesis, 2006.

[42] M. X. Czerny, “Learning-based software testing: Evaluation of Angluin’s L* algorithm and adaptations in practice”, *Batchelors thesis, Karlsruhe Institute of Technology, Department of Informatics Institute for Theoretical Computer Science*, 2014.

[43] M. J. Kearns, U. V. Vazirani, and U. Vazirani, *An introduction to computational learning theory.* MIT press, 1994.

[44] R. E. Schapire, *The Design and Analysis of Efficient Learning Algorithms.* Cambridge, MA, USA: MIT Press, 1992, ISBN: 0-262-19325-6.

[45] F. M. Howar, "Active learning of interface programs", en, Jun. 2012.

[46] M. Isberner, F. Howar, and B. Steffen, "The TTT algorithm: A redundancy-free approach to active automata learning", in *Runtime Verification*, B. Bonakdarpour and S. A. Smolka, Eds., Springer International Publishing, 2014, pp. 307–322, ISBN: 978-3-319-11164-3.

[47] M. Isberner, "Foundations of Active Automata Learning: An Algorithmic Perspective", de, p. 205,

[48] B. Bollig, P. Habermehl, C. Kern, and M. Leucker, "Angluin-style learning of NFA", in *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, ser. IJCAI'09, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Jul. 2009, pp. 1004–1009.

[49] 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.

[50] L. D'Antoni, T. Ferreira, M. Sammartino, and A. Silva, "Symbolic Register Automata", *arXiv:1811.06968 [cs]*, May 2019.

[51] B. Balle and M. Mohri, "Learning Weighted Automata", en, in *Algebraic Informatics*, A. Maletti, Ed., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2015, pp. 1–21, ISBN: 978-3-319-23021-4.

[52] B. Jacobs and A. Silva, "Automata learning: A categorical perspective", in *Horizons of the Mind. A Tribute to Prakash Panangaden*, Springer, 2014, pp. 384–406.

[53] G. van Heerdt, M. Sammartino, and A. Silva, "Calf: Categorical automata learning framework", *arXiv preprint arXiv:1704.05676*, 2017.

[54] A. Groce, D. Peled, and M. Yannakakis, "Adaptive Model Checking", en, in *Tools and Algorithms for the Construction and Analysis of Systems*, J.-P. Katoen and P. Stevens, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2002, pp. 357–370, ISBN: 978-3-540-46002-2.

[55] D. Peled, M. Y. Vardi, and M. Yannakakis, "Black Box Checking", en, in *Formal Methods for Protocol Engineering and Distributed Systems: FORTE XII / PSTV XIX'99 IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX) October 5–8, 1999, Beijing, China*, ser. IFIP Advances in Information and Communication Technology, J. Wu, S. T. Chanson, and Q. Gao, Eds., Boston, MA: Springer US, 1999, pp. 225–240, ISBN: 978-0-387-35578-8.

[56] 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, pp. 447–454.

[57] K. Meinke, F. Niu, and M. Sindhu, "Learning-Based Software Testing: A Tutorial", en, in *Leveraging Applications of Formal Methods, Verification, and Validation*, R. Hähnle, J. Knoop, T. Margaria, D. Schreiner, and B. Steffen, Eds., ser. Communications in Computer and Information Science, Berlin, Heidelberg: Springer, 2012, pp. 200–219, ISBN: 978-3-642-34781-8.

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

[59] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager, "Learning Fragments of the TCP Network Protocol", en, in *Formal Methods for Industrial Critical Systems*, F. Lang and F. Flammini, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2014, pp. 78–93, ISBN: 978-3-319-10702-8.

[60] P. Fiterău-Broştean and F. Howar, "Learning-Based Testing the Sliding Window Behavior of TCP Implementations", en, in *Critical Systems: Formal Methods and Automated Verification*, L. Petrucci, C. Seceleanu, and A. Cavalcanti, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2017, pp. 185–200, ISBN: 978-3-319-67113-0.

[61] M. Tappler, B. K. Aichernig, and R. Bloem, "Model-Based Testing IoT Communication via Active Automata Learning", in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Mar. 2017, pp. 276–287.

[62] J. de Ruiter and E. Poll, "Protocol State Fuzzing of TLS Implementations", en, in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 193–206, ISBN: 978-1-939133-11-3.

[63] P. Fiterău-Broştean, T. Lenaerts, E. Poll, J. de Ruiter, F. Vaandrager, and P. Verleg, "Model learning and model checking of SSH implementations", in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, ser. SPIN 2017, New York, NY, USA: Association for Computing Machinery, Jul. 2017, pp. 142–151, ISBN: 978-1-4503-5077-8.

[64] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana, "HVLearn: Automated Black-Box Analysis of Hostname Verification in SSL/TLS Implementations", in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 521–538.

[65] F. Aarts, J. de Ruiter, and E. Poll, "Formal models of bank cards for free", Mar. 2013, pp. 461–468, ISBN: 978-1-4799-1324-4.

[66] G. Chalupar, S. Peherstorfer, E. Poll, and J. de Ruiter, "Automated Reverse Engineering using Lego®", en, in *8th USENIX Workshop on Offensive Technologies (WOOT 14)*, 2014.

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

[68] M. Isberner, F. Howar, and B. Steffen, "Learning register automata: From languages to program structures", *Machine Learning*, vol. 96, no. 1, pp. 65–98, Jul. 2014.

[69] W. Smeenk, J. Moerman, F. Vaandrager, and D. N. Jansen, "Applying automata learning to embedded control software", in *Formal Methods and Software Engineering*, Springer International Publishing, 2015.

[70] M. Schuts, J. Hooman, and F. Vaandrager, "Refactoring of Legacy Software Using Model Learning and Equivalence Checking: An Industrial Experience Report", en, in *Integrated Formal Methods*, E. Ábrahám and M. Huisman, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2016, pp. 311–325, ISBN: 978-3-319-33693-0.

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

[72] 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, pp. 487–495, ISBN: 978-3-319-21690-4.

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

[74] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, "Test selection based on finite state models", *IEEE Trans. on Software Engineering*, Jun. 1991.

[75] S. Cassel, F. Howar, and B. Jonsson, "RALib: A learnlib extension for inferring EFSMs", *DIFTS. hp://www. faculty. ece. vt. edu/chaowang/dis2015/papers/paper*, vol. 5, 2015.

[76] A. Bainczyk, A. Schieweck, M. Isberner, T. Margaria, J. Neubauer, and B. Steffen, "ALEX: Mixed-mode learning of web applications at ease", in *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, T. Margaria and B. Steffen, Eds., Cham: Springer International Publishing, 2016, pp. 655–671, ISBN: 978-3-319-47169-3.

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

[78] B. Bollig, J.-P. Katoen, C. Kern, M. Leucker, D. Neider, and D. R. Piegdon, “Libalf: The automata learning framework”, in *Computer Aided Verification*, T. Touili, B. Cook, and P. Jackson, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 360–364, ISBN: 978-3-642-14295-6.

[79] A. W. Biermann and J. A. Feldman, “On the synthesis of finite-state machines from samples of their behavior”, *IEEE Transactions on Computers*, vol. C-21, no. 6, pp. 592–597, 1972.

[80] M. Shahbaz, K. Li, and R. Groz, “Learning and integration of parameterized components through testing”, in *Testing of Software and Communicating Systems*, A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 319–334, ISBN: 978-3-540-73066-8.

[81] K. Hiraishi, “Synthesis of supervisors for discrete event systems allowing concurrent behavior”, in *IEEE SMC’99 Conference Proceedings.*, IEEE, vol. 5, 1999, pp. 13–20.

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

[83] X. Yang, M. Lemmon, and P. Antsaklis, “Inductive inference of logical DES controllers using the L* algorithm”, in *Proceedings of 1995 American Control Conference-ACC’95*, IEEE, vol. 5, 1995, pp. 3163–3167.

[84] ——, “Inductive inference of optimal controllers for uncertain logical discrete event systems”, in *Proceedings of Tenth International Symposium on Intelligent Control*, IEEE, 1995.

[85] J. Dai and H. Lin, “A learning-based synthesis approach to decentralized supervisory control of discrete event systems with unknown plants”, *Control Theory and Technology*, vol. 12, no. 3, pp. 218–233, 2014.

[86] C. G. Lee and S. C. Park, “Survey on the virtual commissioning of manufacturing systems”, *Journal of Computational Design and Engineering*, vol. 1, 2014.

[87] K. Bengtsson, B. Lennartson, and C. Yuan, “The origin of operations: Interactions between the product and the manufacturing automation control system”, *IFAC Proceedings Volumes*, vol. 42, 2009.

[88] J. Moerman, “Nominal techniques and black box testing for automata learning”, PhD thesis, [Sl: sn], 2019.

[89] T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, and B. Steffen, “On the correspondence between conformance testing and regular inference”, in *International Conference on Fundamental Approaches to Software Engineering*, Springer, 2005, pp. 175–189.

[90] R. Durrett, *Probability: theory and examples*. Cambridge university press, 2019, vol. 49.

[91] G. Luo, A. Petrenko, and G. v. Bochmann, “Selecting test sequences for partially-specified nondeterministic finite state machines”, in *Protocol Test Systems*, Springer, 1995, pp. 95–110.

[92] L. E. Holloway and B. H. Krogh, “Synthesis of feedback control logic for a class of controlled petri nets”, *IEEE Transactions on Automatic Control*, vol. 35, no. 5, pp. 514–523, 1990.

[93] R. Malik, K. Åkesson, H. Flordal, and M. Fabian, “Supremica–An Efficient Tool for Large-Scale Discrete Event Systems”, in *The 20th World Congress of the International Federation of Automatic Control, 9-14 July 2017*, 2017.

[94] W. Mahnke, S.-H. Leitner, and M. Damm, *OPC Unified Architecture*, 1st. Springer Publishing Company, Incorporated, 2009.

[95] T. M. Inc., *Matlab*, 2020. [Online]. Available: `https://mathworks.com/products/matlab.html`.

[96] *Xcelgo Experior*. [Online]. Available: `https://xcelgo.com/experior/`.

[97] R. E. Bryant, “Symbolic boolean manipulation with ordered binary-decision diagrams”, *ACM Computing Surveys (CSUR)*, 1992.

[98] R. Malik, M. Fabian, and K. Åkesson, “Modelling large-scale discrete-event systems using modules, aliases, and extended finite-state automata”, *IFAC Proceedings Volumes*, vol. 44, no. 1, pp. 7000–7005, 2011, 18th IFAC World Congress, ISSN: 1474-6670.