Bachelor thesis
Computing Science

Radboud University

An Extended Algorithm for Learning Serial Compositions of Mealy Machines

Author:
Reinier Joosse
R.Joosse@student.ru.nl
4698649

First supervisor/assessor:
prof. dr. F.W. Vaandrager
F.Vaandrager@cs.ru.nl

Second assessor:
prof. dr. M.I.A. Stoelinga
m.i.a.stoelinga@utwente.nl

July 4, 2019

Abstract

Some systems which convert input sequences to output sequences consist of two subsystems, A and X, where the input is given to A, the output of A is used as input for X, and the output of X is returned. Abel & Reineke [2] developed an algorithm that, given such a composition for which a model of A is known, learns a model that is equivalent to the behavior of X that can be observed in the composition.

We propose a generalization of this algorithm that can also solve the case where some outputs of A are returned immediately as the output of the composition instead of being forwarded to X. The algorithm is compared to the original algorithm and to a black-box learning algorithm which learns a model of the entire composition. Experiments we performed in which models of three industrial systems are learned in a composition show that the new algorithm is an improvement in terms of run time and the number of required output queries.

Acknowledgments

I would like to thank Frits Vaandrager, without whose ideas and suggested literature my interest in this subject would not have been sparked and who supported me during my thesis project. I would also like to thank Marielle Stoelinga for her feedback on my thesis. Furthermore, I thank Andreas Abel and Jan Reineke, whose initial research enabled me to elaborate on this subject and who provided me with an implementation of their algorithm, which served as the basis for most of my experiments. Lastly, I would like to thank the other (anonymous) people who supported me and reviewed my writings.

Contents

Chapter 1

Introduction

Software or hardware components sometimes need to be redesigned to make them more efficient or to make them fit into new programming frameworks. During such a process, an important goal is that the behavior of the redesigned component is exactly the same as that of the old component. To ensure this, it is necessary to know exactly how the old system responds to all possible inputs. This behavior can be represented by a model. It can be tedious to construct such a model manually, especially when the user is not familiar with the system, as each possible input combination may result in different outputs.

Instead of trying to make a model manually, it can be constructed automatically by applying *model learning* techniques [16]. *Active* model learning algorithms, such as Angluin's L^* algorithm [3] (or Rivest & Schapire's improvement [14], or any other of its improvements; see [8] for an overview), can learn a model of a System Under Learning (SUL) assuming that it is possible to ask *output queries* and *equivalence queries*.

In an output query, the learner asks how the SUL responds to a given input sequence. This can be answered by providing the input sequence to the SUL and recording which outputs it returns. An equivalence query answers the question whether a given hypothesis model is equivalent to the SUL. If it is not, a counterexample must be returned (i.e. an input sequence for which the hypothesis and the SUL return different output sequences). If a model of the SUL is not already available, equivalence queries are usually approximated by comparing the results of the two systems for many output queries: if a counterexample cannot be found after an extensive search, the hypothesis and the SUL are assumed to be equivalent.

The number of output and equivalence queries used by a model learning algorithm should be as small as possible to decrease the run time. While a single output query may not take very long, the total run time is greatly affected if the learning process requires many of them. For example, biometric passports allow only one input symbol per second to prevent brute-force

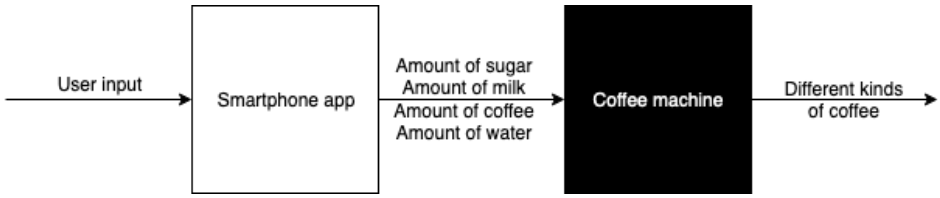


Figure 1.1: A composition of a smartphone app and a coffee machine.

attacks [1]. In such cases, using millions of output queries can increase the learning time to the point that it is unfeasible to complete the learning process. A single equivalence query may also take a long time depending on the method of approximation.

For more extensive background reading on model learning, refer to [3, 12, 9].

To complicate matters, however, interacting with the SUL directly is not always possible: sometimes this is only possible through an interface. Imagine, for example, a coffee machine which can only be used by pressing buttons in a smartphone app (see Figure 1.1). The app generates an input for the coffee machine based on its own state and the pressed button, and the coffee machine determines what coffee to prepare based on the input received from the app. If we already have a model of the app's behavior, we may call this composition a *gray box* (as the app is a known *white box*, and the coffee machine is an unknown *black box*). We cannot use a black-box learning algorithm such as L^* to learn a model of the coffee machine if the app cannot generate all possible input sequences for the coffee machine.

It is possible to learn a model of the entire composition of the app and the coffee machine by considering it as a black box, where the buttons in the app are the inputs and the prepared coffee is the output. A model of the coffee machine may then be inferred from the model of the app and the composition [17]. If the app is complicated, however, learning the entire composition may take too long.

Learning a model of the right system (the coffee machine) in such compositions, where a model of the left system (the app) is known, can be done more efficiently using an algorithm proposed by Abel & Reineke in [2]. They consider the case when all outputs of the left system are passed as input to the right system. The output of the right system is the output of the composition.

Abel & Reineke's algorithm cannot be applied to cases where some outputs of the left system are directly given as output of the composition, without first passing them to the right system. In practice, this is sometimes required. For example, continuing the example of the coffee machine, some buttons may not immediately forward the user's request but first ask the user for confirmation instead. Or it may occur that a company would

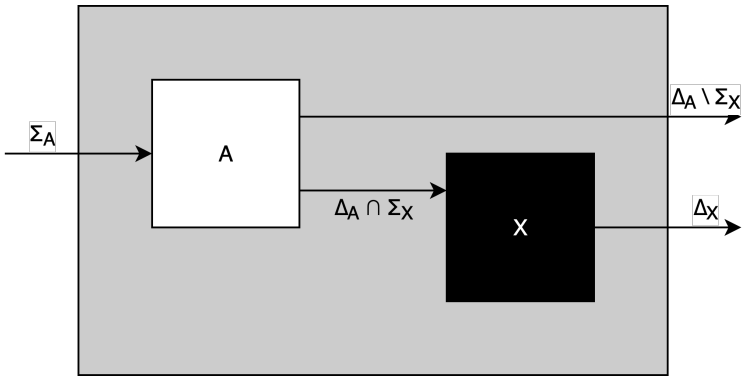


Figure 1.2: A composition $A \gg X$.

like to learn a large amount of software components with which it can only interact through a known security mechanism which sends messages for the SUL given some inputs, but outputs error messages for the user given other inputs.

This thesis presents an extension of Abel & Reineke's algorithm which can learn a model (of minimal size) of the observable behavior of the right system in this more general case as well, displayed in Figure 1.2, where some outputs of A are not forwarded to X. We will call A the *left* machine and X the *right* machine.

In [2], Abel & Reineke only test their algorithm using randomly generated test cases. This thesis compares the proposed extended algorithm to their original algorithm as well as to a black-box learning algorithm for learning the entire composition, implemented in the library LearnLib [9] (namely, [15], according to LearnLib's documentation[1]). Instead of using randomly generated models, we experiment with three industrial use cases as right machines, provided by the company ASML for the RERS 2019 challenge [10]. For the left machine, we use queuing systems of several sizes. Furthermore, we compare the number of output queries used by the three algorithms for an increasingly large left machine. We find that our extension is faster than Abel & Reineke's original algorithm in these experiments. It also uses fewer output queries than the other algorithms for an increasing size of the left machine. We argue that this would make our extension faster than the LearnLib implementation outside of our experimental setup as well.

Hence, we address two research questions in this thesis:

- How can Abel & Reineke's algorithm be extended to make it possible to learn compositions where some outputs of the left machine are directly given as output of the composition instead of being passed to the right machine?

[1] `https://github.com/LearnLib/learnlib/wiki/Learning-Algorithms-in-LearnLib#extensiblelstarmealy`

- How does this extension compare to the original algorithm and to learning the composition as a black box in industrial use cases in terms of run time and number of required output queries?

Section 1.1 will present some required definitions, after which Chapter 2 follows with a formal definition of the problem that must be solved by the algorithm required in the first research question. Chapter 3 presents this algorithm by explaining how Abel & Reineke's original algorithm works and presenting how we extend this algorithm such that it can solve our problem. Next, Chapter 4 will answer the second research question by presenting appropriate experiments and their results. Chapter 5 outlines the literature related to this thesis and Chapter 6 concludes.

1.1 Preliminaries

This section presents some definitions necessary for understanding the following chapters.

Definition 1. *A (deterministic) Mealy machine is a six-tuple* $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$ *where* Q *is a finite set of states,* Σ *is a finite set containing all input symbols (the input alphabet),* Δ *is the (finite) output alphabet,* δ *is a (partial) transition function with signature* $Q \times \Sigma \to Q$*,* λ *is a (partial) output function with signature* $Q \times \Sigma \to \Delta$ *and* $q_0 \in \Sigma$ *is the initial state [7, p. 43]. Let* $\mathcal{M}$ *be the set of all Mealy machines.*

Definition 2. *A transition of a Mealy machine* $(Q, \Sigma, \Delta, \delta, \lambda, q_0) \in \mathcal{M}$ *is a four-tuple* $(q, q', i, o) \in Q \times Q \times \Sigma \times \Delta$ *such that* $\delta(q, i) = q'$ *and* $\lambda(q, i) = o$*. Here,* q *is called the origin state,* q' *is the destination state,* i *is the consumed input symbol and* o *is the produced output symbol.*

Because every transition must have both an origin state and an input symbol, it is meaningless if only one of $\delta(q, i)$ and $\lambda(q, i)$ is defined. Therefore, we require that $\delta(q, i) \downarrow \Leftrightarrow \lambda(q, i) \downarrow$. If, for some $q \in Q$ and $i \in \Sigma$, the functions are not defined, then there is no transition for input i from state q.

An example of a Mealy machine can be seen in Figure 1.3. Circles denote states and arrows denote transitions: an arrow from a state labeled 1 to a state labeled 2 annotated with `i/o` denotes a transition $(1, 2, \mathtt{i}, \mathtt{o})$. The initial state is indicated by an unlabeled incoming arrow.

Nondeterministic Mealy machines may have multiple transitions for the same combination of state and input symbol. Such Mealy machines do not have a λ function; the codomain of their δ function is $\mathcal{P}(Q \times \Delta)$, returning sets of tuples containing the destination state and the output symbol. Other definitions are changed accordingly.

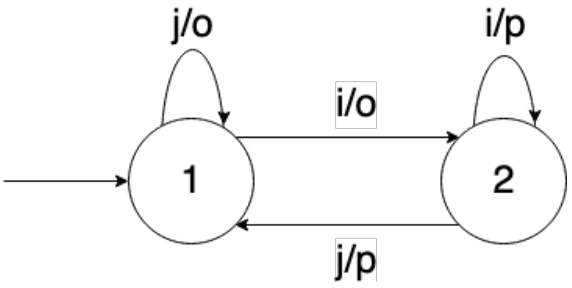


Figure 1.3: A Mealy machine.

Definition 3. *A Mealy machine* $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$ *is* input-complete *if* $\delta(q, i)$ *and* $\lambda(q, i)$ *are defined for all* $q \in Q$ *and* $i \in \Sigma$.

The empty sequence is denoted by ϵ. The concatenation of two sequences w and v is also a sequence and is written as wv. For example, the concatenation of the sequences `a` and `bcd` is written as `abcd`. Sets can be extended to sets of finite sequences using the Kleene star [7, p. 28] (e.g. $\{\mathtt{a}, \mathtt{b}\}^* = \{\epsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}, \dots\}$). For example, Σ^* is a set of input sequences and Δ^* is a set of output sequences.

Definition 4. *Given a Mealy machine* $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$*, let* $\lambda_M^* \colon Q \times \Sigma^* \to \Delta^*$ *be its extended output function, defined as follows:*

$$\lambda_M^*(q, \epsilon) = \epsilon$$
$$\lambda_M^*(q, iw) = \lambda(q, i)\lambda_M^*(\delta(q, i), w)$$

We abbreviate $\lambda_M^*(q, w)$ *as* $\lambda_M^*(w)$*, also called the* output query.

The output query gives the output sequence a Mealy machine produces in response to a given input sequence. For example, to run the input sequence `ijji` on the Mealy machine in Figure 1.3, we start in the initial state, take the `i/o` transition to state 2, the `j/p` transition to state 1, the `j/o` transition to state 1 and finally the `i/o` transition to state 2. The output sequence is thus `opoo`.

Definition 5. *Two Mealy machines* $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ *and* $M' = (Q', \Sigma', \Delta', \delta', \lambda', q_0')$ *are* equivalent *if* $\Sigma = \Sigma'$ *and* $\lambda_M^* = \lambda_{M'}^*$.

The *size* of a Mealy machine refers to the size of its set of states. A Mealy machine is *minimal* if there does not exist an equivalent Mealy machine with fewer states.

Definition 6. *An* equivalence query *for a Mealy machine* M *is a function* $EQ_M \colon \mathcal{M} \to \Sigma^* \cup \{\bot\}$ *such that* $EQ_M(M') = \bot$ *if* M *and* M' *are equivalent; if they are not equivalent, a distinguishing sequence* $EQ_M(M') = w$ *is returned such that* $\lambda_M^*(w) \neq \lambda_{M'}^*(w)$.

Chapter 2

Problem Statement

The problem solved by the algorithm proposed in this thesis is as follows.

Suppose that we have a Mealy machine C which is composed of two deterministic, input-complete Mealy machines A and X, written as $C = A \gg X$. We will call A the *left* machine and X the *right* machine. Any input to C is passed to A. Some outputs of A are subsequently passed as input to X, in which case the corresponding output of X is the output of C. All other outputs of A are directly given as output of C. A visual representation is given in Figure 1.2. Formally,

$$
\begin{aligned}
A &:= (Q_A, \Sigma_A, \Delta_A, \delta_A, \lambda_A, q_{0_A}) \\
X &:= (Q_X, \Sigma_X, \Delta_X, \delta_X, \lambda_X, q_{0_X}) \\
C &:= (Q_A \times Q_X, \Sigma_A, (\Delta_A \setminus \Sigma_X) \cup \Delta_X, \delta, \lambda, (q_{0_A}, q_{0_X})) \\
&\quad \text{where} \\
\delta((q_A, q_X), i) &:= \begin{cases} (\delta_A(q_A, i), \ \delta_X(q_X, o_A)) & \text{if } o_A \in \Sigma_X \\ (\delta_A(q_A, i), \ q_X) & \text{otherwise} \end{cases} \\
\lambda((q_A, q_X), i) &:= \begin{cases} \lambda_X(q_X, o_A) & \text{if } o_A \in \Sigma_X \\ \lambda_A(q_A, i) & \text{otherwise} \end{cases} \\
o_A &:= \lambda_A(q_A, i)
\end{aligned}
$$

To make it clear which outputs originate from A and which originate from X, we require that $(\Delta_A \setminus \Sigma_X) \cap \Delta_X = \emptyset$.

In such a composition, not all states of X may be reachable if not all possible input sequences in Σ_X^* can be generated by A. In such cases, we cannot learn X completely because we cannot query it directly. We can only learn the behavior of X that is visible in the context of the composition.

Definition 7. *Two Mealy machines X_1 and X_2 are right-equivalent in the context of a Mealy machine $A = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ if $\forall w \in \Sigma^*, \lambda^*_{A \gg X_1}(w) = \lambda^*_{A \gg X_2}(w)$.*

Now, the problem statement is as follows. Suppose that we know the internal structure of A (i.e. we know Q_A, Σ_A, Δ_A, δ_A, λ_A and q_{0_A}) but we do not know the structure of X (we only know Σ_X). We cannot directly ask output or equivalence queries for X, but we can ask output and equivalence queries for the composition C.[1] The task is to construct a minimal Mealy machine model that is right-equivalent to X using this information.

[1] We will later show that we can obtain answers to λ_X^* and EQ_X by defining them in terms of λ_C^* and EQ_C.

Chapter 3

Algorithm

The algorithm we propose learns a Mealy machine of minimal size of the behavior of the unknown right system (X) in a composition $A \gg X$. It is the same as Abel & Reineke's algorithm presented in [2], except that the output and equivalence queries are realized in a different way. This chapter will explain the entire algorithm, including the modified way of performing output and equivalence queries, but only the method of performing output and equivalence queries is new in this chapter. The other aspects of how the algorithm works were already presented in [2], where they are described more extensively.

In the first section, an overview will be given of how the algorithm works. The second and third section explain the original algorithm by Abel & Reineke in more detail, which is used in our approach. Section 3.4 presents our new method to realize the output queries. The chapter concludes with a section about the correctness of our approach.

3.1 Overview

To discover the behavior of X, the algorithm first comes up with a number of input sequences for X for which it observes the output using output queries. The results of the output queries are stored in a data structure called the *observation table*. The observation table maps input sequences for X to the corresponding output sequences of X.

Because Mealy machines do not give any output for the empty input sequence, we will only consider non-empty input sequences. Each input sequence can be split into two parts: a (possibly empty) prefix and a suffix. The rows of the observation table are labeled with prefixes of input sequences for X and the columns with suffixes. The cells contain the last output symbol that X gives for the input sequence obtained by appending its column's suffix to its row's prefix. A cell in an observation table in a row with prefix w and in a column with suffix v will contain the last symbol in the result of

	i	j	ii
ϵ	o	o	p
i	p	p	p
j	o	o	p
ji	p	p	p

Table 3.1: An observation table for the Mealy machine in Figure 1.3. The input alphabet is $\{\mathtt{i}, \mathtt{j}\}$; the output alphabet is $\{\mathtt{o}, \mathtt{p}\}$.

$\lambda_X^*(wv)$.

An example of an observation table for the Mealy machine in Figure 1.3 is shown in Table 3.1. To determine the symbol in the cell in the third row and the third column, we have to calculate $\lambda_X^*(\mathtt{jii})$, since we append the row's prefix ($\mathtt{j}$) to the column's suffix ($\mathtt{ii}$). If $\mathtt{jii}$ is not a valid output sequence of the left machine, we put $\perp$ in the cell. Otherwise, looking at Figure 1.3, we use the following transitions:

- $(1, 1, \mathtt{j}, \mathtt{o})$
- $(1, 2, \mathtt{i}, \mathtt{o})$
- $(2, 2, \mathtt{i}, \mathtt{p})$

We obtain the output sequence $\mathtt{oop}$, hence $\lambda_X^*(\mathtt{jii}) = \mathtt{oop}$. We put the last symbol from this output sequence into the cell. Therefore, the cell contains $\mathtt{p}$.

From a full observation table, a hypothesis Mealy machine for X is constructed, which is then compared to X using an equivalence query. If the hypothesis is equivalent to X, we are done. Otherwise, we refine the observation table using the returned counterexample. Two questions arise here:

1. How do we perform output queries for X?
2. How do we perform equivalence queries for X?

We start with the first question. To fill the observation table, we have to know the outputs of X for given input sequences. A problem we face with the composition is that we cannot directly perform output queries for the right machine. Since we know the structure of A, however, we can find specific input sequences for A which cause X to get the desired input sequence (see Section 3.4). It is not always the case that all possible input sequences for X can be generated by A. For those input sequences, it is impossible to observe the corresponding output sequence of X. Cells in the observation table where this is the case will contain $\perp$ cells.

From a correct observation table, the algorithm can construct a hypothesis Mealy machine that exhibits all observed behavior (see Section 3.2). If the observation table did not contain all relevant behavior of X, it needs to be refined further. To find out whether this is the case, the algorithm needs an answer to an equivalence query for this hypothesis. This leads to the second question.

Our problem statement dictates that we cannot directly perform an equivalence query for a hypothesis of X, but only for the composition. One way to implement equivalence queries for X is by expressing them in terms of the equivalence query for the composition $A \gg X$ as follows. The hypothesis H is composed with A into the composition $A \gg H$. This hypothesis composition is used as a hypothesis in an equivalence query for the actual composition, $A \gg X$. If $A \gg H$ is equivalent to $A \gg X$, then H must also be equivalent to X in the context of A. Otherwise, the equivalence query will generate an input sequence c that distinguishes the hypothesis composition from the actual composition.

This counterexample c is an input sequence for the composition (and for A), not for X. To convert c into an input sequence that distinguishes H and X, we first calculate $c' = \lambda_A^*(c)$. Since, in our composition, some outputs of A are directly given as the output of the composition while others are first passed to X, c' may contain symbols which are not passed to X. The input sequence that distinguishes H and X is therefore obtained by removing all symbols which are not in the input alphabet of X from c'.

This way of performing equivalence queries by using equivalence queries for the composition demonstrates that it is possible to implement equivalence queries for X, although it may not be the most efficient method.

To summarize, to construct a Mealy machine representing X's behavior in $A \gg X$, one could in theory start with an observation table containing all symbols in Δ_X as columns and only ϵ as a row and follow the following sequence of actions:

1. Construct a Mealy machine H from the observation table of X.

2. Perform an equivalence query for $A \gg H$. If a counterexample c is returned, then add the sequence of inputs of X in $\lambda_A^*(c)$ as a row to the observation table and go back to step 1. Otherwise, return H.

In practice, however, performing many equivalence queries takes too much time. To reduce the number of equivalence queries, we can try to capture more of X's behavior in the observation table before constructing a Mealy machine and performing an equivalence query. Abel & Reineke [2] presented a number of ways to do this (see Section 3.3).

Section 3.2 describes how the algorithm constructs a hypothesis for X from an observation table; Section 3.3 explains how it extends the observation table before doing so, to obtain more information first. These two

sections reiterate the procedures which the algorithm inherits from [2], where they are described more extensively. Subsequently, Section 3.4 explains how output queries for the right machine are answered in our new setting to fill the observation table.

3.2 Creating a Mealy machine from an observation table

Here, we will describe how to construct a Mealy machine from an observation table: first, how to find the states, and subsequently how to find the transitions. This section summarizes the method to do so described in [2] and does not add anything new.

3.2.1 States

The rows of an observation table can be thought of as states in a Mealy machine. The prefix of a row is the input sequence which causes the Mealy machine to end up in that state. It is possible that following different input sequences brings a Mealy machine to the same state due to loops. For example, in Figure 1.3, using both `i` and `ji` we end up in state 1. Because we would like to learn a minimal Mealy machine that explains the behavior of the system, we must identify all sets of rows which represent distinct states instead of creating a new state for every row. In Table 3.1, the rows labeled ϵ and `j` represent state 1 of Figure 1.3 and the rows labeled `i` and `ji` represent state 2. A set of rows which represent the same state is called an *equivalence class*.

The columns of the observation table are useful when we want to determine whether multiple rows should be seen as the same state. Suppose that we have two rows: one with the prefix a, which leads to state q, and one with the prefix b, which leads to state r. If q and r can be considered to be the same state, then any input sequence processed by the SUL after reaching that state must result in the same output (since it is deterministic). In other words, for any suffix s, $\lambda^*(q, s) = \lambda^*(r, s)$ (provided that both as and bs can be generated by the left machine).

Since the columns of the observation table contain suffixes, two rows will be considered as the same state if they contain the same output symbols for the same columns, except in columns where one of the rows has $\perp$. Such rows are called *compatible*. In an equivalence class, all rows must be compatible.

The algorithm uses a SAT solver to create a partition of the rows in the observation table such that every partition part is an equivalence class, using the same approach as in [2]. How the problem is converted into a satisfiability formula will not be described here; the details can be found in [2].

3.2.2 Transitions

In addition to the states, the transitions of the Mealy machine must be determined.

Definition 8. *Given a row in an observation table with prefix p, a* successor row *under an input symbol i is a row in the observation table with prefix pi.*

If a row corresponds to an origin state, then its successor row under an input i is the destination state when taking the transition labeled with the input i. The generated output is in the column labeled i of the row of the origin state.

For example, in Table 3.1, the row labeled with the prefix `ji` is the successor row of the row labeled `j` under the input symbol `i`. Suppose that the row labeled `j` represents state 1 and the row labeled `ji` represents state 2. Then, the corresponding transition is $(1, 2, \texttt{i}, \texttt{o})$.

As explained, we do not use rows as states, but equivalence classes instead. In the context of equivalence classes, this means: for each equivalence class c and input symbol i, if there is a row in c (with prefix p) which does not have $\perp$ in the column labeled i and which has a successor row under i, then there is a transition from c to the equivalence class which contains the successor row, with the input symbol i and the output symbol in the i-column of the row with prefix p.

For example, in Table 3.1, consider the equivalence class containing the rows labeled ϵ and `j`, and consider the input symbol `j`. There is a row in this equivalence class which does not have $\perp$ in the column labeled `j`: for example, the row labeled ϵ. Additionally, this row has a successor row under `j`, namely the row labeled `j`. Hence, we get a transition from this equivalence class to itself (as `j` is in the same equivalence class) with the input symbol `j`. In this case, the successor row is in the same equivalence class, so the transition has the same destination state. The output symbol of this transition can be found in the `j`-column in the row labeled ϵ: it is `o`. We can see that this corresponds to the transition $(1, 1, \texttt{j}, \texttt{o})$ (since the equivalence class represents state 1).

3.3 Optimizing the observation table

This section describes how the algorithm extends the observation table before performing an equivalence query. If we have a more complete model of the system before performing an equivalence query, we have to do fewer equivalence queries before we have a correct hypothesis. The optimizations described here are repeated from [2].

The first optimization will prevent us from wrongly assigning two rows to the same equivalence class (i.e. assuming that they represent the same state). The algorithm learns a deterministic Mealy machine, which implies that for

each origin state q and input symbol i, there can only be one transition (q, q', i, o). As described before, a transition is added when one of the rows (with prefix p) in equivalence class q has a non-$\perp$ value in the column i, and the row with prefix pi is in equivalence class q'. Hence, to make the resulting Mealy machine deterministic, we must ensure that in each equivalence class, for all rows p which have a non-$\perp$ value in a column i, all rows pi must be in the same destination equivalence class. For example, considering the equivalence class in Table 3.1 with the rows ϵ and `j` again, the successors under input symbol `i` (namely the rows labeled `i` and `ji`) must be in the same equivalence class.

Since all rows in an equivalence class are compatible, a way to work towards this property before creating a partition is making sure that for all pairs of compatible rows in the observation table and for each input symbol i, the successors of both rows under i are also compatible. This property of the observation table is called *consistency*. If two compatible rows do not have compatible successors under some input symbol i, then this means that there is some column labeled with the suffix s in which the successor rows have different output symbols. Hence the column is would distinguish the originally compatible columns. Adding this column to the observation table will prevent us from assuming that the rows belong in the same equivalence class.

As a second optimization, we can use the fact that the Mealy machine we are learning is input-complete: each state has an outgoing transition for each input symbol, although not all of those transitions may be reachable in the right machine in a composition (if no output sequence can be generated by the left machine such that this transition is used). Hence, for each input symbol i and equivalence class c, if there are rows in c (with prefix p) for which pi is a valid output of the left machine, then there must be a successor row under input i in the observation table for at least one of those rows. This property is called *closedness for partitions*. If a partition is not closed, then a row with the mentioned prefix pi can be added to the observation table.

For Table 3.1, let us assume that any input sequence for the right machine can be generated by the left machine. The equivalence class with the rows ϵ and `j` satisfies the closedness property: for example, considering the row labeled `j` and the input symbol `j`, `jj` is a valid output of the left machine, so there must be a successor row for any row of this equivalence class under input symbol `j`. This is indeed the case, as the row labeled `j` is a successor under the input `j` for the row labeled ϵ. However, the equivalence class with the rows `i` and `ji` does not satisfy the closedness property, as it does not have a successor equivalence class under either `i` or `j`.

Finally, it may be possible to create many different partitions from an observation table, which translate into different hypothesis machines. If we have not learned all relevant behavior of the right machine yet, some of

the hypotheses might not be equivalent in the context of the composition. This implies that at least one of the possible hypotheses is incorrect. Before performing an equivalence query for a composition of the left machine with a hypothesis, we can generate all possible partitions and check whether all hypothesis machines are equivalent in the context of the composition, using the fact that we know the structure of the left machine and of the hypotheses. If not all hypotheses are equivalent in the context of the composition, a counterexample can be generated (an input sequence for the composition for which the outputs of the hypotheses differ). Such a counterexample can be translated into an input sequence of the right machine (by processing it using the left machine and filtering the outputs that are passed to the right machine) and added as a row to the observation table.

3.4 Output queries for the right machine

To fill the observation table, we have to perform output queries for X, the right machine. We cannot directly perform such output queries, however. We can only access X through the composition, so we need to define the output query for X in terms of the output query of the composition.

To do so, we first note that in the composition, the right machine can only process input sequences that originated as output of the left machine. Hence, to perform an output query for an input sequence w on X, we need to cause the left machine to output a sequence such that w is passed as input to X. The input sequence w is then processed by X and we can derive its output from the output of the composition.

It is not always possible to generate all possible input sequences for X as output of A. For such input sequences, it is not possible to do an output query. As a consequence, it is not possible to learn the behavior of X for such input sequences. This is why we can only learn a model of the observable behavior of X in the context of A and not a model of X itself.

Now, the task at hand is to cause A to output a sequence such that w, the input sequence of our output query, is passed as input to X. We can do this by examining the (known) structure of A and finding an input sequence for A that achieves this.

In case all output symbols of A are passed to X, which is the case in [2], we can use the following straightforward method for this. We know a Mealy machine model of A. In this model, we perform a depth-first search starting from the initial state, exploring transitions that have the desired output symbols as their output. Once we have found a path of transitions for which the corresponding output sequence equals w (the desired input for X), we can use the input sequence corresponding to the same path of transitions as input sequence for A to cause it to output w. If such a path does not exist, no input for A exists such that it outputs w.

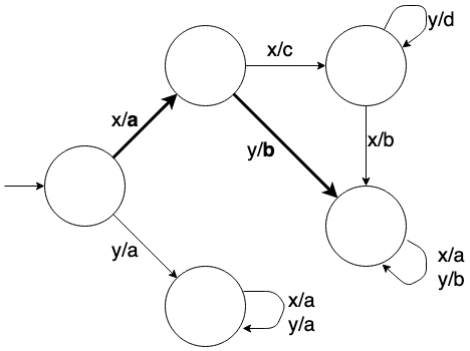


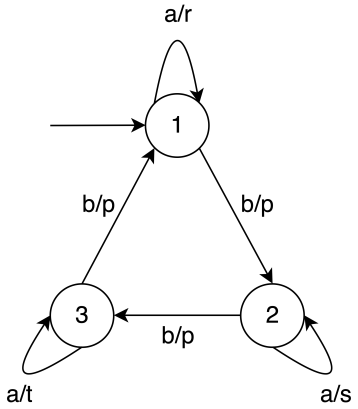
Figure 3.1: An example of how to find an input sequence which generates the output sequence `ab`.

For example, consider Figure 3.1 as the left machine (A). To find an input sequence which generates the output sequence `ab`, we start in the initial state, perform a search and find the transition path which corresponds to the input sequence `xy`. We can then input `xy` to the composition to cause `ab` to be passed to X. But we cannot find out how X responds to the input sequence `ba` as there is no path of transitions in this left machine which corresponds to the output sequence `ba`.

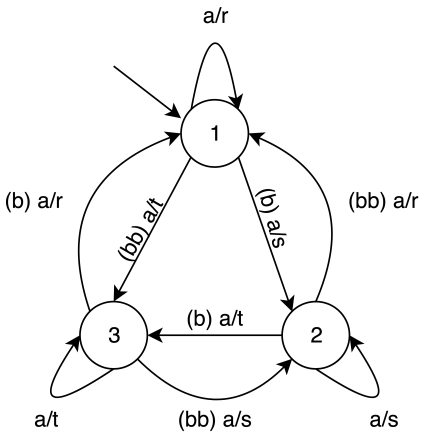
In the more general case, outputs of A are either passed as input to X or directly given as the output of the composition, without being processed by X. Consider the Mealy machine in Figure 3.2a as the left machine. The output `p` is not passed to X but given as the output of the composition. We can generate the input sequence `st` for X by using the input sequence `baba`. This will actually generate the output sequence `pspt`, but since the `p` is not passed to X, the input sequence for X will be `st`. Here, we cannot apply the depth-first search strategy explained above. We would get stuck in the first state as there is no available transition which generates `s`.

We will call the output symbols of A which are passed to X *visible outputs*, and those which are instead directly given as the output of the composition *hidden outputs* (they are hidden from the perspective of X). Similarly, *visible transitions* are transitions which produce a visible output and *hidden transitions* are transitions which produce a hidden output.

To perform output queries in this more general case, we take the following approach. We construct a (nondeterministic) Mealy machine called the *transitive closure machine*. This machine has the same states as A. It does not have hidden transitions; instead, from any state q, it is possible to use any transition from A of which the origin state is either q or a state which can be reached following a path of hidden transitions (corresponding to an input sequence of hidden inputs h) starting in q. These transitions have the same destination state, input symbol and output symbol as the original transitions. In addition, they are annotated with the input sequence h.



(a) A Mealy machine model for A, for the case when not all outputs are passed to X. The outputs `r`, `s` and `t` are passed to X, but `p` is not.



(b) The transitive closure machine for the Mealy machine in Figure 3.2a.

Figure 3.2: A Mealy machine and its transitive closure machine.

Figure 3.2b shows the transitive closure machine corresponding to the machine A in Figure 3.2a. The transitive closure machine may informally be thought of as a model of A seen from the perspective of X. From this perspective, some transitions require multiple input symbols. For example, to produce `t`, the input sequence `bba` is required. To produce the output sequence `st`, the input sequence `baba` must be given to A. As illustrated, we can use this machine to find out for which input sequence A produces any desired input sequence for X. We can use this to perform output queries for X.

To construct the transitive closure machine, we must first know for each state which other states are reachable with hidden transitions. Let T be the hidden transition relation, namely qTr means that either $q = r$ or there exists a hidden transition from q to r. Then its transitive closure is T^+, namely qT^+r means that there is a path of zero or more hidden transitions from q to r.

We can calculate T^+ by running a variant of the Floyd-Warshall algorithm [5] on a graph, where the nodes are the states of A and the edges are the hidden transitions. Algorithm 1 shows how to calculate the transitive closure. In addition to computing T^+, we have to remember for each $(q, r) \in T^+$ the input sequence represented by the hidden transitions we took to get from q to r. This sequence is stored in the d matrix for each pair of states, instead of the length of the path, which the original Floyd-Warshall algorithm uses. When, in Floyd-Warshall's algorithm, the entries in the matrix are compared to see whether a shortest path exists between

two nodes, we compare the length of the sequences stored in the matrix. In the transitive closure machine, we will annotate the transitions with these input sequences.

$Q \leftarrow$ states in A;
$d \leftarrow n \times n$ matrix where $n = Q$.size, with all cells initially $\perp$;
foreach *hidden transition* (q, q', i, o) **do**
  $d[q][q'] \leftarrow i$
end
foreach $q \in Q$ **do**
  $d[q][q] \leftarrow \epsilon$
end
foreach $k \in Q$ **do**
  foreach $q \in Q$ **do**
    foreach $q' \in Q$ **do**
      if `length`$(d[q][k])$ + `length`$(d[k][q'])$ < `length`$(d[q][q'])$ **then**
        $d[q][q'] \leftarrow$ `concatenate`$(d[q][k], d[k][q'])$
      end
    end
  end
end

Algorithm 1: Calculating the transitive closure. Let `length`$(\perp)$ return positive infinity.

After running the algorithm, the matrix d will contain for each pair of states (q, r) a shortest input sequence which follows a path of hidden transitions from q to r, if one exists. For the example in Figure 3.2a, assuming that the outer loop takes the states from Q in numerical order, Table 3.2 shows the d matrix after the iterations of the outer loop of the algorithm.

To construct the transitive closure machine from the matrix, we use the same set of states as A. For each state q (row of the matrix), we look up which other states (q') are reachable with hidden transitions (columns with

	1	2	3
1	ϵ	b	$\perp$
2	$\perp$	ϵ	b
3	b	$\perp$	ϵ

(a) Initialization

	1	2	3
1	ϵ	b	$\perp$
2	$\perp$	ϵ	b
3	b	bb	ϵ

(b) After $k = 1$

	1	2	3
1	ϵ	b	bb
2	$\perp$	ϵ	b
3	b	bb	ϵ

(c) After $k = 2$

	1	2	3
1	ϵ	b	bb
2	bb	ϵ	b
3	b	bb	ϵ

(d) After $k = 3$

Table 3.2: The d matrix after iterations of the transitive closure algorithm. The rows represent origin states; the columns destinations.

a non-$\perp$ value). All visible transitions in A with q' as their origin state are added to the transitive closure machine with q as their origin state. Algorithm 2 shows this procedure.

```
A ← the original left machine;
M' ← new Mealy machine;
M'.Q ← A.Q;
M'.initial_state ← A.initial_state;
foreach q ∈ Q do
    foreach q' ∈ Q do
        if d[q][q'] ≠ ⊥ then
            foreach visible transition (from, to, input, output) where
              from = q' do
                if M' does not contain transition (q, to, input, output)
                 then
                    addTransition(M', (q, to, input, output),
                     d[q][q']);
                end
            end
        end
    end
end
```

Algorithm 2: Constructing the transitive closure machine. `addTransition(m, t, a)` adds the transition `t` to the Mealy machine `m` and annotates the transition with `a`.

This concludes the method of how the transitive closure machine is constructed. It is used when performing an output query for X, $\lambda^*_X(w)$. The procedure is similar to the search on A in the case without hidden outputs. To cause A to produce the input w for X, a depth-first search is performed on the transitive closure machine, exploring the transitions with the required output symbol. If a valid transition sequence v is found, the corresponding input sequence is represented by the inputs (with annotation inputs) of the transitions. Next, v can be used in an output query for the composition. If the transitive closure does not have a matching transition path, $\lambda^*_X(w)$ cannot be answered.

The output sequence of the composition includes the hidden outputs. To find the answer to $\lambda^*_X(w)$, they need to be removed from the composition's output sequence.

We can now define the output query for the right machine X in terms of the output query for the composition $A \gg X$ as follows:

$$\lambda^*_X(w) = \begin{cases} \perp & \text{if } f(w) = \perp \\ e(\lambda^*_{A \gg X}(f(w))) & \text{otherwise} \end{cases}$$

where $f(w)$ performs a search on the transitive closure for a path of transitions for which w is the corresponding output sequence. If no such path exists, f returns $\perp$. Otherwise, it returns the input sequence for A represented by the path of transitions. Here, each transition constitutes a part of the input sequence created by appending its input symbol to its annotation; and e extracts the result of $\lambda_X^*(w)$ from the output of the composition by removing all occurrences of symbols in $\Delta_A \setminus \Sigma_X$ (hidden outputs) from the sequence.

If there are no hidden output symbols, the transitive closure machine is the same as A: there are no hidden transitions or annotations. In that case, using this definition of λ_X^* simplifies to using its original definition, as we perform a depth-first search on A using the f function. Furthermore, e will be the identity function as there are no symbols in $\Delta_A \setminus \Sigma_X$. In the other extreme, if there are no visible transitions, $\lambda_X^*(w) = \perp$ for all w as $f(w)$ will always return $\perp$ because the transitive closure does not have any transitions. In other words, A cannot cause any input sequence of X. This makes it impossible to learn the behavior of X.

This concludes our approach to answering output queries for X.

3.5 Correctness

Here, we will argue that the algorithm presented in this thesis is correct. We will assume that equivalence queries are not approximated but always give the correct answer. If this is not the case, the algorithm will not always give a correct result.

The correctness of our approach relies on the correctness of Abel & Reineke's algorithm. We assume that their approach is correct using the following lemma:

Lemma 1. *Abel & Reineke's algorithm learns a minimal Mealy machine model X' for a system X if a subset S of the input sequences for X can be observed in output and equivalence queries.*

For the output queries, this means that it is possible to answer output queries for any $w \in S$. For the equivalence queries, it means that an equivalence query for a hypothesis H will find a counterexample that distinguishes H and X if such a counterexample exists in S. If a counterexample exists that is not in S, the desired answer to the equivalence query is that H and X are equivalent.

The resulting Mealy machine, X', may not be equivalent to X, but at least we have that $\forall w \in S, \lambda_{X'}^*(w) = \lambda_X^*(w)$.

Abel & Reineke prove the correctness of their approach in [2].

To prove the correctness of our extension, we have to prove the following theorem:

Theorem 1. *Our extension of Abel & Reineke's algorithm learns a Mealy machine model X' that is minimal and right-equivalent to X in the composition $A \gg X$, given that a Mealy machine model of A and the input alphabet of X are known, and given that it is possible to answer output and equivalence queries for the composition $A \gg X$.*

The requirement that the model X' our extension learns is right-equivalent to X in the context of A can be rephrased as follows: X and X' must generate the same output sequence for all input sequences for X which can be generated as output of A. We use this set of input sequences as the set S in Lemma 1. For the system X in the lemma, we use the X we want to learn in our composition.

The only remaining requirement for applying the lemma is that we can answer output and equivalence queries for X considering only the subset of input sequences S. We argue that this requirement is satisfied using Lemma 2 and Lemma 3.

Lemma 2. *Using our extension to Abel & Reineke's algorithm, we can correctly answer output queries for X for any input sequence in S.*

We realize output queries for X using the construction described in Section 3.4. Let w be any input sequence in S. By the definition of the composition, if, in the composition, A outputs all symbols from w in sequence (possibly along with hidden outputs), X is given the input sequence w. The output of the composition will then be $\lambda^*_X(w)$ (interleaved with the hidden outputs, if any). Hence, removing all hidden outputs from this output sequence will indeed result in the correct answer to $\lambda^*_X(w)$.

Our approach to realizing output queries causes A to output the sequence w (possibly with hidden outputs) by using the transitive closure to find an input sequence for A such that it does so, and using this in an output query for the composition. We now need to argue that our method using the transitive closure indeed generates an input sequence for A such that it outputs w (possibly with hidden outputs).

In the transitive closure machine, from any state, it is possible to use any transition that is possible in A, either from the same state, or from a state that is reachable using hidden transitions. It is constructed by finding for each state the set of states that is reachable from that state with hidden transitions using the Floyd-Warshall algorithm. The correctness of this reachability analysis follows from the correctness of the Floyd-Warshall algorithm.

The transitions in the transitive closure machine are annotated with the input symbols from the hidden transitions in A necessary to reach the state where the transition with the desired visible output symbol can be used. Therefore, if a path can be found in the transitive closure machine with w as the visible part of its output sequence, the input sequence corresponding

to this path (including the annotations) is an input sequence for which A generates an output sequence with w as its visible part. If, on the other hand, no such path can be found, then no input sequence for A exists such that it generates w.

This concludes the reasoning behind Lemma 2.

Lemma 3. *Using our extension to Abel & Reineke's algorithm, we can answer equivalence queries for X for any hypothesis H which will return a counterexample if one exists in S, and indicate that they are equivalent otherwise.*

Our algorithm answers equivalence queries for X by translating them into equivalence queries for $A \gg X$. If H is not right-equivalent to X in the context of A, then as a consequence of the definition of right-equivalence, $A \gg H$ is also not equivalent to $A \gg X$. But if H and X are equivalent, then $A \gg H$ must also be equivalent to $A \gg X$.

A counterexample for the hypothesis $A \gg H$ necessarily corresponds to a counterexample for an equivalence query which compares H to X, as a difference for the input sequence v between $\lambda^*_{A \gg H}(v)$ and $\lambda^*_{A \gg X}(v)$ can only be caused in the right machine, since the left machine is the same. The right machines only operate on the output symbols of A which are in their input alphabet. Therefore, the sequence of symbols in the input alphabet of X in the output sequence of A when it processes the counterexample distinguishes H and X.

This concludes the reasoning behind Lemma 3.

We have now satisfied all requirements to apply Lemma 1. We use Abel & Reineke's algorithm to learn a model of X'. By this lemma, the resulting model is both minimal and right-equivalent to X. This concludes the reasoning behind Theorem 1.

Chapter 4

Experiments

4.1 Test cases

To find out how the modified algorithm performs in practice, we used experiments to compare the following implementations when learning compositions of Mealy machines:

- Our extension of Abel & Reineke's algorithm, implemented by modifying the source code of their original implementation;
- Abel & Reineke's original implementation of their algorithm;
- A variant of Angluin's L^* algorithm for learning Mealy machines [15] for learning the entire composition as a black box, implemented in the LearnLib library [9].

Two sets of experiments were used.[1]

In the first set of experiments, we compared the run time and the number of output queries used by the implementations when learning real-world models. We used three Mealy machines representing *queues* of capacity 1 to 3 as left machines. Figure 4.1 shows an example of such a queue Mealy machine. For each composition we tested, we generated a queue Mealy

[1]The tests were run on a VirtualBox virtual machine with 4 GB of memory and a 1.7 GHz Intel Core i7 processor running Ubuntu 18.04.2 LTS.

Model	Number of states	Input alphabet size	Output alphabet size
m217	13	25	13
m34	115	20	17
m85	2220	48	38

Table 4.1: Specifications of the industrial models.

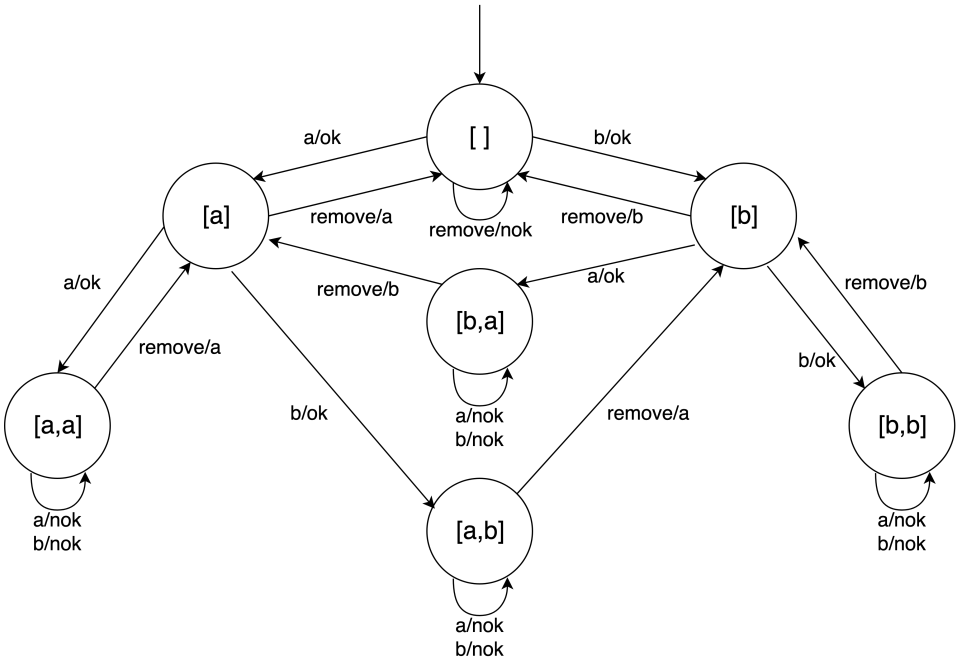


Figure 4.1: A queue Mealy machine that can contain at most two elements from $\{a, b\}$. Instead of $\{a, b\}$, the queue Mealy machines we used could contain any symbol from the input alphabet of the used right machine.

machine which could "contain" any input symbol of the used right machine. As the right machines, we used three models from the company ASML (called $m217$, $m34$ and $m85$). These models from ASML were provided for the RERS 2019 challenge [10].[2] Table 4.1 lists some properties about the models from ASML. All nine combinations of left and right machines were tested.

The number of states of a queue machine which can contain at most n elements from a set of size s is $s^0 + s^1 + \cdots + s^n$. The queue can contain 0 to n elements and each queue position may contain any element from the set. Hence, the number of states of the queue is exponentially related to the number of elements it can contain. In the following, the size of a queue refers to the number of elements it can contain.

The purpose of the second set of experiments was to compare the number of output queries each tested algorithm uses as the size of the left machine increases. We again used Mealy machines representing queues of varying size as the left machines, but we only used the Mealy machine which can be seen in Figure 1.3 as the right machine.

4.2 Implementations

The three tested algorithms are designed to learn different configurations of Mealy machines. The implementations actually learned the following:

- Our extension of Abel & Reineke's algorithm will consider the `ok` and `nok` outputs of the queues as hidden outputs. It will learn a model of the right machine in the context of the composition.
- Abel & Reineke's original algorithm cannot handle hidden outputs. It assumes that all outputs of the queue are passed to the right machine, including the `ok` and `nok` symbols. Learning the right machine in this setup results in a Mealy machine similar to the actual right machine but with loop transitions on each state which echo the `ok` and `nok` inputs.
- LearnLib's implementation will learn the entire composition as a black box.

For all three implementations, we realized output queries by running them on a known Mealy machine of the SUL in memory. In a real use case, this is not possible as such a model would not be available. (Having a correct model defeats the purpose of learning it.) In our setup, however, implementing the output queries this way allowed us to run our experiments

[2] The ASML models were downloaded from `http://rers-challenge.org/2019/index.php?page=industrialTrainingPhase`.

faster. This made it possible to obtain statistics for learning machines that would have taken too long to learn otherwise.

All three implementations cache the results of output queries. This prevents the learner from asking the same output queries more than once.

In case a model of the to-be-learned system is not already available, equivalence queries can be implemented in a number of different ways. To find an input sequence on which the hypothesis and the actual machine differ (or conclude that they are equivalent), a number of output queries must be asked to both the hypothesis and the actual machine. This can be done with a large number of completely random input sequences or in a more sophisticated way. The time taken may vary depending on the implementation choice. Because this thesis does not prescribe the way equivalence queries are approximated, they are performed by simply comparing the hypothesis to an already available correct model of the reference machine. While this causes the observed run times to be much shorter than in a real use case, it makes it possible to compare the algorithms more efficiently.

4.3 Results

Figure 4.2 shows the run times of the implementations for a queue size from 1 to 3 and the right machines. Hourglass icons in the chart indicate that there was no result yet after five minutes. Crosses indicate that the program crashed before reaching the five minutes mark. This chart is based on the values found in Table A.1 in Appendix A. Table A.2 shows the corresponding number of output queries used for the instances that returned a result.

As discussed, the output queries did not take much time in our experimental setup: they would take more time in most realistic scenarios. Figure 4.3 shows the run times from Figure 4.2 in case each output query would have taken an additional 0.1 second (i.e. 0.1 second has been added for each used output query).

Figure 4.4 shows a graph of the number of output queries used by the algorithm from this thesis and the LearnLib implementation for learning a composition of a queue and the Mealy machine in Figure 1.3, for a variable queue size (the second set of experiments). The corresponding data values may be found in Table A.3. The number of output queries of the algorithm from this thesis stays constant as the queue size increases; the output query count of Abel & Reineke's algorithm increases slightly for higher queue sizes. The LearnLib implementation stands out as the relation between the queue size and its output query count seems to be exponential.

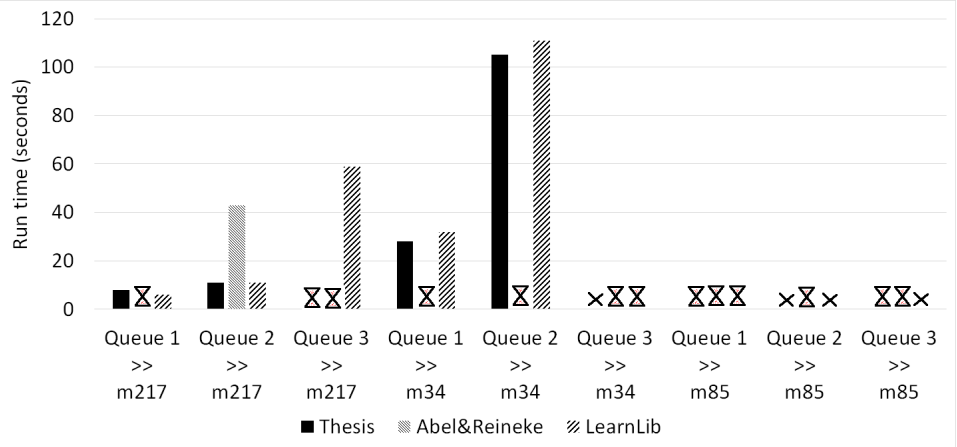


Figure 4.2: The run times of the algorithms for learning the specified compositions.

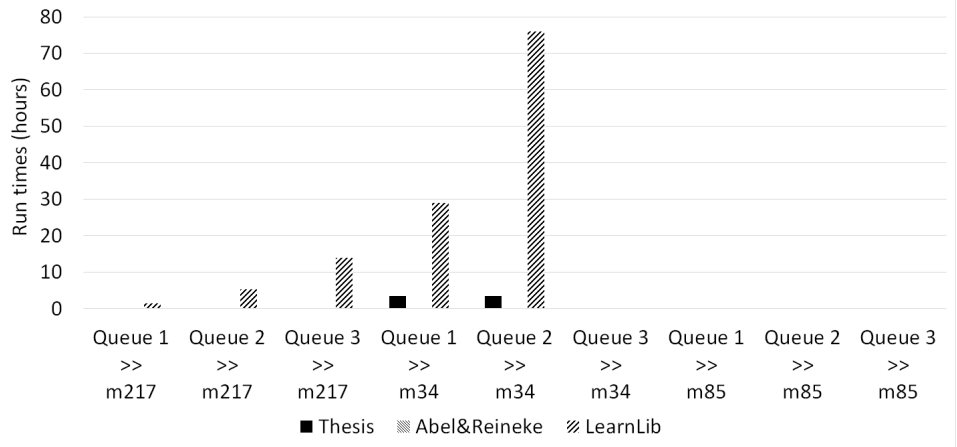


Figure 4.3: The run times for the first set of experiments, with a tenth of a second added for each used output query.

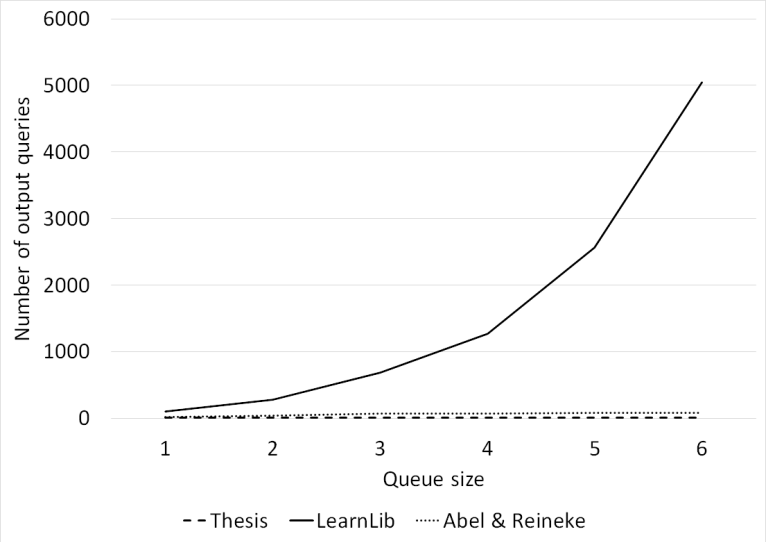


Figure 4.4: The number of output queries used by the algorithms for learning a composition of a queue and the Mealy machine in Figure 1.3, for an increasing queue size.

Chapter 5

Related Work

Petrenko & Avellaneda have proposed a method to learn a model of any component embedded in a known context using a SAT solver [13]. The unknown part in our composition might be seen as an embedded component. It is unclear how their algorithm performs in practice. It would be interesting to find out how their algorithm compares to ours. This has not been done in this thesis due to a lack of time.

In [11], active model learning is applied to a parallel instead of a serial composition topology. In a serial composition, the components of the composition each process the output of the previous Mealy machine. For example, in our composition, the right machine processes the output of the left machine. In contrast, in a parallel composition, the input of the composition is passed to each of its components. Output symbols of the composition can then be split up into an output symbol for each component of the composition.

In this thesis, we learn a model of the right machine in a composition using model learning. If we already have correct Mealy machine models of the entire composition and of the left machine, but not of the right machine, then a model of the unknown right machine can instead be derived from both known models. Solving such equations and others is described in [17].

If we only want to verify some properties of the composition (model checking), we might not have to learn a complete model at all. It was shown previously that some properties in model checking can already be checked without learning a complete model [12]. The model checking is already done using the hypotheses generated by the model learning algorithm. Optimizations for this task have been developed for situations where part of the system is known [6], such as in our composition. Alternatively, if we want to verify some properties of the right machine's behavior given that it is the right machine in a composition, we can use the combination of model checking and model learning described in [4].

Chapter 6

Conclusion

This thesis presented an extension of Abel & Reineke's algorithm that enables it to learn the right machine in our generalized composition. This could not be done using the original algorithm by Abel & Reineke or black-box algorithms without doing post-processing steps, as they were not designed to learn this type of composition. For the compositions we tested, our algorithm was faster than Abel & Reineke's algorithm.

In two test cases, our algorithm was slower than the LearnLib implementation for learning the composition as a black box. Although it uses fewer output queries, it still does not appear to scale to large systems. The cause of this may lie in the SAT-solving approach that Abel & Reineke's algorithm and our extension use to find the states of the Mealy machine.

If the output queries take more time, which is likely in a realistic scenario, our algorithm would be faster than the LearnLib implementation because it uses far fewer output queries. This confirms the experimental results from [2]. Figure 4.4 confirms that learning the composition requires more output queries than learning only the right machine as the size of the left machine increases. This can be explained by the fact that the number of states of the composition increases as the queue size increases. As a result, the LearnLib implementation needs to do more work as the queue size increases, whereas our algorithm still only needs to learn the unchanged right machine.

6.1 Future work

The run time of the learning algorithms in this thesis and in [2] may be improved for usage outside of the experimental setup by finding better ways to implement the equivalence queries for the right machine in compositions. In the experiments, equivalence queries were realized with prior knowledge of the model that must be learned. Without such a model, they can be realized by checking if the hypothesis composition and the SUL agree on

a large amount of random output queries. However, if the systems are not equivalent and the output queries are chosen completely at random, the probability that the implementation will find a distinguishing output sequence is low. To prevent it from wrongly assuming that the systems are equivalent, many output queries would be necessary, which leads to a long run time. Finding better ways to implement the equivalence query would reduce these problems.

Moreover, it would be useful if the algorithm was improved to make it scale better to large systems. In practice, systems are often much larger than the ones learned in our experiments.

As mentioned, it would also be interesting to compare Abel & Reineke's algorithm and our extended version to the algorithm in [13], in terms of run time, number of used output and equivalence queries and the composition topologies it can solve.

Furthermore, it would be useful to make the algorithm capable of learning in more composition topologies by making it more generic. For example, a composition where the left machine is unknown and the right machine is known; a composition where input symbols are either given to the left machine or to the right machine; or a composition where the right machine also sends some of its outputs to the left machine instead of only vice versa.

Bibliography

[1] Aarts, F., Schmaltz, J., and Vaandrager, F. W. Inference and abstraction of the biometric passport. In *Leveraging Applications of Formal Methods, Verification, and Validation* (2010), pp. 673–686.

[2] Abel, A., and Reineke, J. Gray-box learning of serial compositions of Mealy machines. In *NASA Formal Methods* (2016), pp. 272–287.

[3] Angluin, D. Learning regular sets from queries and counterexamples. *Information and Computation 75*, 2 (1987), 87–106.

[4] Cobleigh, J. M., Giannakopoulou, D., and Pasareanu, C. S. Learning assumptions for compositional verification. In *Tools and Algorithms for the Construction and Analysis of Systems* (2003), pp. 331–346.

[5] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.

[6] Elkind, E., Genest, B., Peled, D. A., and Qu, H. Grey-box checking. In *Formal Techniques for Networked and Distributed Systems* (2006), pp. 420–435.

[7] Hopcroft, J. E., and Ullman, J. D. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[8] Isberner, M. *Foundations of active automata learning: an algorithmic perspective.* PhD thesis, Technical University Dortmund, Germany, 2015.

[9] Isberner, M., Howar, F., and Steffen, B. The open-source LearnLib - A framework for active automata learning. In *Computer Aided Verification - 27th International Conference* (2015), pp. 487–495.

[10] Jasper, M., Mues, M., Murtovi, A., Schlüter, M., Howar, F., Steffen, B., Schordan, M., Hendriks, D., Schiffelers, R. R. H., Kuppens, H., and Vaandrager, F. W. RERS 2019: Combining synthesis with real-world models. In *Tools and Algorithms for the Construction and Analysis of Systems* (2019), pp. 101–115.

[11] Moerman, J. Learning product automata. In *Proceedings of the 14th International Conference on Grammatical Inference* (2018), pp. 54–66.

[12] Peled, D. A., Vardi, M. Y., and Yannakakis, M. Black box checking. *Journal of Automata, Languages and Combinatorics 7*, 2 (2002), 225–246.

[13] Petrenko, A., and Avellaneda, F. Conformance testing and inference of embedded components. In *Testing Software and Systems* (2018), pp. 119–134.

[14] Rivest, R. L., and Schapire, R. E. Inference of finite automata using homing sequences. *Information and Computation 103*, 2 (1993), 299–347.

[15] Shahbaz, M., and Groz, R. Inferring Mealy machines. In *FM 2009: Formal Methods* (2009), pp. 207–222.

[16] Vaandrager, F. W. Model learning. *Communications of the ACM 60*, 2 (2017), 86–95.

[17] Yevtushenko, N., Villa, T., Brayton, R. K., Petrenko, A., and Sangiovanni-Vincentelli, A. L. Compositionally progressive solutions of synchronous FSM equations. *Discrete Event Dynamic Systems 18*, 1 (2008), 51–89.

Appendix A

Appendix

	m217	m34	m85
1	8	28	timeout
2	11	45	error
3	timeout	error	timeout

(a) Using the new algorithm

	m217	m34	m85
1	timeout	timeout	timeout
2	43	timeout	timeout
3	timeout	timeout	timeout

(b) Using Abel-Reineke

	m217	m34	m85
1	6	32	timeout
2	11	111	error
3	59	timeout	error

(c) Using LearnLib

Table A.1: The run times in seconds for specified queue sizes and right machines.

	m217	m34	m85
1	8518	123565	
2	8518	123565	
3			

(a) Using the new algorithm

	m217	m34	m85
1			
2	3766		
3			

(b) Using Abel-Reineke

	m217	m34	m85
1	52137	1039265	
2	190066	2734674	
3	500154		

(c) Using LearnLib

Table A.2: The number of output queries for specified queue sizes and right machines.

	New algorithm	Abel-Reineke	LearnLib
1	10	23	103
2	10	38	277
3	10	73	691
4	10	76	1272
5	10	79	2568
6	10	82	5048
10	10	94	75188

Table A.3: The number of output queries used by the three algorithms when learning a composition of a queue and a two-state Mealy machine, for increasing queue sizes.