**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 12, 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 12.

*Barcsa-Szabó Áron*
hallgató

# Kivonat

Informatikai rendszerek tervezése, fejlesztése modell-alapú technológiák segítségével hatékonyabbá tehető. A tervezett rendszerről rendelkezésre álló formális modell lehetővé teszi olyan feladatok automatizált végrehajtását, mint a helyességellenőrzés, kódgenerálás, valamint a rendszer kvalitatív és kvantitatív analízise.

A rendszermodell reaktív komponensei (pl. kommunikációs protokollok) tipikusan állapot-alapú formalizmusokkal modellezhetők. Sokszor azonban a szükséges rendszermodell elkészítése nehéz feladat, hiszen ezen protokollokat kényelmesebb példalefutások alapján megtervezni. Tudomásom szerint az irodalomban még nincs olyan eszköz, mely lehetővé tenné egy rendszer(komponens) megtervezését példalefutások alapján.

Munkám célja a rendszertervezés megkönnyítése, egy olyan szoftver létrehozásával, mely lehetővé teszi egy rendszer példalefutások alapján való megtervezését. A tervezett rendszer magját automatatanuló algoritmusok képzik, melyek erőssége éppen az, hogy állapot-alapú modelleket hoznak létre tervezett viselkedések alapján.

A fenti cél megvalósítására a dolgozatomban bemutatok egy moduláris, kiterjeszthető és más szoftverekkel integrálható keretrendszert. Munkám során két algoritmust is implementáltam, melyek tetszőleges formalizmusokat képesek kezelni. Az elkészült keretrendszer hatékonyságát és alkalmazhatóságát mind elméleti szempontból, mind mérésekkel is igazoltam.
A bemutatott keretrendszer a rendszertervezés elősegítésén kívül az automatatanuló algoritmusok fejlesztését, összehasonlítását és tetszőleges célú felhasználását is lehetővé teszi.

# Abstract


The design and development of technological systems can be made more efficient using model-based technologies. A formal model of the system under design makes the automation of tasks such as verification, code generation and qualitative, quantitative system-analysis possible.

Reactive components of a system model (e.g. communication protocols) are typically modeled using state-based formalisms. However, the construction of such a system-model is often difficult, since these protocols are more straightforward to design using example runs of them. To the best of my knowledge, there is no tool in the literature capable of modeling system (components) using example runs.

The objective of my work is to ease the process of system design by creating a software, which makes the modeling of a system using its example runs possible. The core of this framework is provided by automaton learning algorithms, the strength of which is exactly the creation of state-based models using behavioral information.

To attain the above goal, I present a modular, extensible and easily integrable framework in this thesis. During my contribution, I've implemented two algorithms capable of handling arbitrary formalisms. I have confirmed the efficiency and applicability of these algorithms both theoretically, and by measurements.
The presented framework, while capable of supporting system design, also enables the development, comparison and application of automaton learning algorithms.

# Chapter 1

# Introduction

## 1.1 Context

Model-based technologies can be used to optimize the process of system design and development, by enabling the automatization of various tasks. Given a formal model of a system under design, the verification and analysis of the system can be automated and model-based code generation becomes possible.

## 1.2 Problem Statement

The reactive components of a system, such as communication and security protocols, are typically modeled using state-based formalisms. The manual design of these components can often be a difficult task, since these protocols are more easily defined by their behavior. Specifically, describing the behavior of such a system by example runs proves to be an easier task then modeling them outright. A communication protocol for instance, can be defined by the union of every runs behavior from the viewpoints of every participant of the communication. To the best of my knowledge, there is is no current software, that enables the modeling of a system, or a system component by the description of its example runs.

## 1.3 Objective

The objective of my work was to design a framework capable of supporting system design by modeling a system (component) based on example-runs describing its behavior. Behavioral state-based modeling is the strength of automaton learning algorithms, specifically, active automaton learning algorithms excel at monitoring system behavior, which is why they are the core of the framework that I created.

## 1.4 Contribution

To realize my goal, I have created an automaton learning framework capable of handling arbitrary formalisms in a modular, easily extensible and integrable way. In the framework, I implemented two active automaton learning algorithms, the Direct Hypothesis Construction[7] and the TTT[4] algorithms, both capable of infering a system using behavioral information.

## 1.5 Related work

*LearnLib*[5] provides a Java framework for active and passive automaton learning, with the versatile AutomataLib framework acting as a backbone of it, but has no native support for software-component development. The *libalf* framework also provides learning techniques for finite automata implemented in *C++*. Te Gamma Statechart Composition framework[8] provides automata-based development of software components, but is not capable of learning automata. The Theta framework[11] enables the evaluation of abstraction refinement-based algorithms, but has no support for automaton learning.

## 1.6 Added value

The presented framework, while capable of system design using example runs, also enables the modular implementation of active automaton learning algorithms and the formalisms they might depend on. Additionally, it makes the comparison of different learning algorithms possible in the same environment (and such, with the same overhead), and allows the discretionary use of the automaton learning algorithms implemented therein.

## 1.7 Outline

The paper is organized as follows. Chapter 2 provides an outline of the necessary theoretical background. Chapter 3 gives insight into the contribution presented in this thesis. Chapter 4 evaluates the correctness of the presented framework and the algorithm implemented within both theoretically and experimentally. Chapter 5 concludes the thesis and presents future work goals.

# 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 $\Sigma$ be a finite, non-empty set. $\Sigma$ is an alphabet, its elements are symbols or characters. ▪

**Definition 2 (Word).** If $\Sigma$ is an alphabet, then any finite sequence comprised of the symbols of $\Sigma$ are words (or Strings). $\Sigma^n$ represents the set of every $n$ length word consisting of symbols in $\Sigma$: $\Sigma^n : w_1 w_2 \dots w_n$, 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 $\Sigma^*$. The empty word is denoted by $\epsilon$. ▪

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 $\Sigma$ 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,
- $\Sigma$ 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 $\Sigma$ listed in its transitions, meaning every state behaves deterministically for every possible input.

An example of a DFA (Deterministic Finite Automaton) from[10] 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 $4i + 3a$.*

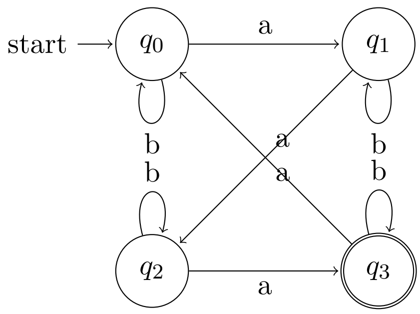



**Figure 2.1.** A simple DFA from [4].

| $\delta$ | $q_0$ | $q_1$ | $q_2$ | $q_3$ |
|---|---|---|---|---|
| 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,
- $\Sigma$ is the input alphabet of the automaton,
- $\Omega$ 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.*

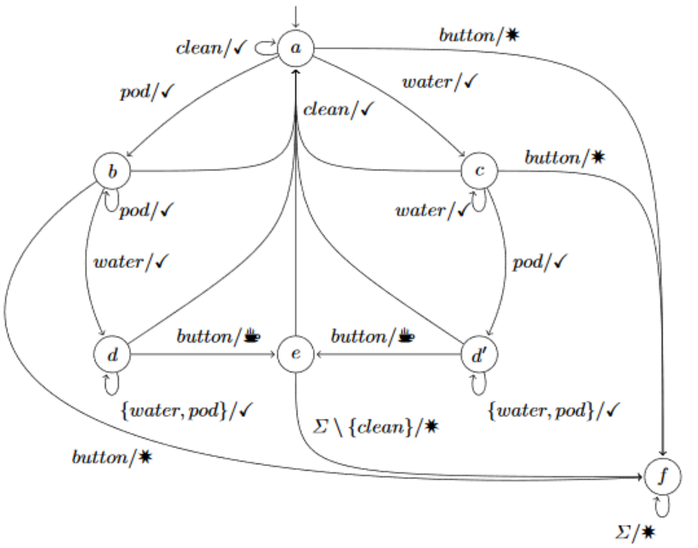



**Figure 2.2.** Mealy machine representing the functionality of a coffee machine.[10]

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 $\delta^*$ for both DFAs and Mealy machines. $\delta^*$ is an extension of the $\delta$ 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 $\Sigma^*$ (when L(M) = $\Sigma$):
$x \equiv_M y \iff \delta^*(s, x) = \delta^*(s, y)$
where $x, y \in \Sigma^*$. This means, that x and y are equivalent with respect to $\equiv_M$.[6]. ▪

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[6], 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[6][9]).** 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 [6][9]. ▪

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 $\delta^*$, 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. $\equiv_P$[10]).** 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. $\equiv_{[\![M]\!]}$:*

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

*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[10]).** 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 $[\epsilon]$.

-$\delta$ is defined by $\delta([u], \alpha) = [u\alpha]$.

-$\lambda$ 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]\!] = $ 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]\!] = $ P, or equivalently, if P has a finite number of equivalence classes, analogously to the previously seen ”classical” regularity.

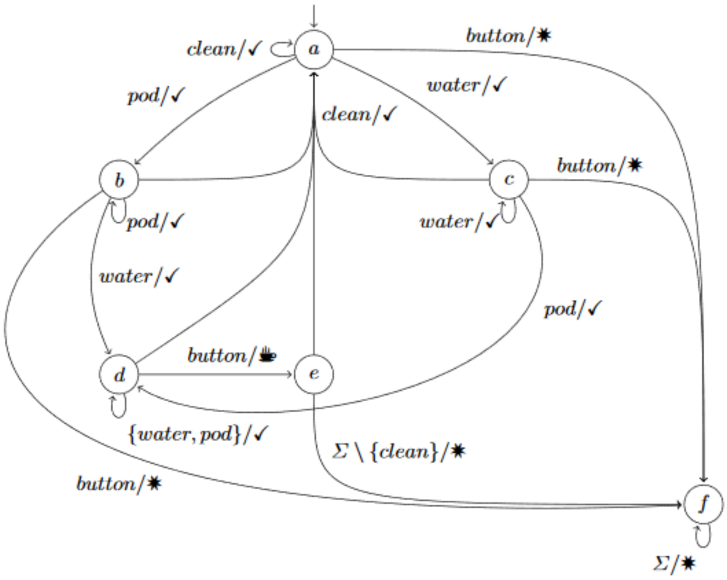



**Figure 2.3.** Minimal version of the Mealy machine seen in 2.2

### 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[10]):** 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[10]).** 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 $\alpha$-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.[10]

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 [10]. 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.[10] 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 a way of modeling a system without having specific knowledge of its internal behavior. To accomplish this, the external behavior of the system needs to

be observed. This learned model is, as the name suggests, an automaton.
Formally: Automata learning is concerned with the problem of inferring an automaton model for an unknown formal language $L$ over some alphabet $\Sigma$[3].

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

**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 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 defines the separation of the algorithm to a teacher and a learner component in a way, 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 o 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 attempts to determine if the hypothesis is behaviorally equivalent to the SUL, and if not, finding the diverging behavior, and return with an example. 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.

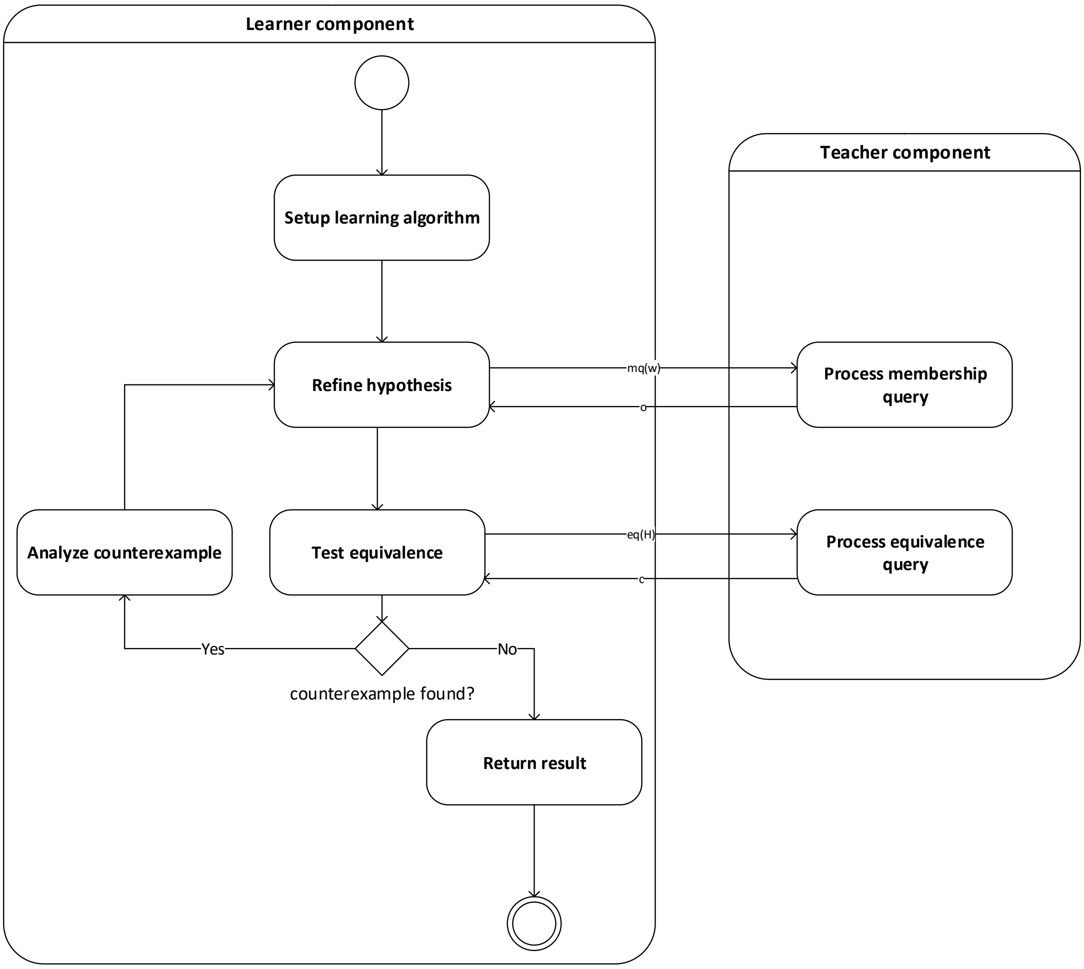



**Figure 2.4.** Active automaton learning

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 driven by the counterexamples produced.

Using an analogous strategy to the minimization of automata seen in the previous section, starting only with a one state hypothesis automaton, all words are explored in the alphabet in order to refine and extend this hypothesis. Here, there is a dual way of characterizing (and distinguishing) between states[10]:

- By words reaching them. A prefix-closed set $S_p$ of 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 $\Sigma^*$. 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), the following definition is introduced.

**Definition 16 (D-epimorphism).** 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[10]

- 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.[10]

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[7]

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 $\epsilon$ and D = $\Sigma$. It then tries to complete the hypothesis: for every state, the algorithm determines the behavior of the state under D. This behavior is called the extended signature of said state. States with a new extended signatures are provably new states, so to guarantee further investigation, all their successors are enqueued. 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. [10][7]

**Algorithm 1:** Hypothesis construction of the Direct Hypothesis Construction algorithm as seen in [10].

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

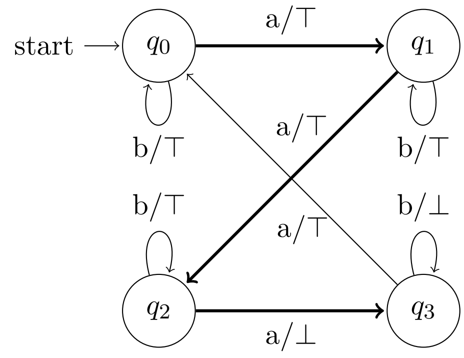
initialize hypothesis H, create a state for all elements of $S_p$
initialize a queue Q with the states of H
**while** *$Q$ is not empty* **do**
    s = dequeue state from Q
    u = access sequence from $s_0$ to s
    **for** $d \in D$ **do**
        o = mq($ud$)
        set $\lambda(s, d) = o$
    **end**
    **if** *exists an $s' \in S$, 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 $\Sigma$ into Q, if not already in $S_p$
    **end**
**end**
Remove entries of $D \setminus \Sigma$ from $\lambda$
**return** $H$

After the execution 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 $S_p$).

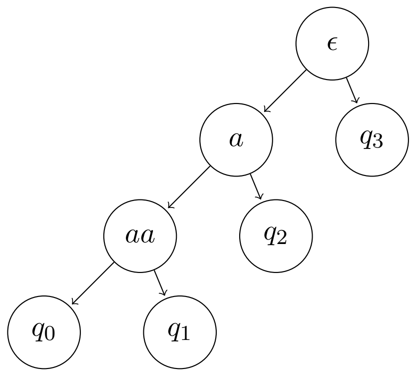
While the DHC algorithm is a straightforward implementation of active automata learning, it struggles with time complexity. It terminates after at most $n^3mk + n^2k^2$ membership queries, and $n$ equivalence queries, where $n$ is the number of states in the final hypothesis, $k$ is the longest set of inputs, and $m$ is the length of the longest counterexample. This renders the DHC algorithm inefficient in practical applications. The next subsection deals with an optimized algorithm, the TTT algorithm.

### 2.2.2 The TTT algorithm[4]

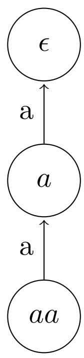
As seen in the Direct Hypothesis Construction algorithm, a large number membership queries are required in order to learn an automaton, causing a bottleneck on the scalability of the algorithm. The main cause of this complexity issue is the way counterexamples are treated. In order to ensure every state is properly found, *all* suffixes of the counterexample are added to the algorithms structure, which, has a huge impact on the number of membership queries in the next iteration of the hypothesis construction. 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.




**(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 constructed simply by the following rule: if a transition ends in $q_3$, the output is $\perp$, else the output is $\top$.*
*TTT keeps the notion of separating states by a spanning tree $S_p$, a prefix-closed set of words from $\Sigma^*$, 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 call 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.

For a more detailed description of the TTT algorithm the reader is referred to[4].

# Chapter 3

# Contribution

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

## 3.1 Automaton learning framework

Active automaton learning algorithms essentially have two endpoints: the input, reached through the teacher component, and the output, where the learned hypothesis automaton is returned. This is illustrated in Fig. 3.1. The teacher and the learner can be adjusted to the required formalisms: the underlying modeling language of the system under learning (the *input formalism*) and that of the learned hypothesis (the *output formalism*).

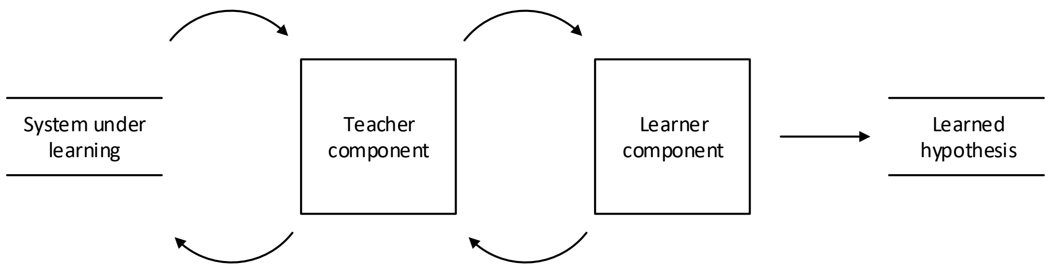



**Figure 3.1.** Data flow of active learning algorithms.

### 3.1.1 Tooling

As seen in Fig. 3.1, the variables of the framework are essentially the input and output formalisms it accepts. In order to provide flexibility in this regard, the following tools were used.

- **Eclipse Modeling Framework (EMF):** EMF, or more specifically, EMF core, is an "abstraction for describing, composing, and manipulating structured informa-

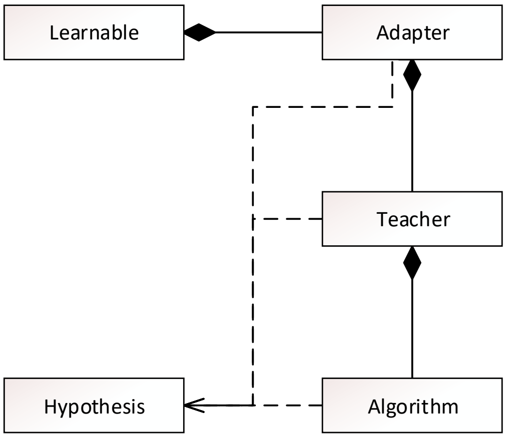



**Figure 3.2.** Structure and relations of the packages comprising the framework.

tion", essentially a tool for system modeling and code generation implemented in Java. The tool it provides allows graphical modeling of UML class diagrams and generating code based on these models.

- **Xtext:** Xtext is a framework for creating programming and domain-specific languages, which also has integration with EMF. This integration allows reading and writing text files in the form of an Xtext grammar using corresponding EMF generated classes.

### 3.1.2 High-level overview

Since the implementation of the framework would be using Java as a language, the high-level view seen in Fig. 3.2 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. 3.2 to Fig. 3.1, 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. 3.1, the Adapter package is used as an abstraction layer to separate the algorithm and the teacher from the input formalism. Since automaton learning algorithms have no direct access to the system under learning, they operate in a black-box way, the Adapter package is a useful addition. Unfortunately no such adapter can be used on the output layer, since Hypotheses are directly accessed by the learning algorithms, and 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, which are later transferred through the Teacher and Adapter components, they both also have a dependency on the Hypothesis package.

### 3.1.3 Detailed overview of abstractions

The diagram in Figure 3.2 showed the modularization of the framework, which depend on the correct implementation of object-oriented architecture so the notated compositions and dependencies in Fig. 3.2 are satisfied. These abstractions are defined and implemented in Java as abstract classes and interfaces seen in Fig. 3.3. 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* interface 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 interface 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.

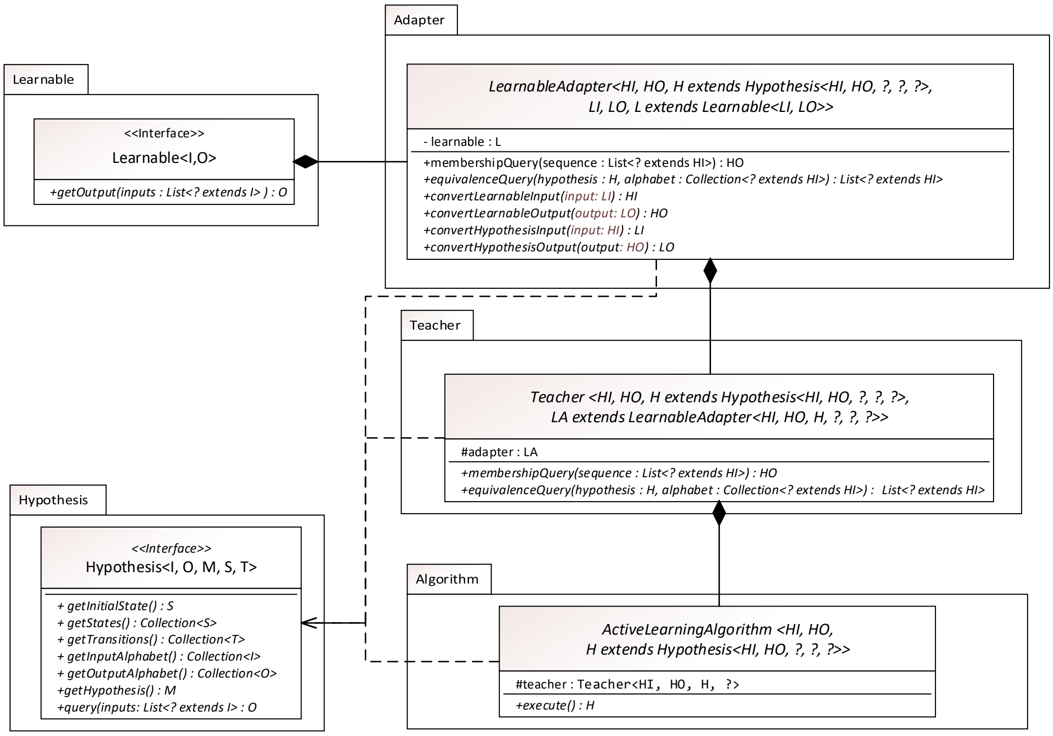



**Figure 3.3.** Overview of the abstract classes and interfaces of the framework.

## 3.2 Implemented learning algorithms

Building upon the abstractions of the framework, I implemented the two active learning algorithms presented in the Background chapter. The first step of the implementation was determining the input and output formalisms to use.

Since automaton learning algorithms can be implemented on any type of automata, the easiest method would be using deterministic finite automata. However, real-life reactive systems are usually 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, which is why the Mealy machine implementation was done in Eclipse Modeling Framework.

Figure 3.4 shows an UML class diagram of the metamodel I've created using the Ecore package of EMF core, and the graphical modeling tool provided by EMF. I also used EMF to generate Java classes corresponding to the ones seen in Fig. 3.4.

In text, the Ecore model 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 has a reference to the source and target states of the transition, as opposed to some models, where states store their own transition information (successors,

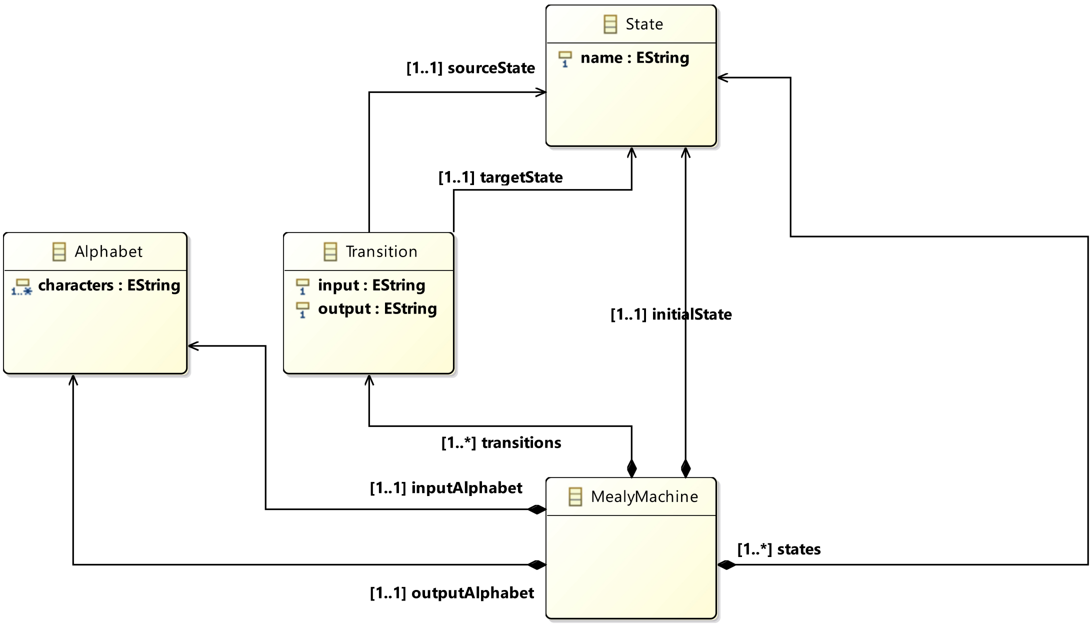



**Figure 3.4.** Ecore metamodel of Mealy machines.

predecessors) like nodes in a graph. This decision was based on the DHC algorithm, which stores traversal information itself, rather than asking for it, which is why ease of access is preferred as opposed to efficiency.

Extending upon the metamodel in Figure 3.4, I generated the Xtext grammar seen in Listing 3.1 utilizing Xtexts EMF integration. In words, this grammar describes a textual grammar in which instances of the metamodel seen in Fig. 3.4 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.

An example of a MealyMachine instance in the form if the grammar in Listing 3.1 can be 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.

### 3.2.1 Direct Hypothesis Construction

While implementing the DHC algorithm, I used the above described MealyMachine formalism using the Xtext input source (later seen as *MealyMachineLearnable*), but I also described a way to programmatically describe an input to ease the process of creating small inputs (later seen as *StringSequenceLearnable*). Both are expanded upon in the following.

The detailed (but not exhaustive) class diagram of the frameworks DHC implementation can be seen in Fig. 3.3. 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 "$|input1|output1|input2|output2|...|inputn|outputn|$", and storing it in a HashMap. One example of this formalism can be seen in Fig. A.1.
- **MealyLearnable:** The *MealyLearnable* class is a *Learnable* which uses the EMF-modeled *MealyMachine* class seen in Fig. 3.4. 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. 3.4. 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.2.
- **StringSequenceToMealyAdapter:** The *StringSequenceToMealyAdapter* class bounds the *Hypothesis* parameters of the *LearnableAdapter* to *DHCMealyMachineHypothesis*, as the dependency relation indicates in Fig. 3.3. 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 Listing 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 3.2.1s *DirectHypothesisConstruction* item.

Note, that the implementatation utilizes the DHCHypothesis abstract class, providing a separation of the DHC algorithm and the output formalism it constructs while learning.

### 3.2.2 TTT

The TTT algorithm is complex, in both theory and implementation. Due to it using multiple specific types of data structures and sub-algorithms, which are not themselves part of TTT, during my imlementation I depended on the original (defining) implementation of it in LearnLib[5].
LearnLib describes an abstract layer of TTT to extend in implementations. I used these abstractions to implement the algorithm into my framework in a (from the perspective of execution) near seamless manner. The current implementation only allows MealyMachines as inputs, but this can be extended upon analogously as with the DHC implementation seen previously.

The implementation itself is built similarly as the DHC implementation, but the algorithm wraps the abstractions of the framework and converts between types of these abstractions and the ones used in LearnLib. More specifically, LearnLib defines abstract types for parts of the TTT algorithms, such as *AbstractTTTHypothesis*,*AbstractBaseDTNode* (abstract discriminator tree node), *AbstractTTTLearner*. While LearnLib has implementations of these, I only depended on abstractions, using the current implementation as reference.

For key steps of TTT, the responsibilities of the current implementation are as follows. The Hypothesis construction and refinement is done by my framework, which provides the methods of hypothesis construction, while also hadling the calculation of output inconsistencies and the refinement of them. LearnLib manages the internal data structures including the dicriminator tree, which stores my implementation of the *AbstractBaseDTNode* class. The state splitting (part of Hypothesis stabilization) and discriminator finalization is handled by LearnLib. Considering the communication with the system under learning, equivalence queries are done in the exact way presented in the DHC algorithm, done by wrapping the *Teacher* class of my implementation into the *EquivalenceOracle* interface of LearnLib. The same solution could not be used with membership queries, since the TTT algorithm internally tries avoiding these queries (done partly by finding output inconsistencies), and has specific behavior in its LearnLib implementation differing from traditional membership queries. The conversion between the formalisms used by the two frameworks allowed me to natively use their *MembershipOracle* interface with the converted *Learnable*.

A summary of the DHC and TTT implementations can be seen in Table 3.1.

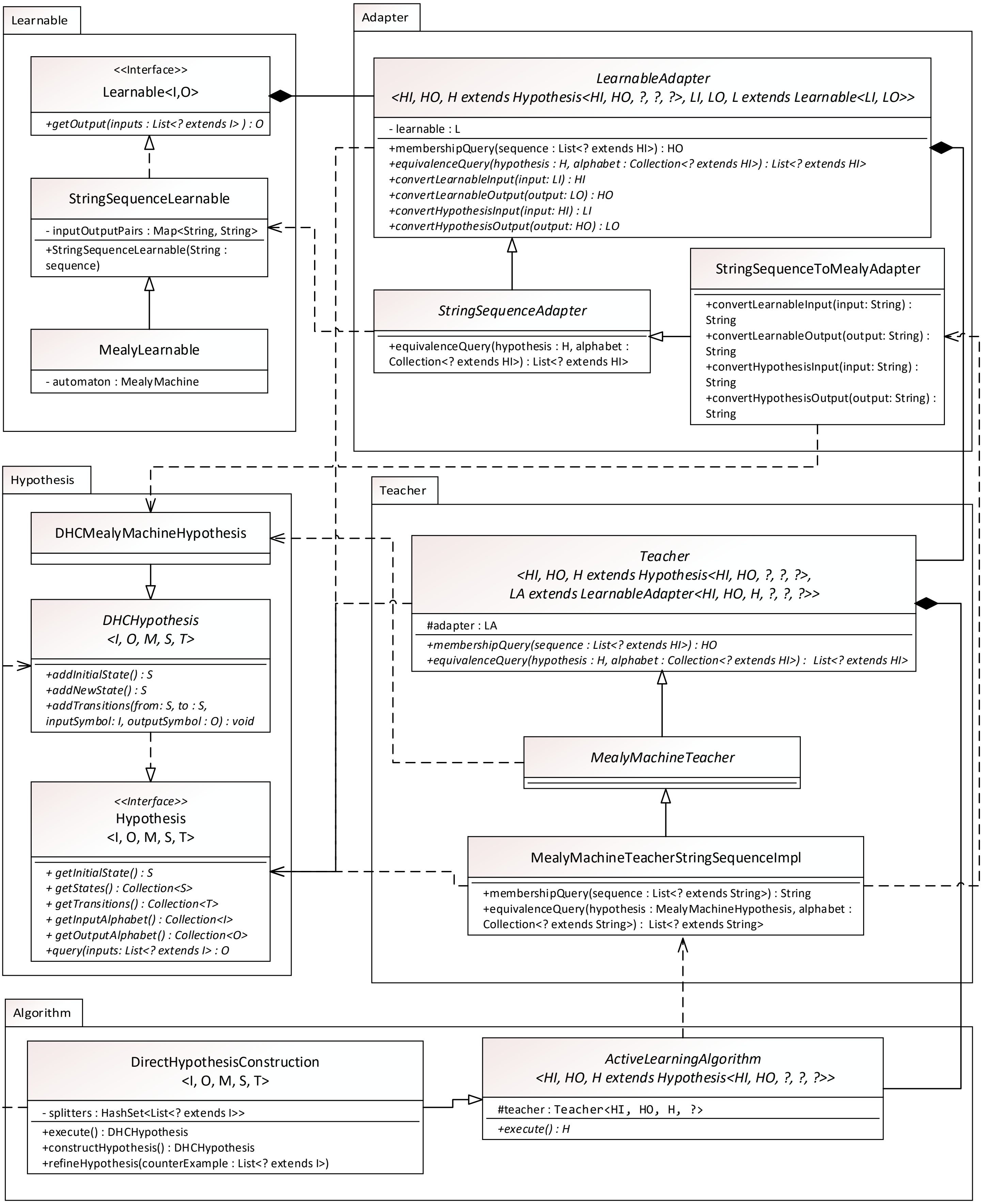



**Figure 3.5.** Non-exhaustive detailed structural overview of the frameworks DHC implementation.

| **Algorithm** | **Input formalism** | **Output formalism** | **Input source** | **Output source** |
|---|---|---|---|---|
| DHC | Mealy machine | Mealy machine | Xtext | Xtext |
| DHC | String sequences | Mealy machine | String | Xtext |
| TTT | Mealy machine | Mealy machine | Xtext | Xtext |

**Table 3.1.** Overview of the algorithm implementations in the framework.

After running the example seen in 3.2, the output of the DHC algorithm can be seen in Listing A.4, while the output of the TTT algorithm can be seen in A.3. As a case study, both these algorithms were ran on the Mealy machine seen in 2.2 contained in the input in Listing A.1, resulting in the outputs Listing A.6 and Listing A.5 repsectively.

# Chapter 4

# Evaluation

## 4.1 Theoretical evaluation

The framework presented in this thesis has both advantages and disadvantages in design and implementation. The modular setup shown in Fig. 3.2, while providing easy extensibility, does require knowledge of the algorithms implemented therein. In contrary to LearnLib, where (from my experience) data structures are easily extended and onboarded to the learning, algorithm implementations are difficult to provide new solutions to, my framework allows extensibility on every front of it, with a steeper learning curve even with new input/output onboarding.

The framework, when implementing similar formalisms does require some redundancy, especially with the *Learnable-LernableAdapter-Teacher* trio of implementation needed with new formalisms. The advantage of this sometimes redundant approach is the overall elimination of typecasting and uncertain genericity, allowing for exact application of methods and classes after bounding the generic parameters required, improving runtime, the compliance with object-oriented paradigms, and understandibility of an implementation without context of its abstrations. The *Learnable* and *Hypothesis* endpoints of input and output formalisms being interfaces provide flexibility by leveraging the multiple-inheritance (implementation) property of interfaces. This is used for example in the TTT implementation in order to extend upon classes of LearnLib, while still implementing my frameworks abstractions.

### 4.1.1 Evaluation of DHC

The Direct Hypothesis Construction algorithm, as theorized and proved in [10] and [7], terminates after at most $n^3mk + n^2k^2$ membership, and $n$ equivalence queries, where $n = |S|$, $k = |\Sigma|$ and m is the longest counterexample. The runtime complexity of these queries are difficult to evaluate, since they highly depend on implementation and context. Reaching the system under learning in the current implementation of input formalisms (String sequece or Mealy machine) takes no overhead, since the system behavior is stored in-memory. This might not be the case in other implementations, where the SUL might be reached through network communication or other such (non-negligible overhead) methods. In terms of membership queries, the current implementations differ.

String sequences (*StringSequenceLearnable*s) are stored in an input-output *HashMap*, which allows $O(1)$ access assuming the values are evenly distributed in the buckets used by the hashing, worst-case scenario being $O(k)$. While access is fast, this method suffers in terms of space complexity, storing a numer of elements identical to $\mathcal{P}(\Sigma) \setminus \emptyset$, or in text, the powerset of $\Sigma$ without the empty set, resulting in a space complexity of $O(2^k)$.

The MealyMachine implementation (the *MealyLearnable* class) provides a more reasonable $O(k)$ space complexity, but it struggles with runtime issues. The implementation, as discussed in the contribution section, provides ease of access to its data, with straightforward implementation of membership queries possible, not storing the automata in a graph-like format reduces efficiency. This results in a worst-case scenario of an $O(kt)$, where $t$ is the number of transitions of the automaton.

From the perspective of equivalence queries, the two implemented input formalisms both provide the same efficiency, since the implementation of this query is in the *StringSequenceAdapter* class, both *MealyLearnable* and *StringSequenceLearnable* are queried using this implementation, which can be seen in Listing A.2. This implementation is a brute-force way of proving equivalence of the hypothesis and the system under learning, operating by taking every permutation of every element in $\mathcal{P}(\Sigma) \setminus \emptyset$ and comparing outputs using membership queries for each of them. In order to mitigate some of this inefficiency, the implementation uses google guavas *Sets.powerSet()* method, providing $O(k)$ space complexity as opposed to a brute-force $O(2^k)$ implementation. For each member of $\mathcal{P}(\Sigma) \setminus \emptyset$, the permutations are calculated using the *Collections2.permutations()* method of guava, implementing the Johnson–Trotter algorithm. This results in a $O(2^k k!)$ number of membership queries considering the "worst case" of finding no counterexamples.

In summary, the best-case scenario of the current DHC implementation has an $O(n^3mk^2 + n^2k^3 + 2^k k!k)$ runtime complexity, the worst case being $O(n^3mk^2t + n^2k^3t + 2^k k!kt)$ depending on the variables presented above.

### 4.1.2 Evaluation of TTT

The TTT algorithm, as presented in [4], requires $O(n)$ equivalence queries and $O(kn^2 + n \log m)$ membership queries, each of which takes $O(n + m)$ time, where $n = |S|$, $k = |\Sigma|$ and m is the longest counterexample. This is a very pessimistic estimate caused by the edge-case of a discriminator tree having $n$ height. The time complexity of the equivalence query implementation seen in A.2 still holds as $O(2^k k!)$, but is lengthened by the complexity of conversion between formalisms, being $O(nt)$ in the worst case, where t is the number of transitions of the current hypothesized automaton. Altogether, the worst case scenario has a $O((kn^2 + n \log m)(n + m) + 2^k k!nt)$ time complexity. TTT also provides an efficient $\Theta(kn)$ space complexity[4].

## 4.2 Experimental evaluation

The experimental evaluation of the implemented algorithms were done using variations of the automaton seen in Fig. 2.5.a. This decision is based on the simplicity of extending the automaton accepting inputs containing $4i + 3a$ to inputs containing $ni + (n - 1)a$, where $n, i \in \mathbb{N}$. An example of this can be seen in Fig. 4.1, where $n = 5$. This solution is easily scalable and straightforward to implement even for large $n$s.

As seen in the theoretical evaluation, the size of the state space is not the only possible bottleneck, both algorithms (and the equivalence query implementation especially) slows with the size of the input alphabet. A simple solution of testing growing input alphabets using the Mealy machine seen in Fig. 2.5.a, is to add characters (transitions) without adding behavior. This does not remove the minimal property of an automaton, contrary

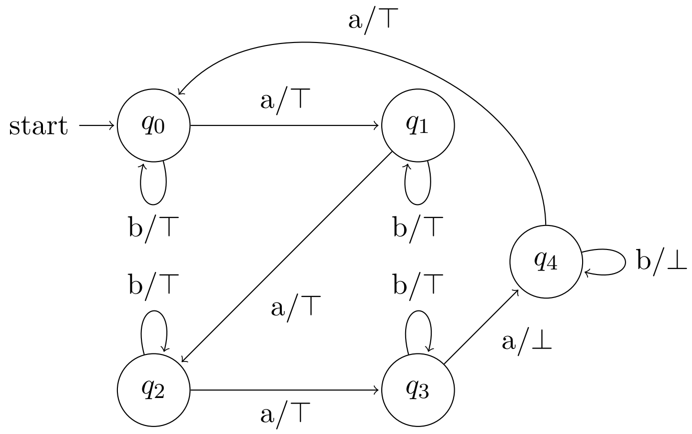



**Figure 4.1.** Example of an automaton accepting inputs containing $5i + 4a$.

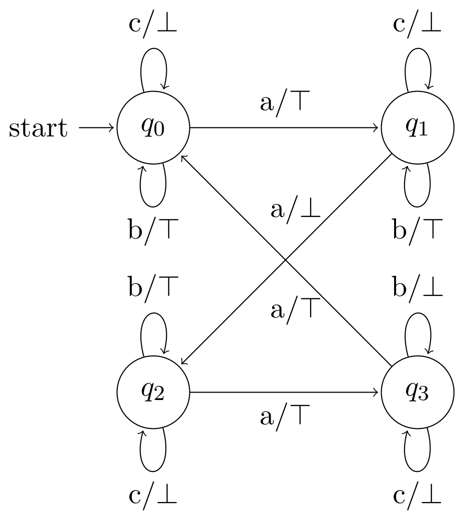



**Figure 4.2.** Example of the automaton seen in Fig. 2.5.a with an extended input alphabet $\Sigma = \{a, b, c\}$.

to adding states without changing behavior, thus increasing runtime. An example of the described input alphabet extension can be seen in Figure 4.2.

After running the experiments, an obvious bottleneck could be found. The current brute-force equivalence query implementation is inefficient, as the theoretical evaluation suggested. Figure 4.3 shows the inefficiency observed when extending the states and the behavior of the learned automaton. The cause of this is the large number of states resulting in a large space of possibilities when brute-forcing equivalence, as theorized in the theoretical evaluation.

Figure 4.4 shows, that this is not the case with the extension of the input alphabet, since with the same number of sates, less future predictions are to be done, extending the input alphabet is not as taxing. In order to show (and compare) my implementation without this bottleneck, since these inefficiencies are not of the learning algorithms, but of the equivalence algorithm, Fig. 4.5 shows the exact same run as Figure 4.3 done with a near linear-time equivalence oracle implemented in LearnLib. This shows, that the learning itself is not part of the bottleneck measured in the previous attempts.

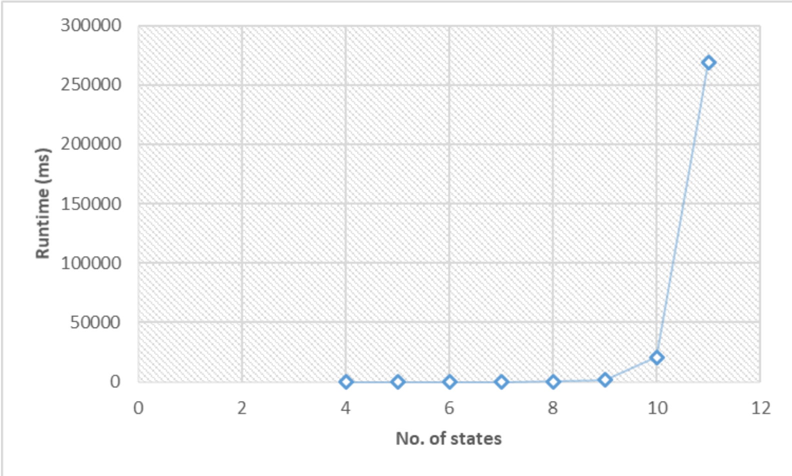



**Figure 4.3.** Figure showing the bottleneck the brute-force equivalence query implementation causes when increasing the state space in a worst-case scenario.

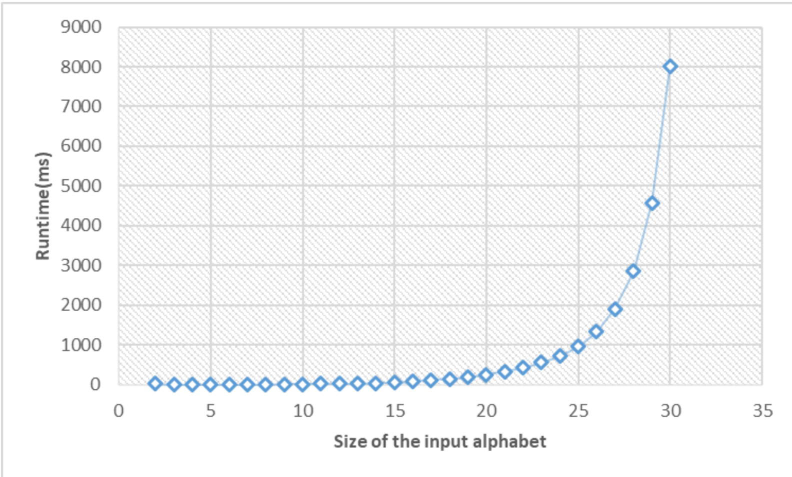



**Figure 4.4.** Figure of the runtime observed when the input alphabet was increased without any behavioral change.

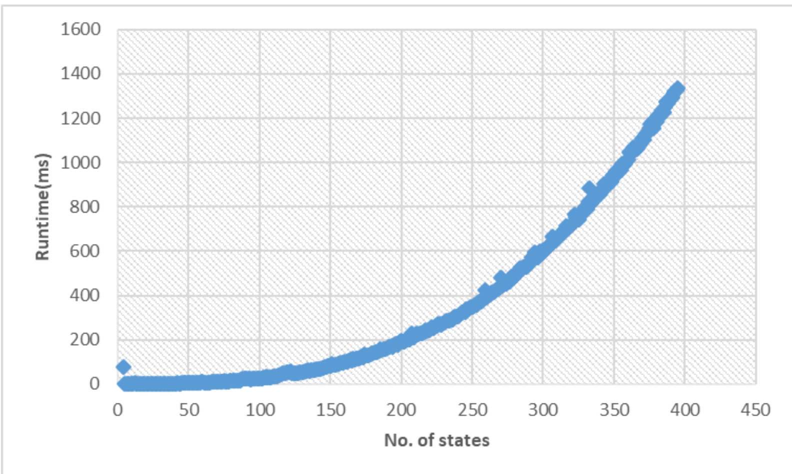



**Figure 4.5**

# Chapter 5

# Conclusions

This chapter concludes the contributions of this thesis, and presents my future goals.

## 5.1 Contributions

In my thesis, I have introduced active automaton learning and two specific learning algorithms. I designed a framework for the development of automaton learning algorithms and implemented a prototype of it. I have demonstrated the correctness of the implementation using case-studies and evaluated its efficiency by experimental evaluations.

As a result, an extensible and modular framework for active automaton learning was created. This framework contains an implementation of the Direct Hypothesis Construction and the TTT algorithms both of which are straightforward to extend using arbitrary formalisms. Utilizing these algorithms and the formalisms already implemented in the framework, system design can be supported by modeling system (components) based on example runs, as show in the case studies presented.

## 5.2 Future work

In terms of future work, the next step is clearing the equivalence query bottleneck shown in the experimental evaluation by implementing a more efficient algorithm. Also, more formalisms and learning algorithms are to be on-boarded into the framework. An integration with the Gamma Statechart Composition framework[8] is planned to enable the development and verification of software component using the automata learned by the framework presented in this thesis.

# 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] Malte Isberner, Falk Howar, and Bernhard Steffen. The open-source learnlib. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 487–495, Cham, 2015. Springer International Publishing. ISBN 978-3-319-21690-4.

[6] 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`.

[7] 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.

[8] Vince Molnár, Bence Graics, András Vörös, István Majzik, and Dániel Varró. The gamma statechart composition framework: design, verification and code generation for component-based reactive systems. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 113–116, 2018. DOI: `10.1145/3183440.3183489`.

[9] 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`.

[10] 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`.

[11] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: a framework for abstraction refinement-based model checking. In Daryl Stewart and Georg Weissenbacher, editors, *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, pages 176–179, 2017. ISBN 978-0-9835678-7-5. DOI: `10.23919/FMCAD.2017.8102257`.

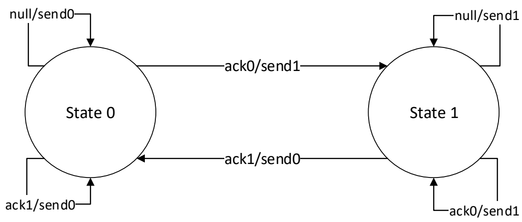
# Appendix

```
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.1.** The Mealy machine seen in Fig.2.2 in the form of the Xtext the grammar described in Listing 3.1.

```
@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.2.** Brute-force implementation of equivalence queries described in Section 3.2.1s *StringSequenceAdapter* item using google guavas Sets.powerSet() function.




**(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.1.** A Mealy machine **(a)**, with its input format **(b)** using the formalism described in Section 3.2.1s *StringSequenceLearnable* item.

```
MealeyMachine{
  initialState State s0
  states {State s0,State s1,State s2,State s3,}
  transitions {
    Transition { input a output top sourceState s0 targetState s1},
    Transition { input b output top sourceState s0 targetState s0},
    Transition { input a output top sourceState s1 targetState s2},
    Transition { input b output top sourceState s1 targetState s1},
    Transition { input a output bot sourceState s2 targetState s3},
    Transition { input b output top sourceState s2 targetState s2},
    Transition { input a output top sourceState s3 targetState s0},
    Transition { input b output bot sourceState s3 targetState s3} }
}
```

**Listing A.3.** The output MealyMachine after running the implemented TTT algorithm with the input seen in Listing 3.1.

```
MealeyMachine{
  initialState State state0
  states {State state0,State state1,State state2,State state3,}
  transitions {
    Transition { input a output top sourceState state0 targetState state1},
    Transition { input b output top sourceState state0 targetState state0},
    Transition { input a output top sourceState state1 targetState state2},
    Transition { input b output top sourceState state1 targetState state1},
    Transition { input a output bot sourceState state2 targetState state3},
    Transition { input b output top sourceState state2 targetState state2},
    Transition { input a output top sourceState state3 targetState state0},
    Transition { input b output bot sourceState state3 targetState state3} }
}
```

**Listing A.4.** The output MealyMachine after running the implemented DHC algorithm with the input seen in Listing 3.1.

```
MealeyMachine{
  initialState State state0
  states {State state0,State state1,State state2,State state3,State state4,State state5,}
  transitions {
    Transition { input water output done sourceState state0 targetState state1},
    Transition { input pod output done sourceState state0 targetState state2},
    Transition { input button output none sourceState state0 targetState state3},
    Transition { input clean output done sourceState state0 targetState state0},
    Transition { input water output done sourceState state1 targetState state1},
    Transition { input pod output done sourceState state1 targetState state4},
    Transition { input button output none sourceState state1 targetState state3},
    Transition { input clean output done sourceState state1 targetState state0},
    Transition { input water output done sourceState state2 targetState state4},
    Transition { input pod output done sourceState state2 targetState state2},
    Transition { input button output none sourceState state2 targetState state3},
    Transition { input clean output done sourceState state2 targetState state0},
    Transition { input water output none sourceState state3 targetState state3},
    Transition { input pod output none sourceState state3 targetState state3},
    Transition { input button output none sourceState state3 targetState state3},
    Transition { input clean output none sourceState state3 targetState state3},
    Transition { input water output done sourceState state4 targetState state4},
    Transition { input pod output done sourceState state4 targetState state4},
    Transition { input button output coffee sourceState state4 targetState state5},
    Transition { input clean output done sourceState state4 targetState state0},
    Transition { input water output none sourceState state5 targetState state3},
    Transition { input pod output none sourceState state5 targetState state3},
    Transition { input button output none sourceState state5 targetState state3},
    Transition { input clean output done sourceState state5 targetState state0} }
}
```

**Listing A.5.** The output MealyMachine after running the implemented DHC algorithm with the input seen in Listing A.1.

```
MealeyMachine{
  initialState State s0
  states {State s0,State s1,State s2,State s3,State s4,State s5,}
  transitions {
    Transition { input water output done sourceState s0 targetState s2},
    Transition { input pod output done sourceState s0 targetState s4},
    Transition { input button output none sourceState s0 targetState s1},
    Transition { input clean output done sourceState s0 targetState s0},
    Transition { input water output none sourceState s1 targetState s1},
    Transition { input pod output none sourceState s1 targetState s1},
    Transition { input button output none sourceState s1 targetState s1},
    Transition { input clean output none sourceState s1 targetState s1},
    Transition { input water output done sourceState s2 targetState s2},
    Transition { input pod output done sourceState s2 targetState s3},
    Transition { input button output none sourceState s2 targetState s1},
    Transition { input clean output done sourceState s2 targetState s0},
    Transition { input water output done sourceState s3 targetState s3},
    Transition { input pod output done sourceState s3 targetState s3},
    Transition { input button output coffee sourceState s3 targetState s5},
    Transition { input clean output done sourceState s3 targetState s0},
    Transition { input water output done sourceState s4 targetState s3},
    Transition { input pod output done sourceState s4 targetState s4},
    Transition { input button output none sourceState s4 targetState s1},
    Transition { input clean output done sourceState s4 targetState s0},
    Transition { input water output none sourceState s5 targetState s1},
    Transition { input pod output none sourceState s5 targetState s1},
    Transition { input button output none sourceState s5 targetState s1},
    Transition { input clean output done sourceState s5 targetState s0} }
}
```

**Listing A.6.** The output MealyMachine after running the implemented DHC algorithm with the input seen in Listing A.1.