Applying state machine learning at ASML

Premchand, J.R.P.

Award date:
2018

Technische Universiteit
Eindhoven
University of Technology

Department of Electrical Engineering
Electronic Systems Group

Applying State machine learning at ASML

Master Thesis

J.R.P Premchand (1283448)

Supervisors:
prof.dr.ir. J.P.M. (Jeroen) Voeten
dr.ir. R.R.H (Ramon) Schiffelers
ir. D. (Dennis) Hendriks

Final Version

Eindhoven, October 8, 2018

Abstract

This report presents the work done as a part of Master graduation project in the Embedded Systems program of Technical University of Eindhoven. ASML is the world's leading provider of complex lithography systems for the semiconductor industry. Over the last few years the complexity of these systems and their architectures have increased at a rapid pace. ASML is adopting Model-Driven Software Engineering techniques to address this challenge. Model-Driven Engineering (MDE) helps software developers specify the requirements using models and then verify the correctness of the software in the early stages of development. Inferring models from software components speeds up this process and helps developers to get closer to the actual behavior of components. There are several model learning algorithms to obtain models. State machine learning is a promising model learning technique which infers models from traces. In this thesis we provide insights on the various state machine learning algorithms.

The various model learning algorithms provides models in different formalisms like State Machines and Petri nets. To compare these different model learning algorithms we need to know how close these models are to each other. Hence we define notions of closeness between models based on language inclusion both qualitatively and quantitatively. Since language inclusion is a partial order, we can obtain a lattice of models giving insights on how close or far away these models are from each other. We also provide a quantitative notion of closeness by computing precision and recall measures for each model. To achieve this we have developed a tool chain which can compare the results of different model learning algorithms. Using this approach we can systematically explore models to find those that are as close as possible to the observed behavior, while still being understandable.

State machine learning algorithms are implemented in the state-of-the-art tool Flexfringe. The Flexfringe tool provides a number of settings to tune the obtained models. We provide a set of guidelines to choose the appropriate algorithm and settings based on the insights from literature and the comparison experiments done using the model comparison approach we developed and its tool chain.

We validated our guidelines and model comparison approach by applying it to learn the behavior of an actual software component from ASML. The case study shows that the approach can be applied for real systems. In addition the case study has resulted in several insights of the interface behavior of the component under study.

Acknowledgement

First of all, I would like to thank God for giving me the strength and patience to complete my master thesis. I would like to sincerely thank my graduation supervisor prof.dr.ir. J.P.M. (Jeroen) Voeten who has been a constant support and guiding light throughout my thesis. His vast knowledge and experience helped me not only in improving my technical skills but also my soft skills which prove to be very essential in the next phase of my career. I have thoroughly enjoyed every discussion with him and i always had something to learn from him.

I would like to express my gratitude to dr.ir. R.R.H (Ramon) Schiffelers for his constant support during my graduation project at ASML. My discussions with him have always been fruitful and motivated me to achieve more. I would like to thank ir. L. (Leonard) Lensink who helped me to achieve my goals by sharing his expertise. He was always there cheerfully to clarify my doubts. I am greatly indebted to him. I sincerely would like to thank ir. D. (Dennis) Hendriks for his constructive feedbacks for my presentations. I would also like to acknowledge him for accepting to be my committee member.

I would also like to thank all my colleagues in the ASML SW Research team for their guidance and feedback which helped me improve my scientific thinking. Finally I would like to thank my parents and friends who were always there for me through thick and thin.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

ASML is the world's leading provider of complex lithography systems for the semiconductor industry. Over the last few years the complexity of these systems and their architectures have increased at a rapid pace due to the demands for higher performance, better quality and speed. A lot of software development in the past have been done using traditional software development processes. These traditional methods have difficulties to cope with the increasing complexity. To cope with this challenge, ASML is adopting Model-Driven Software Engineering techniques. Model-Driven Engineering (MDE) helps software developers specify the requirements using models and then verify the correctness of the software in the early stages of development. Inferring models from software components speeds up this process and helps developers to get closer to the actual behavior of components. Several techniques to infer models exist and ASML is currently exploring and validating these techniques. Combining these techniques help in obtaining models which can approximate the behavioral specification of a real software component. There are static model inference techniques and dynamic model inference techniques to obtain models. Static analysis is a white box approach examining the software code and reasoning about the possible behaviors at run time. Dynamic techniques are black box approaches, of which two main categories exist: Active and Passive learning techniques. Active learning techniques are capable of actively pursuing queries to the System Under Learning (SUL) and learn the automaton based on the SUL's response. Passive learning on the other hand, infers models from observed behavior.

ASML has earlier explored process mining, a passive learning technique to obtain models from software traces. Process mining delivers models in different formalisms such as Petri nets, process trees and causal nets. State machine learning is another promising passive learning technique. This technique has not been explored yet by ASML and ASML would like to validate state machine learning by applying it to a real software component.

In the remainder of this chapter, we list the goals of our thesis in Section 1.1 and the approach to achieve our goals in Section 1.2. In Section 1.3, we describe the industrial case study from ASML and in Section 1.4 we provide the main contribution of the thesis. Finally, Section 1.5 gives the organization of the thesis.

1.1 Thesis Goals

ASML is exploring and validating the various state-of-the-art model inference techniques. The goal of this project is fourfold and is discussed below.

1. Our first goal is to provide insights on State machine learning.

2. Our second goal is to develop a framework which can compare the results of different model learning algorithms and settings.

3. Based on the data available, different state machine learning algorithms and corresponding

settings can be chosen. Our third goal is to provide guidelines to choose the appropriate state machine learning algorithm and corresponding settings.

4. Our fourth goal is to apply state machine learning to an actual industrial case study from ASML using the guidelines.

1.2 Approach

To achieve the goals discussed in the previous section we take the following approach

1. State machine learning is a passive learning technique used to obtain models from software traces. State machine learning is based on the core principle of state merging. State machine learning has different algorithms to obtain models. In this thesis the Hankel, Alergia and Overlap driven algorithms are discussed. These algorithms are implemented in the state-of-the-art Flexfringe tool. The Flexfringe tool provides various settings for obtaining models. Our first goal is to provide insights on these state machine learning techniques and the various settings used. A detailed study on state machine learning technique and the various model learning algorithms is done to obtain insights on how these algorithms work. We also list the various settings used in the tool Flexfringe to obtain models.

2. There are several model learning algorithms to obtain models. These learning algorithms obtain models in different formalisms like State machines, Petri-nets, causal trees and many more. This complicates comparing these models. In this thesis we develop a framework to compare the different model learning algorithms based on formal notions of closeness between models. We define notions of closeness between models based on language inclusion and equivalence both in a qualitative and quantitative way. The qualitative notions of closeness is defined based on language inclusion and equivalence. These induce a structure called a lattice which provides insights in the relative closeness of models in this lattice. To compare models that are not related we develop a quantitative notion of closeness, based on precision and recall measures. To compute these closeness relations between models we developed a tool chain which can compare the results of different learning algorithms. ASML has already validated process mining, a passive model learning algorithm and the results for the industrial case study are available. We will use the results of process mining to validate our tool chain by comparing them with the results of state machine learning.

3. State machine learning technique has several algorithms and settings to learn models. Each algorithm and setting yields different state machine models. To achieve our third goal, to choose the appropriate state machine learning algorithm and setting based on the available input, we create synthetic reference models and automatically obtain traces from it. We use the tool chain developed to apply the various algorithms and setting in state machine learning to obtain models from the traces and determine their closeness to the reference model. The guidelines are obtained from literature and these synthetic case studies. These case studies are also used to validate these guidelines.

4. We apply the guidelines to a real industrial case study from ASML. The ASML case study does not have a reference model and hence we make use of the guidelines and tool chain to obtain models which approximate the observed behavior in traces as close as possible while still obtain understandable models. This industrial case study is discussed in the next section.

1.3 Industrial Case Study

ASML machines contain software consisting of many intercommunicating components. These components interact with each other through interfaces. ASML likes to obtain models of these

interfaces to understand their behavior. The goal is to make use of the guidelines in state machine learning to obtain understandable models which are close to the observed traces of these software components. Figure 1.1 is a schematic view of the various components and interfaces of the case study we consider in this thesis. A and B are two components which implement a common interface protocol I. Components A and B have I1 (Swap) and I2 (SwapAdv) as their interfaces. ASML would like to obtain models of the behavioral specification of the interfaces I1 and I2. ASML has run a number of tests for each interface I1 and I2, logging the sequences of symbols as traces. The two tests are production (tests run in a production environment) and component test (tests run with specific components). These traces are used to learn models using the state machine learning algorithms.

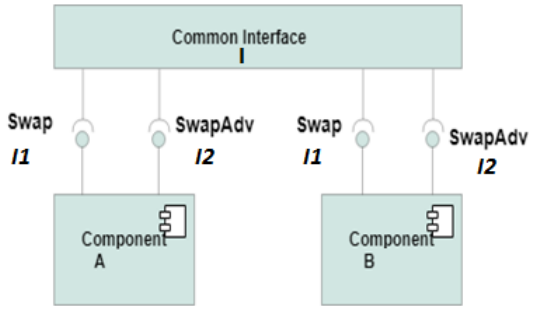


Figure 1.1: Components and interfaces in ASML case study

1.4 Contributions

The main contributions of our work are as follow:

1. **Insights on State machine learning.** In this thesis we explore alternative State Machine Learning (SML) algorithms and their settings. We provide insights on their working and application.

2. **A framework to compare and obtain models.** We develop a framework to compare the different model learning algorithms based on language-based qualitative and quantitative notions of closeness between models. These allow models to be compared independent of the formalism they are expressed in. To compute these closeness relations between models we developed a tool chain which can compare the results of different learning algorithms.

3. **Guidelines to obtain models using State machine learning.**

   State machine learning has several algorithms and a range of settings to obtain models. Guidelines to choose the appropriate state machine learning algorithm and settings are given and these guidelines are validated using synthetic examples.

4. **Applying state machine learning to the industrial case study from ASML.**

   We use the guidelines and framework to apply state machine learning to a case study from ASML to obtain understandable models which approximate the observed behavior in the traces as close as possible.

1.5 Outline

This report is organized as follows:

1. Chapter 2- This chapter explains the various model learning algorithms. We explain the various algorithms in state machine learning and how they work. We also discuss in brief about process mining and the Petri net formalism it utilizes. Finally we explain the state-of-the-art state machine learning tool Flexfringe and its parameter settings.

2. Chapter 3- This chapter develops our framework to formally compare the various model learning algorithms. In this chapter we also introduce a tool chain which materializes this framework.

3. Chapter 4- This chapter provides the guidelines to use state machine learning. The guidelines are validated using several synthetic case studies.

4. Chapter 5- This chapter describes the case study from ASML and also the results obtained by applying state machine learning.

5. Chapter 6- This chapter gives the conclusion and directions for future research.

Chapter 2

Model Learning Algorithms

This chapter discusses the various model learning algorithms applied in this thesis. We first introduce the fundamentals of state machine learning. We provide insights on how the various state machine learning algorithms work and also briefly explain about process mining and its Petri net formalism. In addition we explain the tool Flexfringe used for state machine learning.

2.1 State machine learning

State machine learning is a passive model learning technique which abstracts models in the form of state machines from software traces. There are several state machine learning algorithms to obtain models and they mainly focus on Deterministic Finite Automata (DFA).

A DFA is an automaton that accepts valid sequences and rejects invalid strings of inputs. Its states are either accepting or rejecting, and the transitions between states are labeled with inputs. A state cannot have two outgoing transitions with the same input. A DFA has a single distinguished initial state in which all strings start. Upon receiving an input, it transitions to the corresponding next state. An accepting state is one in which a valid input string ends. If the automaton is in an accepting state when no inputs are left, then the string of inputs is valid. If it ends in a rejecting state, the string is invalid.

Definition of DFA *Formally, a deterministic finite automaton (or finite state machine) is an ordered quintuple (Σ, Q, q0, δ, F), where:*
Σ is a finite alphabet with a set of symbols,
Q is a finite set of states,
q0 $\in$ Q is the initial state,
δ the transition function, is a function from $Q \times \Sigma$ to Q, and
$F \subseteq Q$ is the set of final (accepting) states.

Figure 2.1 shows a simple DFA with three states. A finite state machine M, encodes a language L. Let ω be a string. M is said to accept string ω if the machine starts from the initial state, undergoes transitions corresponding to the symbols in ω , and ends up in an accepting state. We say that the machine M recognizes the language L if M accepts precisely all strings ω that are in L. A language is a regular language if there is a finite automaton that recognizes it. The language of an automaton A is represented as L(A). The language of the automaton A in Figure 2.1 is L(A) = $\{ab^n a \mid n \geq 0\}$.

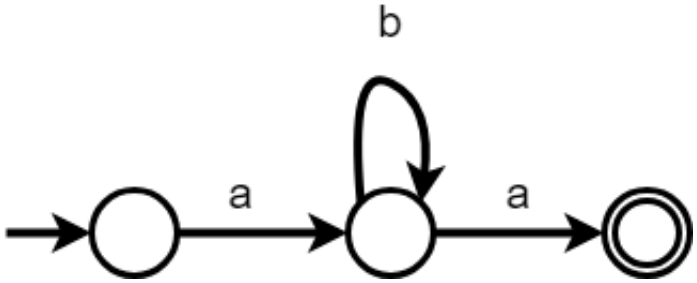


Figure 2.1: A simple DFA

2.1.1 Goal of State machine learning

A string is a finite sequence of symbols from a finite alphabet. Σ^* is the set of all finite strings over alphabet Σ. Given a regular language $L \subseteq \Sigma^*$, let $S^+ \subseteq L$ and $S^- \subseteq \overline{L}$ where $\overline{L} = \Sigma^* \setminus L$. S^+ denotes a set of positive traces that are accepted by the automaton and S^- denotes a set of negative traces that are rejected by the automaton.

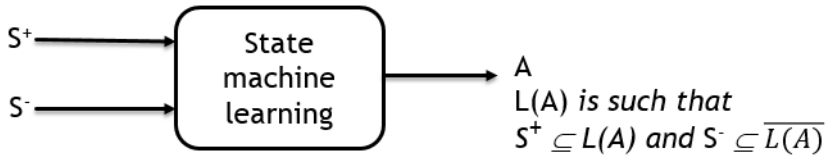


Figure 2.2: Goal of State machine learning

The goal of state machine learning is to find the **smallest** DFA that is **consistent** with the given data. The size of a DFA is determined by the number of states in the DFA. It is desired that this DFA is as small as possible because of an important principle known as Occam's razor, which states that among all possible explanations for a phenomenon, the simplest one is to be preferred. A smaller DFA is simpler, and therefore a better explanation and more likely a model for the observed examples [6]. A DFA is consistent if and only if it accepts all positive strings (S^+) and rejects all negative strings (S^-). Let A be the DFA obtained from state machine learning. Then the language L(A) of DFA A is such that $S^+ \subseteq L(A)$ and $S^- \subseteq \overline{L(A)}$. This is illustrated in Figure 2.2. Hence the main goal of state machine learning is to find a DFA A which is minimal and also consistent:

- **Minimal** - DFA A has the least number of states.
- **Consistency** - DFA A accepts all positive strings (S^+) and rejects all negative strings (S^-).

The idea of having a rejecting state in a DFA is relatively new when compared to the classical finite automaton theory. The idea is that every string in both S^+ and S^- is given a run in the automaton and a string is rejected if and only if the string halts in a rejecting state with no more input left. A typical DFA with a rejecting state is depicted in the Figure 2.3, where strings bba, aa and aab are accepted and a and bb are rejected.

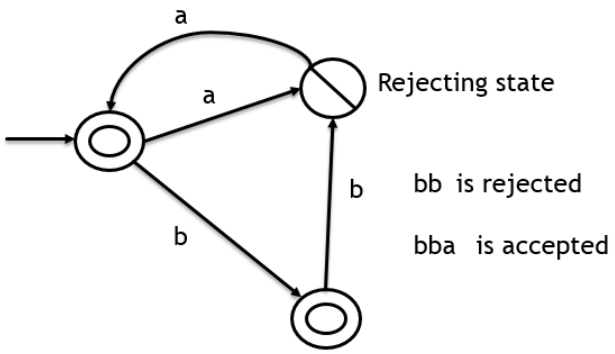


Figure 2.3: A DFA that accepts strings bba, aa, aab and rejects a, bb

2.1.2 State merging algorithms

State machine learning is based on the state merging principle. There are several state merging algorithms to obtain models from system traces. However, all these state merging algorithms have three fundamental steps to obtain models. They are illustrated in Figure 2.4 [1]. In step 1, all

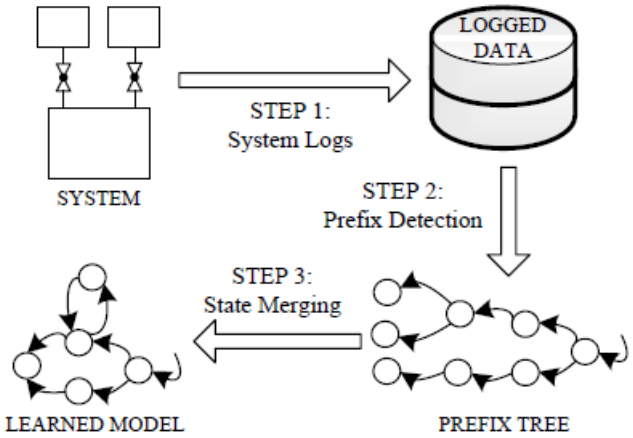


Figure 2.4: State merging approach

relevant data are obtained from a systems execution and these traces are logged in a database. For timed systems, logs also include time stamps. An initial automaton called a Prefix Tree Acceptor (PTA) is then built in step 2 from the traces. Let S be a set of traces from the system. A prefix tree acceptor (PTA) is a tree-like DFA built from the traces in S by taking all the prefixes of traces in S as states and constructing the smallest DFA A for which L(A) = S. Figure 2.5 shows a PTA which accepts strings aa, aba and bba [1].

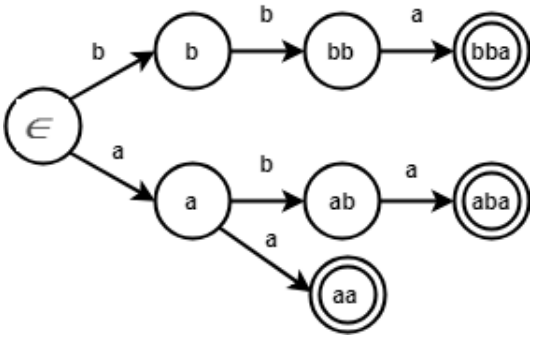


Figure 2.5: A PTA accepting aa, aba, and bba

In step 3, the actual learning takes place using different state merging algorithms. In each learning algorithm compatible pairs of PTA states are merged until the underlying automaton is identified. State merging makes the automaton smaller and generalizes the language encoded by the automaton. Different algorithms use different compatibility tests to merge states. A compatibility test is used to determine if two states can be merged or not in the PTA. The algorithms differ mainly in the conditions used for compatibility test. In general, a learning algorithms can use learning strings that can be both positive (coming from a normal operation) and negative (coming from an abnormal operation) [1]. However, in actual systems the number of negative strings that can be obtained is typically very small.

The state-of-the-art tool for state machine learning is Flexfringe. Flexfringe is a passive automaton learning tool with emphasis on flexibility in merge heuristic [4]. Flexfringe is an open source tool supporting several state merging algorithms like EDSM [5], overlap driven, likelihood, Alergia [3] and DFASAT [6] algorithms. It can also make use of stochastic information for learning models as Probabilistic Finite Automaton (PFA). The Hankel algorithm, the Alergia algorithm and the overlap driven algorithm are the state merging algorithms used in this thesis. These algorithms are discussed in the following sections.

2.1.3 Hankel Algorithm

The Hankel algorithm is a state merging algorithm which makes use of both positive and negative traces to obtain state machine models. The Hankel algorithm is based on the fundamental theorem called the Myhill-Nerode theorem. In the theory of formal languages, the Myhill-Nerode theorem provides a necessary and sufficient condition for a language to be regular. The Myhill-Nerode theorem defines an equivalence relation over the strings in a language [9]. We will explain the Hankel algorithm with two different cases, one is an ideal case in which we obtain a state machine model for a given language and the other case is one in which a limited set of traces is given to obtain a model. The former ideal case yields more evidence for merging states as the complete language is known and the latter is a common case in which evidence for merging is a limited set of traces. Both these cases and the procedure to obtain models are discussed below.

Case 1: Hankel method for a given language :

In this case the procedure to identify a minimal and consistent DFA for a given language is discussed. Let us consider the regular language $L = \{ab^n a \mid n \geq 0\}$ expressed by the regular expression $a\ b^*\ a$. The goal is to learn the minimal and consistent DFA that encodes this language. Here we assume the complete language to be known.

Procedure:

- **Step 1** - Construct a Hankel matrix in which the rows represent the prefixes of the strings in the language and the columns represent extensions of these strings.

- **Step 2** - Fill the Hankel matrix by extending each string $prefix$ in rows with a string $suffix$ in columns to obtain the extended string $prefix \cdot suffix$. If this extended string is in the language then value 1 is assigned to the cell corresponding to the row and column and else value 0 is assigned. In this way all cells in the Hankel matrix are filled with zeros and ones.
- **Step 3** - Iteratively color the rows by giving the same colors to rows for which the cells have the same values.
- **Step 4** Construct a DFA using the table where the number of colors in the table is equal to the number of states in the DFA. For each state s we constructs the outgoing transitions by considering each symbols a, b in the alphabet of the language. Symbol a is appended to any prefix $prefix$ corresponding to state s. If $prefix \cdot a$ is in the language, a transition with label a is created ending in the state corresponding to $prefix \cdot a$. If $prefix \cdot a$ is not in the language, no transition is created.

Using these rules we will now construct the DFA of language L = {ab^{n}a | n$\geq$ 0}. In step 1, we construct the Hankel matrix with the prefixes of strings in the language a, aa, ab, aba, abb..... and so on. The suffixes for the extended strings are the same. Since the language given is infinite the Hankel matrix is also infinite. The skeleton of the infinite Hankel matrix is given in the Figure 2.6.

	∈	a	aa	ab	aba	abb	...
∈							
a							
aa							
ab							
aba							
abb							
...							

Infinite Hankel matrix

Figure 2.6: Hankel matrix skeleton with strings in rows as prefix and strings in columns as suffix.

In step 2, we extend the prefixes in the rows with the suffixes in the columns and check if the extended string is in the language or not. If the string is in the language we assign a value 1 and else we assign 0 to the cell in the Hankel matrix. In this way all the cells of the Hankel matrix are assigned a value 0 or 1. The resulting matrix is given in the Figure 2.7.

	∈	a	aa	ab	aba	abb	...
∈	0	0	1	0	1	0	...
a	0	1	0	0	0	0	...
aa	1	0	0	0	0	0	...
ab	0	1	0	0	0	0	...
aba	1	0	0	0	0	0	...
abb	0	1	0	0	0	0	...
...	...	...	...	...	...	...	...

Figure 2.7: Filled in complete Hankel matrix

In step 3 we color all rows by comparing the cell values of the rows. Rows are assigned the same color if they contain the same values. Rows having the same colors are related and correspond to a state in the final DFA. A colored Hankel matrix for language L = $\{ab^n a \mid n \geq 0\}$ is given in Figure 2.8.

	∈	a	aa	ab	aba	abb	...
∈	0	0	1	0	1	0	...
a	0	1	0	0	0	0	...
aa	1	0	0	0	0	0	...
ab	0	1	0	0	0	0	...
aba	1	0	0	0	0	0	...
abb	0	1	0	0	0	0	...
...	...	...	...	...	...	...	...

Figure 2.8: Colored Hankel matrix with related rows

In step 4 we construct a DFA, which is shown in Figure 2.9. The state corresponding to the empty prefix string is the initial state. For this initial state we consider each of the symbols a and b in the alphabet. Appending a to ϵ yields string a which corresponds to the yellow state. Appending b to ϵ yields string b which does not occur as any prefix in the table. Hence there exists no transition from the empty state labeled with b. In a similar way the outgoing transitions of the other states are constructed. To label a state to be accepting or not, we consider the column with suffix ϵ. A state is accepting if the corresponding cell in this column has value 1. Otherwise the state is not accepting.

In this example the regular language is given and the resulting DFA captures the exact language L = $\{ab^n a \mid n \geq 0\}$. The coloring algorithm for this example is complex and is not solvable in polynomial time and falls under the category 'NP-hard'. This problem is not verifiable in polynomial time as the language is infinite. However in real cases it is difficult to obtain the entire language as input and we ideally want to find the underlying DFA from a set of traces. In the next case the procedure for learning a language from a set of traces is discussed with an example.

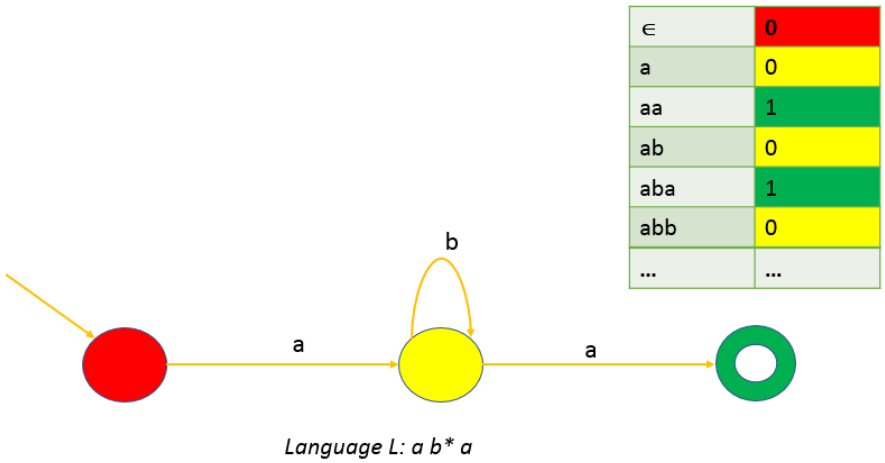


Figure 2.9: Minimal DFA obtained from the Hankel matrix

Case 2: Hankel method for a given set of traces

The goal of state machine learning is to find a smallest DFA that is consistent with the given data. In the previous example we assumed that the complete language was available. This is not true in reality in which we only have a set S^+ of positive traces and a set S^- of negative traces available. The goal is then to learn the smallest DFA that accepts S^+ and rejects S^-. The procedure to obtain models from these sets is as follows:

Procedure:

- **Step 1** - Construct a Hankel matrix in which the rows represent the prefixes of the strings in S^+ and S^- and the columns represent extensions of these strings.
- **Step 2** - Fill the Hankel matrix by extending each string $prefix$ in rows with a string $suffix$ in columns to obtain the extended string $prefix \cdot suffix$. If this extended string is in S^+ then value 1 is assigned to the cell corresponding to the row. If the extended string is in S^- then value 0 is assigned to the cell corresponding to the row. If the extended string is neither in S^+ nor in S^-, the cell is left empty.
- **Step 3** - Rows are given the same color if they do not disagree for each pair of cells in the same column. Two cells disagree if one has value 1 and the other has value 0.
- **Step 4** - Construct a DFA using the table where the number of colors in the table is equal to the number of states in the DFA. For each state s we constructs the outgoing transitions by considering each symbols a, b in the alphabet of the language. Symbol a is appended to any prefix $prefix$ corresponding to state s. If $prefix \cdot a$ is in S^+ or in S^-, a transition with label a is created ending in the state corresponding to $prefix \cdot a$. If $prefix \cdot a$ is neither in S^+ nor in S^-, no transition is created.

As an example consider a set of positive traces $S^+ = \{$ a, aba, bbb, bba $\}$ and negative traces $S^- = \{$ b, aa, abaa $\}$. In step 1, we construct the Hankel matrix with prefixes in $\{$ a, b, ab, aa, bb, aba, bbb, bba, abaa $\}$. The suffixes for the extended strings are the same. The skeleton of the Hankel matrix is given in Figure 2.10.

In step 2, we extend the strings in the rows with the strings in the columns and check if the extended string is in the given sets of traces or not. If the extended string is present as a positive

$S^+ = \{ a, aba, bbb, bba \}\ S^- = \{ b, aa, abaa \}$

	∈	a	b	ab	aa	bb	aba	bbb	bba	abaa
∈										
a										
b										
ab										
aa										
bb										
aba										
bbb										
bba										
abaa										

Figure 2.10: Hankel matrix skeleton with strings in rows as prefix and strings in columns as suffix

trace we assign value 1 and if the extended string is present in a negative trace we assign value 0. The cells of the extended strings for which we do not have such information are left empty. Since we have limited trace data we can only fill a few cells in the Hankel matrix. The resulting Hankel matrix is given in Figure 2.11.

$S^+ = \{a, aba, bbb, bba\}\ S^- = \{b, aa, abaa\}$

	∈	a	b	ab	aa	bb	aba	bbb	bba	abaa
∈		1	0		0		1	1	1	0
a	1	0								
b	0					1				
ab		1			0					
aa	0									
bb		1	1							
aba	1	0								
bbb	1									
bba	1									
abaa	0									

Figure 2.11: Filled in Hankel matrix

In step 3, we merge two rows if and only if their values do not disagree with each other. The rows which do not disagree are given the same color. Once the rows which satisfy this condition are identified the values from one row are copied to the other in order to fill more cells in the Hankel matrix, which helps to find related rows and also distinguish between the states. Rows having the same colors are related and correspond to a state in the final DFA. The colored Hankel matrix for the given traces is given in the Figure 2.12.

$S^+ = \{a, aba, bbb, bba\}$, $S^- = \{b, aa, abaa\}$

	∈	a	b	ab	aa	bb	aba	bbb	bba	abaa
∈		1	0		0		1	1	1	0
a	1	0								
b	0					1				
ab		1	0		0		1	1	1	0
aa	0					1				
bb		1	1							
aba	1	0								
bbb	1	0								
bba	1	0								
abaa	0					1				

Figure 2.12: Colored Hankel matrix with related rows

In step 4 we construct a DFA, which is shown in Figure 2.13. The state corresponding to the empty prefix string is the initial state. For this initial state we consider each of the symbols a and b in the alphabet. Appending a to ϵ yields string a which corresponds to the yellow state. Appending b to ϵ yields string b which corresponds to the green state. In a similar way the outgoing transitions of the other states are constructed. To label a state to be accepting, rejecting or not accepting, we consider the column with suffix ϵ. A state is accepting if the corresponding cell in this column has value 1. A state is rejecting in case the value is 0. Otherwise the state is not accepting. In the example the yellow state is accepting, the green state is rejecting and the other states are non accepting. Notice that we can see that negative strings b, aa and abaa finish in the rejecting green state. The positive traces a, aba, bbb and bba finish in the accepting yellow state. This DFA is minimal and is also consistent with the given set of traces.

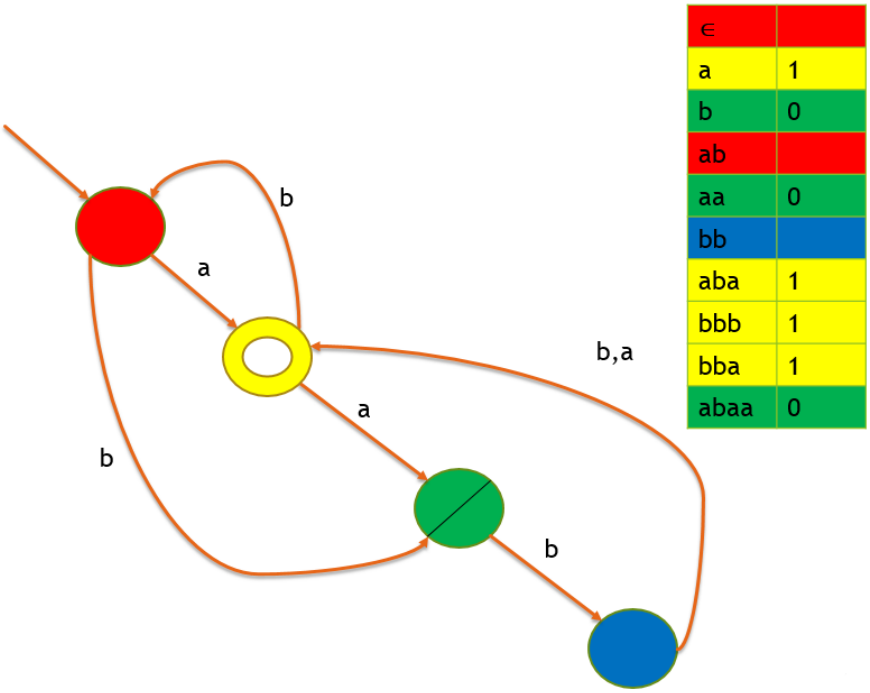


Figure 2.13: Minimal and consistent DFA obtained from the Hankel matrix

Coloring problem and need for heuristics:

The rows in the Hankel matrix are merged only if the values do not disagree with each other at any cell. It can happen that a row A agrees with both rows B and C, but that B and C do not agree. This means that we can merge rows A and B or rows A and C, but not rows A, B and C. Row merging is thus not transitive. Figure 2.14 depicts an example of such a case. In this example rows 1 and 2 do not disagree with each other and hence can be merged to form states with same color. Rows 2 and 3 also do not disagree with each other and can also be merged together. It is not possible to merge rows 1, 2 and 3, since rows 1 and 3 do not agree. Both merging rows 1 and 2 and rows 2 and 3 will result in a consistent DFA but only one of these merges might lead to a minimal DFA. In general, to find the minimal consistent DFA, all different merge options have to be investigated. The fact that row merging is not transitive renders the coloring problem to be NP-hard. This is the reason why state-merging algorithms typically depend on heuristics to search for the minimal consistent DFA.

	0	0	0
1	0		0
1		1	

	0	0	0
1	0		0
1		1	

Figure 2.14: Coloring problem in merging rows

2.1.4 Overlap driven algorithm

Overlap driven state merging is another algorithm to obtain a state machine model from a set of traces. This algorithm learns models only from positive data.

The algorithm first represents the given traces as a Prefix Tree Automaton (PTA). The algorithm then iteratively merges states in the PTA that are equivalent. Two states are considered equivalent if and only if they have the same sequence of symbols following them. The number of symbols that has to be the same for two states to be equivalent can be set using a parameter called lower_bound. For example, when the lower_bound value is set to 2, the algorithm checks if any two states have the same sequences of symbols of length 2 and if so merges these states. The algorithm starts from the initial state and tries to identify a first pair of states that are equivalent. Once the first pair of states is identified, they are merged and the PTA is updated. The algorithm then searches for the next pair of states that are equivalent. The algorithm terminates and outputs the model when no two remaining states are equivalent.

The parameter lower_bound determines the size and generality of the inferred model. A smaller lower_bound value leads to more merges and produces more compact models, while a larger lower_bound restricts the number of states which are equivalent. Overlap driven algorithm can learn recursive automata if the parameter lower_bound is selected in such a way that it identifies the repetition of events in a trace.

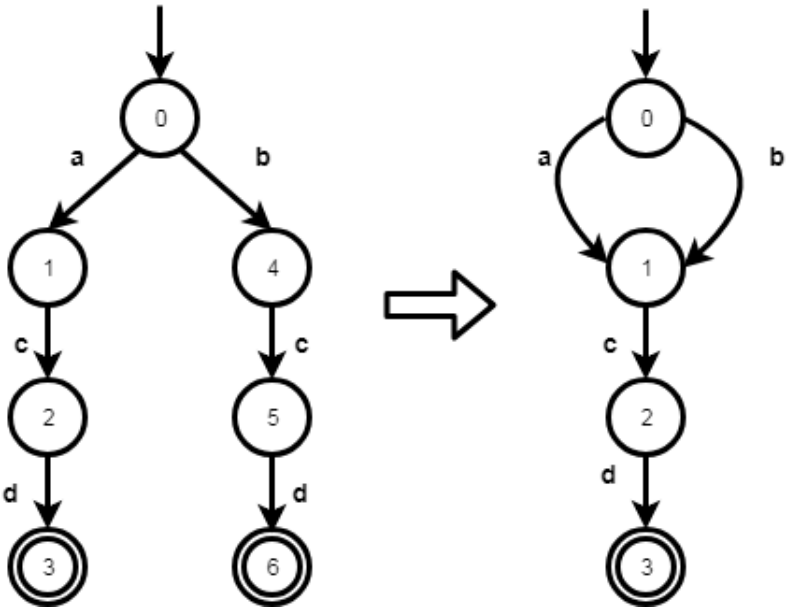


Figure 2.15: Overlap driven state merging algorithm for traces acd and bcd

In Figure 2.15 the state merging process of Overlap driven algorithm for a PTA is shown. The PTA is constructed with two traces, acd and bcd, where 0 is the initial state and 3 and 6 denote the accepting states. We select the lower_bound value to be 2. The algorithms starts from the initial state 0 and examines if any two states have the same sequence of symbols for a length 2. In the PTA, when states 1 and 4 are examined, we see that state 1 is followed by the symbols c and d corresponding to states 2 and 3. Similarly state 4 is also followed by symbols c and d corresponding to states 5 and 6. Hence the equivalence condition is satisfied and states 1 and 4 and the same symbols which follow them (c and d) are also merged.

2.1.5 Alergia Algorithm

Alergia is a state merging algorithm which merges states based on the probability of symbols. Alergia makes use of probabilistic information to compensate for the absence of negative traces. Alergia can learn a Probabilistic Finite Automata (PFA) from a set of traces (positive data).

Definition of PFA *A Probabilistic Finite Automaton is an ordered quintuple (Σ, Q, q_0, P), consisting of an alphabet Σ, a finite set of states $Q= (q_0, q_1, q_2,...q_n)$, where q_0 denotes the initial state, and a set P of probability matrices $p_{ij}(a)$ giving the probability to transition from state q_i to state q_j by symbol a,*

The Alergia algorithm generates a Prefix Tree Automaton (PTA) from the input traces. For every single state, the probabilities of stopping in that state or taking a specific transition to another state is computed based on i) the number of strings that arrive at that state (terminate or pass through), ii) the number of strings that end in that state, and iii) for each symbol the number of strings that pass through and then transition to another state using that particular symbol. Let the number of strings that arrive to state q_i be n_i, the number of strings that end in q_i be f_i, and the number of strings that pass through state q_i transitioning with symbol a be ${f_i}^a$.

The outgoing probability for q_i with symbol a is then

$$\frac{{f_i}^a}{n_i},$$

while the terminating probability is given by

$$\frac{f_{\mathrm{i}}}{n_{\mathrm{i}}}$$

Two states are deemed equivalent if their outgoing and termination probabilities are the same. However experimental data are subject to statistical fluctuations and hence equivalence between states has to be accepted within a confidence range (α). The compatibility between any two states is evaluated using the Hoeffding bound [3]. The confidence range in the Hoeffding bound equation varies from 0 to 1. If the difference between both the probabilities (outgoing and terminating) of two states is less than the confidence range, these states are considered equivalent. For the outgoing probabilities for states q_i and q_j and symbol a the compatibility criteria is given by :

$$\left|\frac{f_i^a}{n_i} - \frac{f_j^a}{n_j}\right| < \sqrt{\frac{1}{2}\log\frac{2}{\alpha}}\left(\frac{1}{\sqrt{n_i}} + \frac{1}{\sqrt{n_j}}\right)$$

For the terminating probabilities for states q_i and q_j the compatibility criteria is given by :

$$\left|\frac{f_i}{n_i} - \frac{f_j}{n_j}\right| < \sqrt{\frac{1}{2}\log\frac{2}{\alpha}}\left(\frac{1}{\sqrt{n_i}} + \frac{1}{\sqrt{n_j}}\right)$$

We present a simple example to show how Alergia works. Let the set of given traces be S = { 110, ∈, ∈, ∈, 0, ∈, 00, 00, ∈, ∈, ∈, 10110, ∈, ∈, 100 }. Let us assume the confidence range for this example to be α= 0.8. The algorithms starts with the construction of a PTA given in Figure 2.16. There are 15 traces and the PTA has 11 states. Each state is assigned a number from 1 to 11. The number of strings arriving and terminating at each state is calculated and shown in square brackets ([]). For example [15,9] in state 1 means there are 15 strings arriving at state 1 and 9 strings terminating in state 1. Each transition has a label with symbol 0 or 1 along with the number of strings using that transition indicated between square brackets.

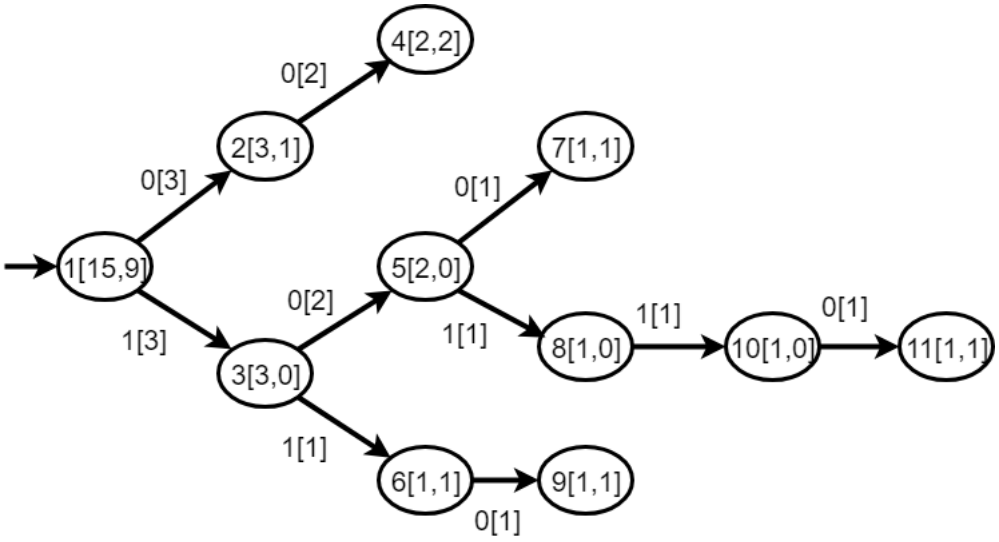


Figure 2.16: A PTA for the given strings

Once the string counts are calculated the algorithm checks if two states are equivalent, starting with the initial state 1. For state 1, $f_1 = 9$, $n_1 = 15$, $f_1^0 = 3$ and $f_1^1 = 3$. Thus $\frac{f_1}{n_1} = \frac{9}{15}$, $\frac{f_1^0}{n_1} = \frac{3}{15}$, $\frac{f_1^1}{n_1} = \frac{3}{15}$, $\frac{f_2}{n_2} = \frac{1}{3}$, $\frac{f_2^0}{n_2} = \frac{2}{3}$, $\frac{f_2^1}{n_2} = \frac{0}{3}$. For states 1 and 2 to be equivalent the following three conditions should be satisfied:

(a) Outgoing probabilities for states 1 and 2 with symbol 0.

$$\left|\frac{3}{15}-\frac{2}{3}\right| = 0.46 < \sqrt{\frac{1}{2}\log\frac{2}{0.8}}\left(\frac{1}{\sqrt{3}}+\frac{1}{\sqrt{15}}\right) = 0.55$$

(b) Outgoing probabilities for states 1 and 2 with symbol 1.

$$\left|\frac{3}{15}-\frac{0}{3}\right| = 0.2 < \sqrt{\frac{1}{2}\log\frac{2}{0.8}}\left(\frac{1}{\sqrt{3}}+\frac{1}{\sqrt{15}}\right) = 0.55$$

(c) Termination probabilities for states 1 and 2.

$$\left|\frac{9}{15}-\frac{1}{3}\right| = 0.26 < \sqrt{\frac{1}{2}\log\frac{2}{0.8}}\left(\frac{1}{\sqrt{3}}+\frac{1}{\sqrt{15}}\right) = 0.55$$

These conditions are satisfied and thus states 1 and 2 are assumed to be equivalent. These states can now be merged, but instead we first check the equivalence between states 2 and 4. Also for these states the conditions are satisfied, implying that states 1, 2 and 4 can be merged. The result is shown in Figure 2.17. The states 1, 2 and 4 are merged to form state 1. After this merge the number of strings that arrive at state 1, the number of strings that end in state 1, and for each symbol the number of strings that pass through and then transition to another state using that particular symbol are calculated again and updated in state 1 of PTA. In fact this can be achieved by adding the numbers in the states to be merged and by adding the numbers of their corresponding transitions.

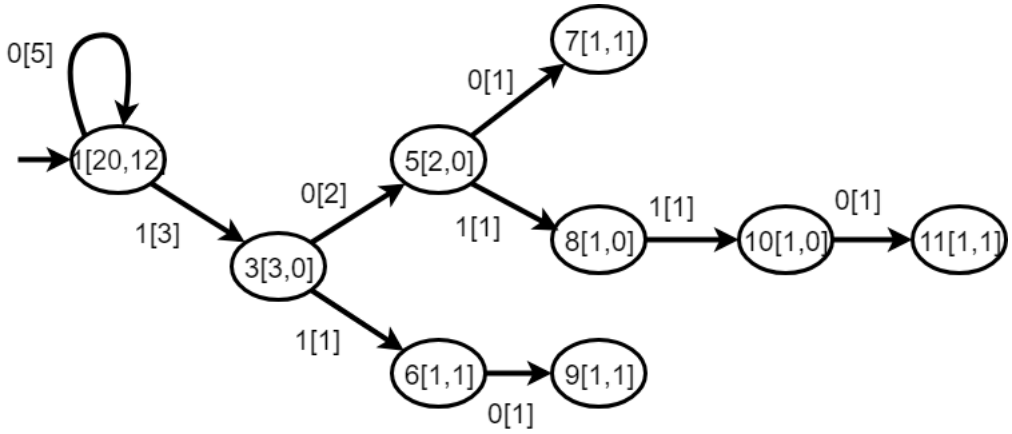


Figure 2.17: PTA after merging states 1, 2 and 4

The algorithm then checks for equivalence between states 1 and 3. But these are found to be non equivalent as the difference between their termination probabilities 0.6 and 0 is greater than the confidence range which is 0.53. Hence states 1 and 3 cannot be merged. We then check for the equivalence between states 1 and 5, These are found to be equivalent. Before merging them we also find equivalences of the following pairs of states 7 and 1, 8 and 3, 10 and 6, and 11 and 9. After the corresponding merges are performed the string calculations described earlier are computed again. The resulting automaton is plotted in Figure 2.18.

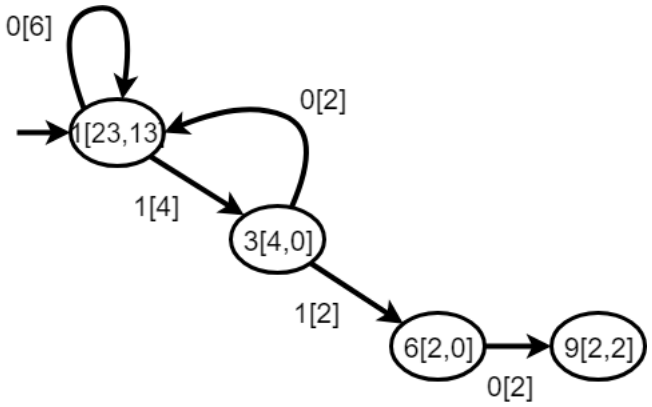


Figure 2.18: Automaton after merging states 5 to 1, 7 to 1, 8 to 3, 10 to 6 and 11 to 9

States 6 and 1 are are found not to be equivalent since the difference between their termination probabilities with label 0 is 0.75 which is larger than the confidence range of 0.61. Finally equivalence between states 6, 3 and 9 are checked and equivalence is satisfied. They are merged resulting in the automaton given in Figure 2.19. The algorithm stops when no more states are deemed equivalent. In state 1 there are 15 strings terminating and there are no strings terminating

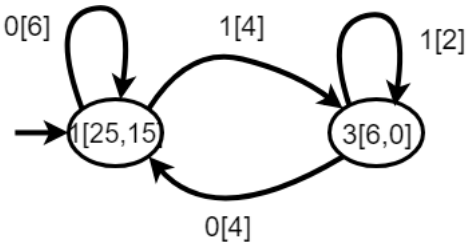


Figure 2.19: Automaton after merging states 6 to 3 and 9 to 3

in state 3. Hence state 1 is marked as the final state. To compute the transition probabilities we consider the numbers on the transitions for each state. For state 0 there are two outgoing transitions, a transition with symbol 0 and number 6 and a transition with symbol 1 and number 4. The probability of occurrence of symbol 0 is then $\frac{6}{6+4} = 0.6$. Similarly, the probability that symbol 1 occurs is $\frac{4}{6+4} = 0.4$. The other transition probabilities are calculated in a similar way and are shown in Figure 2.20.

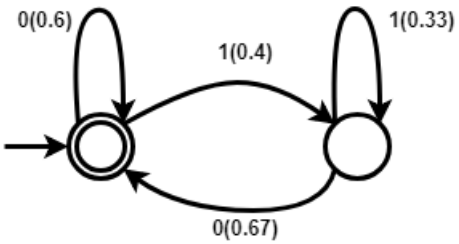


Figure 2.20: Final output PFA with probabilities

2.2 Process mining Algorithm

Process mining is a passive learning technique which infers models from traces. Process mining was initially developed as a tool set to discover processes from traces, primarily applied in the field of business process modeling. Van der Aalst [11] gives an overview and historical perspective of this field in his text book. Process mining is used to discover, monitor and improve real processes by extracting knowledge from traces, readily available in todays systems. There is a family of algorithms using ordering relations between events shown in traces to construct models. The commonly used algorithms in process mining are the alpha algorithm [11] and inductive mining, which is the most advanced process discovery algorithm. Inductive mining uses a divide-and-conquer approach to split the traces recursively into sub traces [11].

Process mining algorithms work only with positive traces. In industrial applications it is difficult to obtain negative traces. Process mining algorithms provide models in different formalisms like Petri nets, process trees and causal nets. In this thesis Petri net models are considered for comparison with state machine learning results.

Petri net formalism

Petri nets are among the oldest and best investigated process modeling language allowing for the modeling of concurrency [11]. A Petri net is a bipartite graph consisting of places and transitions. The state of a Petri net is determined by the distribution of tokens over places and is referred to as its marking. In the initial marking shown in Figure 2.21, there is only one token; place *start* is the only marked place. The square boxes are the transitions which are enabled by the tokens. The circular shapes denote places which can contain tokens. A transition is enabled if each of its input places contains a token. An enabled transition can fire by consuming one token from each input place and producing one token for each output place. The transition a is enabled when place *start* contains a token. The firing of a results in the production of two tokens placed in places $c1$ and $c2$. In this case one token is consumed and two tokens are produced. When the starting place has no tokens anymore, transition a is no longer enabled. However, transitions b, c, and d have become enabled. A token can be placed in $c3$ if any one of the transition b or c is fired. When $c1$ and $c2$ contain a token, firing b and d results in tokens being produced for $c3$ and $c4$ respectively. The transition e can only be enabled if both $c3$ and $c4$ contain a token. A token in place $c5$ can enable transitions g or h and reach the end state or can enable f and loop back to $c1$ and $c2$.

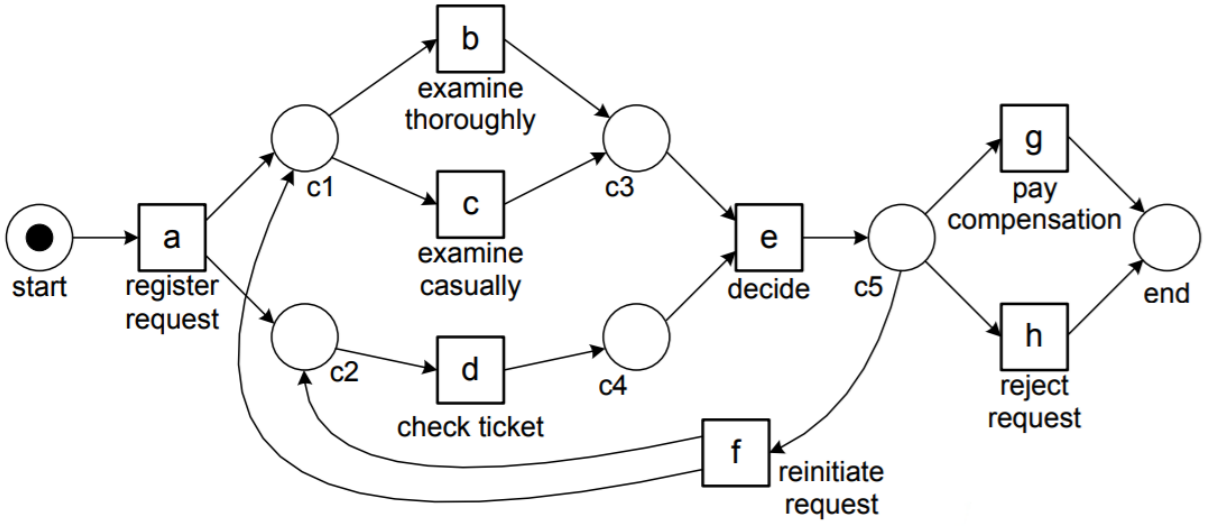


Figure 2.21: Petri-net model

The traces obtained from Petri net models can be used to understand how the system behaves.

In Table 2.1 the accepted sequences of events from the Petri net model is tabulated. In case of id 1, the accepted string is *abdeh* which in the Petri net model corresponds to the sequence of transitions <*register request, check ticket, decide, reject request*> from place *start* to place *end*. Also traces that are not accepted can be identified. Consider for example trace *abeg*. For this trace transition a has to be fired which is possible. Then transition b has to be fired, which is also possible. Then transition e has to be fired, which is not possible since d was not fired yet and $c4$ contains no token. Hence this execution does not result in a token in place *end*. ASML is exploring Petri net models to gain insights in the behavior of components. To this end they are exploring the ProM tools to learn Petri nets from execution logs. A large number of learning algorithms are integrated in ProM. Explaining these algorithms is beyond the scope of this thesis. We refer to [11] for an overview.

Case id	**Trace**
1	$a\ b\ d\ e\ h$
2	$a\ d\ c\ e\ g$
3	$a\ c\ d\ e\ f\ b\ d\ e\ g$
4	$a\ d\ b\ e\ h$
5	$a\ c\ d\ e\ f\ d\ c\ e\ f\ c\ d\ e\ h$
6	$a\ c\ d\ e\ g$
7	$a\ d\ c\ e\ f\ d\ c\ e\ f\ b\ d\ e\ h$

Table 2.1: Accepted traces in Petri net model

2.3 Flexfringe tool

Flexfringe is an open source passive automaton learning tool used for state machine learning. Flexfringe can learn different types of finite state automata, including deterministic and probabilistic finite state automata, mealy machines and register automata. The different parameters used to invoke the learning algorithms with their specific settings are discussed in the next section.

2.3.1 Flexfringe Parameters

The two main parameters are heuristic_name and data_name. They control which algorithm is used, which heuristic is used and the type of output that is generated. The parameter names, their corresponding algorithm, and the type of automata they learn are tabulated in Table 2.2

Algorithm	**Parameters**		**Automata type**
	heuristic_name	**data_name**	
Hankel algorithm	evidence_driven	edsm_data	DFA
Overlap driven algorithm	overlap_driven	overlap_data	DFA
Alergia algorithm	alergia	alergia_data	PFA

Table 2.2: Algorithms and their parameters in Flexfringe

Other settings which can be used for each algorithm are described below:

- **State_count** - The minimum number of arrivals of a string at a state for this state to be considered for state merging. They are used to ignore parts of the tree for which very few arrivals are available. This parameter is applicable in case of the overlap driven and Alergia algorithms.
- **Symbol_count** - The minimum number of traces that trigger a transition in a certain state, for this state to be considered for state merging. This parameter is applicable in case of the overlap driven and Alergia algorithms.

- **Lower_bound** - Lower_bound is used in the overlap driven algorithm to set the length of the string that has to follow each state, to determine state equivalence.
- **Sink** - Sink states are used to to visualize models in a compact way. They are used to group states with low string arrivals to form a single sink node. Sink functionality can be switched on or off by setting the Sink parameter to 1 respectively 0.
- **Sink_count** The Sink_count parameter is used to determine whether a state should be merged to a sink state. In case the number of arrival to a state is less than Sink_count it is merged and otherwise it is not.

All these parameters help to tune the outcome of the learner. The parameters used most often are the symbol and state thresholds, symbol_count and state_count parameters. Notice that the type of algorithm that can be used for learning depends on the input data available. For example, the Hankel algorithm requires both positive and negative data to effectively learn models, whereas Alergia algorithm and overlap driven algorithm can learn from positive data alone.

2.3.2 Input format of Flexfringe

Flexfringe uses traces in .txt format to learn models. The input format of traces in the text file is the Abbadingo format, from the grammar inference community used in competitions. The first line in the file contains the number of traces provided in the file and the total number of symbols occurring. Each subsequent line contains a label telling whether the trace is positive or negative, the length of that particular trace, and the subsequent symbols of the trace. The general format of the text file is as follows:

#No.of Traces #No.of Symbols
label length symbol1 ... symboln

An example trace text file is:
2 2
1 10 a b a b a b a b a b
0 15 a b a b a b a b a b a b a b a

This example has 2 traces and 2 symbols which are specified in the first line. In the second line, the first trace has length 10 and label 1 indicating that the trace forms a positive evidence. The second trace has length 15 and gives negative evidence.

2.3.3 Using flexfringe

The Flexfringe tool is based on the Linux platform and can be invoked via the command line. Assume we like to use the overlap driven algorithm with State_count 15, Symbol_count 25 where we switch the Sink option on with Sink_count 10 applied to file tracefile.txt. These options are specified as:

./flexfringe –heuristic_name overlap_driven –data_name overlap_data state_count 15 –symbol_count 25 –sink 1 –Sink_count 10 tracefile.txt

Chapter 3

Approach to compare model learning results

In Chapter 2 we discussed state machine learning, the various algorithms used in state machine learning and the settings available in the tool Flexfringe. Each learning algorithm results in different models. To compare these models, we like to know how close or far apart these models are from each other. In this chapter we define notions of closeness between models both in a qualitative and quantitative way. We introduce a tool chain which is used to compute closeness relations between models. The tool chain is used to compare the model learning results based on the defined closeness relations.

3.1 Qualitative notion of closeness

We define a qualitative notion of closeness between models based on language inclusion. Since language inclusion is a partial order, we can obtain a structure of models called the lattice giving insights in how close or far aways these models are from each other. By defining the partial order relation, we also obtain an equivalence relation between models in a natural way:

- **Partial order relation** - Assume we have three models named M1, M2 and M3. The languages of these models are L(M1), L(M2) and L(M3) respectively. We define a partial order on these models as follows. M1 $\sqsubseteq$ M2 if and only if L(M1) $\subseteq$ L(M2). In case M1 $\sqsubseteq$ M2 and M2 $\sqsubseteq$ M3 we know that model M1 is at least as far away from M3 as M2 is.
- **Equivalence relation** - Certain models may produce the same languages. For models M1 and M2 this is the case if M1 $\sqsubseteq$ M2 and M2 $\sqsubseteq$ M1. These models are called equivalent, written as M1 $\approx$ M2. When representing the partial order in a picture, equivalent models will be positioned at the same point in the lattice.

The Figure 3.1 shows a lattice having different state machine models with the same alphabet. The lattice has a flower automaton (an automaton which accepts every string with symbols from the alphabet) on top and an empty automaton (an automaton with an empty language) at the bottom. The lattice has state machine models from SM0 to SM18, where SM0 represents the empty automaton and SM18 represents the flower automaton. Since no two models are depicted at the same position in the lattice, we now that they are all different (non equivalent). Assume that SM5 is a 'golden reference' model, capturing the exact behavior of the system. Then we know that SM10 is closer to reference model than SM15 or SM16 are. Hence this lattice gives us qualitative insight in the closeness between these models.

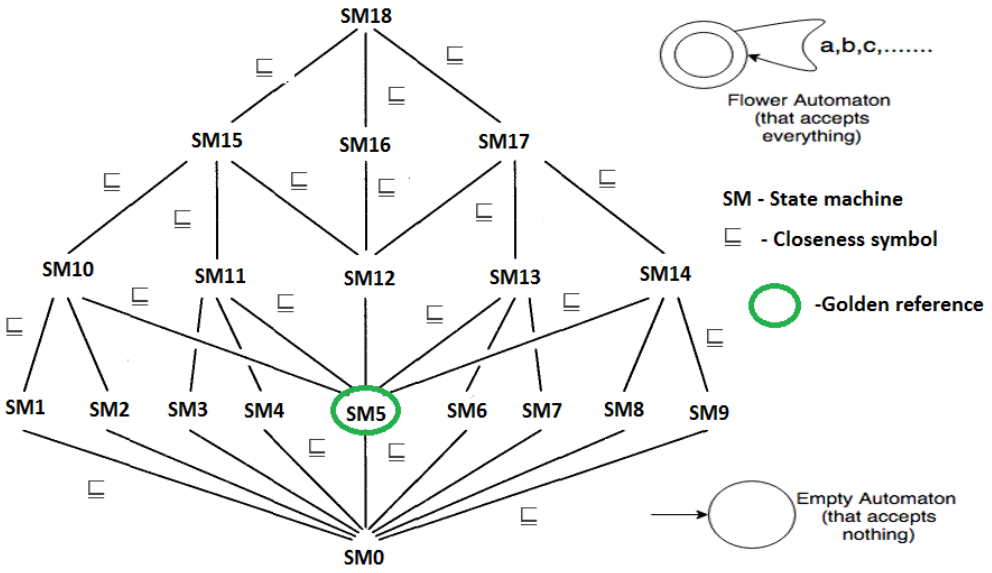


Figure 3.1: Lattice showing a qualitative notion of closeness

3.2 Quantitative notion of closeness

Qualitative notion of closeness between models gives us insights on where the models are placed in the lattice. However to understand the extend of difference between them, we also need a quantitative way to express such a difference. Several measures exist in literature which can quantify the differences between models. They are mainly classified based on two approaches:

- **In terms of their language** - strings of symbols that are in the language or not.
- **In terms of their structure** - the actual states and transitions that govern the behavior.

Language based measures concern precision, recall and distance scores between models. Structural measures are based on structural differences between models, some of which are implemented in LTSDiff [8]. The quantitative relations we use in this thesis are based on languages. This allows us to compare models of different types, namely state machines (obtained through state machine learning) and Petri nets (obtained via process mining). We will exploit the measures based on precision and recall as we will explain in the next subsection.

3.2.1 Language based distance measures

Languages of models can be infinite, making it impossible to exhaustively enumerate every string in the language. Hence in language based methods that quantify model distances we need to generate a suitable finite subset of the language. This can be done using model-based testing techniques [8]. The basic idea is to generate a finite number of strings from a reference model using a model-based testing technique and then replaying those strings on a subject model. Based on the number of strings that can be replayed on the subject model, a binary classifier measure is obtained. These measures are called *precision* and *recall*. To generate the finite set of traces from the reference model, we use the so-called W-method in this thesis.

3.2.2 W-method

The W-Method is a model-based test technique to generate a finite set of test traces that are representative of a given LTS.

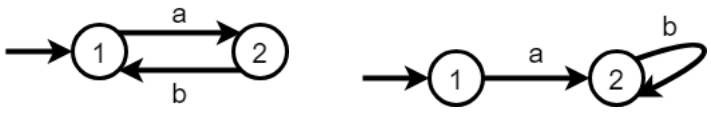
Definition of LTS
A deterministic Labelled Transition System (LTS) is a tuple (Q, Σ, δ, q_0), where Q is the set of states with q_0 as the initial state, Σ is an alphabet and δ is the next state function $\delta : Q \times \Sigma \rightarrow Q$. All sets are assumed to be finite. All states are acceptance states.

The language of an LTS A is a set of strings that are accepted by A. In other words, the language L, represented using an LTS A, accepts string $a_1 \cdots a_n \in \Sigma^*$, $\delta(q_0, a_1) = q_1$, $\delta(q_1, a_2) = q_2$, $\cdots$, $\delta(q_{n-1}, a_n) = q_n$ for some states $q_1, \cdots, q_n$.

The languages of two LTS's A and B can be compared by generating a test set from A, and then measuring the portion of test strings that are classified in the same way by A and B. To generate these test strings, we will use the W-method. The set of test traces constructed by the W-Method is the concatenation of three sets of strings, namely the State Cover Set, the Symbol Permutations Set and the Characterization Set. We will informally describe these sets. For the formal definition we refer to Walkinshaw's paper [8].

A State Cover Set C is a prefixed-closed set containing all the traces required to explore every state of the model from the initial state. This set of traces is used to be able to reach every state of the model. The subject model may however have 'hidden' states that do not exist in the reference model. For this reason the W-Method uses a parameter k denoting the number of potential extra states that can occur in the subject model. With this parameter the Symbols Permutations Set is defined as $\{\epsilon\} \cup \Sigma \cup ... \cup \Sigma^{k+1}$ where Σ denotes the alphabet of the model. By concatenating this set to the State Cover Set we obtain $C \cdot (\{\epsilon\} \cup \Sigma \cup ... \cup \Sigma^{k+1})$ which ensures that the test cases examine not only the expected states but also k extra states. However, the resulting test set can still be insufficient as it cannot ensure that all intended states are reached. It is possible that a state reached by a trace in the subject model is not the intentionally reached state by this trace in the reference model. To this end the W-method also concatenates a Characterization Set W. This set of strings is used to distinguish pairs of states that reach the intended state. By appending W we obtain set $C \cdot (\{\epsilon\} \cup \Sigma \cup ... \cup \Sigma^{k+1}) \cdot W$ which is well able to distinguish the differences between the reference model and the subject model.

We will provide a simple example to illustrate how the W-Method works. Assume that we have a reference LTS with two states and two symbols as depicted in Figure 3.2(a). The subject LTS is shown in Figure 3.2(b).



(a) Reference model (b) Subject model

Figure 3.2: Example for W-method

The State Cover Set for this reference model is $\{\epsilon, a\}$ containing strings that can reach both state 1 and state 2. Parameter k refers to the number of extra states in the subject LTS. Since the subject LTS has two states as well, k equals to zero. Concatenating the State Cover Set and the Symbol Permutations Set yields $\{\epsilon, a\} \cdot (\{\epsilon\} \cup \{a, b\}) = \{\epsilon, a, b, aa, ab\}$. The W-method classifies the generated traces into categories: those accepted by q model and those rejected by a model. The reference model and subject model have the same classification results for each trace in this set. Thus, it fails to detect the difference between the reference and subject models. Although input sequence ab reaches state 2 in the subject model, it does not reach the intended state 1 in the reference. This illustrates the reason why the Characterization Set is essential. The Characterization Set that distinguishes states 1 and 2 is $W = \{a, b\}$. By appending W set, we obtain the complete set of test strings $\{\epsilon, a, b, aa, ab\} \cdot \{a, b\} = \{\epsilon, a, b, aa, bb, ab, ba, aaa, aab, aba, abb\}$. String aba distinguishes these two models as it is classified differently.

3.2.3 Using Binary classifier performance measures to compare languages:

To compare the similarity or difference of two languages, the test set generated using the W-method is categorized in terms of a confusion matrix [7] (a table that separates out true/false positives/negatives). This is shown in Table 3.1. The set of true positives refers to the set of traces that are accepted by both the reference (R) and the subject (S) model. False positives are traces that are accepted by subject model but not by reference model. False negatives are traces that are accepted by the reference model but not by the subject model, and true negatives are traces that are not neither accepted by the subject model nor the reference model.

Classification by Reference model (R)	**Classification by Subject model (S)**	
	$\omega \in$ L(S)	$\omega \notin$ L(S)
$\omega \in$ L(R)	True Positive (TP)	False Negative (FN)
$\omega \notin$ L(R)	False Positive (FP)	True Negative (TN)

Table 3.1: Binary classification - Confusion matrix.

Using the confusion matrix the precision and recall measures can be computed for each pair of models. The precision is defined as the fraction of strings in L(S) that are also in L(R). The recall is the fraction of strings in L(R) that are also in L(S). The formulas for precision and recall are tabulated in Table 3.2. Precision and recall measures give us a quantitative notion of closeness between models.

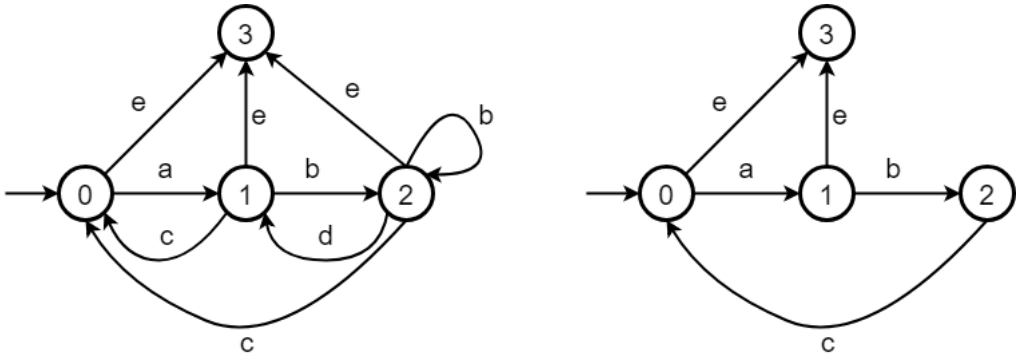
Measure	Formula	Interpretation
Precision	$\mid TP \mid / \mid TP \cup FP \mid$	Fraction of tests in L(S) that are in L(R)
Recall (Sensitivity)	$\mid TP \mid / \mid TP \cup FN \mid$	Fraction of tests in L(R) that are in L(S)

Table 3.2: Formulas to calculate precision and recall.

3.2.4 Example for quantitative analysis

In this section we present an example showing how to use precision and recall measures to quantify closeness between a reference and subject model.

Consider a reference model and a subject model, each with alphabet $\{a, b, c, d, e\}$ and 4 states. The reference model is depicted in Figure 3.3 (a) and the subject model is shown in Figure 3.3 (b).



(a) Reference LTS model (b) Subject LTS model

Figure 3.3: Reference and subject model for quantitative analysis

The W-method is used to obtain test traces from the reference model and these traces are played on the subject model to calculate the precision and recall measures. The w-method is applied to the reference model and 63 traces are generated. These traces are played on the subject model to construct the confusion matrix. The confusion matrix is given in Table 3.3. From this table the precision and recall measures can be calculated for the subject model.

Classification by Reference model (R)	**Classification by Subject model (S)**	
	$\omega \in$ L(S)	$\omega \notin$ L(S)
$\omega \in$ L(R)	True Positives: 2	False Negatives: 12
$\omega \notin$ L(R)	False Positives: 0	True Negatives: 49

Table 3.3: Confusion matrix based W-method

Parameter	**Score**
Precision	1
Recall	0.14

Table 3.4: Measures of subject model

The precision and recall measures are tabulated in Table 3.4. The high precision measure implies that all traces that are classified by the subject model as positive are also classified as positive by the reference model. In this case there are no false positives and the 2 traces that are classified as positive are accepted by both the reference and the subject model. Hence the precision value is 1. On the other hand, the low recall value (0.14) indicates that the subject model rejects (classifies as negative) a large number of traces that must be accepted (classified as positive) according to the language of the reference model. This is because twelve out of fourteen positive traces are classified by the subject model as negatives. Thus the precision and recall measures gives a quantitative notion of how close the subject model is to the reference model.

3.3 Tool Chain

In the previous sections we defined quantitative and qualitative notions of closeness between models. In this section we provide a tool chain to compare models. ASML earlier applied process mining techniques to the industrial case study described in Section 1.3 and these process mining

results are already available. These results have been obtained using one of the many mining settings. We will not alter these settings, as we focus on state machine learning. But since our notions of closeness are based on languages, our framework allows the comparison of the results obtained with process mining to the results obtained by alternative state-machine learning algorithms and settings.

The tool chain we developed therefore supports both process mining, resulting in Petri net models and state machine learning resulting in state machine models. An overview of the developed tool chain and the various steps involved are illustrated in Figure 3.4. The tool chain has three main parts: (1) process mining where models are obtained in the form of Petri nets, (2) state machine learning where models are obtained in the form of state machines and (3) translation and comparison using qualitative and quantitative methods.

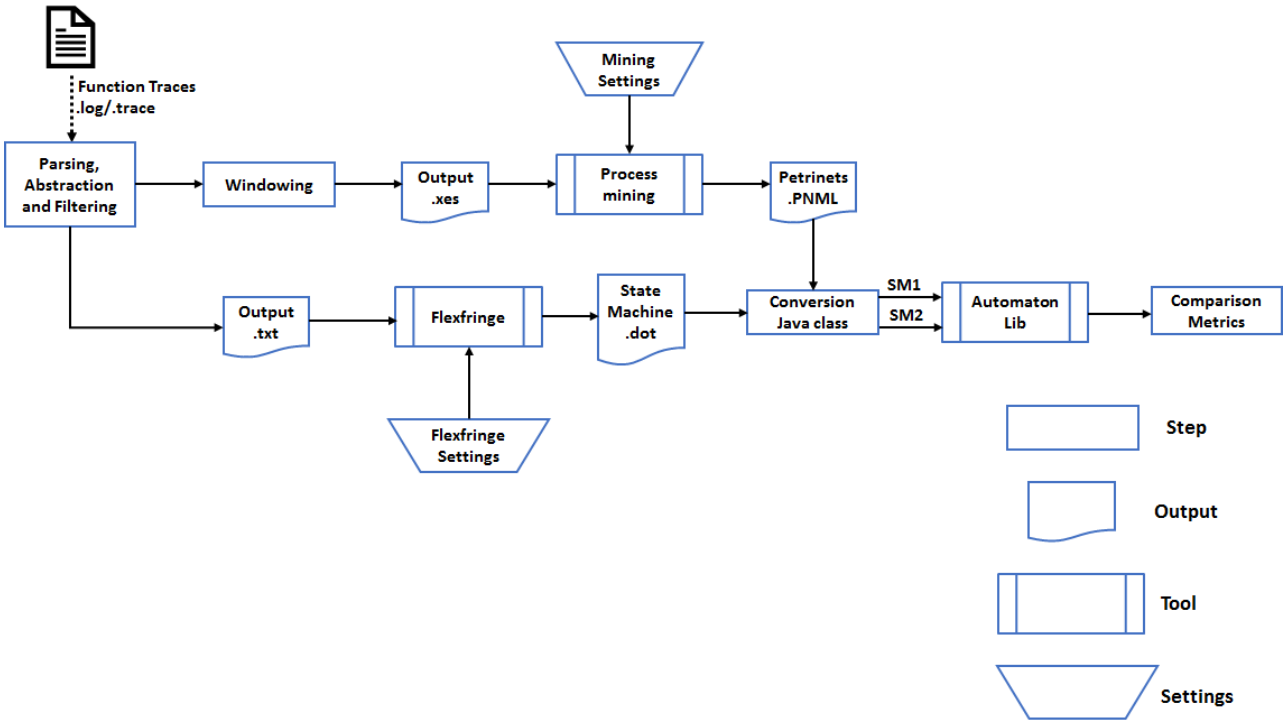


Figure 3.4: Overview diagram of implemented tool chain

3.3.1 Process mining

ASML has been applying process mining, a passive learning technique to obtain models from traces. Earlier experiments were performed using the ProM tool. Traces obtained from test runs run in ASML TWINSCAN machines are first pre-processed by parsing, abstraction, filtering and windowing. In the parsing step, the traces are parsed and the required parameters are extracted for each trace. In the filtering step, the traces with selected symbols are filtered and in the windowing step long traces are split into separate traces based on start and completion of symbols. These pre-processed traces are then converted into .xes files (an XML-based standard for event logs).

There are several process mining settings which are used to obtain models in different formalisms like causal nets, process trees and Petri nets. In our case study the models are expressed as Petri net models which are in the .PNML format (Petri net Markup Language is an XML-based interchange format for Petri nets).

3.3.2 State machine learning

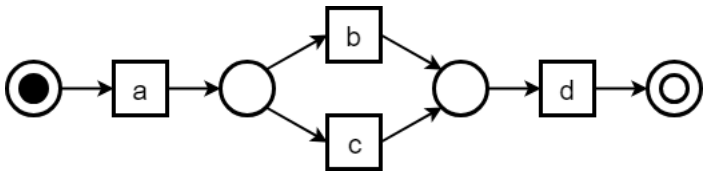
State machine learning algorithms are implemented in the Flexfringe tool. State machine learning algorithms require parsing, abstraction and filtering of traces. The pre-processed traces are converted to .txt files and are inputs to the Flexfringe tool. The input formats and the various settings used in Flexfringe tool were discussed earlier in Section 2.3. Flexfringe delivers state machines in the .dot file (a graph description language) format. In the following chapters we will use the tools chain to explore the different algorithms and settings to obtain alternative state machine models from the traces together with closeness measures.

3.3.3 Translation and comparison

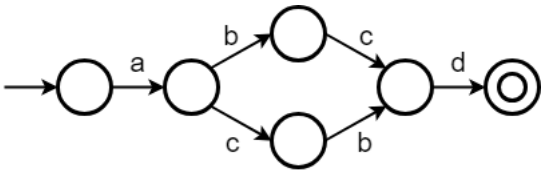
In order to compare models expressed in different formalisms they have to be converted to a common format and formalism to compare these models. Automata-lib is an open source library which can be used to compute relations on state machines. For instance, equivalence and subset relations can be computed in Automata-lib. However the results of process mining and state machine learning are in different formalisms. These results have to be translated to a common format in Java which can be used as input for the Automata-lib tool. The comparison of results of process mining and state machine learning are done based on subset and equivalence relations computed using this library. The precision and recall measures are computed using StateChum, a java framework which provides the implementation of W-method. The translation to Java classes and computing relations on state machines are described below.

Translation from Petri nets to state machines

Process mining results are expressed in Petri nets. Since automata-lib accepts Java classes describing state machine models, we have developed a translation. Petri nets are converted to state machines by preserving the language of the Petri nets as described in [11]. An example of such a translation is given in Figure 3.5. The example Petri net has 4 symbols. The model has a initial marking which indicates the start state. The square boxes are places and the circular ones are transitions. The translation maps places and transitions in the Petri net to states and transitions in the state machine respectively.



(a) Example Petri net model



(b) Translated state machine model

Figure 3.5: Translation example of Petri nets to state machines

Computation of relations in Automaton lib

Once the models are converted to a common format they can be used for comparison. For qualitative comparison language subset and equivalence relations have to be computed. Given two state

machine models S1 and S2. The language represented by the state machine models are L(S1) and L(S2). The formulas used in Automata-lib to obtain pre-order and equivalence relations on state machines are discussed below.

Pre-order relation

Given two state machine models S1 and S2, we know that the language accepted by model S1 is a subset of the language accepted by model S2 if and only if the intersection of the language of S1 and the complement of the language of S2 is empty. Thus

$$\text{L(S1)} \subseteq \text{L(S2)} \text{ iff } \text{L(S1)} \cap \overline{L(S2)} = \emptyset.$$

This translates directly to the corresponding pre-order on state-machines:

$$\text{S1} \sqsubseteq \text{S2} \text{ iff } \text{L(S1} \cap \overline{S2}) = \emptyset.$$

We can thus compute this relation using the complement and intersection operators on state machines. These operators are offered by Automata-lib.

Equivalence relation

The languages of state machine models S1 and S2 are L(S1) and L(S2) respectively. The languages are equivalent if they satisfy the below relations.

$$\text{L(S1)} = \text{L(S2)} \text{ iff } \text{L(S1)} \cap \overline{L(S2)} = \emptyset \text{ and } \text{L(S2)} \cap \overline{L(S1)} = \emptyset.$$

This translates directly to the corresponding equivalence on state-machines:

$$\text{S1} \approx \text{S2} \text{ iff } \text{L(S1} \cap \overline{S2)} = \emptyset \text{ and } \text{L(S2} \cap \overline{S1)} = \emptyset$$

Again this equivalence relation can be computed using the intersection and complement operators offered by Automata-lib.

In our framework we compute these relations for each pair of models, which are then displayed in the lattice as shown earlier in Figure 3.1. Then the precision and recall measures are computed based on a finite set of traces produced by the W-method.

There are several learning techniques to obtain models and each learning technique may yield models in different formalisms. In order to have a common platform to compare these techniques the tool chain can be adapted to learn and compare models obtained via these different techniques.

Chapter 4

Guidelines to apply State Machine Learning

In the previous chapter we explained our approach and tool chain to learn models and to compare them based on the defined notions of closeness. In this chapter we use the tool chain and also the insights gathered from literature to formulate a set of guidelines to effectively use state machine learning. We also validate these guidelines based on experiments on synthetic examples. These guidelines provide insights and are helpful to exploit the available data in the best possible way to learn models.

4.1 Guideline to select appropriate algorithm based on available data

State machine learning algorithms learn models based on the input data available. The data can contain positive and negative traces. The algorithm to use depends on the types (positive and/or negative) of traces available. For each of the SML learning, the required trace type and the resulting model type are tabulated in Table 4.1 and explained below.

- **Hankel algorithm** - The Hankel algorithm requires positive and negative traces to effectively learn models. The evidence in negative traces are used to differentiate the states in the Hankel matrix. The absence of evidence from negative traces in the Hankel algorithm yields a flower automaton (an automaton which accepts all strings with symbols in the alphabet). In theory this is because the flower automaton is the smallest automaton that is consistent with the given positive traces. But also in practice a flower automaton will always be constructed (even though the coloring problem is NP hard). The reason is that the Hankel matrix then only contains either empty cells or cells with value 1, implying that all rows can be merged together, yielding a single state. In case sufficient positive and negative evidence is available, the Hankel algorithms can learn a state machine that is equivalent to the reference model (if it exists) [5].

- **Overlap driven algorithm** - The overlap driven algorithm learns models from positive traces and cannot make use of negative traces. The overlap driven algorithm does not always produce flower automata because states with different outgoing transitions will never be merged (at least if the Lower_bound > 0). No fundamental results are established in literature concerning the learning of a model that is equivalent to the reference model.

- **Alergia algorithm** - The Alergia algorithm learns models from positive traces and cannot deal with negative traces. To compensate for absence of negative information, it uses probabilistic information. In fact it assumes the system under learning to behave as a probabilistic

automaton. In case sufficient information is available, Alergia can learn a probabilistic finite state machine that is equivalent to the reference model (if it exists) [3].

Algorithm	Trace type (positive or negative)	Model inferred
Hankel	Requires both positive and negative traces	DFAs
Overlap driven	Requires positive trace only.	DFAs
Alergia	Requires positive trace only.	PFAs

Table 4.1: Model inferred and its trace requirements for each algorithm

4.2 Guideline to use the Hankel algorithm for model learning

The Hankel algorithm can obtain models that are equivalent to a reference model by exploiting both positive and negative traces. In this section we will show using an example that this is indeed the case. We will further explain the impact of adding positive and negative information on the ordering of corresponding models in the lattice.

Experiment 1

In this experiment the importance of evidence from negative traces for the Hankel algorithm is shown. Let us consider the reference model R in Figure 4.1. Assume we generate the set of positive traces $S^+ = \{$ ab, abab, abababab $\}$ from the given reference model R and apply the Hankel algorithm using these positive traces alone. The algorithm then produces a flower automaton F (as explained before) given in Figure 4.2(a). The flower automaton is consistent with the given traces but is an overapproximation of R, i.e. $R \sqsubseteq F$ but not $R \approx F$.

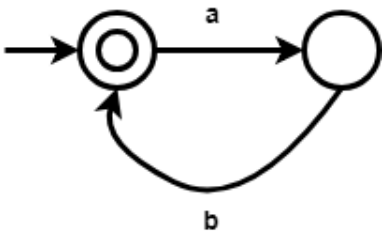
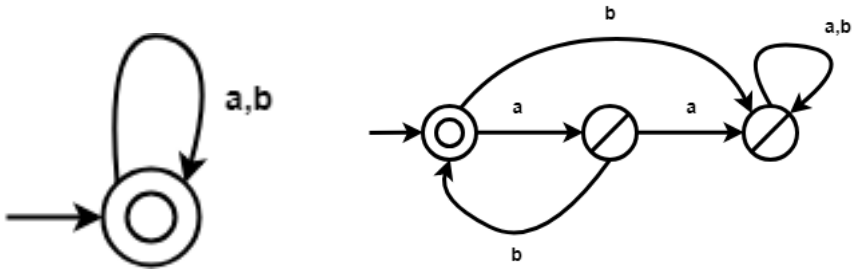


Figure 4.1: Reference model R

Now assume we add negative traces $S^- = \{$a, b, aa, ba, bb, aab, bab, bbb $\}$ and apply the Hankel algorithm again. The model M inferred is shown in Figure 4.2(b). The inferred model is consistent with the given positive and negative trace sets and is even equivalent to the reference model, so $M \approx R$.

When we compute the precision and recall measures of F and M with respect to R we find that both F and M recall value 1 which is consistent with the fact that $R \sqsubseteq F$ and $R \sqsubseteq M$. The precision of F equals 0.2 which together with the fact that $R \sqsubseteq F$ implies that F over approximates R in a severe way. The precision of M is 1, which is consistent with the fact $M \approx R$.



(a) Inferred model F (b) Inferred model M

Figure 4.2: Inferred model F with positive traces and inferred model M with both positive and negative traces

Guideline

In the experiment above the complete set of positive and negative traces were used at once to learn model M. In this case we could compare the result to the reference model R. In general this reference model is not available, implying that one cannot decide whether sufficient positive and negative information is available. To still get information about the sufficiency of the supplied evidence we could create a set of models, based on an increasing amount of evidence, and position them in a lattice. The idea is to alternate the addition of positive and negative traces to the evidence. Each time a positive trace is added a new model N is learned which over approximates the previous model P, i.e. $P \sqsubseteq N$. Each time a negative trace is added a new model N is learned which under approximates P, i.e. $N \sqsubseteq P$. Using the recall measure we can compute the rise/drop in the amount of over/under approximation. If the rise/drop is below a certain bound we could assume to have sufficient evidence.

4.3 Guideline to use Alergia algorithm for model learning

The Alergia algorithm can learn a probabilistic finite state machine that is equivalent to the reference model (if it exists) if sufficient information is available and if the system under learning behaves as a probabilistic automaton. In order to validate this we setup an experiment with a reference model and generated positive traces for such a probabilistic automaton.

Experiment 2

The goal of this experiment is to apply the Alergia algorithm to a set of positive traces obtained from a reference model to learn about the amount of data required to learn a model which is equivalent to the reference model. We consider an example PFA with 4 states and 6 symbols $\{a, b, c, d, e, f\}$. The state machine is depicted in Figure 4.3. In order to obtain traces, the reference model PFA was modeled using the Parallel object-oriented specification language (POOSL) [12] and tool [10]. The traces were generated probabilistically in correspondence to the PFA in the figure. This POOSL model is configurable with respect to the number of traces it generates. Also an upperbound and lowerbound of the trace length is configurable. The length of a trace is determined in a homoegenous way between this lower and upperbound.

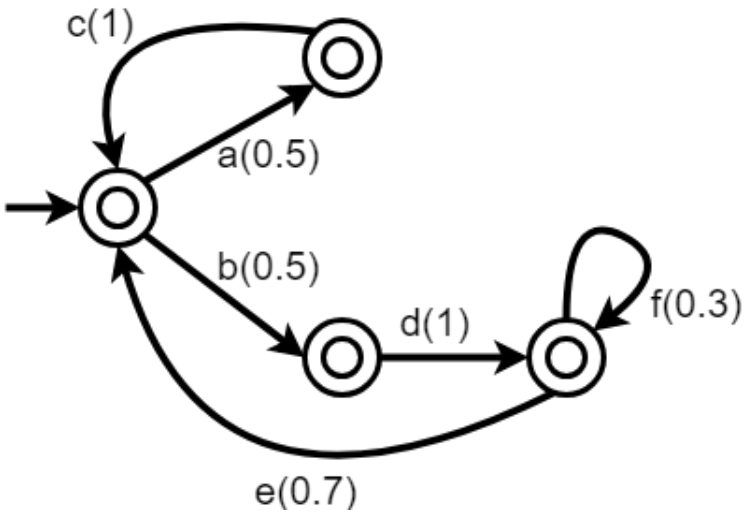


Figure 4.3: Reference model PFA

In this experiment, the trace length was set to a constant 10 (lowerbound and upperbound set to 10) and we varied the number of traces. Trace set of sizes 50, 100, 150, 300 and 500 were generated and offered as input to the Flexfringe tool. The number of states obtained and model convergence results are tabulated in Table 4.2. The algorithm produces a model that is equivalent to the reference model after using 150 traces of length 10. This experiment shows that the Alergia algorithm can learn a model which is equivalent to the reference model if sufficient data is available and if the system under learning behaves as a probabilistic automaton.

Number of traces	**Length of each trace**	**Number of states**	**equivalence**
50	10	8	No
100	10	6	No
150	10	4	Yes
300	10	4	Yes
500	10	4	Yes

Table 4.2: Convergence of Alergia algorithm

Experiment 3

In this experiment the the length of traces is varied to learn a model which is equivalent to the reference model. The number of traces is kept as constant at 100 and the length of the trace is varied from 6 to 25. The results of convergence of Alergia algorithm with varying trace lengths is tabulated in Table 4.3. The algorithm converges and produces a model that is equivalent to the reference model after increasing the length of the trace to 15. This indicates that if several long traces are available Alergia can converge to the reference model with fewer traces.

Number of traces	Length of each trace	Number of states	Model convergence
100	6	8	No
100	10	7	No
100	15	4	Yes
100	20	4	Yes
100	25	4	Yes

Table 4.3: Effect of trace length in convergence of Alergia

Guideline

The Alergia algorithm does not use explicit negative evidence, but can compensate for this by using probabilistic information. The downside of this is that compared to the Hankel approach, a relatively large amount of information is required. This implies that we either should provide many short traces or several long ones. Needless to say, this techniques is only effective if the system under learning has an underlying probabilistic execution mechanism.

To identify if the number of traces is sufficient to learn a model. We could use the same experiment as mentioned in the Hankel algorithm. The idea is to learn with increasingly large sets (also subset wise) and then compute between two iterations the quantitative measures between learned results. If this measure is getting smaller than a certain bound, we can conclude that we have sufficient information to learn a model otherwise we need to add more information.

4.4 Guideline to learn models from a single long trace

Overlap driven algorithms can obtain models from a single long trace. Neither the Hankel technique nor Alergia do so. As described earlier the Hankel algorithm requires a set of positive and negative traces to learn models, while the Alergia algorithm requires a large number of traces. We set up an experiment to use Overlap driven algorithm to obtain a model from a single long trace.

Experiment 4

The goal of this experiment is to show that the overlap driven algorithm can learn models from a single long trace. We generated a single long trace of length 100 from the reference model given in Figure 4.4. We set the lower_bound parameter to 2. For this example, the overlap driven algorithm learns a model that is equivalent to the reference model. Overlap driven algorithm can exploit the repetition of symbols to identify recursions and learn models from a single long trace. If we would have applied the Hankel approach, we would obtain the flower automaton with symbols a, b and c. In case of Alergia, we arrive at a PTA with one long branch.

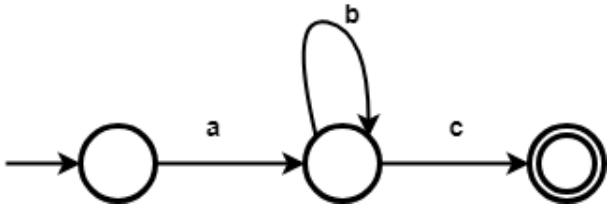


Figure 4.4: Reference model

Guideline

In case we have a single long trace it is suggested to use the overlap driven algorithm and tune the parameter lower_bound to identify recursions. The resulting models can be positioned in

a lattice including a reference model representing the *observed* traces. This lattice should be completed using the infimum and supremum operator, see also Section 4.6. For each model, the precision and recall measures relative to this reference model, Understandability of a model can be determined based on its number of states, which can be annotated in the lattice. Using this information, a model can be selected which is as close as possible to the reference, while still being understandable. Notice that a trade-off between understandability and closeness might have to be made. For example, the flower automaton is the most easy to understand abstraction of the reference, but it might be a far distance away from it.

4.5 Guidelines to tune parameters of state machine learning

The Flexfringe tool provides algorithmic settings to obtain different models. Many parameters control which states are considered for state merging. These are described below.

- **State_count** - The minimum number of arrivals of a string at a state for this state to be considered for state merging. State count is used to ignore states for state merging which are less frequent as they will have an undesired influence on states considered for state merging. When there are a number of traces available and if the traces have states which are reached rarely it is advised to ignore such states for state merging. Hence it is suggested to set the value of State_count to 10 [2]. For cases where the number of traces are limited it is suggested to set it to lower values.
- **Symbol_count** - The minimum number of traces that trigger a transition in a certain state, for this state to be considered for state merging. It is advised to set a value of 4 when there are symbols which occur rarely in a trace [2]. Symbols that have such a low occurrence in the traces have a negative impact on the merging of states and may lead to more complex state machines. Hence ignoring such symbols might helps in obtaining understandable models.
- **Lower_bound** - Lower_bound is used in overlap_driven algorithm to set the number of symbols that follows each state, which has to be the same for two states to be merged. The algorithm merges more and generalizes the model when the Lower_bound values are less. The language represented by the inferred model is bigger when the Lower_bound value is 1 as it merges two states which have one outgoing symbol the same. Hence to obtain smaller models it is suggested to set the Lower_bound value to 1 [2].
- **Sink** - Sink states are mainly used to obtain clear and concise models. In models which have a lot of concurrency, it is difficult to analyze the models. Setting sink parameter to 1 helps to get concise and easier to visualize models [2].

4.6 Guidelines to obtain understandable models close to a reference

Assume we have a set of observed traces from the system under learning. For each of these traces, any prefix will also be a trace form the system under learning. One can therefore take the prefix closure of this set and consider this set of traces to be a reference model R. We would like to find a model that is as close as possible to this reference model, but is yet understandable. The approach developed in this thesis gives us a systematic way to find this model.

The idea is to learn from the observed set of traces a number of different models by

1. applying different state machine learning techniques;
2. varying the algorithmic parameters as described in Subsection 4.5;
3. vary the input to the algorithms by taking subsets of the observed traces as described in Sections 4.2 and 4.3.

The partial order relation $\sqsubseteq$ is then used to order these models and position them in a lattice, including the flower automaton representing the top $\top$ of the lattice and the empty automaton representing the bottom $\bot$ of the lattice. Some of these models will be over approximations of R, some will be under approximations, but there can also be models that are unrelated to R. The figure 4.5 shows a lattice with top and a bottom, with models $M1$ and $M2$ which are over approximations of R.

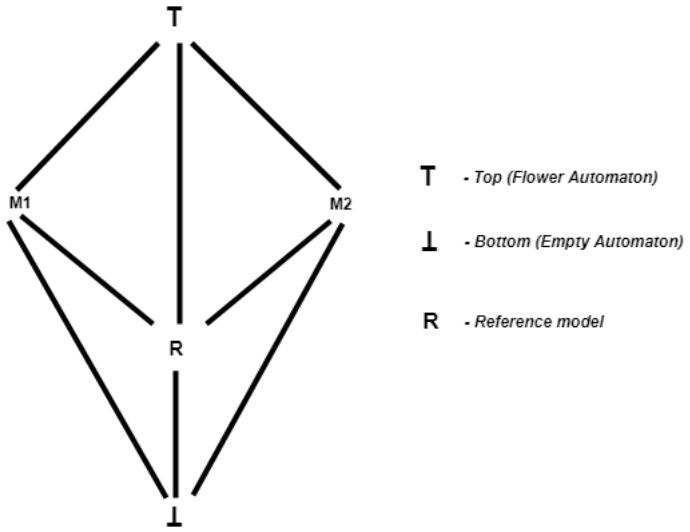


Figure 4.5: A lattice with top and bottom

The models that are now in the lattice can be used to compute *new* models in this lattice by using the supremum operator $\sqcup$ and infimum operator $\sqcap$. If M_1 and M_2 are models then $M_1 \sqcup M_2$ is such that $L(M_1 \sqcup M_2) = L(M_1) \cup L(M_2)$ and $M_1 \sqcap M_2$ is such that $L(M_1 \sqcap M_2) = L(M_1) \cap L(M_2)$. Using these operators we can make the lattice complete.

Assume we have two over approximations M_1 and M_2 of R which are unrelated, i.e. $M_1 \not\sqsubseteq M_2$ and $M_2 \not\sqsubseteq M_1$. Then we know that $R \sqsubseteq M_1 \sqcap M_2$ still holds, but that $M_1 \sqcap M_2$ will be closer (not further away) to R than M_1 and M_2. The new obtained models are placed in the lattice in Figure 4.6.

Vice versa if we have two unrelated under approximations M_3 and M_4 of R, then we now that $M_3 \sqcup M_4$ is still an under approximation of R (i.e. $M_3 \sqcup M_4 \sqsubseteq R$) which is closer (not further away) to R than M_3 and M_4. These models helps to complete the lattice in Figure 4.6.

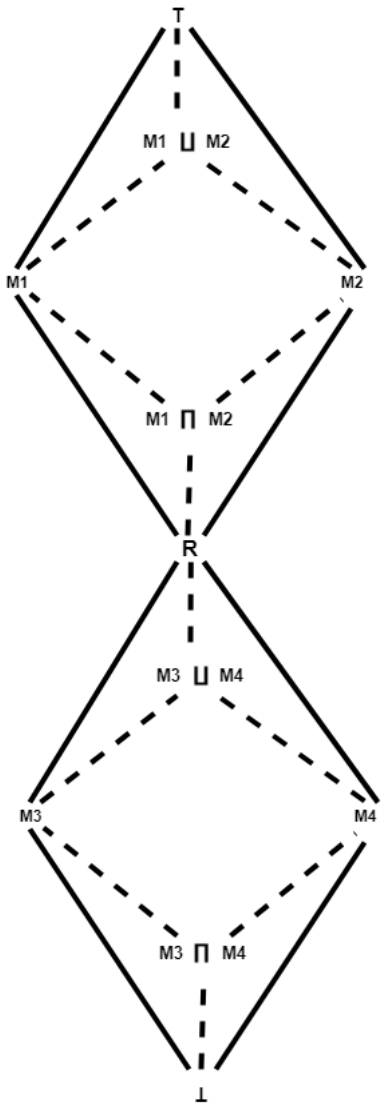


Figure 4.6: Lattice constructed using infimum and supremum operators

We can now decorate the lattice with precision and recall measures. For each model M for which $R \sqsubseteq M$ we have a (theoretical) recall measure of 1. The precision measure will be between 0 and 1. If M is a recursive model, its language will be infinite, implying that the precision will be (theoretically) 0 (since R has a finite language). Vice versa for each model M for which $M \sqsubseteq R$ we have a (theoretic) precision measure of 1 and a recall measure between 0 and 1. If M is recursive the recall value will be 0.

Finally we notate the lattice with model complexity measures in terms of the number of states of a model. The parameters that are used to annotate the lattice are given in the Figure 4.7. Using the decorated lattice we can select a model which is close to the reference, while still having a reasonable number of states. Notice that the selection can either be an over approximation or under approximation of R or might even be unrelated to R.

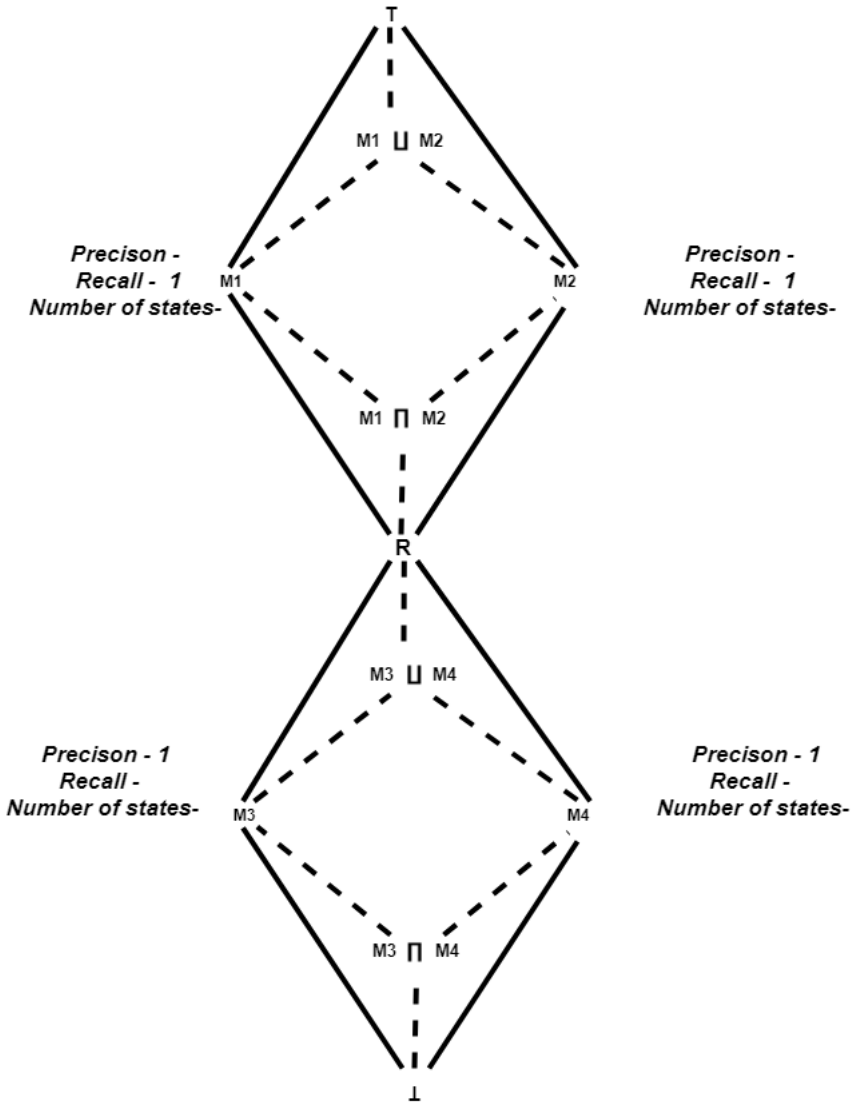


Figure 4.7: Lattice annotated with precision, recall and number of states.

Chapter 5

ASML Case studies

In this chapter we apply state machine learning to an industrial case study from ASML. We apply the various guidelines suggested in Chapter 4 to infer models. The reference model for these case studies are currently not available and ASML is working on them. However we have the process mining results of the case study in Petri net formalism. We apply the state machine learning techniques to obtain understandable models using the guidelines and also compare them with the results of process mining using our tool chain. The case study from ASML has two components A and B which implement the common interface protocol(CI). The components A and B have several interfaces out of which two interfaces, Swap and Swap Advanced (Swap Adv) are the same. In this case study we approximate the interface behavior of these two interfaces Swap and Swap Advanced (Swap Adv). An illustration of the interfaces and components are given in Figure 5.1.

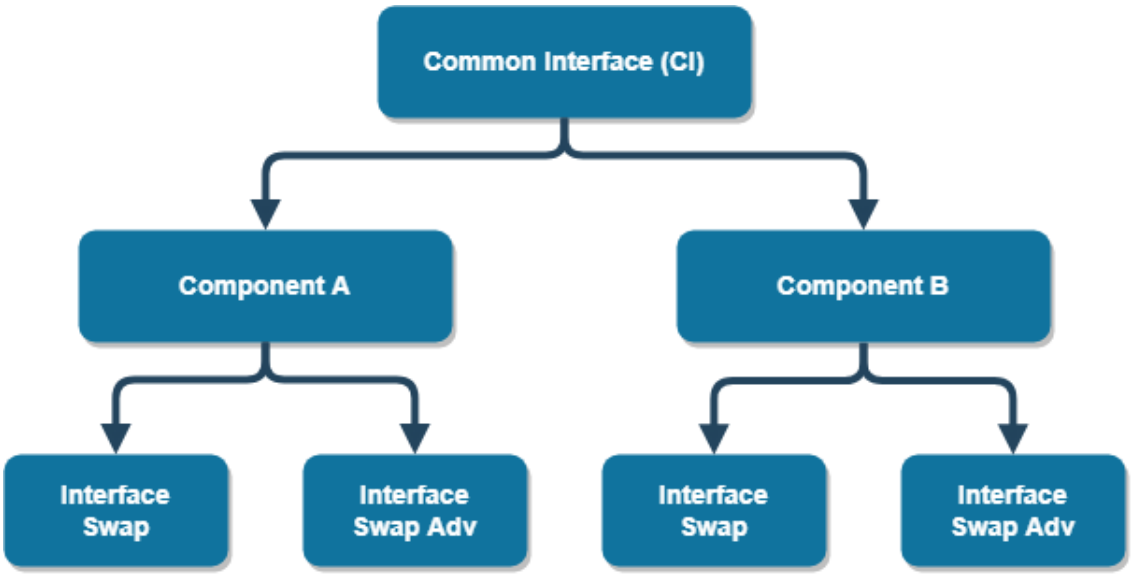


Figure 5.1: Various interfaces and components of the ASML case study

Traces are obtained from these components by running production and component tests. Production tests are run in the production environment and component tests are tests which are run with specific components enabled or disabled. The components A and B have 4 logs each resulting in 8 case studies. An illustration of the case study with the interfaces, the type of test conducted and the resulting trace files for component A is given in Figure 5.2. The same applies for component B as well. The list of log files obtained from ASML are listed below

1. Component A - Swap production test.

2. Component A - Swap component test.
3. Component A - Swap Advanced production test.
4. Component A - Swap Advanced component test.
5. Component B - Swap production test.
6. Component B - Swap component test.
7. Component B - Swap Advanced production test.
8. Component B - Swap Advanced component test.

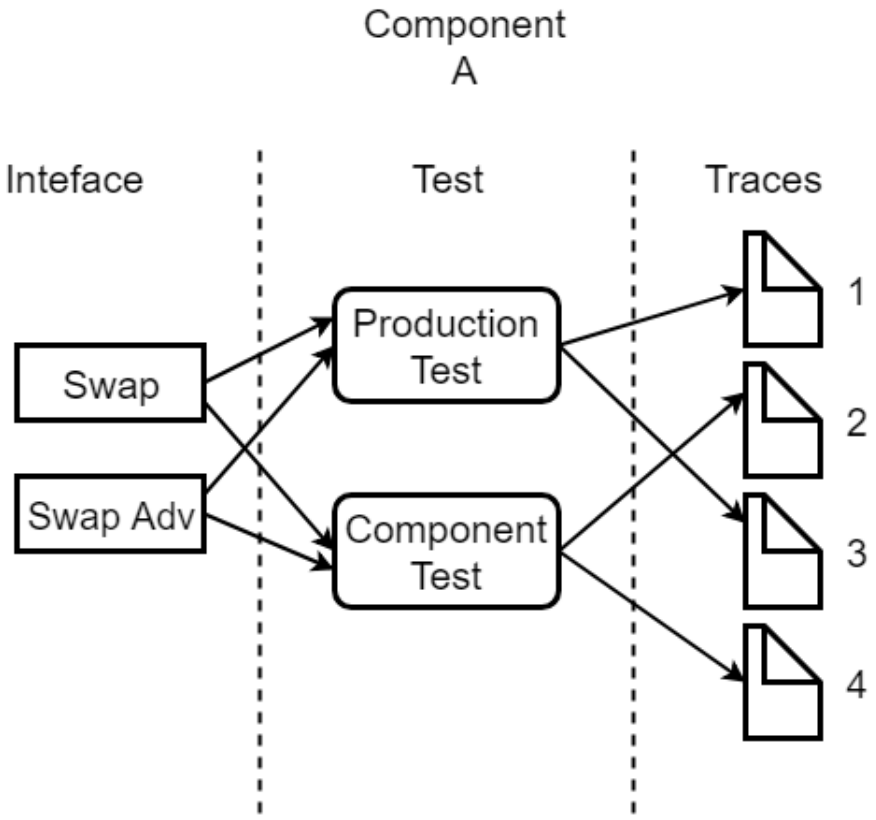


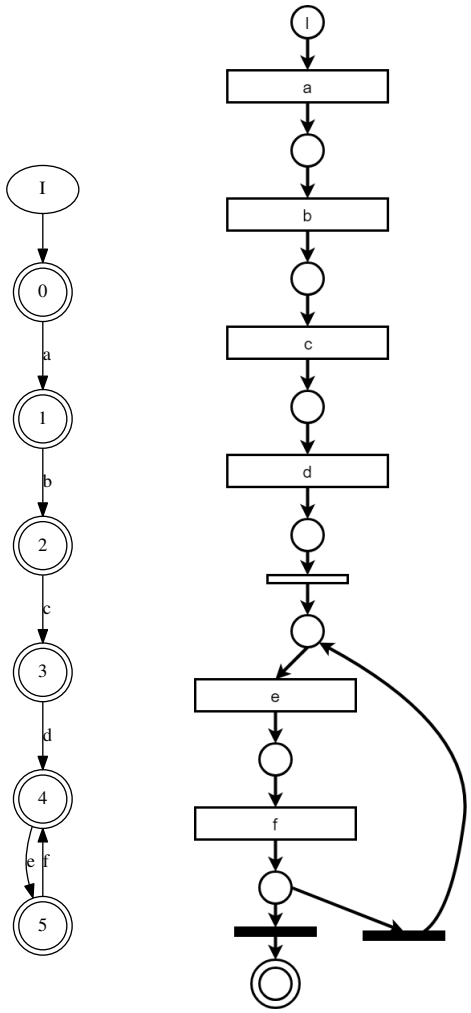
Figure 5.2: Tests and trace files for component A

The function calls (symbols in SML) in each trace file are converted to symbols with different names for confidentiality reasons. The traces in the ASML case study have limited number of positive traces only. Hence based on our guideline we decide to use the Overlap driven algorithm on all the case studies with different settings to learn models. The given logs contain traces which are obtained from the ASML machine and hence all these function calls in the traces are accepted by the machine. Hence these function calls which are actually symbols, should be accepted by the automata we learn. We use prefix closed traces for all the case studies indicating that all symbols are accepted in the automata we learn. The traces from each case study is applied in the tool chain to obtain models. The log files from ASML are converted into text files which have the traces in Flexfringe format discussed in Subsection 2.3.2. In the next section the models obtained for the various case studies are discussed.

5.1 Case study results

1. **Component A - Swap production test**

   The trace file of component A with interface Swap having undergone the production test has 6 symbols. The symbols are {a, b, c, d, e, f}. The trace file which is the input to Flexfringe has 16 positive traces. The symbols in the trace file follow a set pattern in which symbols e and f are repeated continuously forming a loop after the execution of event d. As the trace file follows this set pattern we apply the Overlap driven algorithm with settings lower_bound = 2 to obtain the state machine model M in Figure 5.3 (a). The Petri net model P for this case study is given in Figure 5.3 (b). To compare the SML result of this case study with the Petri net model, we import both these models in our tool chain. In this case the language represented by the Petri net and the state machine are the same. The equivalence relation is satisfied which means $M \approx P$ and when the lattice is constructed both these models lie in the same point.



(a) State machine model M

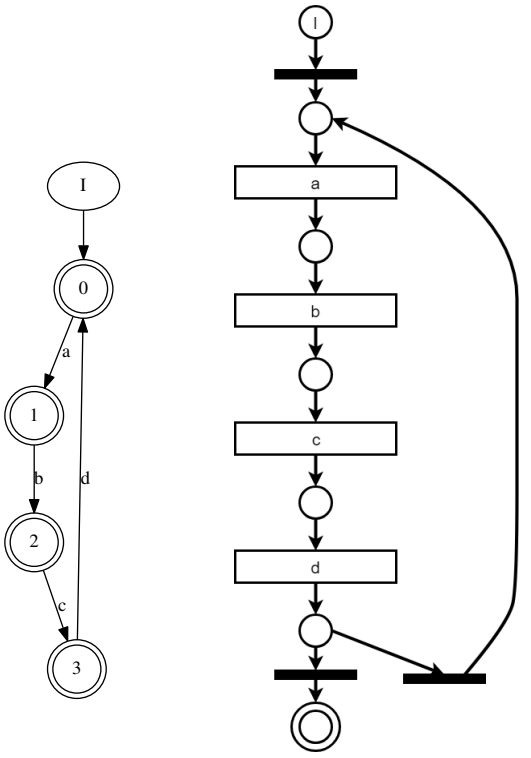
(b) Process mining - Petri net model P

Figure 5.3: Obtained state machine and Petri net model for Component A Swap production test

2. **Component A - Swap component test**

The trace file of component A with interface Swap having undergone the component test has 4 symbols. The symbols used are {a, b, c, d}. The trace file which is the input to Flexfringe has 13 positive traces. The string *abcd* is repeated continuously in the trace file. To identify recursion, Overlap driven algorithm is used with the settings lower_bound = 4 to obtain the state machine model M in Figure 5.4 (a). The Petri net model P for this case study is given in Figure 5.4 (b). To compare the SML result of this case study with the Petri net model, we import both these models in our tool chain. In this case the language represented by the Petri net and the state machine models are the same. The equivalence relation is satisfied which means $M \approx P$ and when the lattice is constructed both these models lie in the same point.

Observation - *Analyzing the state machine models of component A's production and component test we see that the component test does not have the symbols e and f. The rest of the events happen in the same order for both these tests.*



(a) State machine model M

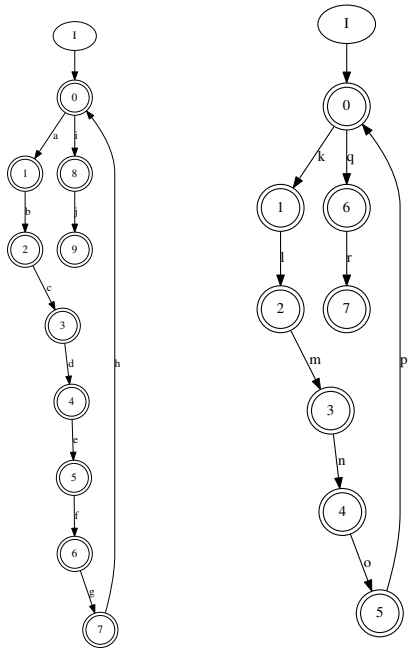
(b) Process mining - Petri net model P

Figure 5.4: Obtained state machine and Petri net model for Component A Swap component test

3. **Component A - Swap Advanced production test**

   The trace file of component A with interface Swap Advanced having undergone the production test has 18 symbols. The symbols are {a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r}. However we noticed that the production log is composed of two different swap events occurring in an interleaved manner. So ASML had split the production test trace further into two sub traces based on the start and stop events of the swap. This split traces are named swaplow and swaphigh which has 10 {a, b, c, d, e, f, g, h, i, j} and 8 {k, l, m, n, o, p, q, r} symbols respectively. The traces for swaplow and swap high are a single long trace. The overlap driven algorithm was used with lower_bound = 2 to obtain models. The models for both swaplow and swaphigh are given in Figure 5.5. The Petri net results for swaplow

and swaphigh are also available. These results were compared in the tool chain and the models were found to be equivalent. Thus by splitting the entire log based on the occurrence of swap events , helps in obtaining insights on the interface behavior of component A.



(a) State machine learning result for swaplow

(b) State machine learning result for swaphigh

Figure 5.5: Obtained state machine models for Component A advanced production test

4. **Component A - Swap Advanced component test**

   The log file for component A with Swap Advanced interface which has undergone component test has 20 symbols. The trace file used as input for SML has about 105 traces. As the number of symbols is more and there are lot of concurrent events. The Overlap driven algorithm is used to obtain understandable models.

   The various settings used to obtain models are given in Table 5.1. All these models are unrelated and hence placed at different points in the lattice. Hence we apply quantitative methods to identify a model in the lattice which is a lesser approximation of the behavior in traces. The models obtained are consistent with the traces but are over approximations of the trace behavior. As there is no reference model there are two ways to investigate the over approximation results of these models. (1) to investigate the merges in the model that creates a self loop leading to over approximation. (2) to use the PTA to generate traces . We use approach (1) to identify the over approximation in the models. We see that SML1 which is the most over approximated model is close to the flower automaton in the lattice. SML3 has the least over approximation and is placed at the bottom. Model SML2 is placed between SML1 and SML3 in the lattice. The constructed lattice is given in Figure 5.6.

Model	Algorithm	Parameter 1 State_count	Parameter 2 Symbol_count	Parameter 3 Lower_bound	Number of states in model
SML 1	Overlap driven	10	4	1	13
SML 2	Overlap driven	10	4	2	24
SML 3	Overlap driven	10	4	3	30

Table 5.1: Algorithms and settings used to obtain understandable models for Component A - Swap Advanced component test

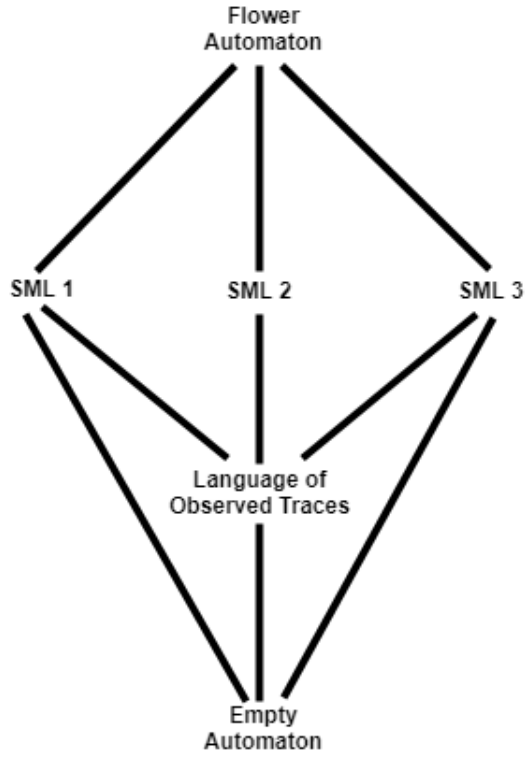


Figure 5.6: Lattice of models constructed for Component A - Swap Advanced component test

The model SML1 has the least number of states , which also explains the over approximation behavior as overlap driven algorithm merges more states. This model is considered to be more understandable because of the lesser number of states. However this model in Figure 5.7 has a huge number of transitions making it difficult to understand. The numbers in each state denoted with an # is the number of times a symbol has entered that state. The numbers denoted along with the symbols , for example $a : 81$ is the symbol count of a that arrives to the corresponding state.

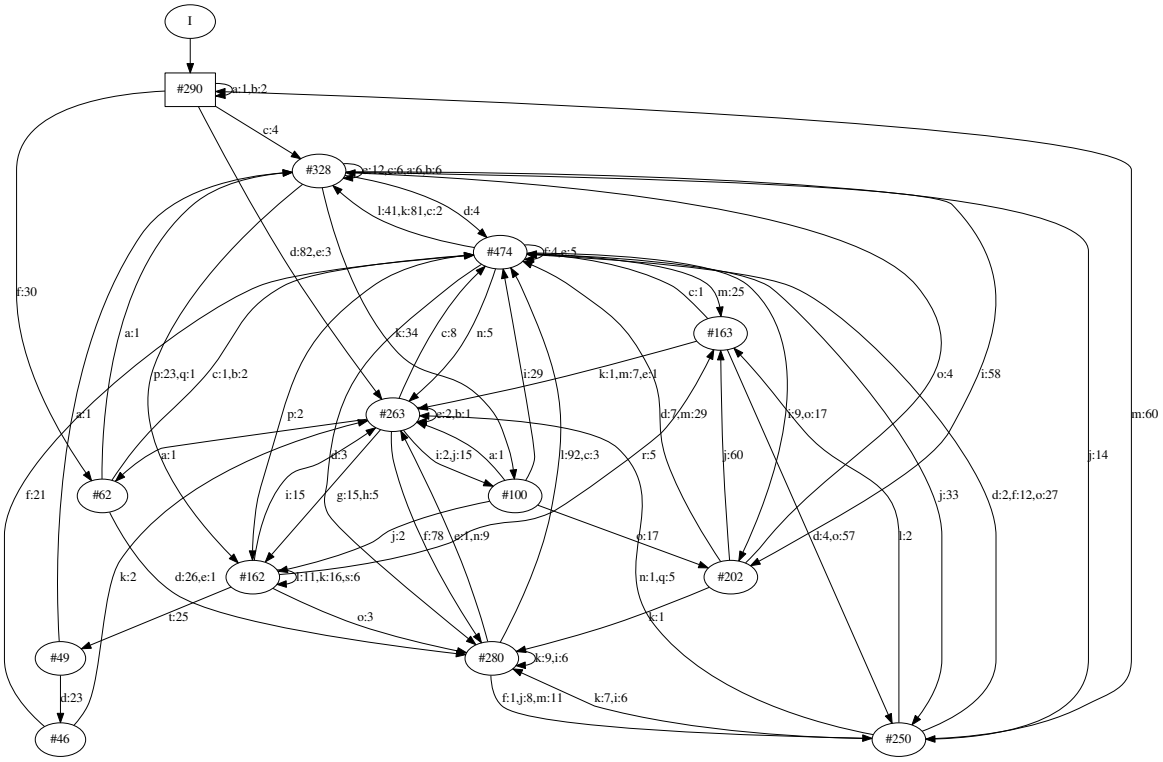


Figure 5.7: SML model 1 that has the least number of states

Based on the guideline to obtain concise and easy to visualise models, we activate the sink functionality and set the sink_count. There are several symbols in each traces that occur less frequently. Hence such symbols have to be grouped to the sink node for better understandability. The sink_count is set to 10. This gives us an understandable model shown in Figure 5.8. This model shows the most frequent execution of the interface behavior under study.

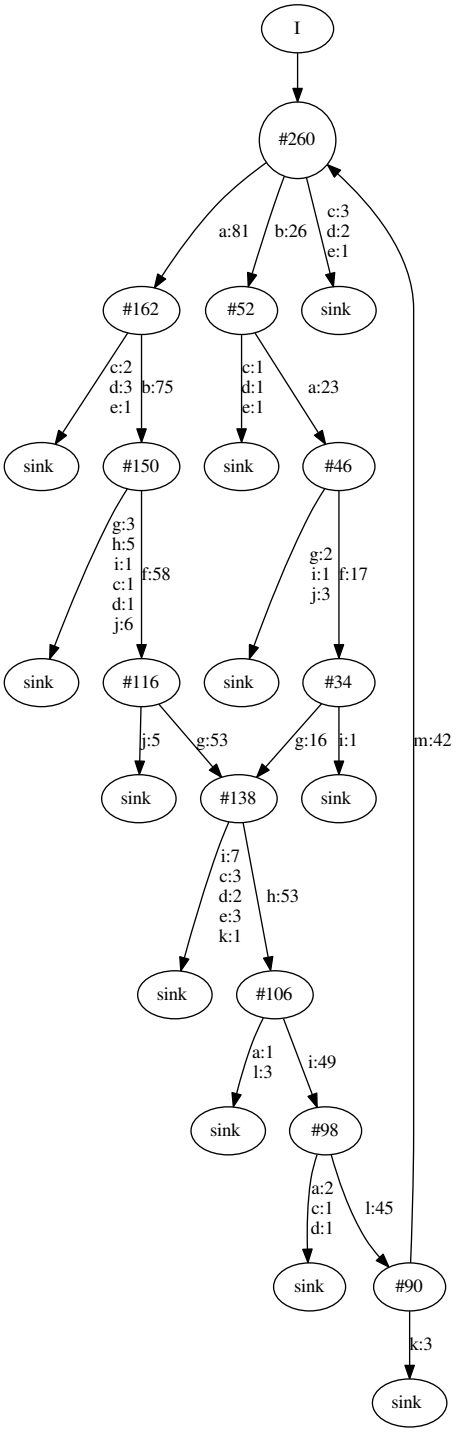


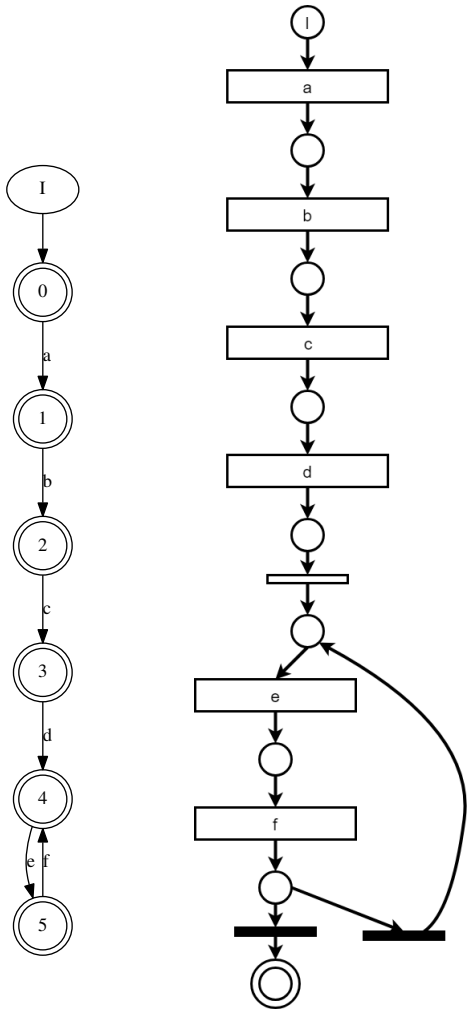
Figure 5.8: SML model 1 with Sink functionality on.

5. **Component B - Swap production test**

   The trace file of component B with interface Swap having undergone the production test has

6 symbols. The symbols are {a, b, c, d, e, f}. The trace file which is the input to SML has 13 positive traces. The trace file is very similar to the one in Component A - Swap production test. The Overlap driven algorithm is applied to obtain the state machine model in 5.9 (a).

Observation - *The results of component A and component B, swap production test are the same. This indicates that they have the same interface behavior for both component A and component B*



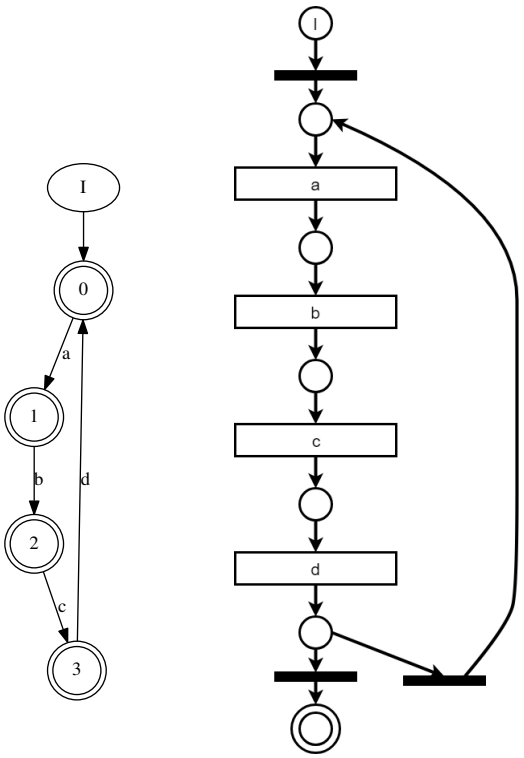
(a) State machine learning result

(b) Process mining - Petri net result

Figure 5.9: Results of SML and Petri net for Component B production test

6. **Component B - Swap component test** The trace file of component B with interface Swap having undergone the component test has 4 symbols. The symbols are {a, b, c, d }. The trace file which is the input to SML has 15 positive traces. The trace file is very similar to the one in Component A - Swap component test. The Overlap driven algorithm is applied to obtain the state machine model in 5.10 (a).

   Observation - *The results of component A and component B, swap component test are the same. This indicates that they have the same interface behavior for both component A and component B*



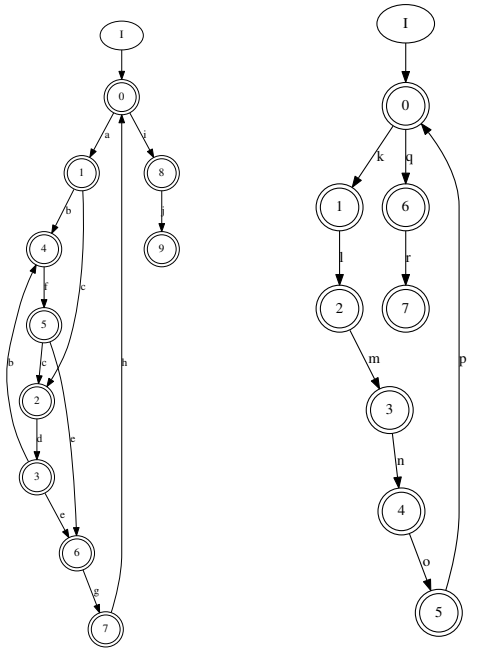
(a) State machine learning result

(b) Process mining - Petri net result

Figure 5.10: Results of SML and Petri net for Component A Swap component test

7. **Component B - Swap Advanced production test**

   The trace file of component B with interface Swap Advanced having undergone the production test has 18 symbols. The symbols are {a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r}. Similar to component A swap advance production test this trace is composed of two different swap events occurring in an interleaved manner. So ASML had split the production test trace further into two sub traces based on the start and stop events of the swap. This split traces are named swaplow and swaphigh which has 10 {a, b, c, d, e, f, g, h, i, j} and 8 {k, l, m, n, o, p, q, r} symbols respectively. The traces for swaplow and swaphigh are a single long trace. The overlap driven algorithm was used with lower_bound = 2 to obtain models. The models for both swaplow and swaphigh are given in Figure 5.5. The Petri net results for swaplow and swaphigh are also available. These results were compared in the tool chain and the models were found to be equivalent. Thus by splitting the entire log based on the occurrence of swap events , helps in obtaining insights on the interface behavior of component B.



(a) State machine learning result for swaplow with 10 symbols

(b) State machine learning result for swaphigh with 8 symbols

Figure 5.11: Results of SML and Petri net for Component B production test

8. **Component B - Swap Advanced component test** The log file for component B with Swap Advanced interface which has undergone component test has 10 symbols. The trace file used as input for SML has about 12 positive traces. As the number of traces available is less and the trace file exhibit a lot of concurrent events we apply Overlap driven algorithm to obtain models.

   The various settings used to obtain models are given in Table 5.2. All these models are unrelated and hence placed at different points in the lattice. We apply quantitative methods to identify a model in the lattice which is a lesser approximation of the behavior in traces. The models obtained are consistent with the traces but are over approximations of the trace behavior. As there is no reference model we investigate the merges in the model that creates a self loop leading to over approximation as done for Component A. We see that SML1 which is the most over approximated model is close to the flower automaton in the lattice. SML3 has the least over approximation and is placed at the bottom. Model SML2 is placed between SML1 and SML3 in the lattice.

   The model SML1 has the least number of states , which also explains the over approximation behavior in this case. This model is considered to be more understandable because of the lesser number of states and is shown in Figure 5.12.

Model	Algorithm	Parameter 1 State_count	Parameter 2 Symbol_count	Parameter 3 Lower_bound	Number of states in model
SML 1	Overlap driven	5	2	1	6
SML 2	Overlap driven	5	2	2	18
SML 3	Overlap driven	5	2	3	20

Table 5.2: Algorithms and settings used to obtain understandable models for Component B - Swap Advanced component test

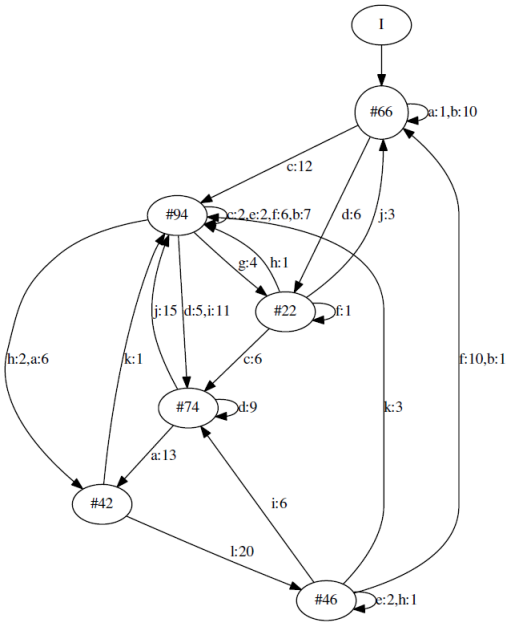


Figure 5.12: SML model 1 for component B.

5.2 Conclusion

In this section we discuss the results and the conclusions of the case study from ASML.

- The results of the case study shows that the guidelines and approach to find models close to the observed behavior in traces can be applied to an actual software component.
- We had to apply overlap driven algorithm, because of the lack of negative evidence and lack of probabilistic information; suggestion is that tests can be extended with probabilistic information and/or negative evidence to allow the use of more advanced techniques for which fundamental convergence results are established in literature.
- To use the Alergia algorithm the tests have to be obtained in such a way that the SUL behaves as a probabilistic automaton. This can be done by creating test generator block to trigger events in the system with a defined probability.
- The results of the case study shows that our approach to compare models based on language inclusion is possible and the obtained results in SML are same as the earlier obtained results of process mining.
- With this approach it is possible to identify the amount of data required for the Hankel or Alergia algorithm to obtain models that are close to the observed behavior in traces.

Chapter 6

Conclusions and Future Work

In Chapter 4, we provided a set of guidelines to apply state machine learning and also validated these guidelines with synthetic examples. In chapter 5, we used the guidelines to apply state machine learning to a real case study from ASML. In this chapter we discuss the conclusions of the thesis and reflect over the contributions made. In Section 6.1, we draw the conclusions of the thesis. Finally, we suggest possible topics and ideas as extensions of this thesis as future work, in Section 6.2.

6.1 Conclusions

The thesis started with the objective to understand the passive learning technique state machine learning and to provide insights on the various algorithms used. In this thesis we have made a detailed study on the SML technique and the various state merging algorithms used in it. In Chapter 2 we provide insights on these algorithms and explain how they work.

There are several model learning algorithms to learn models. Our second goal is to develop a framework which can compare the results of different model learning algorithms and settings. In this thesis we provide a systematic approach to compare models with different formalisms based on language inclusion. We developed a framework to compare these models and also defined notions of closeness between models both qualitatively and quantitatively. We used different model learning algorithms and settings in state machine learning to construct a lattice of models. The lattice is made complete by computing new models using the infimum and supremum operators. Using the qualitative notion of closeness we select a model from the lattice that is the least over/under approximation of the observed behavior. Since language inclusion is a partial order, all the models in the lattice are not related. Hence we introduced quantitative methods to compare these unrelated models by computing precision and recall measures and identify which model approximates the observed behavior as close as possible. In this thesis we have provided a systematic approach to compare models and form the lattice, using which we can select models that are close to the observed behavior in traces.

To make these comparisons and to form the lattice we have a tool chain which is used to compare the different model learning results. To validate this tool chain a comparison between state machine learning and process mining results have been done. Comparison between state machine models and Petri net models were validated for the given ASML case study and they were found to be equivalent.

Our third goal was to provide guidelines to choose the appropriate state machine learning algorithm and corresponding settings. We provide a set of guidelines from both literature and our experiments to use SML. These guidelines help to use state machine learning technique in the best possible way. They help to attain models which are as close as possible to the observed behavior in traces. They provide information on which state merging algorithm to use for certain types of input data. It also helps us to determine if the amount of data used is sufficient to learn a model

that is close to the behavioral specification. These guidelines were also validated using synthetic examples.

We apply state machine learning to an industrial case study from ASML. We analyze the given data and make use of the guidelines to obtain models for the software component under study. We applied the SML technique to the case study to obtain models that are close to the behavior in traces and still being understandable. These models provide a lot of insights for ASML to understand the interface behavior of the software component.

There are several model learning techniques to learn models. Combining these techniques helps us to get close to the original behavior of a software component. State machine learning technique together with our model comparison approach promises a lot in obtaining models closer to the observed behavior in traces. The SML technique, the guidelines and the approach to compare models discussed in this thesis provides a lot of insights for ASML and also brings us a step closer to learning the behavior of systems from software traces.

6.2 Future work

As suggestions for future work we leave the following topics:

- **Automating the creation of models to form the lattice**

  The different learning algorithms can learn different models. Currently the guidelines are used to tune the settings of SML to obtain models which are close to the original behavior of a software component. Automating this process of obtaining models to construct the complete lattice and identifying the model that approximates the behavioral specification as close as possible can be very helpful. It gives users a systematic way to explore models and to find those that are as close as possible to the observed behavior, while still being understandable.

- **Exploring and validating other algorithms in SML:**

  State machine learning has several other algorithms to obtain models like DFASAT, likelihood, conflict-driven. Hence these algorithms in SML need to be explored.

Bibliography

[1] Alexander Maier, Asmir Voden Carevic and Oliver Niggemann. Evaluating Learning Algorithms for Stochastic Finite Automata Comparative Empirical Analyses on Learning Models for Technical Systems, 1994. 7, 8

[2] Flexringe automaton learning tool. https://automatonlearning.net/flexfringe. 36

[3] R. Carrasco and J. Oncina. Learning stochastic regular grammars by means of a state merging method. pages 139-152.Springer-Verlag., 1994. 8, 17, 32

[4] Sicco Verwer and Christian A. Hammerschmidt. Flexfringe: a passive automaton learning package, International Conference on Software Maintenance and Evolution (ICSME), 2017. 8

[5] Price, Lang K. and Pearlmutter B. Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In: Honavar, V.G., Slutzki, G. (eds.). LNCS (LNAI), vol. 1433, pp. 112. Springer, Heidelberg , 1998. 8, 31

[6] Sicco Verwer and Marijn J. H. Heule. Exact DFA Identification Using SAT Solvers, Grammatical Inference: Theoretical Results and Applications. ICGI 2010. Lecture Notes in Computer Science, vol 6339. Springer, Berlin, Heidelberg, 2010. 6, 8

[7] Guy Lapalme and Marina Sokolova. A systematic analysis of performance measures for classification tasks, 2009. 26

[8] Kirill Bogdanov and Neil Walkinshaw. Automated Comparison of State-Based Software Models in terms of their Language and Structure, 2010. 24, 25

[9] Anil Nerode. Linear Automaton Transformations, Proceedings of the AMS,JSTOR 2033204, 1958. 8

[10] Tool used for trace generation. http://poosl.esi.nl. 33

[11] Wil MP van der Aalst. Process mining: data science in action, 2016. 20, 21, 29

[12] M. Geilen, J. Voeten, P. van der Putten, L. van Bokhoven and M. Stevens. Object-Oriented Modelling and Specification using SHE. Computer Languages, volume 27, pp 19-38, 2001. 33