Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Supporting system design with automaton learning algorithms

Bachelor's Thesis

Author
Áron Barcsa-Szabó

Advisor
Rebeka Farkas
Dr. András Vörös

December 6, 2019

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Barcsa-Szabó Áron*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2019. december 6.

Barcsa-Szabó Áron
hallgató

Kivonat

Jelen dokumentum egy diplomaterv sablon, amely formai keretet ad a BME Villamosmérnöki és Informatikai Karán végző hallgatók által elkészítendő szakdolgozatnak és diplomatervnek. A sablon használata opcionális. Ez a sablon LaTeX alapú, a *TeXLive* TeX-implementációval és a PDF-LaTeX fordítóval működőképes.

Abstract

This document is a LaTeX-based skeleton for BSc/MSc theses of students at the Electrical Engineering and Informatics Faculty, Budapest University of Technology and Economics. The usage of this skeleton is optional. It has been tested with the *TeXLive* TeX implementation, and it requires the PDF-LaTeX compiler.

Chapter 1

Introduction

1.1 Overview

1.2 Problem Statement

1.3 Objective

1.4 Contribution

1.5 Outline

Chapter 2 provides an outline of the necessary technical background.

Chapter 2

Background

This chapter provides some theoretical background of the contributions presented in this thesis. First of all, the necessary basics of formal language and automaton theory are introduced, afterwards, automaton learning algorithms are discussed.

2.1 Basics of automaton theory

First, we introduce the fundamentals of formal language theory, on which automaton theory is based on.

2.1.1 Fundamentals of formal language theory

Atomic elements of formal languages are alphabets, characters and words.

Definition 1 (Alphabet). Let Σ be a finite, non-empty set. Σ is an alphabet, its elements are symbols or characters. ▪

Definition 2 (Word). If Σ is an alphabet, then any finite sequence comprised of the symbols of Σ are words (or Strings). Σ^n represents the set of every n length word consisting of symbols in Σ: $\Sigma^n : w_1 w_2 \dots w_n$, where $\forall 0 \leq i \leq n : w_i \in \Sigma$. The set of every word under an alphabet, formally $\bigcup_{n>0} \Sigma^n$ is denoted by Σ^*. The empty word is denoted by ϵ. ▪

Words can be constructed using other words. The following definition defines these relations.

Definition 3 (Prefixes, Substrings and Suffixes). Let an arbitrary $w = uvs$, where $w, u, v, s \in \Sigma^*$. u is the prefix, v is the substring, and s is the suffix of w. Formally:

- $w \in \Sigma^*$ is a prefix of $u \in \Sigma^*$ iff $\exists s \in \Sigma^* : s = wu$,
- $w \in \Sigma^*$ is a suffix of $u \in \Sigma^*$ iff $\exists s \in \Sigma^* : s = uw$,
- $w \in \Sigma^*$ is a substring of $u, v \in \Sigma^*$ iff u is the prefix and v is the suffix of w. ▪

Using these atomic elements of formal language theory, formal languages can be defined.

Definition 4 (Formal Language). An arbitrary set of words under an Alphabet Σ is a Language. Formally: $L \subseteq \Sigma^*$. ▪

Definition 5 (Prefix-closure). Let $L \subseteq \Sigma^*$ and $L' = \{u \in \Sigma^*, v \in \Sigma^* : uv \in L\}$. In other words, L' is the set containing all the prefixes of every word of L. L is prefix-closed if $L = L'$. ▪

Formal language theory is closely linked with automata theory, which we will introduce in the following subsection.

2.1.2 Properties of deterministic automata

Informally, automata are mathematical constructs which read characters from an input and classify them into "accepted" and "rejected" categories. A bit more precisely, automata consist of states, one of which is always active. Starting from an initial state, based on the inputs received, the automaton changes, transitions between states. Essentially, for each of the inputs, the automaton determines whether to keep, or change its current state. In order to determine if an input sequence should be accepted or not, some states are distinguished, accepting states. If after processing a sequence of inputs, the final state of the automaton is an accepting state, the input sequence is accepted. If not, the input is rejected.

One of the most simple automata is the Deterministic Finite Automaton.

Definition 6 (Deterministic Finite Automaton). A Deterministic Finite Automaton is a Tuple of $DFA = (S, s_0, \Sigma, \delta, F)$, where:

- S is a finite, non-epty set containing the states of the automaton,
- $s_0 \in S$ is the initial state,
- Σ is a finite Alphabet,
- $\delta : S \times \Sigma \to S$ is a transition function,
- $F \subseteq S$ is a set of the accepting states of the automaton. ▪

The deterministic in the name refers to a property of every state having exactly one transition for every input. In other words, every state must have every member of Σ listed in its transitions, meaning every state behaves deterministically for every possible input.

An example of a DFA (Deterministic Finite Automaton) from[8] can be seen in Example 1.

Example 1. *See Fig. 2.1. This example has four states, $S = \{q_0, q_1, q_2, q_3\}$ (hence $|S| = 4$). The initial state is marked by the start arrow, so $s_0 = q_0$. The alphabet can be inferred as $\Sigma = \{a, b\}$. Transitions are visualized as $q_0 \xrightarrow{a} q_1$ given by the transition function (in this example) $\delta(q_0, a) = q_1$. The complete transition function in a table form can be seen in Table 2.1. Finally, the accepting states, or in this case, accepting state of the automaton is $F = \{q_3\}$.*

The semantics of automata are defined via runs. A run of an automaton is to test for a certain input (word), if it is accepted or rejected. See Example 2.

Example 2. *In accordance with the transition function, a run of Fig. 2.1 with an input of $\{a, a, a\}$ would end in state q_3 meaning the input is accepted. A rejected input could be $\{a, b, b\}$, which would stop at state q_1, a non-accepting state. On deeper examination, one can see, that this automaton only accepts runs with inputs containing $4i + 3a$.*

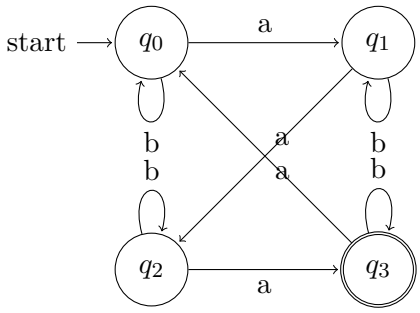


Figure 2.1. A simple DFA from [4].

δ	q_0	q_1	q_2	q_3
a	q_1	q_2	q_3	q_0
b	q_0	q_1	q_2	q_3

Table 2.1. The transition function of the automaton seen in Fig. 2.1

DFAs are useful to model system behavior based on inputs, but in order to work with reactive systems, we also need to handle outputs. Mealy machines are automata designed to communicate with output symbols instead of accepting and rejecting states.

Definition 7 (Mealy machine). A Mealy machine or Mealy automaton is a Tuple of $M = (S, s_0, \Sigma, \Omega, \delta, \lambda)$, where:

- S is a finite, non-empty set containing the states of the automaton,
- $s_0 \in S$ is the initial state,
- Σ is the input alphabet of the automaton,
- Ω is the output alphabet of the automaton,
- $\delta : Q \times \Sigma \to Q$ is the transition function and
- $\lambda : Q \times \Sigma \to \Omega$ is the output function. ▪

Mealy machines can be regarded as deterministic finite automata over the union of the input alphabet and an output alphabet with just one rejection state, which is a sink, or more elegantly, with a partially defined transition relation.

An example of a deterministic Mealy machine can be seen in Example 3.

Example 3. *An example of a deterministic Mealy machine can be seen in Fig. 2.2. The formal definition of the automaton can be seen below.*

- $S = \{a, b, c, d, d', e, f\}$
- $s_0 = a$
- $\Sigma = \{water, pod, button, clean\}$
- $\Omega = \{\checkmark, ☕, \star\}$

The transitions, as seen in Fig. 2.2 are visualized as $s_0 \xrightarrow{input/output} s_1$*, which denotes the machine moving from state* s_0 *to state* s_1 *on the specified input, while causing the specified output. Also, some simplifications are done, e.g. in this transition:* $d \xrightarrow{\{water,pod\}/\checkmark} d$ *we see a visual simplification of having both transitions merged to one arrow, this is only for visual convenience. Fig. 2.2 is also a great example of sinks, as seen in state f, the machine accepts anything, and never changes. This is a variation of the accepting state seen in DFAs.*

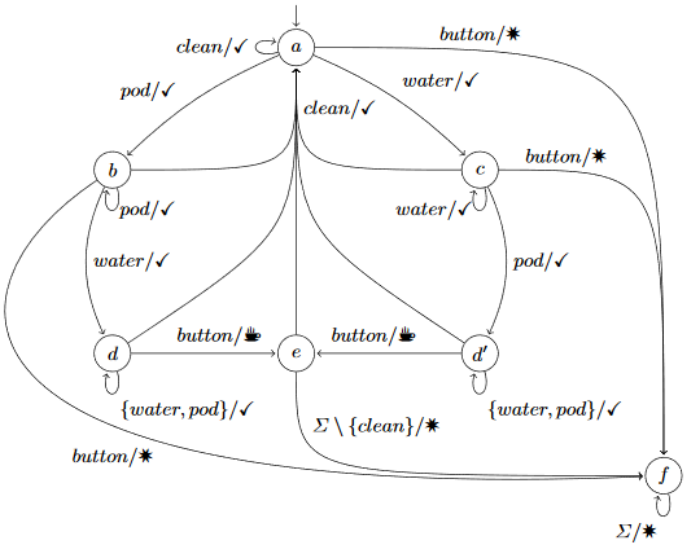


Figure 2.2. Mealy machine representing the functionality of a coffee machine.[8]

Since automaton-based formalisms deal with alphabets, formal language theory is essential not only to define them, but to construct them in a way that is efficient in practical applications. Often automata are used to design and analyze real-life systems. Naturally, questions of efficiency and correctness arise, which is why the relations of automatons and formal languages are discussed more in-depth in the following subsection.

2.1.3 Relations of formal languages and automata

Definition 8 (Recognized language of automata). The language $L \subseteq \Sigma$ containing all the accepted words by an automaton M is called the recognized language of the automaton. It is denoted by L(M) = L. ▪

Definition 9 (Regular language). A formal language L is regular, iff there is a Deterministic Finite Automaton M, for which L(M) = L, in other words, iff there is a DFA with the recognized language of L. ▪

Let us now introduce a semantic helper δ^* for both DFAs and Mealy machines. δ^* is an extension of the δ transition function, as $\delta^* : S \times \Sigma^* \to S$ defined by $\delta^*(s, \epsilon) = s$ and $\delta^*(s, \alpha w) = \delta^*(\delta(s, \alpha), w)$, essentially providing the state of the automaton after running an input sequence from a specified state.

Definition 10 (Myhill-Nerode relation). A DFA $M = (S, s_0, \Sigma, \delta, F)$ induces the following equivalence relation $\equiv_M$ on Σ^* (when L(M) = Σ):

$x \equiv_M y \iff \delta^*(s, x) = \delta^*(s, y)$

where $x, y \in \Sigma^*$. This means, that x and y are equivalent with respect to $\equiv_M$.[5]. ▪

In words, the Myhill-Nerode relation states, that two words are equivalent wrt. $\equiv_M$ iff runs of both words would end in the same state on the automaton M. The Myhill-Nerode relation is an equivalence relation with some additional properties[5], which can be seen in the following.

- The properties of equivalence relations:
  - Reflexivity: $x \equiv_M x$.
  - Symmetry: $x \equiv_M y \implies y \equiv_M x$.
  - Transitivity: if ($x \equiv_M y$ and $y \equiv_M z$) $\implies x \equiv_M z$.
- Right congruence: $\forall x, y \in \Sigma^* : (x \equiv_M y \implies \forall a \in \Sigma : xa \equiv_M ya)$
  also, by induction, this can be extended to:
  $\forall x, y \in \Sigma^* : (x \equiv_M y \implies \forall w \in \Sigma^* : xw \equiv_M yw)$.
- It respects membership wrt. R:
  $\forall x, y \in \Sigma^* : x \equiv_M y \implies (x \in R \iff y \in R)$.
- $\equiv_M$ is of finite index, has finitely many equivalency classes. Since for every state $s \in S$, the sequences which end up in s are in the same equivalence class, the number of these classes is exactly $|S|$, which is a finite set.

Using this relation, we can introduce the Myhill-Nerode theorem, which neatly ties together the previous definitions.

Theorem 1 (Myhill-Nerode theorem[5][7]). Let $L \subseteq \Sigma^*$. The following three statements are equivalent:

- L is regular.
- there exists a Myhill-Nerode relation for L.
- the relation $\equiv_L$ is of finite index.

For proof, see [5][7]. ▪

The same concepts can be applied to Mealy machines, which are somewhat more complex in this regard. As before, a semantic helper is needed similar to δ^*, but considering the output function of Mealy machines. $\lambda^* : S \times \Sigma^* \to \Omega$, defined by $\lambda^*(s, \epsilon) = \varnothing$ and $\lambda^*(s, w\alpha) = \lambda(\delta^*(s, w), \alpha)$.

When monitoring the behavior of Mealy machines, one of the most important metrics given an input is the specific output given by the input. The behavior of a Mealy machine, a specific run of it, has a pattern of $i_1, o_1, i_2, o_2, .., i_n, o_n$, where i are inputs and o are outputs. In order to characterize these runs, we actually do not need every output from this pattern, we only need the final one. Also note, that essentially the final output of a run is given by $\lambda^*(s_0, inputs)$. Let us introduce a $[\![M]\!]$: $\Sigma^* \to \Omega$ semantic functional as $[\![M]\!](w) = \lambda^*(s_0, w)$. This provides the final output given by a run of an automaton for an input sequence w. Using $[\![M]\!]$, the behavior of Mealy machines can be captured, as discussed in the following.

Example 4. *Given the Mealy machine $M_{coffeemachine}$ in Fig. 2.2, the runs:*
<clean, ✓>,
<pod water button, ☕>
are in $[\![M_{coffeemachine}]\!]$, since the given input words cause the corresponding outputs, while the runs
<clean, ☕> and
<water button button, ✓>
are not, since these input sequences do not produce those outputs.

Similarly to the Myhill-Nerode relations in DFAs, equivalence relations over the $P : \Sigma^* \to \Omega$ functional can be introduced, where P is an abstraction of $[\![M]\!]$ that can be applied to any state, rather than just the initial state.

Definition 11 (Equivalence of words wrt. $\equiv_P$[8]). Given a Mealy machine $M = (S, s_0, \Sigma, \Omega, \delta, \lambda)$, two words, $u, u' \in \Sigma^*$ are equivalent with respect to $\equiv_P$:
$u \equiv_P u' \iff (\forall v \in \Sigma^* : P(s, uv) = P(s, u'v))$.
We write [u] to denote the equivalence class of u wrt. $\equiv_P$. ▪

This definition is more along the lines of the right congruence property observed in the Myhill-Nerode relations. The original formalism: $u \equiv_P u' \iff P(s, u) = P(s, u')$ of the Myhill-Nerode relation still stands as a special case of the above definition: if $v = \epsilon$ and $v' = \epsilon$, $P(s, uv) = P(s, u)$ and $P(s, u'v) = P(s, u')$.

Example 5. *Taking Fig. 2.2 as an example, the following words are equivalent wrt. $\equiv_{[\![M]\!]}$:*

$$\begin{array}{ll} & water, pod \\ \equiv_{[\![M]\!]} & water, water, pod \\ \equiv_{[\![M]\!]} & pod, pod, water. \end{array}$$

The first two of $\equiv_{[\![M]\!]}$ are straightforward, since both words lead to the same state, d′, while the third input ends in state d. Observably, state d and d′ wrt. outputs operate exactly the same regardless of continuation, hence the equivalence holds.

Theorem 2 (Characterization theorem[8]). Iff mapping $P : \Sigma^* \to \Omega$ $\equiv_P$ has finitely many equivalence classes, there exists a Mealy machine M, for which P is a semantic functional.

Proof($\Longleftarrow$): As seen in the case of the Myhill-Nerode finite index property for DFAs, same states in Mealy machines will obviously be in same equivalence classes. This implies, that the number of classes in (or in other words, the index of) $\equiv_P$ is at most the number of states the Mealy machine contains, which is finite by definition.
Proof($\Longrightarrow$): Consider the following Mealy machine: $M_P = (S, s_0, \Sigma, \Omega, \delta, \lambda)$:
-S is given by the equivalence classes of $\equiv_P$.
-s_0 is given by $[\epsilon]$.
-δ is defined by $\delta([u], \alpha) = [u\alpha]$.
-λ is defined by $\lambda([u], \alpha) = o$, where $P(u\alpha) = o$.
A Mealy machine constructed this way fulfills what the theorem states, P is a semantic functional of it, in other words, $[\![M]\!] = \mathrm{P}$. ▪

With this theorem, regularity for mappings $P : \Sigma^* \to \Omega$ can be defined. A $P : \Sigma^* \to \Omega$ mapping is regular, iff there is a corresponding Mealy machine for which $[\![M]\!] = \mathrm{P}$, or equivalently, if P has a finite number of equivalence classes, analogously to the previously seen "classical" regularity.

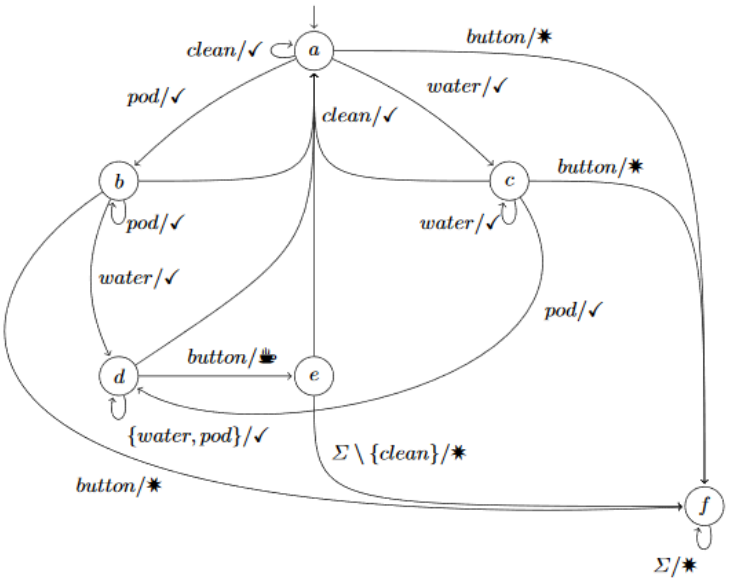


Figure 2.3. Minimal version of the Mealy machine seen in 2.2

2.1.4 Minimization of automata

The introduction of regularity is useful in the construction of automata, specifically, the construction of canonical automata.

Definition 12 (Canonical automaton (Minimal automaton)). An automaton M is canonical (i.e. minimal) iff:

- every state is reachable: $\forall s \in S : \exists w \in \Sigma^* : \delta^*(s_0, w) = s$,
- all states are pairwisely separable, in other words behaviorally distinguishable. For Mealy machines, this is formalized as: $\forall s_1, s_2 \in S : \exists w \in \Sigma^* : \lambda(s_1, w) \neq \lambda(s_2, w)$ ▪

The minimal version of the Mealy machine in Fig. 2.2 can be seen in Fig. 2.3.

Constructing automata to be canonical, especially in the case of Mealy machines is important with regards to efficiency and is the backbone of automaton learning. The next proposition comes straightforward from the previously presented characterization theorem.

Proposition (Bounded reachability[8]): Every state of a minimal Mealy machine with n states has an access sequence, i.e., a path from the initial state to the given state, of length at most n-1. Every transition of the model can be covered by a sequence of length at most n from the initial state.

The process of constructing automata uses the concept of partition refinement. It works based on distinguishing suffixes, suffixes of words which mark, witness the difference between two access sequences. The following notion is introduced to formalize this.

Definition 13 (k-distinguishability[8]). Two states, $s, s' \in S$ are k-distinguishable iff there is a word $w \in \Sigma^*$ of length k or shorter, for which $\lambda^*(s, w) \neq \lambda^*(s', w)$. ▪

Definition 14 (exact k-distinguishability). Two states, $s, s' \in S$ are exact k-distinguishable, denoted by $k^=$ iff s and s' are k-distinguishable, but not (k-1)-distinguishable ▪

Essentially, if two states, s and s' are k-distinguishable, then when processing the same input sequence, from some suffix of the word w with length at most k, they will produce differing outputs. Using this, we can observe, that whenever two states, $s_1, s_2 \in S$ are (k+1)-distinguishable, then they each have a successor s_1' and s_2' reached by some $\alpha \in \Sigma$, such that s_1' and s_2' are k-distinguishable. These successors are called α-successors. This suggests, that:

- no states are 0-distinguishable and
- two states s_1 and s2 are (k+1)-distinguishable iff there exists an input symbol $\alpha \in \Sigma$, such that $\lambda(s_1, \alpha) \neq \lambda(s_2, \alpha)$ or $\delta(s_1, \alpha)$ and $\delta(s_2, \alpha)$ are k-distinguishable.[8]

This way, if we have an automaton M, we can construct its minimal version, by iteratively computing k-distinguishability for increasing k, until stability, that is until the set of exactly k-distinguishable states is empty.

Example 6. *Given the Mealy machine seen in Fig.2.2, we can use k-distinguishability to refine its partitions. The initial state, the initial partition would be:*
$P_1 = \{a, b, c\}, \{d, d'\}, \{e\}, \{f\}$
since when k=1, a, b and c are not 1-distinguishable, but d and d' separate on the behavior of the button input, while e and f are separated by the suffix clean. Let's see the k=2 scenario.
$P_2 = \{a\}, \{b\}, \{c\}, \{d, d'\}, \{e\}, \{f\}$
Here, water and pod separate a, b and c, while d and d' can still no longer be separated. If observed, even if k is increased, d and d' can not be refined. This means, that they are indistinguishable, they can be merged together without altering behavior. This shows the process of acquiring the minimal machine seen in Fig. 2.3.

The process explained in Example 6 is partition refinement, the exact algorithm and proof of its validity can be seen in [8]. Partition refinement is a version of the minimization algorithm for DFAs proposed by Hopcroft[2].

Let us define one last relation which will be useful in the next section to compare automata minimization and automata learning.

Definition 15 (k-epimorphisms). Let $M = (S, s_0, \Sigma, \Omega, \delta, \lambda)$ and $M = (S', s_0', \Sigma, \Omega, \delta', \lambda')$ be two Mealy machines with shared alphabets. We call a surjective function $f_k : S \to S'$ existential k-epimorphism between M and M', if for all $s' \in S', s \in S$ where $f_k(s) = s'$ and with any $\alpha \in \Sigma$, we have: $f_k(\delta(s, \alpha)) = \delta'(s', \alpha)$, and all states, that are mapped by f_k to the same state of M' are not k-distinguishable. ▪

It is straightforward to establish that all intermediate models arising during the partition refinement process are images of the considered Mealy machine under a k-epimorphism, where k is the number of times all transitions have been investigated.[8] Essentially this establishes P_1 and P_2 from Example 6 as images of the Mealy machine seen in Fig. 3 under k-epimorphisms where k=1 and k=2 respectively.

Active automaton learning algorithms operate in a similar way, but they do not have access to the automata they are learning.

2.2 Automaton learning

Automaton learning is modeling a system without having specific knowledge of its the internal behavior. To accomplish this, a model needs to be inferred by observing

the external behavior of the system. This learned model is, as the name suggests, an automaton.
Formally: Active automata learning is concerned with the problem of inferring an automaton model for an unknown formal language L over some alphabet Σ[3].

In order to monitor a system, a way of access to its behavioral information is required. There are two approaches, which separate the two types of automaton learning as well.

Passive automaton learning In case of passive automaton learning, the gathering of information is not part of the learning process, but rather a prerequisite to it. The learning is performed on a pre-gathered positive an/or negative example set of the systems behavior. In passive automaton learning, the success of the process is determined not only by the efficiency of the algorithm, but the methodology and time used to gather the data.

Active automaton learning In case of active automaton learning, the behavioral infromation is gathered by the learning algorithm in an "active" way via queries. In order to accomplish this, learning is separated to two components: the learner, which learns, and the teacher, which can answer questions about the system under learning.

Active automaton learning follows the MAT, or the Minimally Adequate Teacher model proposed by Dana Angluin[1]. It separates the algorithm to a learner and a teacher, where the teacher can only answer the minimally adequate questions needed to learn the system. These two questions, or queries are are follows:

Membership query Given a $w \in \Sigma^*$ word, the query return the $o \in \Omega$ output corresponding to it, treating the word as a string of inputs. We write $mq(w) = o$ to denote that executing the query w on the system under learning (SUL) leads to the output o: $[\![SUL]\!](w) = o$ or $\lambda^*(s_0, w) = o$.

Equivalence query Given a hypothesis automaton M, the query tries to determine if the hypothesis is behaviorally equivalent to the SUL, and if not, finding the diverging behavior, and return with the example of it. We write $eq(H) = c$, where $c \in \Sigma^*$, to denote an equivalence query on hypothesis H, returning a counterexample c. The counter example provided is the sequence of inputs for which the output of system under learning and the output of the hypothesis differ: $[\![H]\!](c) \neq mq(c)$.

The learner component uses membership queries to construct a hypothesis automaton, then refines this hypothesis by the counterexamples provided by equivalence queries. Once counterexamples can not be found this way, the learners hypothesis is behaviorally equivalent to the SUL, the learning can terminate and the output of the learning is the current hypothesis.

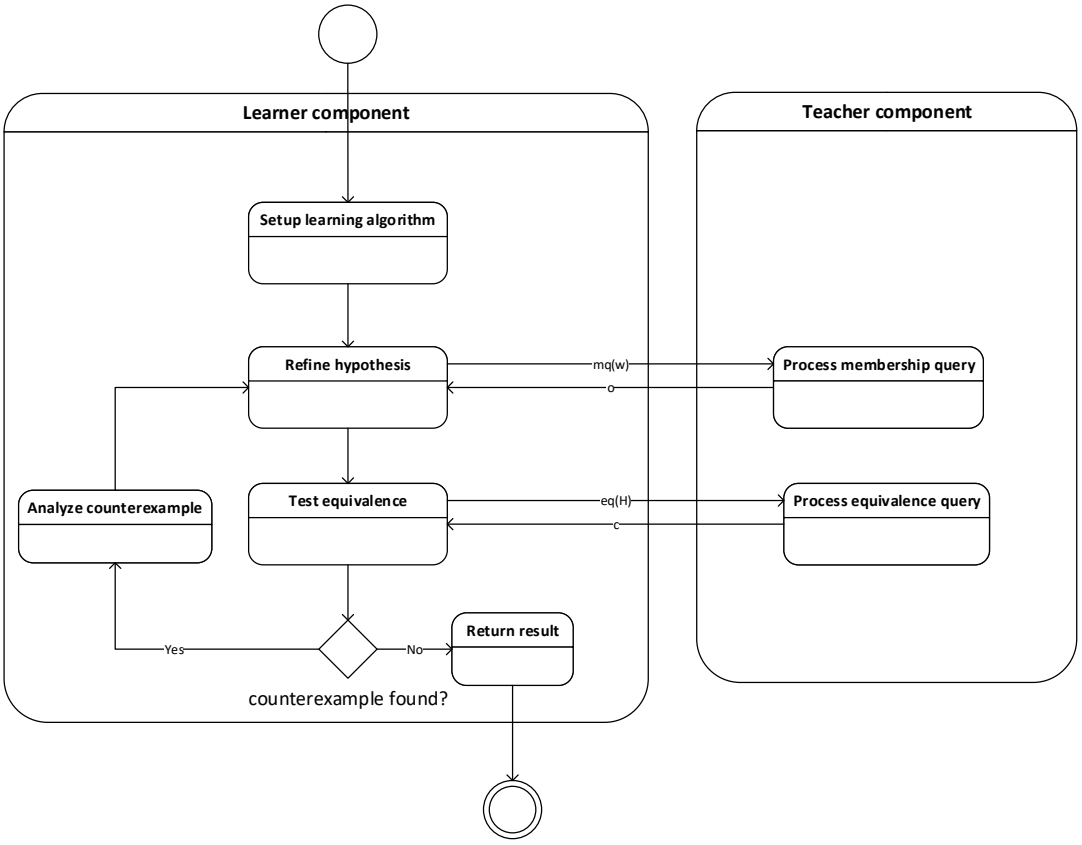


Figure 2.4. Abstract model of active automata learning algorithms

As seen on Fig. 2.4, the learning proceeds in rounds, generating and refining hypothesis models by exploring the SUL via membership queries. As the equivalence checks produce counterexamples, the next round of this hypothesis refinement is steered by the counterexamples produced.

Using an analogous strategy to the minimization of automatons seen in the previous section, starting only with a one state hypothesis automaton, we explore over all words in the alphabet in order to refine and extend this hypothesis. Here, there is a dual way of characterizing (and distinguishing) between states[8]:

- By words reaching them. A prefix-closed set of S_p words, reaching each state exactly once, defines a spanning tree of the automaton. This characterization aims at providing exactly one representative element from each class of $\equiv_P$ on the SUL. Active learning algorithms incrementally construct such a set S_p.
  This prefix-closedness is necessary for S_p to be a "spanning tree" of the Mealy machine. Extending S_p with all the one-letter continuations of words in S_p will result in the tree covering all the transitions of the Mealy machine. L_p will denote all the one-letter continuations that are not already contained in S_p.

- By their future behavior with respect to an increasing vector of words of Σ^*. This vector $< d_1, d_2, ..., d_k >$ will be denoted by D, and contains the "distinguishing suffixes". The corresponding future behavior of a state, here given in terms of its access

sequence $u \in S_p$, is the output vector$< mq(u * d_1), ..., mq(u * d_k) > \in \Omega^k$, which leads to an upper approximation of the classes of $\equiv_{[\![SUL]\!]}$. Active learning incrementally refines this approximation by extending the vector until the approximation is precise.

While the second characterization defines the states of the automaton, where each output vector corresponds to one state, the spanning tree on L_p is used to determine the transitions of these states. In order to characterize the relation between the SUL $M = (S, s_0, \Sigma, \Omega, \delta, \lambda)$ and the hypothesis model $M' = (S', s'_0, \Sigma, \Omega, \delta', \lambda')$ (note, that M and M' only share alphabets) let $D \subseteq \Sigma^*$. We call a surjective function $f_D : S \to S'$ existential D-epimorphism (surjective homomorphism) between M and M' if, for all $s' \in S'$ there exists an $s \in S$ with $f_D(s) = s'$ such that for all $\alpha \in \Sigma$ and all $d \in D$: $f_D(\delta(s, \alpha)) = \delta'(s', \alpha)$, and $\lambda^*(s, d) = \lambda^*(s', d)$.
Note, that active learning deals with canonical Mealy machines, in other words, the canonical form of SUL, and not, the perhaps much larger Mealy machine of SUL itself.

Since active learning algorithms maintain an incrementally growing extended spanning tree for $H = (S_H, h_0, \Sigma, \Omega, \delta_H, \lambda_H)$, i.e., a prefix-closed set of words reaching all its states and covering all transitions, it is straightforward to establish that these hypothesis models are images of the canonical version of SUL under a canonical existential D-epimorphism, where D is the set of distinctive futures underlying the hypothesis construction[8]

- define $f_D : S_{SUL} \to S_H$ by $f_D(s) = h$ as following: if $\exists w \in S_p \cup L_p$, where $\delta(s_0, w) = s$, then $h = \delta_H(h_0, w)$. Otherwise h may be chosen arbitrarily.
- It suffices to consider the states reached by words in the spanning tree to establish the defining properties of f_D. This straightforwardly yields:
  - $f_D(\delta(s, \alpha)) = \delta_H(h, \alpha)$ for all $\alpha \in \Sigma$, which reflects the characterization from below.
  - $\lambda^*(s, d) = \lambda^*_H(h, d)$ for all $d \in D$, which follows from the maintained characterization from above.[8]

In basic logic, D-epimorphisms and k-epimorphisms do not differ, they both deal with establishing constructed models being images of the model they are based on. D-epimorphisms could replace k-epimorphisms where $D = \Sigma^k$, it can be suggested, that there is no need to differentiate. However, there is in important difference of complexity between the two. While k-distinguishability supports polynomial time, black-box systems do not. Also, the "existential" in existential D-epimorphism is important: f_D must deal with unknown states, ones that haven't been encountered yet. This implies that characterization can only be valid for already encountered states.

Active learning algorithms can be proven correct using the following three-step pattern:

- Invariance: The number of states of each hypothesis has an upper bound of $\equiv_{[\![SUL]\!]}$.
- Progress: Before the final partition is reached, an equivalence query will provide a counterexample, where an input word leads to a different output on the SUL and on the hypothesis. This difference can only be resolved by splitting at least one state, which increases the state count.
- Termination: The refinement terminates after at most the index of $\equiv_{[\![SUL]\!]}$ many steps, caused directly by the described invariance and progress properties.

The following subsection introduces the first active automaton learning algorithm this thesis covers.

2.2.1 Direct Hypothesis Construction[6]

The Direct Hypothesis Construction algorithm, which hypothesis construction can be seen in Algorithm 1 follows the idea of the breath-first search of graph theory, it constructs the hypothesis using a queue of states, which is initialized with the states of the spanning tree to be maintained. Explored states are removed from this queue, while the discovered successors are enqueued, if they are provably new states. The algorithm starts with a one-state hypothesis, including only the initial state, reached by ϵ and D = Σ. It then tries to complete the hypothesis, for every state we determine the behavior of the state under D, we will call this behavior the extended signature of said state. States with a new extended signatures are provably new states, so to guarantee further investigation, we enqueue all their successors. Initially, $D = \Sigma$, so only the $1^{=}$-distinguishable states are revealed during the first iteration. This is extended straightforwardly to comprise a prefix closed set of access sequences. [8][6]

Algorithm 1: Hypothesis construction of the Direct Hypothesis Construction algorithm as seen in [8].

Input: S_p: a set of access sequences, D: a set of suffixes, an input alphabet Σ
Output: A Mealy machine $H = (S, s_0, \Sigma, \Omega, \delta, \lambda)$

```
initialize hypothesis H, create a state for all elements of S_p
initialize a queue Q with the states of H
while Q is not empty do
    s = dequeue state from Q
    u = access sequence from s_0 to s
    for d ∈ D do
        o = mq(ud)
        set λ(s, d) = o
    end
    if exists an s' ∈ S, where the output signature of s' is the same as s then
        reroute transitions of s to s' in H
        remove s from H
    else
        create and enqueue successors of s for every input in Σ into Q, if not
          already in S_p
    end
end
Remove entries of D \ Σ from λ
return H
```

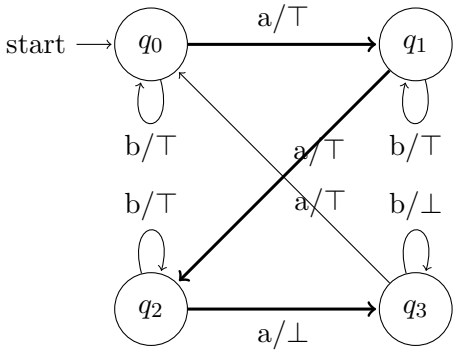
After the run of the Hypothesis construction seen in Algorithm 1, the output automaton H is used in an equivalence query eq(H) = c, to find if a counterexample c exists. If no counteraxamle can be found, the learning terminates, H is the learned automaton. Else, if a counterexample c is found, for which $\lambda_H(s_0, c) \neq mq(c)$, c is used to enlarge the suffixes in D and a new iteration of Algorithm 1 begins, using the now extended set D and all the access sequences found in the previous iteration, the current spanning tree S_p.

While the DHC algorithm is a straightforward implementation of active automata learning, it struggles with time complexity, it terminates after at most $n^3mk + n^2k^2$ membership

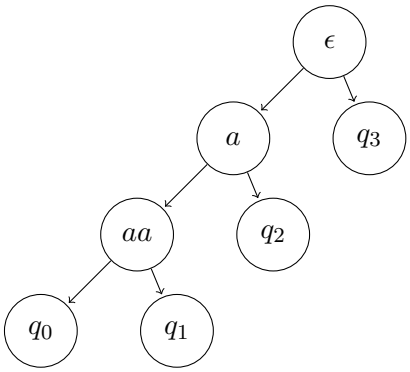
queries, and n equivalence queries, where n is the number of states in the final hypothesis, k is the longest set of inputs, and m is the length of the longest counterexample. This of course is far from efficient, the next subsection deals with a far more optimal algorithm, the TTT algorithm.

2.2.2 The TTT algorithm[4]

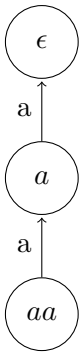
As seen in the Direct Hypothesis Construction algorithm, a large number membership queries are required in order to learn an automaton, causing a bottleneck on the scalability of the algorithm. The main cause of this complexity issue is the way counterexamples are treated. In order to ensure every state is properly found, *all* suffixes of the counterexample are added to the algorithms structure (denoted D in the previous subsection), which, in the next iteration of the hypothesis construction, has a huge impact on the number of membership queries. TTT resolves this inefficiency by fine-tuning exactly what needs to be further investigated by membership queries.



(a) A Mealy-machine version of the DFA seen in Fig. 2.1. Constructed simply by the following rule: if a transition ends in q_3, the output is $\bot$, else the output is $\top$.



(b) Final discrimination tree.



(c) Discriminator trie of the final hypothesis.

Figure 2.5. The three namesake trees of the TTT algorithm, (a) being a spanning **T**ree, (b) a disctimination **T**ree, and (c) a discriminator **T**rie.

Example 7. *As an example, we'll use the automaton portrayed on Fig.2.5.a a Mealy machine version of the DFA previously seen in Fig.2.1. TTT keeps the notion of separating states by a S_p spanning tree, a prefix-closed set of words from Σ^*, this tree defines the access sequences of each state. The spanning tree S_p is indicated by bold transitions in Fig.2.5.a. States are separated by the notion of discriminators. The algorithm maintains a set of discriminator suffixes D, but uses them differently from the DHC algorithm. D in TTT is used by a n-ary tree, where $n = |\Omega|$, so in this example a binary tree. We cal this tree a discrimination tree (seen in Fig.2.5.b), which maintains information about which inputs (discriminators) separate states: for every distinct pair of states, a separator can be obtained by looking at the label of the lowest common ancestor of the corresponding leaves. This way, the label of inner nodes act as discriminators. These labels form a suffix-close set and are stored in a trie seen in Fig.2.5.c. Each node of the trie is a word, which can be constructed by going up the tree towards the root.*

Discriminator trees, as seen in the previous example, are rooted n-ary trees, where n is the size of the output alphabet. Leaves are labeled by states of the automaton, while inner states are labeled by discriminators (suffixes). When adding a new word to the tree T, we sift the word $w \in \Sigma^*$ into T by starting at the root, and for every inner node $v \in D$, we branch depending on the output of $\lambda^*(wu)$. This limits the number of membership queries to the height of the tree, while DHCs limit was $S_p \times D$.

Key steps of the TTT algorithm:

- **Hypothesis construction.** The initial state is initialized and membership queried for all inputs, the results of which are sifted into the discriminator tree T.
- **Hypothesis refinement.** If a counterexample is found, the corresponding state is split (analogously to the DHC algorithm), in the discriminator tree, the leaf corresponding to this new state is split by a temporary discriminator $v \in \Sigma$.
- **Hypothesis stabilization.** The hypothesis provided by the previous example might contradict information of the discrimination tree. This step searches for counterexamples between them (without the need of equivalence queries), and if any such counterexample exist, states are split accordingly.
- **Discriminator finalization.** TTT treats discriminators derived directly from counterexamples temporary, keeping track of the maximal sub-tree of the discrimination tree containing temporary discriminators. This sub-tree is split as temporary discriminators are replaced with final ones. Only when this finalization happens, is an inner node of the discrimination tree considered a member of the discriminator trie, in other words, when finalizing a temporary discriminator, it is added to the discriminator trie.

A more thorough view on the TTT algorithm can be seen here[4].

Chapter 3

Contribution

The main contribution in this thesis is to provide an active automaton learning framework, which can be used to support system design and analysis. This framework, since used in system design, is needed to be easily modifiable and extensible, while having the capability of handling any variation of formalisms systems might use.

In order to tackle the problem of creating such a framework, I first needed to understand active automaton learning algorithms themselves.

3.1 Implementing a learning algorithm

After processing the background knowledge seen in [8], I decided to implement the most straightforward learning algorithm presented therein, the direct hypothesis construction algorithm.
The first step was determining the formalisms used in this implementation, more specifically, the input and the output formalisms. Active automaton learning algorithms essentially have two endpoints: one being the input, reached through the teacher component, and the other being the output, where the learned hypothesis automaton is returned. This is illustrated in Fig. 3.1. The teacher and learner component can handle abstraction, meaning the formalisms of the system under learning, or from now the input formalism, and the formalism of the learned hypothesis, or from now, the output formalism can both be arbitrary. Only implementing a specific algorithm, these

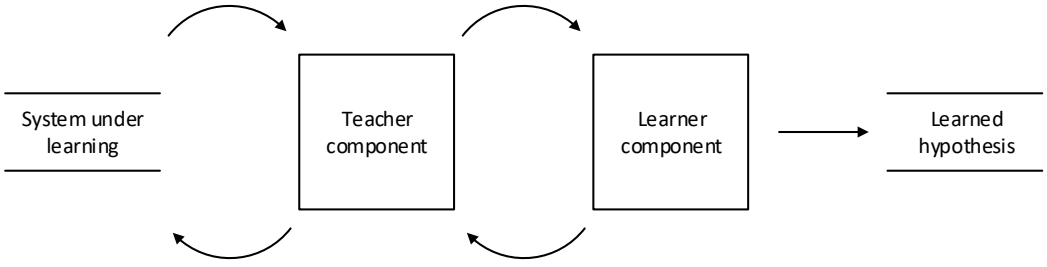


Figure 3.1. Data flow of active learning algorithms.

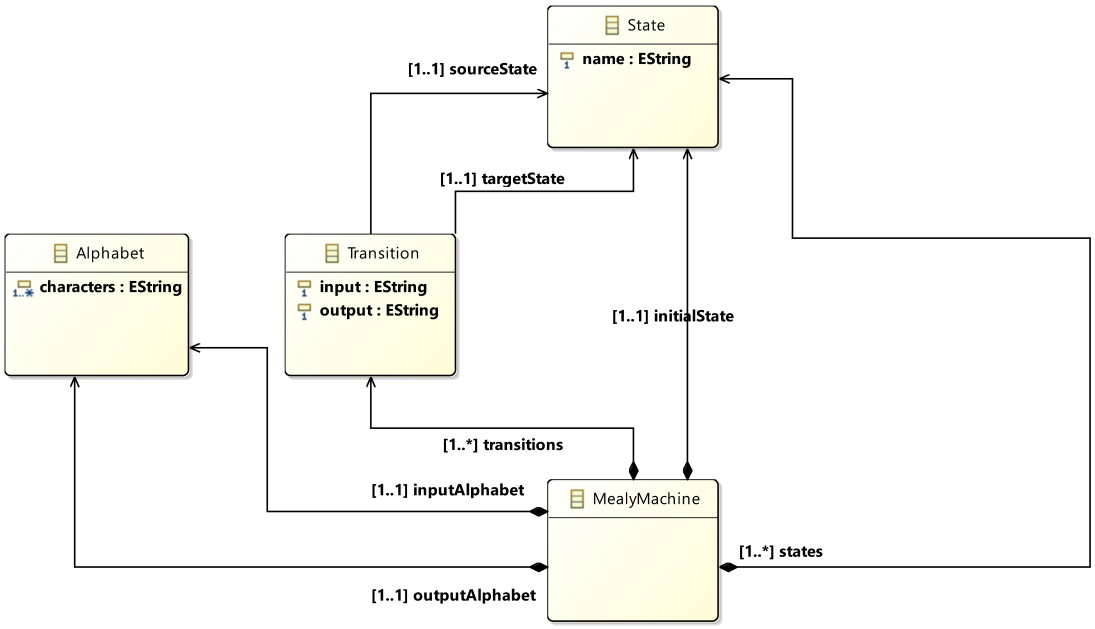


Figure 3.2. Ecore metamodel of Mealy machines.

formalisms needed to be specified.

Even if at first I was only implementing a single algorithm, the decisions on the formalism of this implementation was made with the future framework in mind. Since automaton learning algorithms can be implemented on any type of automata, the easiest method would've been to implement DHC using deterministic finite automata. However, real-life reactive systems usually are better modeled using Mealy machines, hence the implementation was made using Mealy machines as both input and output formalisms.
The first step of the implementation process was choosing the right tooling. This decision was also formed by the future system design perspective, which is why the Mealy machine implementation was done in Eclipse Modeling Framework.

Eclipse Modeling Framework (EMF), or more specifically, EMF core, is an "abstraction for describing, composing, and manipulating structured information", essentially a tool for system modeling and code generation implemented in Java. Figure 3.2 shows an UML class diagram of the metamodel I've created using the Ecore package of EMF core.

In text, the Ecore model I've created uses Strings as distinguishers (on code generation, EString is converted to java.lang.String). States of the automaton, i.e. State objects are differentiated based on the name field. Note, that this is a transition-driven model of mealy machines, the Transition class stores the source and target states of the transition, as opposed to some models, where states store their own transition information (successors, predecessors) like nodes in a graph. This decision was based on the DHC algorithm, which while learning, stores traversal information itself, rather than asking for it, which is why ease of access is preferred as opposed to efficiency.

Using EMF, I generated the class diagram seen in Fig. 3.2 into Java code. With this code, I implemented the direct hypothesis construction algorithm (in Java as well), without any framework whatsoever, only a single class accepting a MealyMachine object and constructing a Hypothesis MealyMachine object based off of it. In order to run and

test this implementation, an example was needed, for which I chose the automaton seen in 2.2. Programmatically creating this example as an object is not a scalable solution, espetially regarding the future framework to be built, so I used Xtext, another tool, to solve this issue.
Xtext is a framework for creating programming and domain-specific languages, which also has integration with EMF metamodels. Using this integration, I generated the Xtext grammar seen in Listing 3.1. In words, this grammar describes a textual grammar in which instances of the metamodel seen in Fig. 3.2 can be stored.

```
MealyMachine returns MealyMachine:
  'MealyMachine'
  '{'
    'initialState' initialState=State
    'states' '{' states+=State ( "," states+=State)* '}'
    'inputAlphabet' inputAlphabet=Alphabet
    'outputAlphabet' outputAlphabet=Alphabet
    'transitions' '{' transitions+=Transition ( "," transitions+=Transition)* '}'
  '}';

State returns State:
  {State}
  'State'
  name=EString;

Alphabet returns Alphabet:
  'Alphabet'
  '{'
    'characters' '{' characters+=EString ( "," characters+=EString)* '}'
  '}';

Transition returns Transition:
  'Transition'
  '{'
    'input' input=EString
    'output' output=EString
    'sourceState' sourceState=[State|EString]
    'targetState' targetState=[State|EString]
  '}';

EString returns ecore::EString:
STRING | ID;
```

Listing 3.1. Xtext grammar describing Mealy machines.

Utilizing the Xtext grammar in Listing 3.1, I created the input to be used by the implemented DHC, containing the formalized version of the automaton shown in Fig. 2.2. The input file can bee seen in Listing 3.2

```
  MealyMachine{
    initialState
    State q0 states { State q0, State q1, State q2, State q3}
    inputAlphabet Alphabet { characters { a , b } }
    outputAlphabet Alphabet { characters { top , bot } }
    transitions{
    Transition { input a output top sourceState q0 targetState q1 } ,
    Transition { input b output top sourceState q0 targetState q0 } ,
    Transition { input a output top sourceState q1 targetState q2 } ,
    Transition { input b output top sourceState q1 targetState q1 } ,
    Transition { input a output bot sourceState q2 targetState q3 } ,
    Transition { input b output top sourceState q2 targetState q2 } ,
    Transition { input a output top sourceState q3 targetState q0 } ,
    Transition { input b output bot sourceState q3 targetState q3 } } }
```

Listing 3.2. The Mealy machine seen in Fig.2.5.a in the form of the Xtext grammar described in Listing 3.1.

Using the input file in Listing 3.2 and converting it to a MealyMachine object using Xtext/EMF integration, the implemented DHC algorithm worked as intended. The execution

of the algorithm returned with a new MealyMachine hypotesis object, which was the minimal version of the input automaton (seen in Fig. 2.3). After the learning, the program wrote the learned hypothesis into a file in the same Xtext grammar the input was described in.
The overview of this implementation can be seen in Table 3.1.

Algorithm	Input formalism	Output formalism	Input source	Output source
Direct Hypothesis Construction	Mealy machine	Mealy machine	Xtext	Xtext

Table 3.1. Overview of the prototype DHC algorithm implementation.

I used the experience gained during this prototype implementation to extend it into a framework.

3.2 Automaton learning framework

After implementing a singular active learning algorithm using fixed formalisms, I started designing an abstract framework to implement any algorithm using arbitrary formalisms. My desing decisions are mostly illustrated by Fig. 2.4. The ”Learner” and the ”Teacher” components are both abstractions upon an algorithm, with two endpoints, one being the input, the other the output. Following this mindset, I first created a high-level model of the framework.

3.2.1 High-level overview

Since the implementation of the framework would be using Java as a language, the high-level view of it seen in Fig. 3.3 is an UML class diagram of the packages and the relations between them, essentially being an overview of the modularization of the framework.

Note, that when comparing Fig. 3.3 to Fig. 2.4, the data flows identically. The Learnable package containing the input formalisms, and the Hypothesis package containing the output formalisms are used by the teacher (Teacher package) and the learner (Algorithm package). The package not represented on Fig. 2.4, the Adapter package is used as an abstraction layer by which the algorithm and the teacher get separated from the input formalism. Since automaton learning algorithms have no direct access to the system under learning, they operate in a black-box way, this Adapter package is a useful addition, which unfortunately can not be used on the output layer. Hypothesis are directly accessed by the learning algorithms, they are constructed during the learning. While more specific abstractions were made (and can be seen later), no singular abstraction layer can be provided for every type of automaton and every type of learning algorithm the same way as for the input formalisms.

The relations between the packages (modules) are straightforward. Composition is used, to indicate, that there is no Algorithm (learner) without a Teacher, there is no Teacher, without an Adapter, and there is no Adapter, without an input, a Learnable, to adapt. Algorithms of course depend on Hypothesis, and because of equivalence queries,

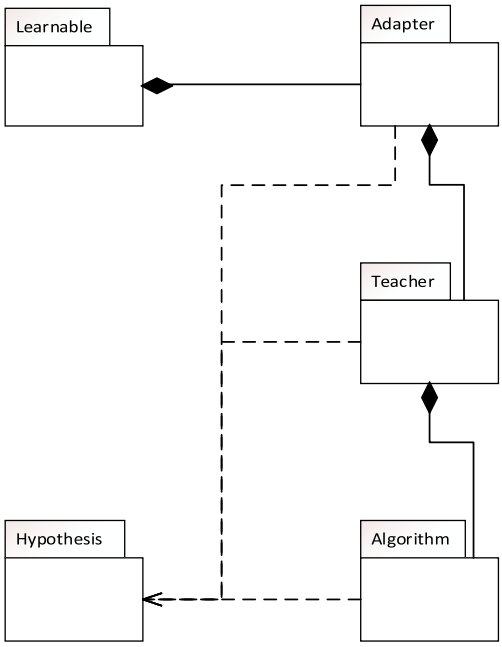


Figure 3.3. Structure and relations of the packages comprising the framework.

which are later queried through the Teacher and Adapter components, they both also have a dependency on the Hypothesis package.

3.2.2 Detailed overview of abstractions

The diagram in Figure 3.3 showed the modularization of the framework, which depend on the correct implementation of object-oriented architecture so the notated compositions and dependencies in Fig. 3.3 are satisfied. These abstractions are defined and implemented in Java as abstract classes and interfaces seen in Fig. 3.4. The detailed descriptions of these abstractions are as follows.

- **Learnable:** The *Learnable* interface is the input type used by the framework. It defines two generic parameters, I is the character type of the regular language the Learnable represents, while O is the character type by which actions are defined for the system under learning. The only method the interface defines is the *getOutput()* method, which returns the output O given by the system under learning for a specific sequence of inputs. Note, that while the word ”output” is used, the O it returns is the action the system takes for an input. If the system is represented as a DFA, this would be the state it moves to, if the system is represented as a Mealy machine, it would be an output.

- **Hypothesis:** The *Hypothesis* abstract class is the output type used by the framework. The generic parameters it takes are the automaton it uses (M), and the input (I), output (O), state (S) and transition (T) types used by M. The Hypothesis class does not enforce bounds on these parameters to allow flexibility in implementation. Beyond the getters, the *query()* method takes a sequence of inputs, and returns the

output given by the hypothesis automaton. Just as with the *Learnable* interface, this output represents action in the automaton.

- **LearnableAdapter:** The *LearnableAdapter* abstract class is the abstraction of the adaptation between any *Learnables* and *Hypotheses*. The generic parameters of it are the *Hypothesis* type (H) and the input, output types of H (HI, HO), similarly, the *Learnable* type (L) and the input, output types of L (LI, LO). The *LearnableAdapter* class stores a *Learnable*, the object which represents the data of the system under learning. It gives access to the system using the *membershipQuery()* method and the abstract *equivalenceQuery()* method. The abstract *covert..()* methods are used to do the actual adapting between *Hypotheses* and *Learnables* character by character.
- **Teacher:** The *Teacher* abstract class defines the "middle ground" between a learning algorithm and the input. The *Teacher* class as generic parameter, takes a *Hypothesis* (H) with input and outputs types of HI and HO, also a *LearnableAdapter* (LA) which has the same *Hypothesis* type as H. It defines two query types, *membershipQuery()* and *equivalenceQuery*, both of which it delegates to the current adapter object in its adapter field. The *Teacher class* is needed for easy extensibility, specifically, for any algorithm and input to work together without modification on them.
- **ActiveLearningAlgorithm:** The *ActiveLearningAlgorithm* class provides abstraction for active automaton learning algorithms. It defines only a *Hypothesis* (H), and its input types (HI, HO). The teacher field bounded to the *Hypothesis* H is used for queries, while the *execute()* method executes the algorithm and returns with the learned *Hypothesis*. Note, that the algorithm is fully separated from the system it is learning, it does not even know the generic type which with the communication, queries are executed.

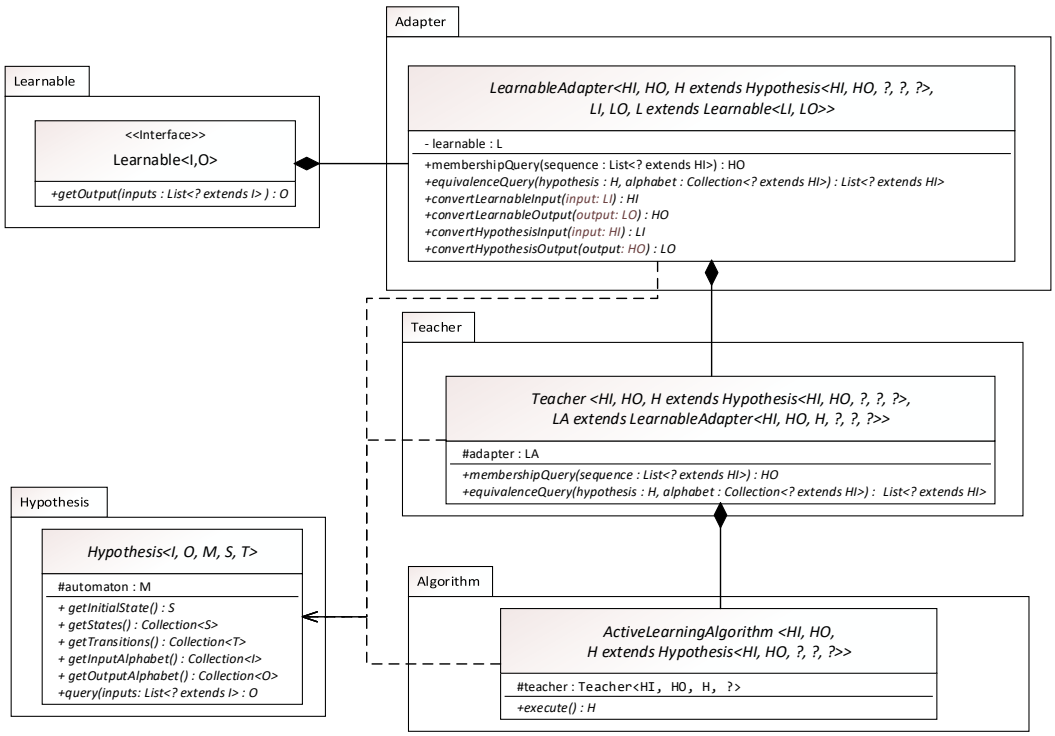


Figure 3.4. Overview of the abstract classes and interfaces of the framework.

3.2.3 Detailed overview of implementation

The detailed (but not exhaustive) class diagram of the frameworks implementation can be seen in Fig. 3.4. The figure is self-explanatory in some regards, while the generic specifics and implementation details are explained in the following.

- **StringSequenceLearnable:** The *StringSequenceLearnable* class is a realization of the Learnable interface, defining simply any type of learnable, which use Strings (java.lang.String) as both input and output types. It contains a simple implementation for describing behavior, accepting a String in the format of "|*input*1|*output*1|*input*2|*output*2|...|*inputn*|*outputn*|", and storing it in a HashMap. One example of this formalism can be seen in Fig. A.6.
- **MealyLearnable:** The *MealyLearnable* class is a *Learnable* which uses the EMF-modeled *MealyMachine* class seen in Fig. 3.2. Since this implementation of Mealy machines uses Strings as both input and output characters, it extends upon the generic bounds provided by the *StringSequenceLearnable* class.
- **DHCHypothesis:** The *DHCHypothesis* abstract class contains the abstractions the DHC algorithm needs to be separated from the implementation of the output formalism. Output here is also implementation-dependent, for DFAs it would be the state after running an input, for Mealy machines it would be output after running an input. Contrary to the *Adapter* layer of input formalisms, every algorithm must define its own abstract hypothesis type, an example being this (the *DHCHypothesis*) class.
- **DHCMealyMachineHypothesis:** The *DHCMealyMachineHypothesis* class straightforwardly extends *DHCHypothesis* using the EMF-modeled *MealyMachine* class seen in Fig. 3.2. Essentailly, *DHCMealyMachineHypothesis* is the Mealy machine extension of the DHCHypothesis abstract class, implementing every abstract method of both its superclasses.
- **StringSequenceAdapter:** The *StringSequenceAdapter* abstract class adapts *StringSequenceLearnable*s (bounds these using generics), while not defining any generic bounds to the *Hypothesis* it adapts to. This makes it possible to implement equivalance queries only once for every input formalism, using the abstract *convert...()* methods to be implemented by subclasses. The current implementation uses a brute-force method of comparing the outputs of the *Learnable* and the *Hypothesis* for every possible input under the input alphabet. This implementation can be seen in A.5.
- **StringSequenceToMealyAdapter:** The *StringSequenceToMealyAdapter* class bounds the *Hypothesis* parameters of the *LearnableAdapter* to *DHCMealyMachineHypothesis*, as the dependency relation indicates in Fig. 3.4. It only implements the *convert...()* functions.
- **MealyMachineTeacher:** The *MealyMachineTeacher* abstract class bounds the generic parameters of the *Hypothesis* (H) to *DHCMealyMachineHypothesis*, while leaving the *Learnable* (LA) parameter unbound, analogously to *StringSequenceAdapter*. This is done, so algorithm implementations are completely separated from input types.

- **MealyMachineTeacherStringSequenceImpl:** The *MealyMachineTeacherStringSequenceImpl* class defines the LA generic parameter to be a *StringSequenceAdapter*, and delegates both its methods to it.
- **DirectHypothesisConstruction:** The *DirectHypothesisConstruction* class implements the DHC algorithm using the *DHCHypothesis* abstract class to build its hypothesis. The *constructHypothesis()* method is an implementation of Algorithm 1, constructing a *DHCHypothesis* in a black-box way using the splitters field. The splitters field is initialized and constructed the way described in Algorithm 1, on the first run initialized by the input alphabet, then extended on hypothesis refinement. The *refineHypothesis()* method does exactly this: it takes a counterexample, and adds the suffixes of it to the *splitters*, so the next run of *constructHypothesis()* will split states as needed. The *execute()* method ties everything together by executing *constructHypothesis()*, equivalence queries and the *refineHypothesis()* method. The implementation of the *execute()* method can be seen in Fig. 3.3.

```
public DHCHypothesis execute() {
  List<? extends I> counterExample = null;
  DHCHypothesis<I, O, M, S, T> h = null;
  do {
    if(counterExample != null) {
      refineHypothesis(counterExample);
    }
    h = constructHypothesis();

    counterExample = teacher.equivalenceQuery(h, alphabet);
  }while(counterExample != null);

  return h;
}
```

Listing 3.3. The *execute()* function of the DHC algorithm implementation described in Section 3.2.3s *DirectHypothesisConstruction* item.

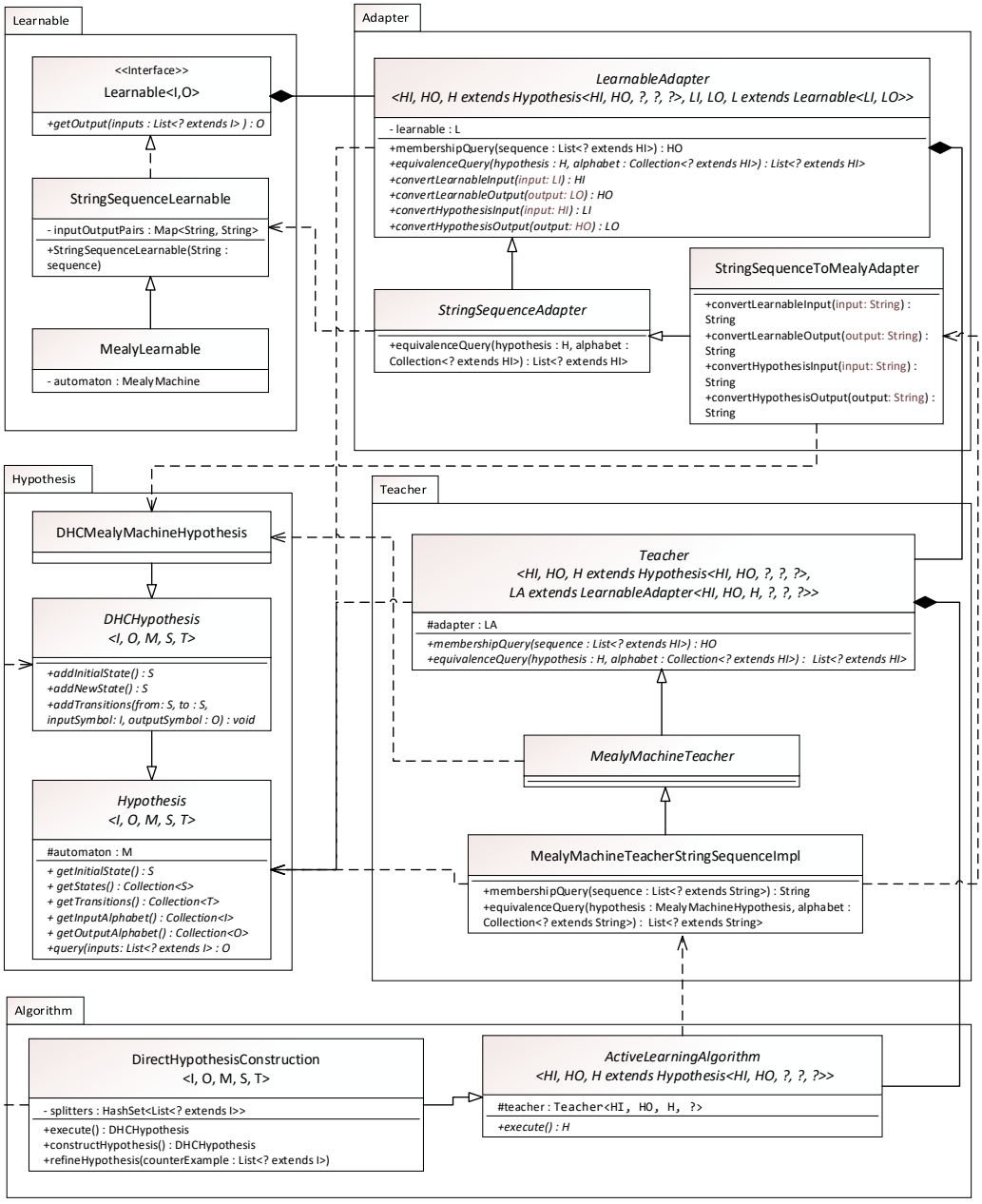


Figure 3.5. Non-exhaustive detailed overview of the implementation and structure of the framework.

Acknowledgements

Ez nem kötelező, akár törölhető is. Ha a szerző szükségét érzi, itt lehet köszönetet nyilvánítani azoknak, akik hozzájárultak munkájukkal ahhoz, hogy a hallgató a szakdolgozatban vagy diplomamunkában leírt feladatokat sikeresen elvégezze. A konzulensnek való köszönetnyilvánítás sem kötelező, a konzulensnek hivatalosan is dolga, hogy a hallgatót konzultálja.

Bibliography

[1] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87 – 106, 1987. ISSN 0890-5401. URL https://doi.org/10.1016/0890-5401(87)90052-6.

[2] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In Zvi Kohavi and Azaria Paz, editors, *Theory of Machines and Computations*, pages 189 – 196. Academic Press, 1971. ISBN 978-0-12-417750-5. DOI: https://doi.org/10.1016/B978-0-12-417750-5.50022-1. URL http://www.sciencedirect.com/science/article/pii/B9780124177505500221.

[3] Falk Howar and Bernhard Steffen. *Active Automata Learning in Practice*, pages 123–148. Springer International Publishing, Cham, 2018. ISBN 978-3-319-96562-8. DOI: 10.1007/978-3-319-96562-8_5. URL https://doi.org/10.1007/978-3-319-96562-8_5.

[4] Malte Isberner, Falk Howar, and Bernhard Steffen. The ttt algorithm: A redundancy-free approach to active automata learning. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, pages 307–322, Cham, 2014. Springer International Publishing. ISBN 978-3-319-11164-3.

[5] Dexter C. Kozen. *Myhill—Nerode Relations*, pages 89–94. Springer Berlin Heidelberg, Berlin, Heidelberg, 1977. ISBN 978-3-642-85706-5. DOI: 10.1007/978-3-642-85706-5_16. URL https://doi.org/10.1007/978-3-642-85706-5_16.

[6] Maik Merten, Falk Howar, Bernhard Steffen, and Tiziana Margaria. Automata learning with on-the-fly direct hypothesis construction. In Reiner Hähnle, Jens Knoop, Tiziana Margaria, Dietmar Schreiner, and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 248–260, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-34781-8.

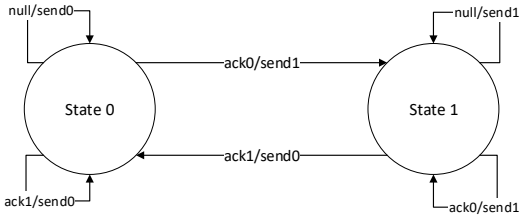
[7] A. Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958. ISSN 00029939, 10886826. URL http://www.jstor.org/stable/2033204.

[8] Bernhard Steffen, Falk Howar, and Maik Merten. *Introduction to Active Automata Learning from a Practical Perspective*, pages 256–296. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-21455-4. DOI: 10.1007/978-3-642-21455-4_8. URL https://doi.org/10.1007/978-3-642-21455-4_8.

Appendix

```
MealyMachine{
initialState
State a states { State a, State b, State c, State d, State e, State dd, State f
}inputAlphabet Alphabet { characters { water , pod , button , clean } }
outputAlphabet Alphabet { characters { done , coffee , none } }
transitions{
Transition { input clean output done sourceState a targetState a } ,
Transition { input pod output done sourceState a targetState b } ,
Transition { input water output done sourceState a targetState c } ,
Transition { input button output none sourceState a targetState f } ,
Transition { input pod output done sourceState b targetState b } ,
Transition { input water output done sourceState b targetState d } ,
Transition { input button output none sourceState b targetState f } ,
Transition { input clean output done sourceState b targetState a } ,
Transition { input clean output done sourceState c targetState a } ,
Transition { input pod output done sourceState c targetState dd } ,
Transition { input button output none sourceState c targetState f } ,
Transition { input water output done sourceState c targetState c } ,
Transition { input water output done sourceState d targetState d } ,
Transition { input pod output done sourceState d targetState d } ,
Transition { input clean output done sourceState d targetState a } ,
Transition { input button output coffee sourceState d targetState e } ,
Transition { input water output done sourceState dd targetState dd } ,
Transition { input pod output done sourceState dd targetState dd } ,
Transition { input button output coffee sourceState dd targetState e } ,
Transition { input clean output done sourceState dd targetState a } ,
Transition { input clean output done sourceState e targetState a } ,
Transition { input button output none sourceState e targetState f } ,
Transition { input pod output none sourceState e targetState f } ,
Transition { input water output none sourceState e targetState f } ,
Transition { input clean output none sourceState f targetState f } ,
Transition { input button output none sourceState f targetState f } ,
Transition { input pod output none sourceState f targetState f } ,
Transition { input water output none sourceState f targetState f } } }
```

Listing A.4. The Mealy machine seen in Fig.2.2 in the form of the Xtext the grammar described in Listing 3.1.



(a) Mealy machine representation of the alternating-bit protocol.

```
String sequence =
  "null|send0|ack1|send0|ack0|send1"
  + "|ack0null|send1|ack0ack0|send1|ack0ack1|send0"
  + "|ack1null|send0|ack1ack0|send1|ack1ack1|send0"
  + "|nullnull|send0|nullack0|send1|nullack1|send0"
  + "|ack0nullnull|send1|ack0nullack0|send1|ack0nullack1|send0"
  + "|ack0ack0null|send1|ack0ack0ack0|send1|ack0ack0ack1|send0"
  + "|ack0ack1null|send0|ack0ack1ack0|send1|ack0ack1ack1|send0"
  + "|ack1nullnull|send0|ack1nullack0|send1|ack1nullack1|send0"
  + "|ack1ack0null|send1|ack1ack0ack0|send1|ack1ack0ack1|send0"
  + "|ack1ack1null|send0|ack1ack1ack0|send1|ack1ack1ack1|send0"
  + "|nullnullnull|send0|nullnullack0|send1|nullnullack1|send0"
  + "|nullack0null|send1|nullack0ack0|send1|nullack0ack1|send0"
  + "|nullack1null|send0|nullack1ack0|send1|nullack1ack1|send0";
```

(b)

Figure A.6. A Mealy machine **(a)**, with its input format **(b)** using the formalism described in Section 3.2.3s *StringSequenceLearnable* item.

```
@Override
public List<? extends I> equivalenceQuery(H hypothesis, Collection<? extends I> alphabet) {
  for(Set<I> s : com.google.common.collect.Sets.powerSet(new HashSet<I>(alphabet))) {
    if(!s.isEmpty()) {
      for(List<I> permutation : com.google.common.collect.Collections2.permutations(s)) {
        if(!hypothesis.query(permutation).equals(this.membershipQuery(permutation))) {
          O a = hypothesis.query(permutation);
          O b = this.membershipQuery(permutation);
          return permutation;
        }
      }
    }
  }
  return null;
}
```

Listing A.5. Brute-force implementation of equivalence queries described in Section 3.2.3s *StringSequenceAdapter* item using google guavas Sets.powerSet() function.