Towards Automatic Learning of Discrete-Event Models from Simulations

Towards Automatic Learning of Discrete-Event Models from Simulations

Ashfaq Farooqui, Petter Falkman and Martin Fabian
Department of Electrical Engineering
Chalmers University of Technology, Göteborg, Sweden 412 96
Email: {ashfaqf, petter.falkman, fabian} @ chalmers.se

***Abstract*— Model-based techniques are, these days, being embraced by the manufacturing industry in their development frameworks. While model-based approaches allow for offline verification and validation before physical commissioning, and have other advantages over existing methods, they do have their own challenges. Firstly, models are typically created manually and hence are prone to errors. Secondly, once a model is created, tested, and put into use on the factory floor, there is an added effort required to maintain and update it. This paper is a preliminary study of the feasibility of automatically obtaining formal models from virtual simulations. We apply the foundational algorithm from the active automata learning community to study the requirements and enhancements needed to be able to derive discrete event models from virtual simulations. An abstract model in the form of operations is learned by applying this algorithm on a simulation model composed of discrete operations. While a major bottleneck to be solved is the generation of counterexamples, the results seem promising to apply model learning in practice.**

I. Introduction

Model-based techniques, that offer design, validation, verification, and testing, are being actively adopted within the manufacturing industry [1], [2] to formally ensure correctness of complex systems. The last few years have seen a drastic advancement in model-based algorithms that are practical to use on common manufacturing systems. These techniques are usually coupled with virtual technologies such as simulations [3] and virtual reality [4] in the early phases of manufacturing systems development or the virtual commissioning phase [5], thereby leading to shorter physical commissioning effort. However, creating formal models is a challenging task that requires skill, in-depth knowledge of the system, and creativity.

The automotive manufacturing industry, and manufacturing industries in general, are gradually moving towards simulation based techniques during the initial phase of setting up the systems known as the virtual commissioning phase [5]. During the virtual commissioning phase, a virtual replica of the manufacturing station is first created in a simulation software and is tested in simulation to ensure correctness, before physically commissioning the station. Testing in a virtual environment can be enabled using formal models [3]. However, the formal models and the specifications are typically created manually, which is a daunting task.

This work has been supported by ITEA3 VINNOVA ENTOC (2016-02716), and VR SyTeC (2016-06204).

Models specifying the intended behaviour of the system need to be created early during the specification and design phase of virtual commissioning. Incorrect or incomplete models are misleading, and can unnecessarily complicate the development process. In reality, these models do not always capture the complete behaviour and, even when they do, become outdated as the system evolves if they are not updated regularly.

A possible method to deal with incorrect and outdated models is to create them automatically. Several methods have been proposed in the literature that deal with automatically creating models. These methods work by either observing the behaviour of the components in the system or by actively interacting with the system, and then using specially designed techniques to construct a model. These techniques are now receiving increased attention within the verification and testing communities [6], [7], and are known as automata learning (a.k.a grammatical inference) [8], [9].

Previously, in [10], a method for learning models was presented using *passive learning*. The focus was on collecting data from the manufacturing system and then processing this date to create the model. In this paper we present the first steps toward automatically creating models by interacting with a virtually commissioned simulation system using a method called *active learning*. To this end, we investigate the possibility of using grammatical inference techniques, specifically active learning techniques, to automatically learn a formal model representation of a given simulation. The results of these first steps will provide directions for future research.

A. Outline

We will first introduce Dana Angluin's seminal algorithm [11] and will this use this algorithm to learn a model of a simulated robotic arm. The learning outcome will be the evaluation of possibilities and limitations in applying these techniques to larger manufacturing systems.

In order to be able to apply automata learning to simulated systems we use the abstraction of *operations* [12] to build the simulated controller. To achieve this, the functionality present in the simulated controller is made up of several smaller, unique, actions that actuate a specific component. An operations is then a sequence of actions that accomplish a task. The learning algorithm takes the set of operations as the input for the learning task.

Section II provides a brief literature survey regarding the available work in the field of learning automata. Section III provides the basic definitions and preliminaries used in the remainder of the paper, and also briefly highlights the L* algorithm with an example. We then present the basic framework to enable learning models from simulation systems in Section IV. Thereafter, to practically demonstrate the approach discussed in this paper Section V describes how the model of a simulated robotic arm is learnt using the defined setup. Finally, Section VI concludes with some remarks about future work.

II. Related Literature

Grammatical inference [9] is associated with various fields of study like computational linguistics, machine learning, formal learning theory, and computational biology, to name but a few. Hence, it is also known by different names depending on the field, such as Automata learning, Grammatical induction, Grammar learning etc. An in-depth survey of grammar inference techniques is provided by [8], [13], [14].

A large body of work already exists in this field, and can be classified into the two categories *Active* and *Passive* learning. In this paper we deal with Active learning, sometimes called learning with interaction, which deals with submitting queries to the target system and forming a hypothesis based on the responses obtained. The *learner* first starts with an initial hypothesis which is iteratively improved by first finding counterexamples that invalidate the hypothesis, and then querying the target system to revise the hypothesis.

A seminal paper in the field of Active learning is Dana Angluin's work on learning minimal automata using queries and counterexamples [11]. From an algorithmic perspective, there have been only a handful of improvements and new approaches suggested in this field. Schapire [15] improves the algorithm by handling the counterexamples in a smarter way. Kearns and Vazirani [16] introduce the idea of discrimination trees, which is further used by Malte et.al [17] who suggest the TTT algorithm.

From a more practical perspective, [11] has inspired a tremendous amount of work that has yielded positive results. Active automata learning has been applied to verify communication protocols using Mealy machines [18], [19]. By using a suitable abstraction interface Arts [20] learn IO automata. Other techniques are directed towards learning models of software systems. [21] apply active automata towards learning models of software programs modeled as register automata, [22] focus on learning embedded software programs.

III. Prerequisites

In this section, we introduce the concepts and terminology used throughout the paper. First, the modeling formalisms – Operations and Automata – are defined. Subsequently, a brief explanation of the L* algorithm is provided to highlight its workings.

A. Operations

The system to be learnt can perform several tasks, and these need to be pre-defined. To define the tasks we use the abstraction of *operations*. Each operation corresponds to one specific function performed by the target system.

Definition 1 (Operation): An *operation* is defined as a 4-tuple $\langle \text{PreGaurd}, \text{PreActions}, \text{PostGaurd}, \text{PostActions} \rangle$ where:
PreGuard, is a predicate over the state that defines when the operation is allowed to execute;
PreActions, defines assignments to configuration parameters in the state that will execute the operation in the target system;
PostGuard, is a predicate over the state that defines when the operation completes;
PostActions, defines assignments to configuration parameters in the state that ensures completion of the operation.

For simplicity, we will consider operations that are two-state. That is, when an operation is executed it transforms the system from one state to another when the operation completes.

B. Alphabets, Words and Languages

Let Σ, known as the *alphabet*, represent a finite set of symbols, then Σ^* is used to denote the set of *words* of finite length over Σ including the empty word ε, i.e a set of sequences of symbols formed by *concatenation*. Concatenation of sets is represented using the dot $(\cdot)$ operator, and concatenated symbols are written together without a space. For example, for two sets $A = \{a\}$ and $B = \{b\}$, set concatenation is represented by $A \cdot B$ and symbol concatenation by ab. Note that $A \cdot B = \{ab\}$. A language $\mathcal{L} \subseteq \Sigma^*$ contains the set of words over Σ including the empty word ε.

A set of words is said to be prefix-closed if the prefix of every member in the set is also a member. Suffix-closed sets are defined analogously.

C. Deterministic Finite State Automata

Definition 2 (DFA): An *automaton* is defined as a 4-tuple $\langle S, \Sigma, \delta, s_q \rangle$, where:
S, is the set of states;
Σ, is the alphabet;
$\delta : S \times \Sigma \to S$, is the transition function;
$s_q \subseteq S$, is a set of states representing the *marked* states.

The marked language given by $\mathcal{L}_m \subseteq \mathcal{L}$, is the set of traces that reach marked states. There are many automata that represent the same language, but it is known [23] that among all of these automata there is a *minimal* one, with the smallest number of states and transitions. This automaton is unique.

D. The L* Algorithm

The L* algorithm [11] learns a minimal automata accepting a regular language $\mathcal{L}_m \subseteq \Sigma^*$ over a finite alphabet Σ. Two types of queries need to be answered to make the

algorithm work, and these are answered by a *teacher*. For pedagogic reasons, we will refer to the algorithm as the learner that interacts with the teacher using queries. The two types of queries made by the learner are:

- **Membership queries:** given a word $w \in \Sigma^*$, the teacher replies positively if w belongs to $\mathcal{L}_m$, else the reply is negative.
- **Equivalence queries:** given a *hypothesis* automata $\mathcal{H}$, the teacher must verify if $\mathcal{H}$ accurately represents the language $\mathcal{L}_m$. If not, the teacher must provide a *counterexample* $c \in \Sigma^*$, such that, c is incorrectly accepted or rejected by $\mathcal{H}$.

The algorithm terminates when the teacher cannot find such a counterexample. The L* algorithm is outlined in Figure 1.

At any given time the learner updates its knowledge about the target language. Internally, this knowledge is represented as an *observation table*. The observation table has three parts, a non-empty finite prefix-closed set $S \subseteq \Sigma^*$, a non-empty finite suffix-closed set $E \subseteq \Sigma^*$, and a transition function T mapping $((S \cup S.\Sigma).E)$ to $\{0, 1\}$.

Result: A Hypothesis automata $\mathcal{H}$
initialisation $S, E \leftarrow \varepsilon$;
repeat
  while *the table is not closed or not consistent* **do**
    if *table is not closed* **then**
      find $u \in S, a \in A$ such that $row(ua) \neq row(s) \forall s \in S$;
      $S \leftarrow S \cup \{ua\}$;
    end
    if *table is not consistent* **then**
      find $s_1, s_2 \in S$, $a \in A$ and $e \in E$ such that $row(s_1) = row(s_2)$ and $row(s_1 ae) \neq row(s_2 ae)$;
      $E \leftarrow E \cup \{ae\}$;
    end
  end
  Construct the hypothesis $\mathcal{H}$ to the teacher **if** *the teacher replies* no *with a counterexample* c **then**
    $S \leftarrow S \cup prefixes(c)$
  end
until *the teacher replies* yes;
return $\mathcal{H}$

Fig. 1: The L* Algorithm

The learner uses the observation table to construct a hypothesis automata from the rows of the table, where the rows represent the states of the automata. Rows with the same content result in a single state. The marked states are represented by $row(s)$ where $s \in S$ and $T(s)(\varepsilon) = 1$. The initial state corresponds to a row with the empty word, i.e $row(\varepsilon)$.

In order to be able to construct a hypothesis, the observation table needs to be *closed* and *consistent*.

- An observation table is said to be **closed** if for all $t \in S, a \in \Sigma$ there is an $s \in S$ such that the $row(s) = row(t\,a)$. In other words, each transition reaches some state in the hypothesis.
- A table is **consistent** if for $s_1 \in S$ and $s_2 \in S$ and $row(s_1) = row(s_2)$ then for all $a \in \Sigma, row(s_1\,a) = row(s_2\,a)$. In other words, there is no ambiguity in the transition.

The learner ensures that the table is closed and consistent by updating the sets S and E according to the algorithm in Figure 1. Once the table is closed and consistent, the learner submits a hypothesis to the teacher requesting an equivalence query. If the teacher provides a counterexample, the observation table is updated with the traces from the counterexample, new queries are made, and a revised hypothesis is submitted when the table is again closed and consistent. This process is repeated until no more counterexamples can be found by the teacher, at which point the current hypothesis is the DFA representing the model of the system.

IV. The learning framework

The previous section discussed the general framework that is applied to learn an unknown regular language of a system. In this section, focus will be on the system to be learnt. More specifically, the components needed by the learner in order to apply the discussed learning technique to learn an automata model using the simulation of the system.

A. The Simulation

The first component is a simulator capable of simulating the system to be learnt. The target system must be modeled in a simulator to create the simulation model. Simulations provide several advantages in comparison to using the real system. Unlike the real system, the simulation can be run faster than real-time, even multiple instances in parallel, thereby speeding up the learning process. Dangerous collisions and unforeseen events are avoided and confined to the simulation, providing a safer environment to learn. The financial investment needed, once a simulation is obtained, relates to obtaining powerful computers – which in today's world is relatively cheap. Additionally, simulation models are easier to reset to a known predefined state. This ability to reset helps the teacher while answering queries.

The simulation model and its controller must be created depending on the level of detail required to be captured in the model. The basic requirement, of the simulation model, needed to learn a behavioural model of the system is the ability to accurately simulate the sensors and actuators. Furthermore, a simulation capable of capturing physics can be used to capture collisions and other such unwanted behaviour in the system.

B. Modeling operations for the learner

In order for the learning algorithm to be able to interface with the simulation a common interface needs to be defined. Operations are a sufficiently good abstraction to use while creating the simulated model. The learning algorithm takes the operations as defined in Section III, where each operation

must correspond to a specific function effectuated by a sequence of actions in the simulated model.

Each operation has a name by which it can be identified, and guards and actions that start and stop it. The learner uses the name of each operation while constructing the observation table. These names are then matched with the corresponding operations by the teacher while querying the simulation environment. Thus, the operation names are the symbols of the alphabet.

Apart from the operations, the algorithm requires a "goal" – a predicate over the system state – that defines the marked states of the system. The goal is used by the teacher while evaluating queries as explained in the following section.

C. The teacher

As defined earlier, L* is an interactive query based algorithm which consists of a learner and a teacher. The teacher is responsible for answering two types of queries, membership and equivalence queries. In order to answer these, the teacher needs to interact with the simulated model, making the teacher an interface between the learner and the simulation. For this, the simulator must have an accessible interface enabling communication. Practical details about the communication are presented in Section V.

1) Membership queries: The learner poses membership queries by providing a sequence of operations to the teacher. The teacher is then responsible to execute this sequence in the simulator, and to check if it results in a marked state – i.e if the sequence reaches the goal.

The teacher first ensures that the simulator is in the initial state, else the simulation is reset. For simulations where reset is not possible a homing sequence can instead be used [24]. The sequence of operations received from the teacher is then executed one operation at a time. By first evaluating the PreGuard to check if the operation is allowed, and then, setting the parameters defined as PreAction, the operation is started. The teacher then monitors the state until the operation has ended by evaluating the PostGuard and stopping the operation by setting the parameters as defined by the PostAction.

If the resulting state after executing the SOP is the goal state, the teacher sends a positive response. In all other cases it replies negatively.

2) Equivalence queries: Once the learner has completed one cycle of learning – by fulfilling properties of the closedness and consistency – it presents a hypothesis to the teacher. The role of the teacher is to either provide a suitable counterexample existing in the system to be learnt, but not in the hypothesis; or the teacher approves the hypothesis to be an acceptable model of the system.

Generation of counterexamples is not easy, the teacher needs to exhaustively try all possibilities in order to provide a counterexample. Presence of loops in the model add complexity to the task. A survey of various counterexample generation strategies applied to communication protocols are discussed in [25]. More generally, communication protocols are usually modelled as Mealy machines. Counterexamples are generated by comparing the output strings obtained from the target system and the hypothesis, for a given input string. One of the main differences while applying counterexample generation strategies using the setup defined in this paper is to perform the comparison against the final state reached. The example discussed in Section V will clarify how counterexamples are obtained.

V. L* LEARNING APPLIED TO A ROBOTIC ARM

In this section the L* algorithm is applied to construct a model of a simulated robotic arm. The robotic arm is programmed using ladder logic and controlled by a PLC. The arm can move in the X and Y directions, and is fitted with a gripper that allows the arm to grip objects. The gripper has the ability to extend and retract in order to grip. The program contains the required logic to control the arm, which includes moving the arm in four directions: up, down, left, and right; and extending, retracting, closing, and opening the gripper.

The different operations are controlled, in the PLC, by a variable that can start the execution of its corresponding action. Additionally, the PLC program keeps track of the current position of the arm in both the X and Y directions using sensors. Figure 2 provides an image of the simulation environment.

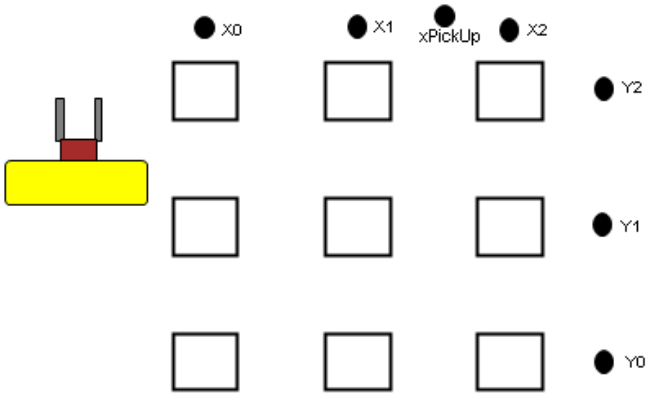

Fig. 2: Simulation of the robotic arm.

In order to learn the model of this robotic arm, the teacher must be able to communicate with the simulation environment. Furthermore, the algorithm requires the alphabet (the operations) as input, and also a technique to generate counterexamples. In this section we explore these different components and provide the end result from the learning algorithm.

The example will be treated as a purely discrete system with no parallelism. That is, only one operation is allowed to execute at any given time.

A. Communicating with the PLC

A communication system should be in place allowing the teacher and the PLC to communicate. This communication is achieved using the OPC protocol [26]. OPC provides bidirectional access to all the variables internal to the PLC. By monitoring the changes on the OPC, the state of the

target system is captured and observed. The observed state is maintained as a map of the variables and their current values, this map is called as a StateMap. The StateMap is a digital replica of the system's state.

The bidirectional nature of OPC provides the means to update variables on the PLC by changing their counterparts in the StateMap, thereby allowing the teacher to start an operation in the PLC by assigning the correct values in the StateMap.

B. Defining the system

The operations in the PLC need to be defined for the teacher and the learner in the format specified in Section III. The guard predicates for each operation must specify exactly when the operation is allowed to execute and when it has finished. Correspondingly, the actions need to specify the variable assignments to start and stop the operation. The guards ensure that only one operation can execute at a time, even though it is possible that several operations are enabled. Additionally, the guards must ensure that the arm cannot move when it is extended, and can open and close the gripper only when it has already extended.

The goal is constructed according to what we want to learn about the system. For simplicity, we define the goal to be the initial state of the arm. An interesting modification would be to define every state as part of the marked language. By doing this, as we will see, the learning takes place with fewer queries.

C. Generating counterexamples

Given a hypothesis, the teacher needs to find a counterexample. A counterexample is generated using a random walk algorithm. The teacher randomly selects one executable operation and executes it in the simulator. The resulting StateMap obtained is then compared with the corresponding operation in the hypothesis. If either such a trace does not exist in the hypothesis, or the StateMaps do not match, a counterexample is found. Since such a random walk can go on for an unbounded time we explicitly define a maximum number of allowed steps and also restart the algorithm at random instances to avoid the possibility of getting stuck in loops and to ensure randomness.

D. Results and Discussions

Using the setup as defined above the learning algorithm was capable of learning a model of the robotic arm. In order to learn how the algorithm scales as the simulation grows, the arm was allowed to move on different sized grids. Furthermore, the operations were grouped depending on the function they performed to see if it was possible to learn smaller subsystems rather than the complete system.

Figure 3 shows the learnt model while learning the four operations related to gripping. Figure 4 shows the model obtained while learning only the operations involved in motion. Both these were learnt against a grid size of 3×3 as seen in Figure 2.

The learning algorithm was also capable of learning the complete model of the robotic arm. The time taken to learn

Fig. 3: learnt model of extending and gripping

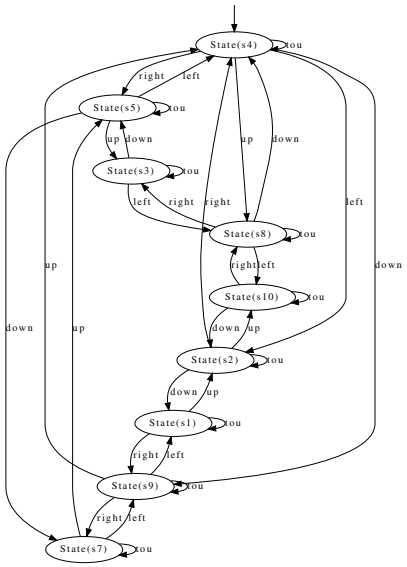

Fig. 4: learnt model consisting of the four movements

a 3×3 grid was around 5 minutes. However, the execution times of both cases vary significantly between different runs, so evaluating based on time does not provide good metrics. The main reason for this is the dependence on the simulated environment and the random walk algorithm. A more obvious way to evaluate the algorithm is based on the number of queries made. By increasing the grid size we could scale up the system to get a glimpse into the performance and applicability to real world systems.

Table I shows some metrics obtained for several trials while learning a model of the robotic arm. Table Ia shows the results for learning a prefix-free system, while Table Ib corresponds to the prefix-closed system. The 'Grid' column in the tables refers to the grid size within which the robot arm was allowed to move. 'Eq' and 'Mq' refer to the number of equivalence and membership queries made, and 'States' gives the total number of states in the learnt automata. The numbers in the tables are not absolute. They depend on the counterexamples provided, which in turn depend on the parameters of the random walk algorithm. Hence, all experiments were performed with fixed random walk parameters. For example, learning a grid of size 4×4 in the prefix-free setting resulted in an incomplete solution, missing several states and transitions, this was due to the current implementation of the random walk algorithm that terminates after traversing a fixed number of steps. The inference we draw here is the need to have better techniques to generate valid and useful counterexamples. A more in-depth study towards understanding properties of useful counterexamples and how to generate them is needed.

As can be seen in the tables, learning a prefix-closed

Grid	Eq	Mq	States
2x2	10	9476	17
3x3	24	118208	37
4x4	38	219431	56

(a) Prefix-free system

Grid	Eq	Mq	States
2x2	5	2981	17
3x3	8	17060	37
4x4	8	29021	65

(b) Prefix-closed system

TABLE I: Observations on the performance of L* while learning the model of the simulated robotic arm.

system results in fewer queries. This is contrary to the experiments performed using random automata in [27], which shows that learning a prefix-closed language results in higher number of queries. We hypothesise that this contradiction is due to the structure of our system. Probably, the cyclic nature due to operation pairs (up-down, left-right, extend-retract) is the cause for this behaviour.

VI. Conclusion

In conclusion, this paper presents an approach to learn discrete event models using simulation environments. To this end, the L* algorithm, a fundamental active learning algorithm, is briefly described. A framework is presented that enabled learning of models from simulations of systems. Furthermore, the presented framework was applied to learn the model of a simulated robotic arm. Through this, we could identify the different components that will enable us to apply the L* techniques to learn models of real manufacturing systems.

According to the experiences from these experiments, several lines of work are identified to be explored that will help towards applying these algorithms practically. These include improvement of the learning algorithm. More specifically, most of the existing algorithms and techniques are specifically designed to address the needs of either communication protocols or software programs, and need to be adapted or newer ones need to be explored that can be used on simulated manufacturing systems.

Another line of work to be explored concerns generation of counterexamples. Generating counterexamples is the bottleneck towards learning. An approach using the predefined specifications of the system or model-based testing may improve the quality of the counterexample generation.

The third line of work, and maybe the most challenging one, is to learn richer formalism in particular Extended Finite State Machines [28].

References

[1] J. Campos, C. Seatzu, and X. Xie, *Formal methods in manufacturing*. CRC press, 2014.

[2] G. Frey and L. Litz, "Formal methods in PLC programming," in *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, vol. 4, 2000.

[3] M. Dahl, K. Bengtsson, P. Bergagård, M. Fabian, and P. Falkman, "Integrated virtual preparation and commissioning: Supporting formal methods during automation systems development," *IFAC-PapersOnLine*, vol. 49, no. 12, pp. 1939–1944, 2016.

[4] M. Dahl, A. Albo, J. Eriksson, J. Pettersson, and P. Falkman, "Virtual reality commissioning in production systems preparation," in *22nd IEEE International Conference on Emerging Technologies And Factory Automation*, 2017.

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

[6] C. Y. Cho, E. C. R. Shin, D. Song *et al.*, "Inference and analysis of formal models of botnet command and control protocols," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 426–439.

[7] 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.

[8] C. de la Higuera, "A bibliographical study of grammatical inference," *Pattern Recognition*, vol. 38, no. 9, 2005.

[9] ——, *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.

[10] A. Farooqui, K. Bengtsson, P. Falkman, and M. Fabian, "From factory floor to operation models: An approach to generate, transform, and visualise manufacturing systems," 2018, submitted for possible publication.

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

[12] 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.

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

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

[15] R. E. Schapire, *The Design and Analysis of Efficient Learning Algorithms*. Cambridge, MA, USA: MIT Press, 1992.

[16] M. J. Kearns and U. V. Vazirani, *An Introduction to Computational Learning Theory*. Cambridge, MA, USA: MIT Press, 1994.

[17] 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. Cham: Springer International Publishing, 2014, pp. 307–322.

[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*. Springer, 2011, pp. 256–296.

[19] B. Jonsson, *Learning of Automata Models Extended with Data*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 327–349.

[20] F. Aarts and F. Vaandrager, "Learning I/O automata," in *CONCUR 2010 - Concurrency Theory*, P. Gastin and F. Laroussinie, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 71–85.

[21] 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.

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

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

[24] R. Rivest and R. Schapire, "Inference of finite automata using homing sequences," *Information and Computation*, vol. 103, no. 2, pp. 299 – 347, 1993.

[25] R. Dorofeeva, K. El-Fakih, S. Maag, A. R. Cavalli, and N. Yevtushenko, "FSM-based conformance testing methods: A survey annotated with experimental evaluation," *Information and Software Technology*, vol. 52, no. 12, pp. 1286 – 1297, 2010.

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

[27] "Insights to Angluin's learning," *Electronic Notes in Theoretical Computer Science*, vol. 118, pp. 3 – 18, 2005.

[28] 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.