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

Complex Requirements in Automata Learning-Based Software Engineering

Scientific Students' Association Report

Author:

Balázs Várady

Advisor:

dr. András Vörös

2021

Contents

Kivonat **i**

Abstract **ii**

1 Introduction **1**

2 Background **3**

2.1 Model-Based Engineering 3

2.2 Foundations of Automata Theory 4

2.2.1 Fundamentals of Formal Language Theory 4

2.2.2 Finite Automata and ω-automata 5

2.2.3 Relations of Formal Languages and Automata 9

2.2.4 Minimization of Mealy Automata 12

2.3 Automata Learning 14

2.3.1 Traditional Automata Learning Algorithms 17

2.4 Specifying Complex Requirements 19

2.4.1 Requirements and Properties 19

2.4.2 Linear-Time Temporal Logic 22

2.5 Reactive Synthesis from Temporal Logic 24

2.5.1 Synthesis Problem for Reactive Systems 24

2.5.2 Games and Reactive Systems 24

2.5.3 Synthesis through Parity Games 25

3 Interactive Learning and Related Work **28**

3.1 Interactive Learning 28

3.1.1 Overview of the Methodology 28

3.1.2 Overview of the Architecture 31

3.1.3 The Oracle 32

3.1.4 The Learning Algorithm 33

3.2 Related Work 35

4 Temporal Logic Specification in Interactive Model Synthesis 36
4.1 Overview of the Extended Methodology . . . 36
4.2 Challenges of Model Synthesis from LT Properties . . . 37
4.3 Applications of LTL in Interactive Learning . . . 38
4.3.1 Constraining Queries Using Büchi Automata . . . 39
4.3.2 Learning from Mealy Automata . . . 39
4.3.3 Generalized Learning Algorithms . . . 40
4.4 Model Refinement in Automata Learning . . . 43
4.4.1 Extending the Alphabets . . . 43
4.4.2 Refinement of the Symbols . . . 44

5 Implementation 49
5.1 Tooling . . . 49
5.1.1 Eclipse Modeling Framework . . . 49
5.1.2 Xtext Framework . . . 49
5.1.3 Owl . . . 49
5.1.4 Strix . . . 50
5.1.5 LearnLib . . . 50
5.1.6 Automata Learning Framework . . . 50
5.2 Extensions to the Framework . . . 51
5.2.1 Interactive Learning . . . 51
5.2.2 Refinement-based Learning . . . 52
5.2.3 Components of the Learning Algorithm . . . 52

6 Case Study: Alternating Bit Protocol 54
6.1 Introduction . . . 54
6.2 Initial LTL Specification . . . 55
6.3 Extending the Model with Interrupts . . . 55
6.4 Refinement of the Interrupt Handling . . . 57
6.5 Evaluation of the Results . . . 59

7 Evaluation 61
7.1 Complexity of the Learning Algorithms . . . 61
7.2 The Interactive Learning Entity . . . 62

8 Conclusion 64
8.1 Contribution . . . 64
8.2 Future Work . . . 65

Kivonat

A szoftverfejlesztés a magas szintű követelmények szintjén kezdődő folyamat, amelyből a szoftvermérnökök megtervezhetik a rendszermodelleket és implementálhatják a szoftvert. Ez egy igen összetett eljárás, amely során a bonyolultság kezelésében többek közt a modell alapú technikák is segíthetnek. A magas minőségű modellek tervezése azonban továbbra is fáradságos feladat.

Állapot alapú modelleket elsősorban reaktív rendszerek viselkedésének hatékony leírására használunk, melyek példákból történő szintetizálására kézenfekvő megoldást nyújtanak az automatatanuló algoritmusok. Az aktív automatatanulás lekérdezések és ellenpéldák felhasználásával állít elő modelleket, és így természetesen kiterjeszthető a mérnökök tudásából történő modellszintézisre is. Ezt a módszert interaktív tanulásnak nevezzük. A mérnökök azonban - szemben az implementációk lekérdezésével, amelyekhez gyakran használnak aktív automatatanulást - általában a komplex követelmények megfogalmazásával kezdik a fejlesztési folyamatot, és azok alapján készítik el a rendszermodelleket.

A komplex követelmények alkalmazása az automatatanulásban komoly kihívást jelent. Még viszonylag egyszerű követelményeket is gyakran végtelen halmazokon lehet csak definiálni - ilyenek például az élőségi követelmények.

Ezen munka célja az automatatanuláson alapuló szoftverfejlesztés elősegítése azon komplex követelmények meghatározásával, melyek algoritmikus módszerekkel támogathatók. Ezt követően a megfelelő formalizmusok integrálása egy létező keretrendszerbe, amely rendszerek és komponensek tervezését segíti elő interaktív automatatanulással.

Ez a dolgozat egy interaktív állapot alapú modellezési keretrendszert mutat be, amely támogatja a komplex bemeneteket – például LTL kifejezéseket –, valamint a modellek finomítását, mint a rendszertervezés hatékony eszközét. A bemutatott keretrendszer egyesíti a mérnöki gyakorlat és az automatizált megoldások előnyeit, támogatva az absztrakció finomításon alapuló modellezési folyamatot, amely a modellezett rendszer magas szintű követelmények tekintetében vett helyességét is képes figyelembe venni.

Abstract

Software engineering is a process, which starts at the level of high-level requirements, from which the software engineers design the system models and implement the software. Engineering software is a complex procedure, where model-based techniques can help to manage the complexity. However, designing high-quality models is still a tedious task.

State-based modeling supports the efficient description of the behavior of a system and automata learning algorithms can be used to synthesize such models from examples. Active automata learning constructs models by using queries and counterexamples, and thus can naturally be extended to synthesize models from the knowledge of engineers. We call this method interactive learning. However – as opposed to querying implementations, for which automata learning is frequently used – engineers usually start the development process by formulating complex requirements, and creating system models from them.

Using complex requirements in automata learning can be challenging. Even relatively simple requirements are often defined upon possibly infinite sequences, and cannot be described by finite means – as in case of liveness properties.

The objective of this work is to aid automata learning-based software engineering by defining the set of complex requirements that can be supported, then integrate corresponding formalisms into a framework to support the design of systems and components by interactive automata learning, allowing the engineer to focus on the expected behavior of the system and specify its behavioral requirements declaratively.

This thesis presents an interactive state-based modeling framework, which also supports complex inputs – such as LTL expressions – and also model refinement as an efficient means to design a system. The introduced framework combines the advantages of manual engineering practices and automated solutions, supporting the refinement-based modeling process that keeps the continuously evolving system correct-by-design.

Chapter 1

Introduction

Context Software engineering is a long and complex process, which starts with defining high-level requirements the envisioned software product must fulfill. The design models and the implementation are created based on these requirements and they also play an important role in verification. During these steps, model-based techniques may help with managing the complexity by the formalized application of modeling.

State-based modeling is a convenient way for characterizing the behavior of a system, but acquiring a correct model is still challenging. Apart from manual design, automated and semi-automated solutions also exist for this purpose. Examples include model synthesis techniques from various temporal logics and property specification languages, automata learning, and its extension utilizing human knowledge: interactive learning.

Problem Statement The application of automated techniques for model synthesis is often difficult in practical applications. On one hand, model synthesis from specification might be impossible due to the inherent abstraction in high-level requirements. On the other hand, techniques utilizing automata learning work best with trace-based requirements, mapping several inputs directly to outputs. This mapping is often infeasible to acquire from certain types of requirements, especially those, which are defined upon possibly infinte sequences of behavior – such as liveness and certain fairness properties.

Objective The objective of this work is to aid automata learning-based software engineering by exploring requirement types and formalisms and defining those which can be supported by algorithmic means. Then, to design the corresponding algorithms and integrate them into an existing interactive automata learning framework to support system and component synthesis possibly utilizing complex requirements extensively. This results in a modified version of interactive learning: a semi-automated solution driven by iterative refinement of declarative behavioral requirement specifications, allowing the designing engineer to focus only on the then relevant parts of the behavior and gradually complete the model.

Contribution This thesis presents an interactive state-based modeling framework combining the advantages of manual and automated solutions, which supports complex input formalisms – such as Linear-Time Temporal Logic – and also an iterative, abstraction-refinement-based workflow. It also demonstrates the capabilities of the thus extended framework on a case-study.

Outline The thesis is organized as follows. Chapter 2 provides an outline of the necessary theoretical background. Chapter 3 presents interactive learning, its reference architecture and the related approaches. Chapter 4 gives an overview of the possible applications of temporal logic in the methodology. Chapter 5 describes the tools and steps taken in creating the current implementation, emphasizing the main design decisions during the implementation of interactive learning, as well as its current extensions. Chapter 6 presents a case study to demonstrate the capabilities of the approach. Chapter 7 evaluates the designed algorithms and their effect on the whole framework. Chapter 8 provides concluding remarks and possiblities for further improvement.

Chapter 2

Background

This chapter provides the theoretical background of the thesis. Section 2.1 introduces model-based engineering, Section 2.2 and 2.3 discuss automata theory and automata learning and Section 2.4 describes different types of requirements. Finally, Section 2.5 presents state-of-the-art model synthesis techniques from these requirements.

2.1 Model-Based Engineering

Due to the extensive application of the modeling concept throughout this thesis, first of all, we need to define the meaning of *model* in this context.

Definition 1 (Model). A model is the simplified view of an element of the real or a hypothetical world (the system), that replaces the the system in certain considerations. ▪

For a model to be interpretable, executable or formally verifiable, it must be described according to predefined rules in the given domain. This set of rules is provided by *modeling languages*.

Definition 2 (Modeling Language). A modeling language consists of the following elements:

- *Metamodel:* a model defining the building blocks of the modeling language as well as their relationships.
- *Concrete syntax:* a set of rules defining a graphical or textual notation for the element and connection types defined in the metamodel.
- *Well-formedness constraints:* a set of constraints that models have to meet in order to be deemed valid in the modeling language.
- *Semantics:* a set of rules that define the meaning of the element and connection types defined in the metamodel. Semantics can be either *operational* (what should happen during execution) or *denotational* (given by translating concepts in a modeling language to another modeling language with well-defined semantics). ▪

Models can grasp various aspects of a system. Structural models describe the structure of the system, representing knowledge regarding the parts of the system and the properties and connections of these parts. This means that the model describes static knowledge

and not temporal change. On the other hand, behavioral models describe the change of the system over time through its changing of states and execution of processes. These categories do not cover every aspect of a system, and usually cannot be separated this well in practical applications. For instance, action languages of state-based models describe the behavior of the system in a procedural way. There are several possible formalisms for both kinds of models, some of which are discussed in the following section.

Model-Based Systems Engineering (MBSE) is the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases[13]. This concept can also be applied to software engineering. Note, that the models may be the primary artifact of the development process, in which case precisely defined formal models are required. When the models are the primary artifacts, the process is called *Model-Driven Engineering*.

2.2 Foundations of Automata Theory

In order to provide the theoretical background of behavioral modeling, this section discusses the necessary basics of formal language and automata theory.

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

2.2.1 Fundamentals of Formal Language Theory

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

Definition 3 (Alphabet). Let Σ be a finite, non-empty set. Σ is an alphabet, its elements are symbols or characters. ▪

Definition 4 (Word). If Σ is an alphabet, then any finite sequence comprised of the symbols of Σ are words. 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 finite word under an alphabet, formally $\bigcup_{n>0} \Sigma^n$ is denoted by Σ^*. The empty word is denoted by ϵ. Any word w with a length n can be viewed as a function $w : \{0, 1, ..., n-1\} \to \Sigma$. ▪

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

Definition 5 (Prefixes, Substrings and Suffixes). Let w be an arbitrary word, s.t. $w = uvs$ and $w, u, v, s \in \Sigma^*$. 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 6 (Formal Language). An arbitrary set of words under an Alphabet Σ is a Language. Formally: $L \subseteq \Sigma^*$. ▪

Definition 7 (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 $L = L'$. ▪

Especially in case of describing the behavior of reactive systems (or non-terminating programs), it makes sense to introduce the infinitary counterparts of these definitions. In the following, we use ω to denote the first infinite ordinal – as it is customary in set theory.

Definition 8 (ω-Sequence). If Σ is an alphabet, then any infinite sequence comprised of the symbols of Σ are ω-sequences (also called infinite sequences or infinite words). Any ω-sequence α can be viewed as a function $\alpha : \mathbb{N} \to \Sigma$. The set of all ω-sequences is denoted Σ^ω. ▪

Definition 9 (ω-Language). An arbitrary set of infinite words under an Alphabet Σ is an ω-Language. Formally: $L \subseteq \Sigma^\omega$. ▪

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

2.2.2 Finite Automata and ω-automata

Informally, automata are mathematical constructs which read words from an input and classify them into "accepted" and "rejected" categories. A bit more precisely, automata consist of states, some of which are accepting. Starting from an initial state, based on the inputs received, the automaton transitions between states. If after processing a sequence of inputs, the final state of the automaton is an accepting one, the input sequence is accepted. If not, the input is rejected.

One of the simplest automata is the so-called Deterministic Finite Automaton.

Definition 10 (Finite Automaton). A Finite Automaton is a tuple $DFA = (S, s_0, \Sigma, \delta, F)$, 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. ▪

One kind of finite automata, called *Deterministic Finite Automata* (DFA) refer to a property of every state having exactly one transition for every input. Formally: $\delta(s, a) \in S$, where $s \in S$ and $a \in \Sigma \cup \{\epsilon\}$.

The other kind of finite automata, called *Nondeterministic Finite Automata* (NFA) refer to a property of every state possibly having multiple transitions or maybe none for any given input. Formally: $\delta(s, a) \in S$, where $s \subseteq S$ and $a \in \Sigma$.

An example of a DFA (Deterministic Finite Automaton) from [29] can be seen in Example 1.

Example 1. *See Figure 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 Figure 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* $4i + 3a$*.*

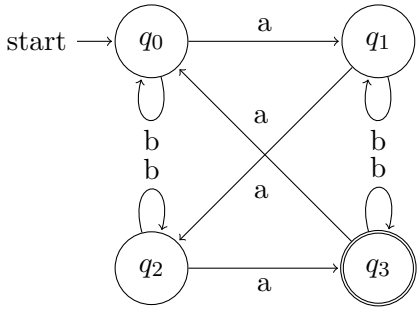


Figure 2.1: A simple DFA from [17].

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 Figure 2.1

A slightly different formalism can be defined for cases where the acceptance of the input (word) is not necessary to consider, called Labeled Transition System.

Definition 11 (Labeled Transition System). A Labeled Transition System is a tuple $LTS = (S, Act, \rightarrow)$, where:

- $S = q_0, q_1, ..., q_n$ the finite, non-empty set of states, q_0 being the initial state,
- $Act = a, b, c, ...$ the finite set of actions,
- $\rightarrow \subseteq S \times Act \times S$ the labeled transitions between the states.

In the beginning, the initial state is active. The active state may change after each transition.

The *path* of an LTS is the $\pi = (q_0, a_1, q_1, a_2, ...)$ alternating sequence of states and actions, where q_0 is the initial state and the subsequent states are the results of the transitions labeled with the actions of the same index, starting from the state with the previous index. ▪

Example 3. *See Figure 2.2. This example has three states,* $S = \{q_0, q_1, q_2\}$ *(hence* $|S| = 3$*). The initial state is* q_0*, also marked by the start arrow. The set of actions is* $Act = \{money, coffee, tea\}$*. Transitions are visualized as* $q_0 \xrightarrow{money} q_1$ *given by the transition (in this example)* $(q_0, money, q_1)$*. The set of transitions* $(\rightarrow)$ *also contains* $(q_1, coffee, q_2)$ *and* (q_1, tea, q_3)*.*

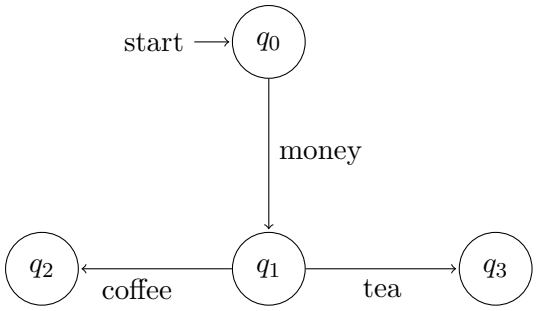


Figure 2.2: A simple LTS.

Finite automata and LTSs 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 (usually deterministic) automata designed to communicate with output symbols instead of accepting and rejecting states.

Definition 12 (Mealy Machine). A Mealy machine or Mealy automaton is a tuple $M = (S, s_0, \Sigma, \Omega, \delta, \lambda)$, 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,
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function and
- $\lambda : Q \times \Sigma \rightarrow \Omega$ 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.[29]

An example of a deterministic Mealy machine can be seen in Example 4.

Example 4. *An example of a deterministic Mealy machine can be seen in Figure 2.3. 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 Figure 2.3 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. Figure 2.3 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.*

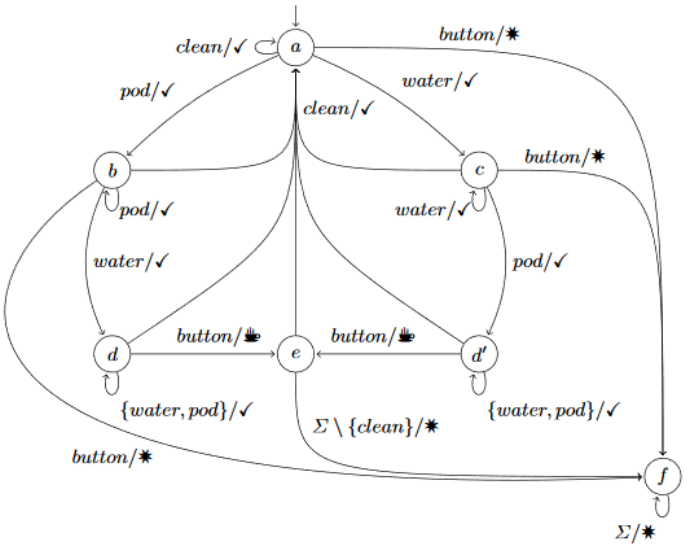


Figure 2.3: Mealy machine representing the functionality of a coffee machine. [29]

Another possible extension of finite automata is the acceptance of infinite input sequences. These automata are called *ω-automata*, most of which are nondeterministic and differ only in their acceptance condition. The most notable kinds of ω-automata are Büchi, Co-Büchi, Rabin, Streett, parity and Muller automata, some of which are defined here.

Definition 13 (ω-Automaton [30]). An ω-automaton is a tuple $A = (S, s_0, \Sigma, \delta, F)$, where:

- $S = \{0, 1, ..., k\}$ for some $k \in \mathbb{N}$. A finite, non-empty set containing the states of the automaton,
- $s_0 \subseteq S$ is the set of initial states,
- Σ is a finite alphabet,
- $\rho : S \times \Sigma \to 2^S$ is the nondeterministic transition function,
- $F \subseteq S$ is a set of the final states of the automaton.

A run of the ω-automaton A is the $r = (s_0, s_1, s_2, ...)$ infinite series of states as a result of an $a_0, a_1, a_2, ...$ infinite input (word), where $s_0 \in S_0$ and $\forall s_{i+1} = \rho(s_i, a_i)$.

The characteristic of an infinite run is the set of $s \in S$ states, which occur infinitely many times during the run. Formally: $lim(r) = \{s | \nexists j \geq 0 : \forall k > j : s \neq s_k\}$.

Büchi Acceptance Condition: A run of the Büchi automaton is accepting, if $lim(r) \cap F \neq \emptyset$. A w infinite word is accepted by the automaton, if there exists a run of the automaton that accepts w.

Generalized Büchi Acceptance Condition: A Generalized Büchi automaton has zero or more final states, denoted $F_i \subseteq S$. A run of the Generalized Büchi automaton is accepting, if $\forall i : lim(r) \cap F_i \neq \emptyset$. A w infinite word is accepted by the automaton, if there exists a run of the automaton that accepts w.

Co-Büchi Acceptance Condition: A run of the co-Büchi automaton is accepting, if $lim(r) \subseteq F$. A w infinite word is accepted by the automaton, if there exists a run of the automaton that accepts w.

Parity Acceptance Condition: A run of the parity automaton is accepting, if $min(lim(r))$ is even. A w infinite word is accepted by the automaton, if there exists a run of the automaton that accepts w.

It is also possible to define ω-automata with a set of final transitions $F \subseteq \rho$. This requires the definition of runs using transitions, but can be used in a quite similar way. This type of ω-automata are often referred to as *transition-based ω-automata*, and generally perform better in implementations. ▪

Example 5. *Figure 2.4 shows a simple Büchi automaton. Notice the nondeterminism and how the automaton accepts only infinite runs.*

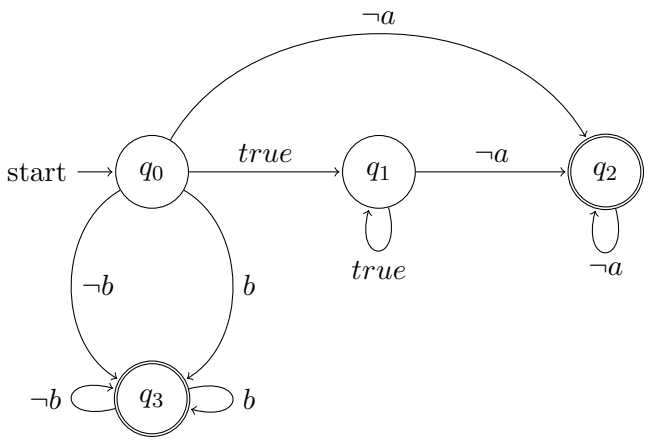


Figure 2.4: A simple Büchi automaton.

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 automata and formal languages are discussed more in-depth in the following subsection.

2.2.3 Relations of Formal Languages and Automata

Definition 14 (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 15 (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 16 (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 $\equiv_M$.[19]. ▪

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[19], 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[19][26]). 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 [19][26]. ▪

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 for the input. Each specific run has a pattern of $i_1, o_1, i_2, o_2, .., i_n, o_n$, where i are inputs and o are outputs, called a *trace*. In order to characterize these runs, we actually do not need every output from the corresponding trace, 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 6. *Given the Mealy machine $M_{coffeemachine}$ in Figure 2.3, the runs <clean, ✓> and <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 17 (Equivalence of words wrt. $\equiv_P$[29]). 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 7. *Taking Figure 2.3 as an example, the following words are equivalent wrt. $\equiv_{[\![M]\!]}$:*

$water, pod$
$water, water, pod$
$pod, pod, water$.

The first two 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[29]). 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. ▪

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.

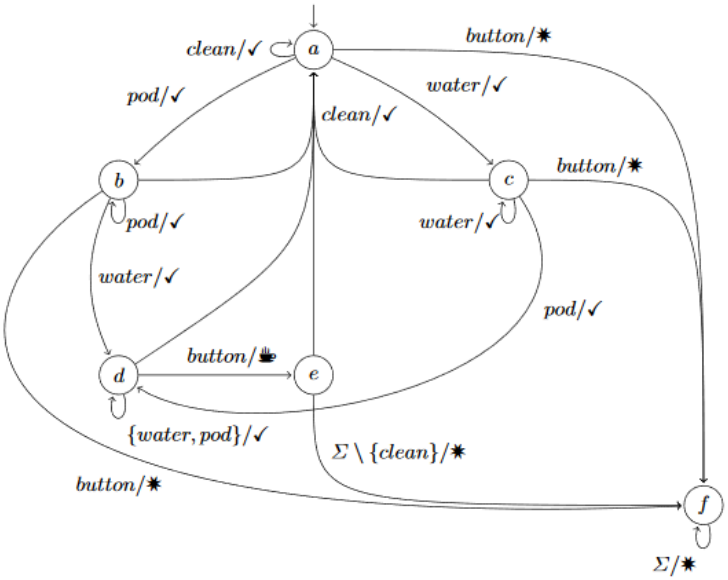


Figure 2.5: Minimal version of the Mealy machine seen in Figure 2.3.

Definition 18 (ω-Regular Language). A formal language L is ω-regular, iff there is a Büchi Automaton M, for which L(M) = L.

As we have already seen, several kinds of ω-automata have the same expressive power, thus, ω-regular languages could also be defined using any formalism equivalently expressive as Büchi automata. ▪

Note, that the Myhill-Nerode theorem does not hold for ω-regular languages.

2.2.4 Minimization of Mealy Automata

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

Definition 19 (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 Figure 2.3 can be seen in Figure 2.5.

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

Proposition (Bounded reachability[29]): 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 20 (k-distinguishability[29]). 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 21 (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 different 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 s_2 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. [29]

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 8. *Given the Mealy machine seen in Figure 2.3, 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 Figure 2.5.

The process explained in Example 8 is partition refinement, the exact algorithm and proof of its validity can be seen in [29]. Partition refinement is a version of the minimization algorithm for DFAs proposed by Hopcroft[14].

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

Definition 22 (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 [29]. Essentially this establishes P_1 and P_2 from Example 8 as images of the Mealy machine seen in Figure 4 under k-epimorphisms where k=1 and k=2 respectively.

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

Note, that although it is possible – and often quite useful – to define minimal ω-automata (the one having the least number of states), however, several problems arise during the minimization process, due to not having a theorem resembling Theorem 1 in this case. Firstly, there is no "canonical" acceptance condition: an automaton of one acceptance condition can have exponentially more states than another one with a different acceptance condition (both accepting the same language). In addition, there may be several, topologically different minimal automata with the same acceptance condition accepting the same language. This results in an NP-complete problem out of the scope of this thesis, thus, not discussed futher. For this reason, traditional active automata learning algorithms for finite automata are not applicable for ω-languages.

2.3 Automata Learning

Automata 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 Σ [15].

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

Passive Automata Learning In case of passive automata learning, the gathering of information is not part of the learning process, but rather a prerequisite. The learning is performed on a pre-gathered positive an/or negative example set of the systems behavior. In passive automata 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 Automata Learning In case of active automata learning, the behavioral information 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 automata learning follows the MAT, or the Minimally Adequate Teacher model proposed by Dana Angluin [4]. 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 returns 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 attempts to determine if the hypothesis is behaviorally equivalent to the SUL, and if not, finding the diverging behavior, and returning 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 counterexample 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 cannot be found this way, the learner's hypothesis is behaviorally equivalent to the SUL. The learning can terminate and the output of the learning is the current hypothesis.

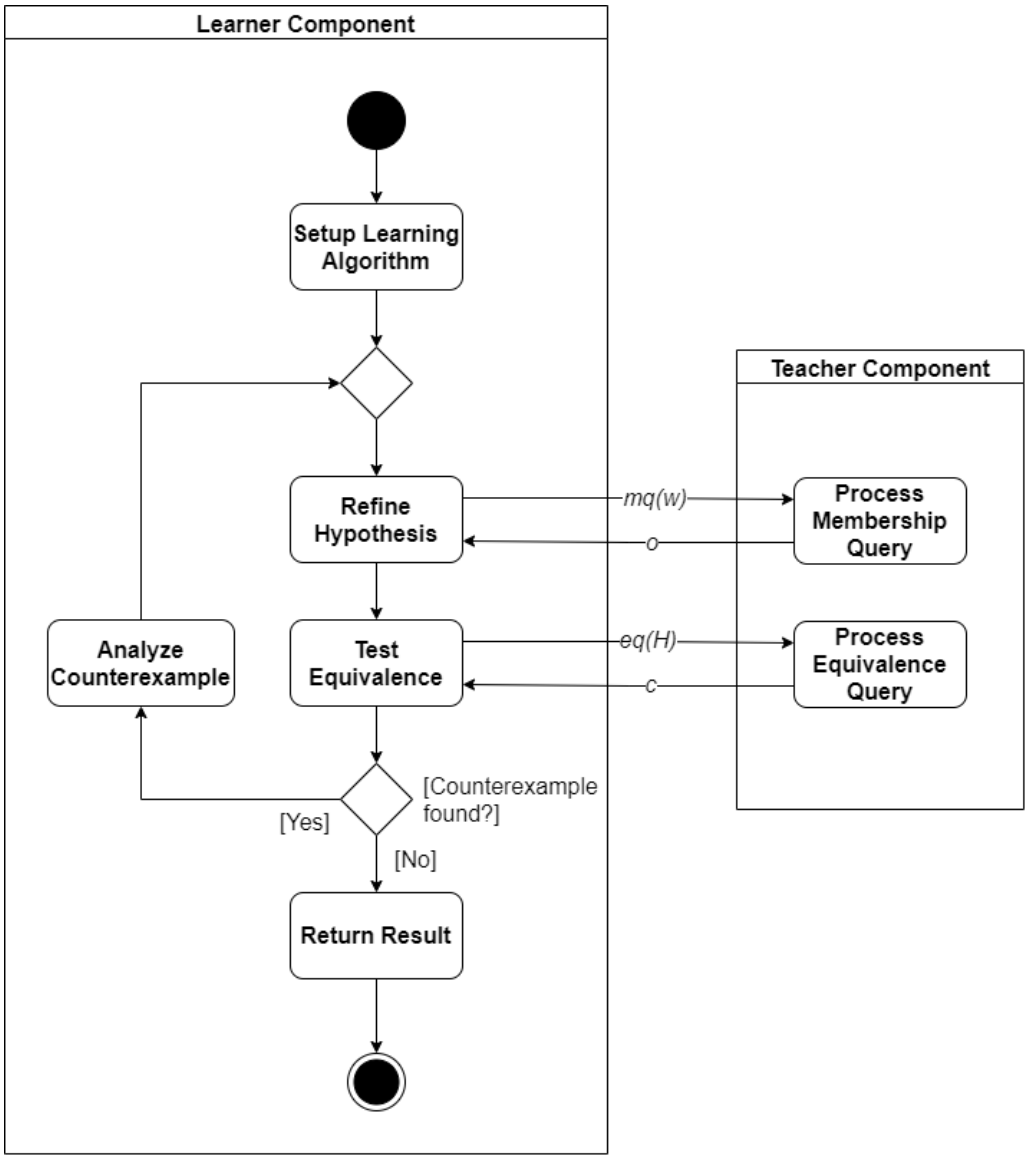


Figure 2.6: Active automata learning.

As seen in Figure 2.6, 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 [29]:

- 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 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), the following definition is introduced.

Definition 23 (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 the SUL, and not the perhaps much larger Mealy machine of the 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 [29]

- 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 [29].

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 basic active automata learning algorithms this thesis builds on.

2.3.1 Traditional Automata Learning Algorithms

Direct Hypothesis Construction (DHC): The Direct Hypothesis Construction algorithm, for which the hypothesis construction can be seen in Algorithm 1, follows the idea of the breath-first search (BFS) algorithm in 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, 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. [29][23]

Algorithm 1: Hypothesis construction of the Direct Hypothesis Construction algorithm as seen in [29].

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

```
1  initialize hypothesis H, create a state for all elements of S_p
2  initialize a queue Q with the states of H
3  while Q is not empty do
4  |  s = dequeue state from Q
5  |  u = access sequence from s_0 to s
6  |  for d ∈ D do
7  |  |  o = mq(ud)
8  |  |  set λ(s, d) = o
9  |  end
10 |  if exists an s' ∈ S, where the output signature of s' is the same as s then
11 |  |  reroute transitions of s to s' in H
12 |  |  remove s from H
13 |  else
14 |  |  create and enqueue successors of s for every input in Σ into Q, if not
   |  |    already in S_p
15 |  end
16 end
17 Remove entries of D \ Σ from λ
18 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 whether a counterexample c exists – similarly to the process in Figure 2.6. If no counteraxamle can be found, the learning terminates, H is the learned automaton. 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).

The DHC algorithm is a straightforward implementation of active automata learning. 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[23].

L^*_M Algorithm: The L^*_M algorithm – seen in Algorithm 2 – is similar to the DHC algorithm, however, it maintains a table thoroughout its entire run containing the previously gathered information. Informally, the rows of the table correspond to states and state candidates of the hypothesis, and are labeled with a possible access sequence of the given state (candidate). The columns of the table correspond to the suffixes of the access sequence containing separating behavior. Thus, the fields of the table are the results of $mq(prefix\ suffix)$. The fields are indexed with $Obs(prefix, suffix)$.

More precisely, the rows are divided into two disjoint sets: S_p contains unique rows, which correspond to the states during the hypothesis construction. L_p contains rows, which are labeled with the labels of S_p plus an element from the input alphabet – i.e. the continuation of the access sequences of S_p.

Before each hypothesis construction, the consistency and closedness of the table must be ensured. This means, that S_p must only contain unique rows, and each row in L_p must

have a corresponding row in S_p with the same elements. This can be reached by iteratively adding unique rows of L_p to S_p and adding their continuations to L_p.

This algorithm results in a canonical automaton of the learned system, but does not ensure, that the intermediate hypotheses are minimal. That property requires D to be semantically suffix-closed before the equivalence queries. We say that D is semantically suffix-closed for the hypothesis H, if for any two states $u, u' \in S_p$ and any decomposition $v_1v_2 \in D$ of any suffix with $Obs(u, v_1v_2) \neq Obs(u', v_1v_2)$. It is possible to ensure this property using a simple algorithm described together with the original one in [29].

Algorithm 2: Hypothesis construction of the L^*_M algorithm as seen in [29]

```
   Input: Σ: an input alphabet
   Output: A Mealy machine H = (S, s_0, Σ, Ω, δ, λ)
 1 initialize table T (...)
 2 while true do
 3 |   construct H by algorithm Close Table
 4 |   while Check Semantic Suffix-Closedness returns a separator d do
 5 |   |   D := D ∪ {d}
 6 |   |   construct H by algorithm Close Table
 7 |   end
 8 |   counterExample := eq(H)
 9 |   if counterExample is empty then
10 |   |   return H
11 |   end
12 |   d := Process CounterExample
13 |   D := D ∪ {d}
14 end
15 return H
```

The L^*_M algorithm is more refined implementation of automata learning than the DHC algorithm. Let n denote the number of states in the final hypothesis and k the size of the input alphabet. The algorithm terminates after at most $n^2k + k^2n + nlog(m)$ membership queries, the first two terms resulting from the maximum size of the table and the last term as a result of processing the equivalence queries. It also asks at most n equivalence queries. Also, all computation that is required on the table and the construction of hypothesis models is polynomial in the size of the final observation table, which contains maximum $n^2k + k^2n$ elements.

2.4 Specifying Complex Requirements

The previous sections introduced different modeling types and techniques. We now discuss the requirements used in model-based engineering which the models need to satisfy.

2.4.1 Requirements and Properties

The concept of requirement is widely used in connection with interactive learning, therefore, it is essential to define it precisely.

Definition 24 (Requirement[1]).

1. A condition or capability needed by the user to solve a problem or achieve an objective.
2. A condition or capability that must be met or possessed by a system component to satisfy a contract, standard, specification or other formally imposed documents.
3. A documented representation of a condition or capability as in (1) or (2). ▪

Basically, requirements are properties of the system, which the system must satisfy. They can be specified in many different ways, the most common being textual requirements in traditional feature lists. This method is an informal way of requirement specification, as the structure of this format is hard to analyze due to it lacking a precise definition. Attempts were made to formalize this type of requirement specification by defining patterns and mapping them to formal semantics, however, there are also more general, abstract approaches, such as *temporal logics*.

The rationale behind the precise formalization is the wide range of automated applications, especially in *formal methods* – such as validation, formal verification, test oracle generation, documentation generation or even code generation. The rest of this subsection defines basic concepts concerning properties, based on [6].

Definition 25 (Linear-Time Property). A linear-time property (LT property) P over the set of atomic propositions AP is defined as $P \subseteq (2^{AP})^{\omega}$

An LT property is thus an ω-language over the alphabet 2^{AP}. ▪

LT properties can be used to describe restictions on the traces of transition systems, such as LTSs – see Subsection 2.2.2 – or Kripke-structures [21].

Definition 26 (Satisfaction of LT Properties). Let $TS = (S, Act, \rightarrow)$ be a (labeled) transition system, $AP = Act$ the set of atomic propositions and P an LT property over AP. TS satisfies P – denoted $TS \vDash P$ – iff $Traces(TS) \subseteq P$.

This means, that a transition system TS satisfies the LT property P, if all its traces respect P. ▪

Example 9. *Figure 2.7 depicts a system of two fully synchronized traffic lights, each only having two possible actions: red and green. The set of atomic propositions is* $AP = \{red_1, green_1, red_2, green_2\}$.

Consider the LT property: $P =$ *"The first traffic light is infinitely often green." This corresponds to the set of infinite words of the form* $A_0A_1A_2...$*, such that* $green_1 \in A_i$ *holds for infinitely many* i.

Examples of such infinite words include:

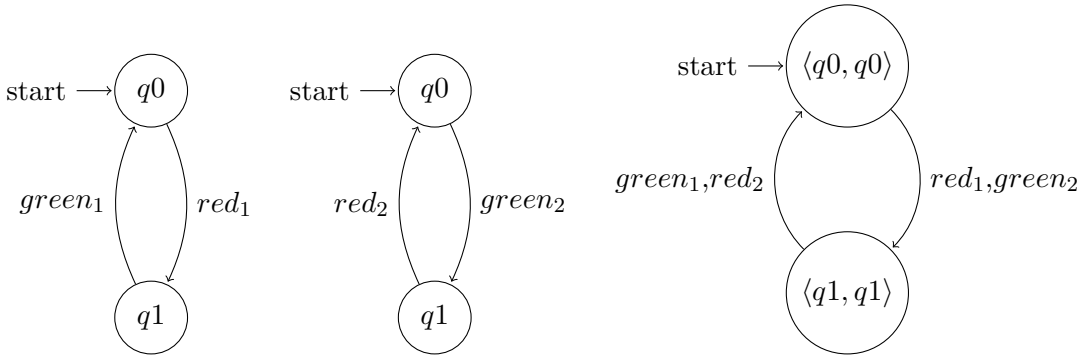
- $\{green_1\}\emptyset\{green_1\}\emptyset\{green_1\}\emptyset\{green_1\}\emptyset...$
- $\{red_1, green_2\}\{green_1, red_2\}\{red_1, green_2\}\{green_1, red_2\}...$

Examples of infinite words not included in P*:*

- $\{red_1, green_1\}\{red_1, green_1\}\emptyset\emptyset\emptyset...$ – *it contains finitely many occurrences of* $green_1$
- $\{red_1, red_2\}\{red_1, red_2\}\{red_1, red_2\}...$ – *it contains no occurrences of* $green_1$

Naturally, finite words do not satisfy P – by definition, they never satisfy LT properties.

It is easy to see, that both the model of the first controller (Figure 2.7a) and the product controller (Figure 2.7c) satisfy P.



(a) 1st controller **(b)** 2nd controller **(c)** Product automaton

Figure 2.7: A simplified model of two fully synchronized traffic lights based on [6].

Following the natural distinction between what a modeled system *should* and what it *should not* do, it is possible to introduce the terms *safety* and *liveness* properties.

Definition 27 (Safety Property). An LT property P_{safe} over AP is called a *safety* property iff for all words $\sigma \in (2^{AP})^\omega \backslash P_{safe}$ there exists a finite prefix $\hat{\sigma}$ of σ such that $P_{safe} \cap \{\sigma' \in \sigma \in (2^{AP})^\omega | \hat{\sigma} \text{ is a finite prefix of } \sigma'\} = \emptyset$.

In that case, $\hat{\sigma}$ is called a *bad prefix*. A *minimal bad prefix* is a bad prefix of minimal length. ▪

Definition 28 (Liveness Property). An LT property P_{live} over AP is called a *liveness* property iff for all finite words $w \in (2^{AP})^*$ there exists an infinite word $\sigma \in (2^{AP})^\omega$ satisfying $w\sigma \in P$. ▪

Theorem 3. The only LT property over AP that is both a safety and a liveness property is $(2^{AP})^\omega$. [6] ▪

Theorem 4. For any LT property P over AP, there exists a safety property P_{safe} and a liveness property P_{live} such that $P = P_{safe} \cap P_{live}$. [6] ▪

Example 10. *Consider the property "the machine provides tea infinitely often after initially providing coffee three times in a row" regarding a vending machine. This can be decomposed into two parts.*

The first part – "provides tea infinitely often" – is a liveness property: any finite trace can be extendid such that the property holds.

The second part – "initially providing coffee three times in a row" – is a safety property: any finite trace in which one of the first three drinks is tea (or generally, not coffee) violates it.

There is also a third aspect frequently mentioned in the literature: *fairness* properties. These are assumptions that rule out infinite behaviors that are considered unrealistic, and are often necessary to establish liveness properties. However, they are out of the scope of this thesis and are not detailed further.

The class of ω-lanugages resulting from the above definition of LT properties is too general to handle. For simpler (i.e. decidable) processing and verification of these properties, we restrict the language of the properties to ω-regular languages. We will see, that this is enough in most practical use-cases.

Definition 29 (ω-Regular Properties). An LT property P over the set of atomic propositions AP is called ω-regular, if P is an ω-regular language over the alphabet 2^{AP}.

More complex constructs also exist in the literature, but these raise serious problems – such as undecidability – like context-free extensions introduced in [10].

In the following subsection, we are going to introduce languages for the specification of different LT properties.

2.4.2 Linear-Time Temporal Logic

Linear-Time Temporal Logic (LTL), also called Propositional Linear-Time Temporal Logic (PLTL) is the extension of propositional logic with temporal connectives over *paths* of a *base model*, e.g. an LTS. It is common to use definitions using Kripke-structures [21] as base models too, depending on the application. The syntax of LTL expressions over paths of LTSs is defined as follows:

Definition 30 (Syntax of LTL Expressions). Let $\pi = (s_0, a_1, s_1, a_2, ...)$ be a path of an LTS. Then the valid LTL expressions can be derived using the following production rules:

- L_1: if $a \in Act$, then (a) is an LTL expression.
- L_2: if p and q are LTL expressions, then $p \wedge q$ and $\neg p$ are LTL expressions.
- L_3: if p and q are LTL expressions, then pUq and Xp are LTL expressions.

With the operator precedence: $\leftrightarrow$ « $\rightarrow$ « $\wedge$ « $\neg$ « X, U .

Additional operators can also be defined using the already defined ones:

- $true$ holds for every state,
- $false$ does not hold for any state,
- $p \vee q$ as $\neg((\neg p) \wedge (\neg q))$,
- $p \rightarrow q$ as $(\neg p) \vee q$,
- $p \leftrightarrow q$ as $(p \rightarrow q) \wedge (q \rightarrow p)$,
- Fp as $trueUp$,
- Gp as $\neg F(\neg p)$,
- pWq as $\neg((\neg p)Uq)$, also denoted as $p\ WB\ q$
- pBq as $\neg((\neg p)Uq) \wedge Fq$

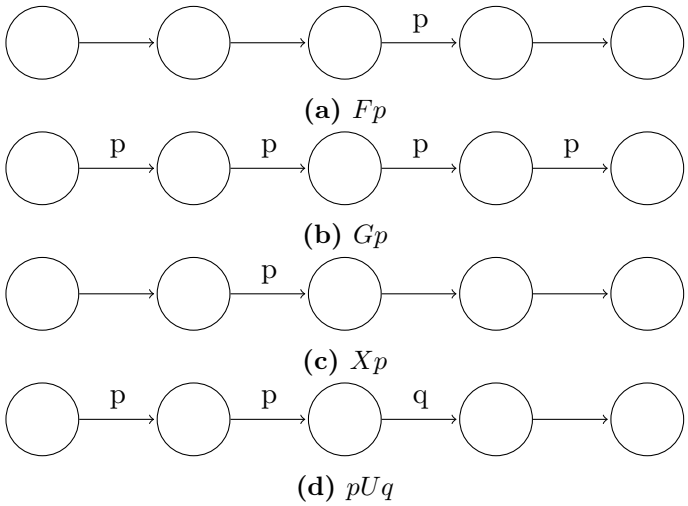
The semantics of LTL expressions are defined as follows:

Definition 31 (Semantics of LTL Expressions). Let $\pi = (s_0, a_1, s_1, a_2, ...)$ be a path of an LTS model $M = (S, Act, \rightarrow)$. Then the formal semantics to the LTL expressions a, p and q is given recursively, with regard to syntactic production rules as:

- L_1: $M, \pi \vDash (a) \leftrightarrow a_1 = a$
- L_2: $M, \pi \vDash p \wedge q \leftrightarrow M, \pi \vDash p$ and $M, \pi \vDash q$;
  $M, \pi \vDash \neg q \leftrightarrow$ not $M, \pi \vDash q$
- L_3: $M, \pi \vDash (pUq) \leftrightarrow \exists j \geq 0 : \pi^j \vDash q$ and $\forall 0 \leq k \leq j : \pi^k \vDash p$;
  $M, \pi \vDash Xp \leftrightarrow \pi^1 \vDash p$ ▪

Where $\vDash$ is the logical entailment operator, $M, \pi \vDash q$ denoting: for trace π of model M, q holds.

Example 11. *Based on these definitions, Figure 2.8 shows examples for the intuitive meanings of different LTL operators.*



(a) Fp **(b)** Gp **(c)** Xp **(d)** pUq

Figure 2.8: Intuitive examples for the meanings of different LTL operators.

LTL expressions over paths of LTS models can be used to formulate requirements for any system interpretable as LTSs (such as Mealy machines) in a formalized way resembling conventional propositional logic – which is widely used among engineers. LTL is a well-researched area of mathematics, and has an extensive tooling available for different purposes. One such application is the transformation to ω-automata, which then can be executed parallel with the modeled system to verify its behavior.

Example 12. *Figure 2.9 shows a possible transition-based Generalized Büchi Automaton corresponding the LTL specification* $(GFa) \leftrightarrow (GFb)$*. A run is accepting if it visits both edges marked with 0 and 1 infinitely often.*

Based on Definition 31, it can be shown that LTL expressions are only capable of describing ω-regular LT properties. On the other hand, LTL only possesses the expressive power to describe a proper subset of all ω-regular languages [11]. This makes room for extensions and different property specification approaches, which are discussed in the following subsections.

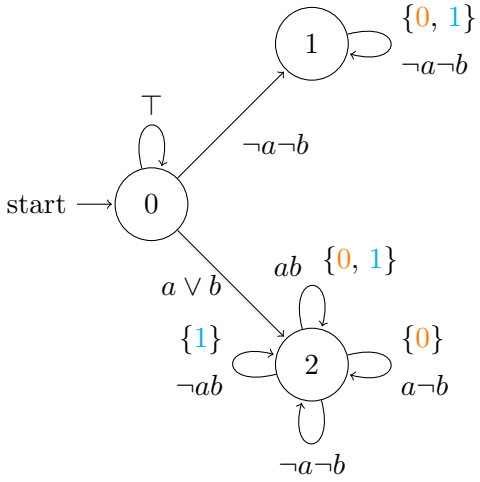


Figure 2.9: A transition-based GBA for the specification $(GFa) \leftrightarrow (GFb)$.

2.5 Reactive Synthesis from Temporal Logic

The previous section defined requirements and properties. A straightforward way of utilizing these requirements in connection with automata learning is to transform them to its targeted formalism – a Mealy automaton. Luckily, model synthesis from ω-regular LT properties is a well-researched area with an extensive literature. This section introduces the most common synthesis techniques from such properties to Mealy automata.

From the definition of ω-regular properties it follows, that an ω-automaton must exist for each ω-regular property. We have also seen, that several formalisms exist for the formulation of ω-regular properties, each fit for a different subset of these properties. Thus, we are only going to discuss the problem of finding an appropriate controller – in this case a Mealy machine – for a given ω-automaton.

2.5.1 Synthesis Problem for Reactive Systems

Formally, an instance of the *reactive synthesis problem* is a triple (I, O, L), where I and O are two disjoint sets of input and output events (words) respectively, and L is an ω-regular language over the alphabet $2^{I \cup O}$ – given as an ω-automaton.

A *controller* is a function $C : (2^{I \cup O})^* \times 2^I \rightarrowtail 2^O$. An infinite word $u = u_0 u_1 u_2 \cdots \in (2^{I \cup O})^\omega$ is *consistent* with controller C if, for every $n \in \mathbb{N}, u_n \cap O = C(u_0, \ldots, u_n \cap I)$. A controller C is said to *enforce* L if every infinite word consistent with C is in L. Given I, O and L, the reactive synthesis problem asks, whether there exists a controller enforcing L, and also asks for a witness if the answer is positive [24].

From this triple-based definition of the reactive synthesis problem, game theory and the formal definition of games is a natural association.

2.5.2 Games and Reactive Systems

A *game* is an ω-automaton, where the set of states Q is partitioned into two sets Q_A and Q_B for two players A and B. We refer to the underlying automaton (seen as an automaton and not as a game) as its *arena*. Runs of the arena are called *plays* in the game setting. By convention, the acceptance condition of the arena is the winning condition for the second

player (B) in the game: a play is won by B iff it is accepting in the arena. Otherwise, the play is won by A (in particular, finite plays are won by A). It is also convenient to partition Σ into Σ_A (letters played by A) and Σ_B (letters played by B). This restricts Δ (the transition relation) of the ω-automaton so that transitions from Q_A and Q_B are labelled with letters from Σ_A and Σ_B respectively.

A *strategy* for A (resp. B) is a function mapping finite plays ending in Q_A (resp. Q_B) to a valid transition extending the play. Given a strategy σ for A (resp. B), a σ-play is a (finite or infinite) play ρ where every transition taken by A (resp. B), say at position i, is given by $\sigma(\rho_0 \dots \rho_i)$. A *positional* (or *memoryless*) strategy is a strategy whose value depends only on the state in which the given run ends. Given one strategy for each player, say σ_A and σ_B, there is a unique longest play that is both a σ_A-play and a σ_B-play, which is called the outcome of σ_A and σ_B. A winning strategy for a player is a strategy that makes that player win its outcome against every strategy for the other player. A game is *turn-based* if Δ alternates between Q_A and Q_B (i.e. no player plays twice in a row).

It is now clear, that the defined games and the reactive synthesis problem are closely related – just swap the players A and B to the I and O of the synthesis problem – and the old and well-researched game-theory can be applied to efficiently solve it.

In fact, most state-of-the-art approaches – such as LTLSYNT [24] and STRIX [22] – use parity-game solving for model synthesis, which we discuss next.

2.5.3 Synthesis through Parity Games

The ω-automata, based on which the controller is constructed, does not necessarily separate input and output symbols. This results in smaller automata – perfect for verification – that are hard to utilize in model synthesis. Thus, we need to define a corresponding intermediate formalism, which takes this into consideration.

Definition 32 (Split Word, Language, Automaton). Let a *word* $u = (u_i)_{i\in\mathbb{N}} \in (2^{I\cup O})^\omega$. We define $split_{I,O}(u) = (v_i)_{i\in\mathbb{N}} \in (2^I 2^O)^\omega$ as follows: for all $i \in \mathbb{N}$, $v_{2i} = u_i \cap I$ and $v_{2i+1} = ui \cap O$. This operation naturally extends to *languages*: $split_{I,O}(L) = \{split_{I,O}(u) | u \in L\}$.

The split of an *automaton* $A = (Q, 2^{I\cup O}, \Delta, q_0, F)$ is the automaton, noted $split_{I,O}(A)$, $(Q \cup Q \times 2^I, 2^{I\cup O}, \Delta_s, q_0, F)$ where each transition $(q, a, f, q_0) \in \Delta$ gives, for each $i \in a \cap 2^I$, two transitions in Δ_s: $(q, i, \emptyset, (q, i))$ and $((q, i), a \cap 2^O, f, q')$. ▪

Example 13. *Figure 2.10 shows an example of a split automaton for the Büchi automaton of Example 12. Notice, how the splitting separates the edges to ones containing only input symbols and those containing only output symbols. Also note, that the split automaton is nondeterministic and exhibits the acceptance condition of the original automaton: the transition-based generalized Büchi condition.*

For the construction of an appropriate controller, the following two theorems give clues. For the proofs, see [24].

Theorem 5. Let G be a game whose arena is complete and deterministic for player A and that recognizes $split_{I,O}(L)$. If player B wins G, then there is a controller enforcing L. ▪

Theorem 6. Let G be a game whose arena is complete for A, deterministic (for both players) and recognizes the language $split_{I,O}(L)$. If there is a controller enforcing L, then B wins the game G. ▪

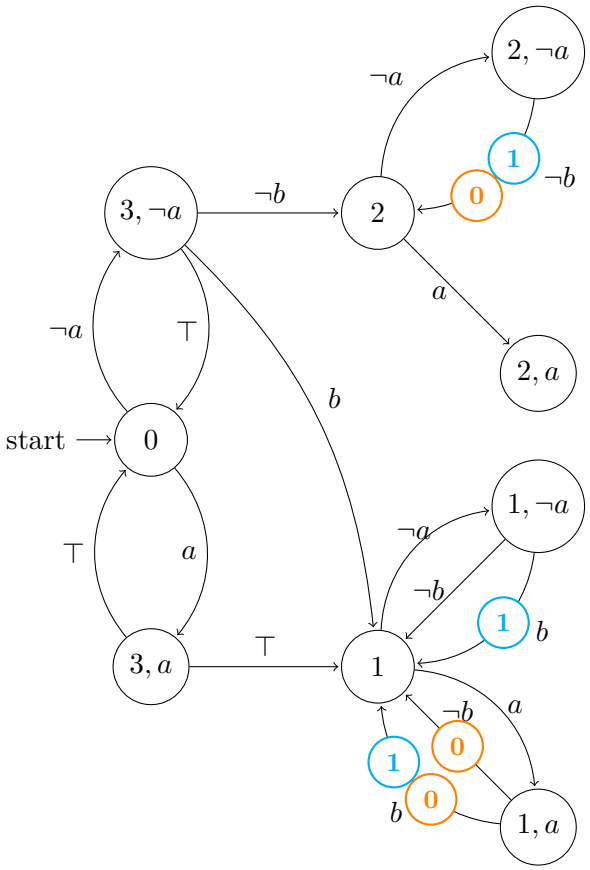


Figure 2.10: Split automaton of the automaton from Figure 2.9 based on [24].

Note, that both theorems require the arena of the game to be deterministic, and no algorithms are known for the nondeterministic case. This restricts the applicability of certain ω-automata: for instance, deterministic Büchi automata are strictly less expressive than their nondeterministic counterparts. Thus, most approaches use deterministic parity automata (DPA) as the arena of the games to solve. Deterministic and nondeterministic parity automata are equally able to express all ω-regular languages and they also have a natural association to turn-based games. Thus, the next step of these synthesis algorithms is the determinization of the input ω-automata by applying a transformation to a DPA. A possible algorithm for the transformation of transition-based Büchi automata to transition-based DPA is that of Redziejowski [27].

This is followed by the solution of the parity game. An appropriate algorithm is Zielonka's recursive algorithm [33], with the time complexity of $O(n^d)$, where d is the number of different priorities in the game. There are other, quasi-polinomial algorithms too, which can be applied with little improvement in most practical cases.

Example 14. *Figure 2.11 shows a parity game constructed from the automaton in Example 13 – with a deterministic parity automaton as its arena. The diamond nodes belong to Player A – the environment player – and the circular nodes belong to player B – the system player. Player B wins, if the minimum priority encountered infinitely often is odd.*

The solution is also shown: the highlighted nodes and transitions are in the winning strategy, while the grey elements are unreachable.

When Player B has a winning strategy σ from the initial state, we can extract a controller from σ that ensures realisation of the specification. It corresponds to an incompletely

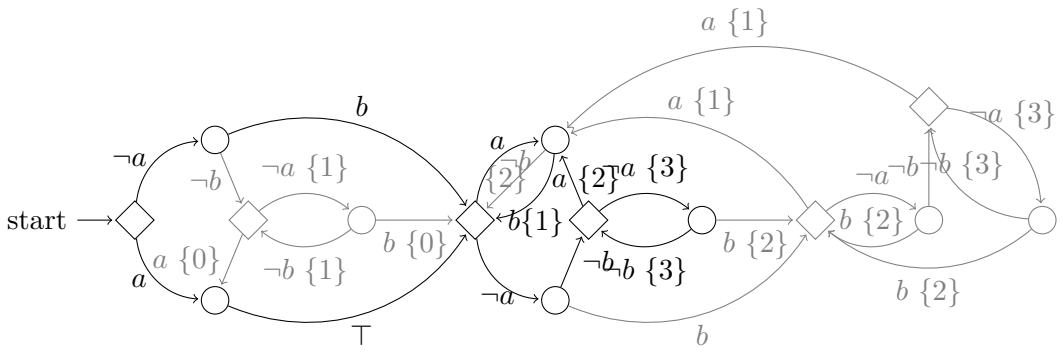


Figure 2.11: Parity game from the automaton in Figure 2.10 based on [24].

specified Mealy machine, where some outputs might not be specified and could be instantiated either way. This enables more compact representations and retains the possiblities in the implementations.

Let (V_A, V_B, E) be the parity game arena where B wins from the initial state with the strategy σ. We can use $Q = \{q \in V_A |$ Player B wins the game applying the strategy $\sigma\}$ as the set of states. For the transition function, let $\delta(q, i) := q'$ by choosing some q' where $((q, I), O, q') \in \sigma$ for some $I \subseteq \Sigma_{in}$ with $i \in I$ and any $O \subseteq \Sigma_{out}$. By construction, and as σ is a winning strategy for all $q \in Q$, such a q' always exists. However, there may be multiple applicable q'. This may result in several possible – not necessarily minimal – controllers. For the output function, let $\delta(q, i) = q'$ be an edge of the Mealy machine. Then, let $\lambda(q, i) = \{o \mid \exists I, O : i \in I$ and $o \in O$ and $((q, I), O, q) \in \sigma\}$ – a Boolean formula encoding all the possible corresponding outputs in a nondeterministic way.

Example 15. *Figure 2.12 shows a Mealy automaton constructed from the winning strategy of the parity game in Example 14.*

Note, that the automaton is nondeterministic – not fully specified – and non-minimal – q_1 and q_2 could be merged. The minimal automaton can be seen in Figure 2.13.

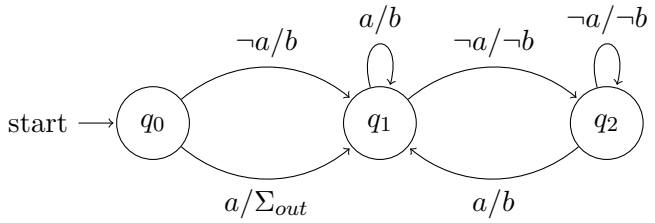


Figure 2.12: (Non-minimal) Mealy machine from the parity game in Figure 2.11.

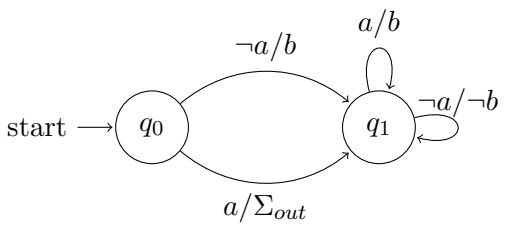


Figure 2.13: Minimal, non-deterministic Mealy machine from the non-minimal version in Figure 2.12.

Chapter 3

Interactive Learning and Related Work

This thesis is built on three main pillars. First and foremost, it is the extension of a joint work on interactive learning [8], summarized in Section 3.1. Second, it is heavily based on model synthesis techniques from LTL formulae, already introduced in Section 2.5. The third one is model refinement, which is a well-known concept in engineering, also undefined in the context of automata learning.

This chapter summarizes interactive learning in Section 3.1, and also reviews the related approaches in Section 3.2.

3.1 Interactive Learning

Previous works proposed an interactive approach for designing reactive systems, the most relevant aspects of which are summarized in this section. In Subsection 3.1.1, the application of this methodology is presented from the users' point of view: the intended usage of the interactive automata learning framework – also called *Interactive Learning Entity* or ILE - and its utilization in the design of reactive systems in a declarative way. Then, in Subsection 3.1.2, the relevant parts of the architecture, software components, algorithms and data structures are presented.

3.1.1 Overview of the Methodology

The methodology is heavily based on the interaction of the user with the ILE. The different types of interacitons are summarized in Figure 3.1. These interactions take place in a predefined order – the *proposed workflow*, illustrated in Figure 3.2. Its most important steps are explained later in this section. This workflow consists of two phases: first, an *offline* one, and then an *online* one, and ends with the serialization of the models. During the offline phase, the ILE offers little assistance, the designing engineer must determine the required details by other means. The interactive system design happens during the online phase.

The main characteristic of the input formalisms of individual steps in both the offline and the online phases is the declarative way of describing the system components. This allows the engineer to focus solely on the expected behavior and acquire a minimal model exhibiting the specified functionality.

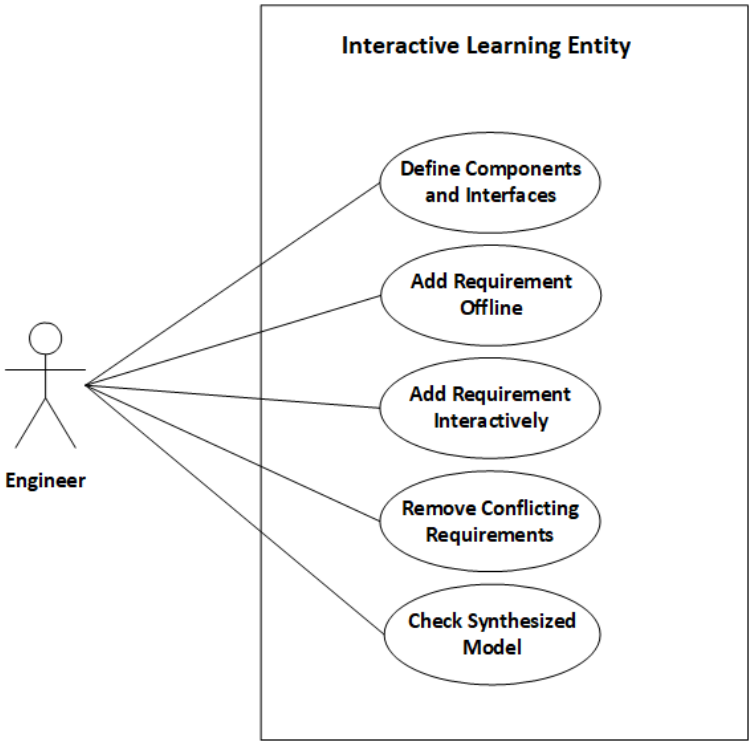


Figure 3.1: Interaction types between the engineer and the ILE.

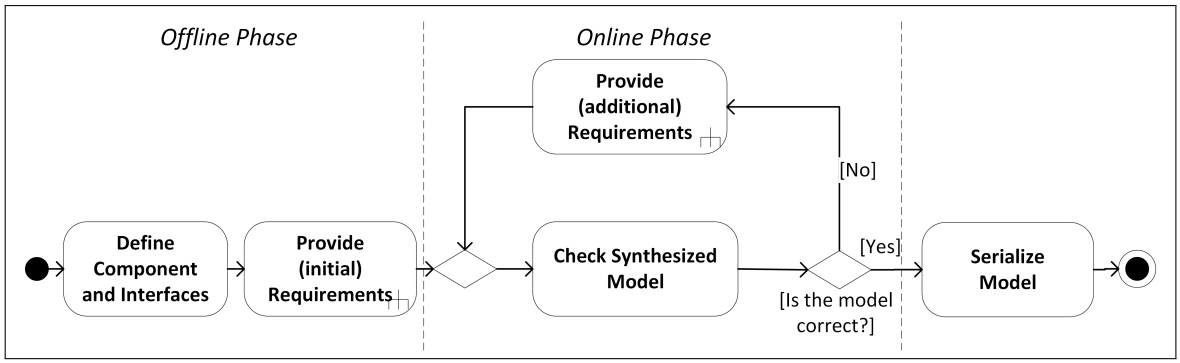


Figure 3.2: The proposed workflow of Interactive Learning from [8].

Components and Interfaces:

The workflow enables the modeling of composite systems by declaring the components during the first step and iteratively synthesizing a definition for each one. These components are handled as independent systems, for which the interfaces – the input and output alphabets – have to be defined in advance. This results – among others – in the arbitrary ordering of the online behavior-learning phases, and the behavioral faults being limited to their components of origin (although this does not limit the propagation of errors through messages resulting from incorrect behavior).

Providing Requirements:

Specifying requirements for the component under learning is present in both the offline and the online phases. In the offline phase, it is useful for more general requirements, with the scope of the whole component, conveniently formulated as logic expressions or long traces of the expected behavior. In the online phase, it is guided by the queries of the

algorithm regarding a yet unspecified behavior at a specific place in the trace currently examined. This can also be answered using various, more complex types of the supported requirements, but giving the corresponding output containing just the answer for the specific question is the most convenient option.

Several supported requirement types have been defined for the ILE, attempting to provide a generic structure open for extension. Among trace-based requirements, valid and invalid traces were specified along with just the corresponding output for a given trace, and also sequence diagrams. Logic-based requirements have also been included in the initial specification through LTL expressions with the simplest possible model and little success – which this thesis attempts to improve. Due to handling numerous requirements in several different formalisms even during a simple learning, a best-effort conflict resolution procedure was also introduced. The workflow of providing requirements can be seen in Figure 3.3.

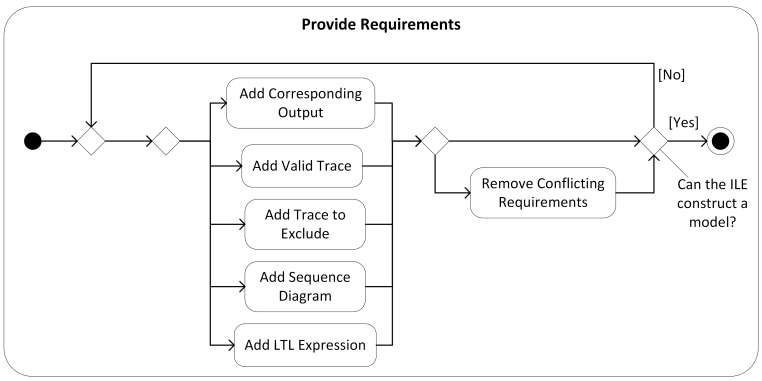


Figure 3.3: The process of adding a requirement.

Checking the Synthesized Model:

During the online phase, whenever the ILE assumes that it has gathered enough information to construct a model for the given component, the engineer if offered with a visualized representation of the current state of the model being synthesized – the equivalent of an equivalence query in automata learning algorithms. The user can either approve this model – in which case the learning, and therefore the design of the behaviour is complete for the component – or provide a counterexample where the model does not meet the – not yet specified – requirements.

The proposed equivalence model is a deterministic automaton, which, based on the information provided by the user, can be incomplete in multiple ways. The behavior of the desired model can differ from that of the learned system because of lacking information, in which case the user (acting as the equivalence oracle of the learning) needs to provide the separating behavior. Another reason for incompleteness can be newly discovered states, whose behavior is unknown based on their input signatures. This case prompts the user to evaluate the validity of state separation and to provide the lacking information. If, for some reason the hypothesized behavior is contradicting that of the desired system (by the users' oversight in providing requirements), the actual, currently conflicting requirement can be provided to guide the learning algorithm through the process of conflict resolution described above. Examples for equivalence models can be seen in Figure 3.4.

If the model is accepted, the design phase is complete for the current model. When all models are complete, the composite system model can be serialized in various formalisms, including that of the Gamma Statechart Composition Framework [25].

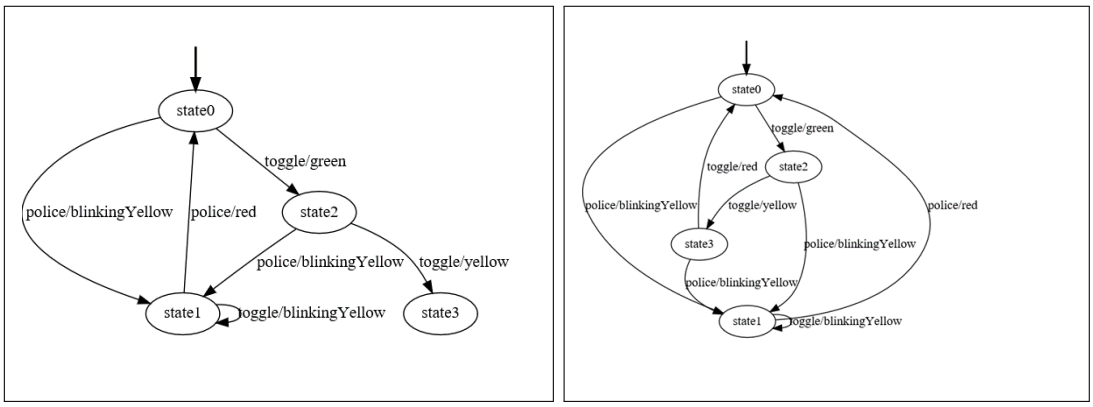


Figure 3.4: Equivalence query for an incomplete model (*left*) and the final model (*right*).

3.1.2 Overview of the Architecture

The architecture of the ILE consists of two main components: the *learning algorithm* - which is responsible for the model synthesis procedure, thus the course of the learning - and the *interactive oracle* - investigating the membership of the given input sequences in the languages of the models given by the user on one side and interacting with the user on the other side. A functional overview of this architecture is depicted in Figure 3.5. The following subsections elaborate on the details and connections of these components.

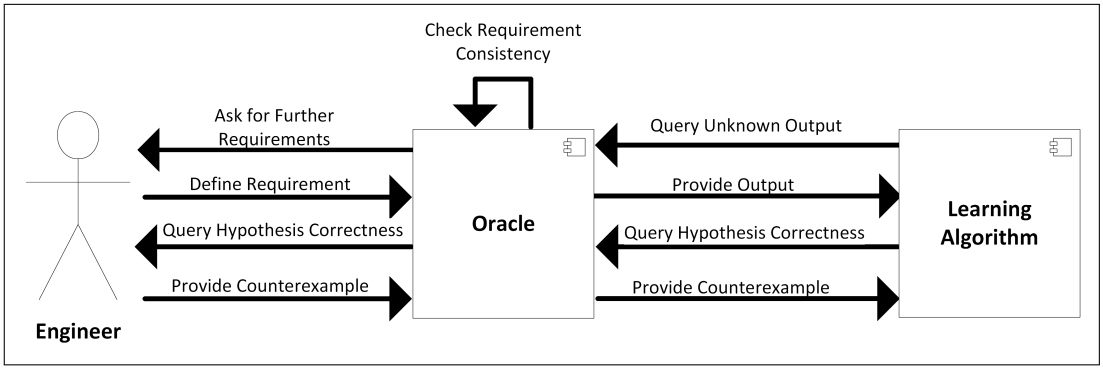


Figure 3.5: High-level architecture of the components of the ILE.

In the case of the learning algorithm, active automata learning algorithms were chosen as a design direction. Active automata learning enables complete separation of the learner algorithm and the system under learning through a teacher component – enabling the system under learning to be made from multiple, separate requirement-models provided by the user. Since active automata learning works through queries, the query can go through arbitrary layers of logic – allowing the proposed oracle-based interactive learning.

As discussed in Chapter 2, active automata learning algorithms work through a teacher and a learner component. While traditionally, the queries asked through the teacher are automated - by known or derived information and equivalence algorithms - in order to achieve an interactive algorithm, we created a new approach. Figure 3.5 shows the Learning Algorithm delegating its queries through the oracle, which delegates the questions to the user. The abstract approach presented in Figure 3.5 does enable interactive learning, but implementing it using traditional approaches (by delegating every single query to the user) proves to be infeasible in practical use cases because of the overwhelming amount of queries needed to learn a model. In order to overcome this boundary, we made optimiza-

tions to the ILE to automate a subset of queries, and we designed a new, *adaptive* active automata learning approach to heuristically control the design space.

3.1.3 The Oracle

The oracle is responsible for interacting with the user, managing the provided requirements and extracting information based on the queries posed by the learning algorithm. Its most important component is the interactive learnable.

The interactive learnable stores the requirements received from the user in the form of *partial models* and answers the queries of the learning algorithm. The architecture of of the interactive learnable can be seen in Figure 3.6.

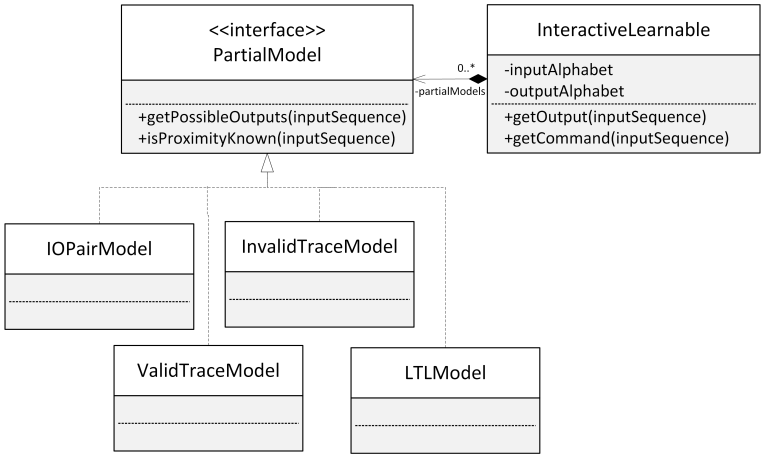


Figure 3.6: Architecture of the Interactive Learnable component.

Partial models have two responsibilities: providing the set of possible outputs to the given input sequences based on the information contained within, and providing information about the proximity of the input sequence. The intersection of these possible outputs provides the output based on the given requirements, and also reveals conflicting requirements without additional overhead. In the current approach, the inconclusive outputs are delegated to the user – highlighting the non-conflicting choices – creating the loop in Figure 3.3.

Telling whether partial models contain additional information enables the interactive learnable to track the easly explorable parts of the design space, thus facilitating significant optimization opportunities for the entire workflow through attaching an adaption command to the answer to each query. For instance, when a given valid trace is queried for an output somewhere in the middle of its contained sequence, the exploration of the rest of its contained behaviors can be automatically queried without requiring the input of the user.

Preserving the requirements in separate partial models has several advantages. First of all, it ensures the traceability between the user input and the learning algorithm, which is essential, as the user may not understand feedback or questions about information derived from their input. Also, translating each requirement to a common formalism and merging these models might be possible, but the addition and removal of models - which are frequently used operations in the workflow - would be severely ineffective. Additionally, in this manner the exploration of the individual models may be adjusted to their own internal logic.

Supporting model conflict handling – as proposed in Section 3.1.1 – introduces two potential challenges: models have to be able to be removed just as easily as they can be added, and inconsistencies need to be handled when removing a model. The first problem is solved by the partial model pattern, but the other one requires further consideration. Model inconsistency arises, when a model has been used to answer behavior-related queries, then it is removed. In this case, the already extracted information remains in the system, but its source disappears. Our solution to this problem, is to restart the automata learning – retaining the models already provided by the user, thus hiding this restart – by attaching a *reset* command to the answer to the queries of the adaptive learning algorithm.

3.1.4 The Learning Algorithm

There are a variety of approaches to active automata learning, differing in the number of queries and the logical order in which they are asked. To fully utilize the proposed interactive learning, the used automata learning algorithm is required to:

- allow the addition, removal and modification of requirements (and thus the learned behavior), and
- to be easily extensible and open for modification, such that the adaptive consideration of explorable and inferable behaviors - enabled through the proposed interactive learnable - can be utilized.

To support the above requirements – as an initial solution – we built upon and designed a variant of the Direct Hypothesis Construction algorithm, already introduced in Chapter 2. The DHC algorithm learns through rounds of hypothesis creation, in which every round starts from the ground-up. This approach has the benefit of allowing the system under learning to behaviorally change through the run of the algorithm, and - in the case of interactive learning - allows the designing engineer(s) to add, remove and modify requirements during the online phase of the workflow.

This results in adaptive learning algorithms, which receive both the desired output and the learning heuristic to utilize - keeping an identical, but extended automata learning architecture. The resulting algorithm can be seen in Algorithm 3.

As *line 7* shows, the membership query returns both the output and the adaption heuristic. If the adaption command is *reset*, the learning round begins again while keeping the current inputs. The enqueueing of successors is only possible, if the received heuristic is *optimistic*, allowing the fine-tuning of exactly which states to explore greedily.

Algorithm 3: Adaptive Direct Hypothesis Construction algorithm

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

```
1  initialize hypothesis H, create a state for all elements of S_p
2  initialize a queue Q with the states of H
3  while Q is not empty do
4      s = dequeue state from Q
5      u = access sequence from s_0 to s
6      for d ∈ D do
7          output, adaptionCommand = mq(ud)
8          if adaptionCommand is RESET then
9              go to line 1
10         end
11         set λ(s, d) = output
12     end
13     if exists an s' ∈ S, where the output signature of s' is the same as s then
14         reroute transitions of s to s' in H
15         remove s from H
16     else
17         if adaptionCommand is OPTIMISTIC then
18             create and enqueue successors of s for every input in Σ into Q, if not
               already in S_p
19         end
20     end
21 end
22 remove entries of D \ Σ from λ
23 return H
```

Naturally, the complete architecture of the framework introduced in this chapter contains several other – from our current perspective irrelevant – components. For the sake of completeness, a complete architectural overview can be seen in Figure 3.7.

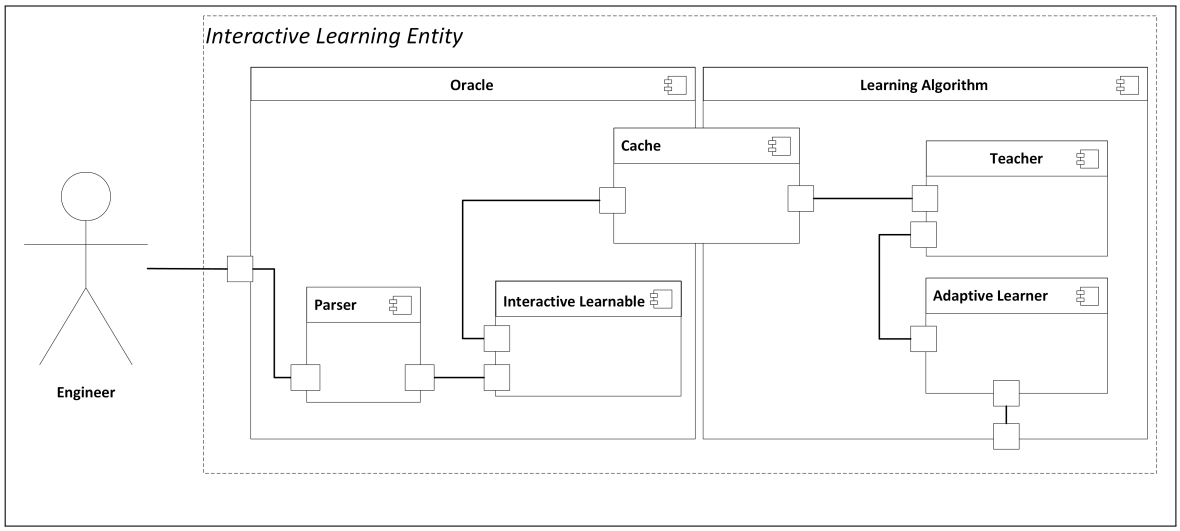


Figure 3.7: Architectural overview of the ILE.

3.2 Related Work

There are multiple automata learning frameworks in the literature, including *LearnLib*[18] that provides a Java framework for active and passive automata learning or *libalf*, which provides learning techniques for finite automata implemented in C++.

Tomte[2] is a tool built to support the automata learning of realistic software components with large state-spaces due to data variables. It attempts to simplify automata learning by applying counterexample-guided abstraction refinement on the alphabet of the system under learning, refining the symbols only when the abstract representation would introduce nondeterminsm. Processing the queries and answers between the learner and the system in such a way results in an abstract model – synthesized through automata learning – and an abstraction from Tomte. Together, these can be used to construct a realistic model for the system under learning.

A more direct approach to the same problem by Howar et al. in [16] is to apply abstraction refinement inside the learning algorithm: it becomes part of the learning process. This results in a more effective solution, as nondeterminism does not lead to failure and the learning can simply continue until a complete, deterministic system is achieved.

An interesting survey on the applicability of automata learning in formal methods can be found in [31] by Wang et al. The paper focuses on two topics: one is learning for formal verification – for which automata learning can be used – and the other is learning formal specifications. They provide an up-to-date overview on these subjects and possible future research topics. A somewhat similar work exists by Steffen et al. [29], in which the authors only focus on the current state of automata learning.

In [32], Zhu et al. focus on the synthesis problem for safety LTL. They present a highly effective technique for the said fragment of LTL, which significantly outperforms the currently available LTL synthesis tools.

Ferdowsifard et al. introduce an approach similar to interactive learning in [12], called *Small-Step Live Programming by Example*. They synthesize actual program code in an interactive way by combining live programming – a paradigm where the environment displays runtime values – and programming by example – a program synthesis technique generating program candidates based on examples. They only use this approach to synthesize single statements, as opposed to big-step approaches with the scope of the entire program. An implementation generating Python code was also created for this.

Chapter 4

Temporal Logic Specification in Interactive Model Synthesis

In this chapter, the challenges and the solutions are detailed, in connection with the application of temporal logic in interactive learning. It is mainly focused on LTL as a convenient formalism of defining properties while being relatively simple to manage by algorithmic means. After a brief overview of the context in Section 4.1, in Section 4.2 the challenges of the practical application are discussed, for which in Section 4.3 and Section 4.4 various solutions are proposed.

4.1 Overview of the Extended Methodology

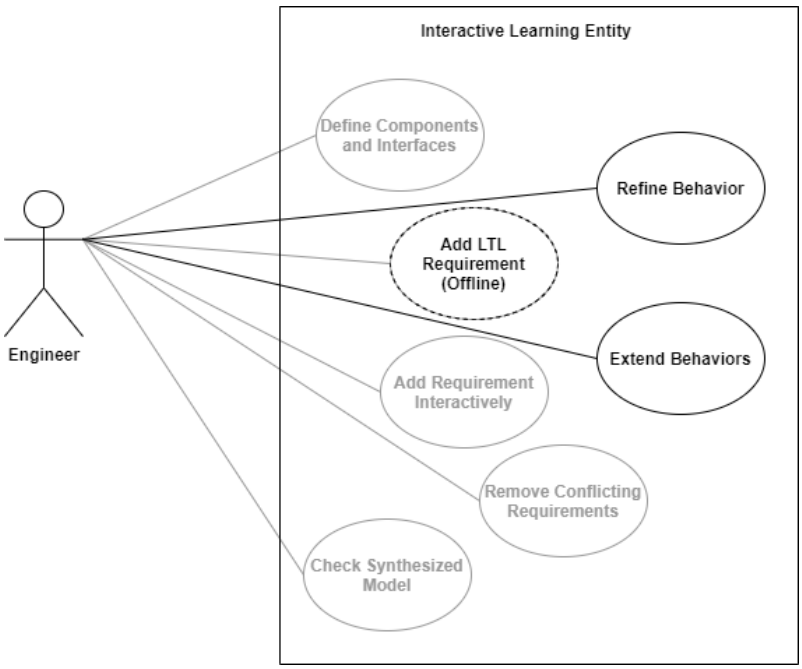


Figure 4.1: Interaction types between the engineer and the ILE. The highlighted elements are specific to the application of temporal logic (cf. Figure 3.1).

The extensive application of temporal logic in interactive learning brings about some modified and also some extra functionality from the perspective of the designing engineer.

These changes can be seen in Figure 4.1 and are elaborated on in the following sections. In short, LTL formulae among the offline requirements are handled in a special way, and the holistic model synthesis workflow of each component is extended to an iterative, abstraction refinement-based one. This modified workflow can be seen in Figure 4.2.

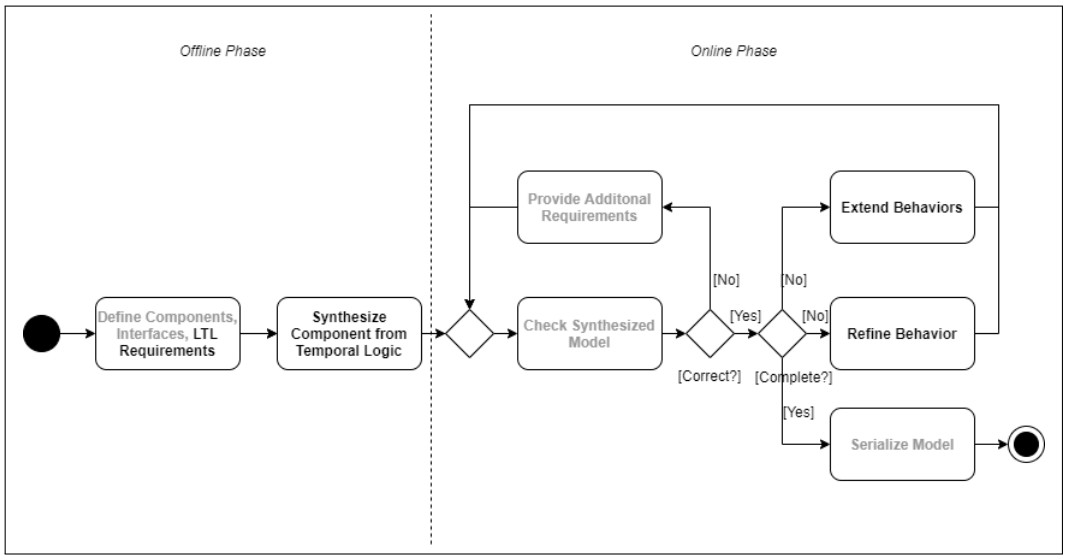


Figure 4.2: The extended workflow (cf. Figures 3.2).

Even though this workflow is a possible way of synthesizing component models form LTL expressions, it might not be obvious why is the *extended online phase* necessary – especially after comprehending the model synthesis techniques introduced in Chapter 2. The next section discusses this in detail.

4.2 Challenges of Model Synthesis from LT Properties

Chapter 2 has already introduced ways to synthesize executable automaton models directly from LTL expressions. However, these approaches are difficult to use in practice for several reasons.

In practice, temporal logic is used to formulate certain high-level properties for the main behavior the system being modeled. On one hand, this results in behaviors inevitably missing – either by abstraction of the whole model, or for the convenience of the modeler – which would be needed for a complete, executable specification. On the other hand, certain behaviors may be modeled only on an abstract level, that would have to be refined in later stages of the development.

The missing behaviors represent behavior that may be more convenient to define using different formalisms – such as the expected output symbol for a specific trace. Though, this raises the question how these radically different formalisms should be reconciled. Luckily, Chapter 3 already describes a possible solution, which can be extended appropriately. Also, at the time of formulating the high-level properties and also in the early stages of design, these behaviors may not even be ignored, but totally unknown. Thus, the atomic propositions of the initial properties might need to be extended with extra ones later.

Example 16. *In Subsection 2.4.2 we have already seen, that LTL cannot express all ω-regular languages. For instance, over the atomic propositions $\{j, k\}$, the ω-regular expression $((j+k)k)^\omega$ meaning "there is k in all the even places" cannot be expressed using LTL.*

In case the system under learning should exhibit such behavior, the automata learning algorithm must receive this information using a non-LTL formalism.

The modeled abstract behaviors must also be refined at some point. Refining the set of atomic propositions of the temporal properties may result in difficulties both theoretical and practical in nature. First and foremost, behavioral logic not expressible in a given formalism could not be modeled at all – see Example 16. Second, it may result in highly complex properties that are difficult to understand and maintain, even though it would be possible to synthesize the correct models. On the other hand, inappropriate refinements on a (semi-)finished model may result in an incorrect model with regard to the initial properties.

Example 17. *An example for the infeasibility of pure LTL refinement could be the Mealy machine in Figure 2.12, where the desired refinement is that of the proposition $\{b\}$ with $\{j, k\}$ such that the ω-regular expression from Example 16 holds. However, such an automaton clearly exists, as it can be seen in Figure 4.3.*

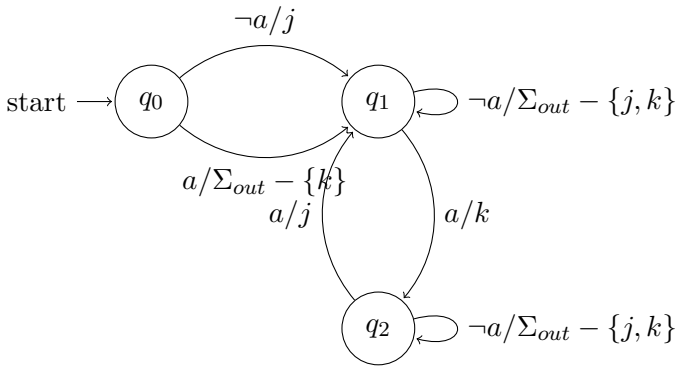


Figure 4.3: Refinement of the Mealy machine in Figure 2.13.

Lastly, a challenge specific to LTL is that by definition, it is used to define properties for paths in a system model. This means, that the *scope* of these properties can be deliberately defined by the application: it can be used to formulate properties universally true during the operation of the system as well as properties that should be possible for the system to exhibit under certain circumstances. This makes the formalism very flexible. However, in order to avoid confusion, this scope should always be clearly defined.

The rest of this chapter addresses each of these difficulties, ultimately proposing a refinement-based interactive workflow.

4.3 Applications of LTL in Interactive Learning

This section proposes and compares different solutions for the usage of LTL expressions in interactive automata learning. The following subsections describe the possible applications in different parts of the workflow: Subsection 4.3.1 introduces an approach that handles these requirements on the user input level (i.e. in the *interactive oracle*). Subsection 4.3.2 evaluates the approach where the property is moved closer to the teacher component of the algorithm (i.e. in an *automaton cache*). These solutions integrate well into the original interactive learning workflow – Figure 3.2 – not requiring any modifications. Finally, Subsection 4.3.3 describes a solution with LT properties handled *inside the learning algorithm*, which can be used in the workflow in Figure 4.2.

It is common in each of these solutions that the base model of the LTL expressions is the LTS interpretation of the system under learning. This means, that the set of atomic propositions that can be used in these expressions is the subset of the possible labels of the transitions, which are the elements of the input and the output alphabets of the component. The model synthesis takes place assuming event semantics – exactly one input and one output proposition happening at any given step during the execution of the system.

4.3.1 Constraining Queries Using Büchi Automata

Transforming LTL expressions to Büchi automata is a well-researched area due to its applicability in model checking: there exists a very simple algorithm for checking the reachability of a property given by a Büchi automaton on finite transition systems. This transformation and model checking algorithm are a possible starting point for using LTL expressions in automata learning.

The main idea of this solution is to transform the LTL formulae into a – non-deterministic – Büchi automaton, then, on each membership query posed to the engineer, check the possible answers on this automaton first. This can be done by executing the automaton for the input sequence in question, then checking all the possible runs from the resulting state(s). In case the run can be made accepting from a given state, the output for the last element of the input sequence is a possible answer to the query. This constrains the range of the query, and in the cases where exactly one output is possible in such a manner, the query can be automatically answered without asking the engineer.

The main benefit of this approach is its simplicity. Assuming an efficient algorithm for the LTL-to-Büchi transformation, it only requires basic graph traversal algorithms for checking the reachability of accepting states in Büchi automata. Also, additional meta-information can be easily attached to these Büchi automata *partial* models to control the scope for which these properties should hold. On the other hand, this method can only be used for safety properties: it only filters out the *bad prefixes* and cannot ensure that liveness (or certain fairness) properties are ever met. A possible implementation of this concept can be included in LTL Models in Figure 3.6.

Example 18. *The Büchi automaton in Figure 2.9 – obtained from an LTL expression specifying a liveness property – only has states from which any run can be made accepting. This means, that querying the automaton does not constrain the set of possible answers for any input sequences, thus demonstrates the shortcomings of this approach for general LTL properties.*

4.3.2 Learning from Mealy Automata

Mealy automaton-based controller synthesis from LTL expressions – introduced in Section 2.5 – has also been researched in recent years, and it could also be a natural starting point for interactive learning. As the original concept of automata learning queries existing implementations, it is fairly simple to carry out fully automated learning phases at certain points of the interactive learning workflow. The illustration of this concept can be seen in Figure 4.4. This fits perfectly into the architecture of interactive learning, but the cache now plays an essential role, even in the high-level architecture.

The benefits of this solution are similar to those of the previous one. In particular – assuming an efficient algorithm for the Mealy automaton synthesis – the required algorithm

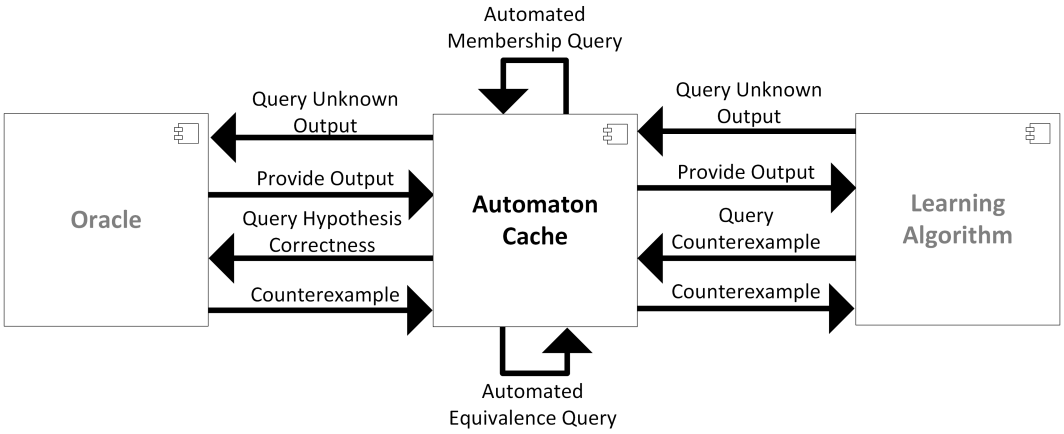


Figure 4.4: Automated Learning in Interactive Learning. The cache is a vital component between the learning algorithm and the interactive oracle.

is very simple yet again: a (possibly non-deterministic) Mealy automaton must be queried before delegating to the user. In addition, this approach also solves the general LTL case: it is not only useful for safety properties, but also for liveness properties, as the behavior of the correct-by-design automaton of the LTL property is fully learned. Also, this solution guarantees that the property holds for every run of the automaton.

The automated learning phase may result in several automated membership and equivalence queries in the workflow. These membership queries would be present even without the intermediate automaton, however, the automated equivalence queries bring extra complexity into the system in the form of equivalence algorithms. Also, as a common solution to them is to check the equivalence to traces of a predefined length, it is possible – although highly unlikely – that the learning algorithm does not find a distinguishing behavior that is included in the synthesized Mealy automaton.

4.3.3 Generalized Learning Algorithms

The previous solutions attempted to tackle the inclusion of LT properties without exploiting the full capabilities of the learning algorithm: the learning took place by traditional membership and equivalence queries. This resulted in several disadvantages.

Both algorithms described in Subsection 2.3.1 started the learning from the equivalent of a one-state Mealy automaton, then incrementally extended it with additional information by systematically querying the possible traces. However, it is possible to *generalize* these algorithms to start the learning from arbitrary – possibly non-empty and incomplete – Mealy automata. Then, these generalized algorithms can be used to extend an automaton synthesized from a temporal property step-by-step using the interactive learning methodology. In the rest of this section, these generalized algorithms are defined.

For these algorithms, the starting automaton is constrained to minimal automata with non-determinism removed: this is the part where the behavior needs – thus possibly incomplete – and the input and output alphabets as subsets of those of the model being synthesized.

Generalized DHC Algorithm:

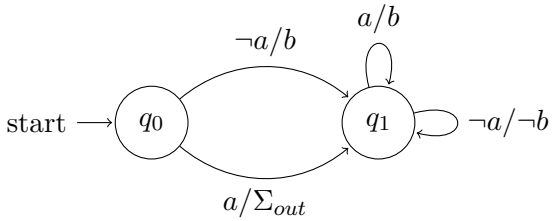
The algorithm only differs from the original one in its hypothesis construction. The generalized version can be seen in Algorithm 4 (cf. Algorithm 1).

Algorithm 4: Hypothesis construction of the Generalized DHC Algorithm

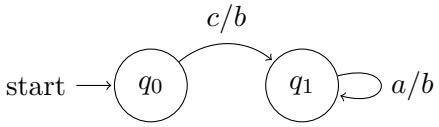
Input: D: a set of suffixes,Σ: an input alphabet,
$M_0 = (S_{M0}, s_{M0}, \Sigma, \Omega, \delta_{M0}, \lambda_{M0})$: an initial Mealy machine
Output: A Mealy machine $H = (S, s_0, \Sigma, \Omega, \delta, \lambda)$

```
1  initialize S_p, a set of access sequences for each state of M_0
2  initialize hypothesis H from M_0: copy its states and transitions
3  initialize a queue Q with the incomplete states of H
4  while Q is not empty do
5      s = dequeue state from Q
6      u = access sequence from s_0 to s
7      for d ∈ D do
8          if ud in H then
9              o = mq_hypothesis(ud)
10         else
11             o = mq_teacher(ud)
12         end
13         set λ(s, d) = o
14     end
15     if s existed in M_0 then
16         for each ud not in H do
17             create and enqueue successor if not already in S_p
18         end
19     else
20         if exists an s' ∈ S, where the output signature of s' is the same as s then
21             reroute transitions of s to s' in H
22             remove s from H
23         else
24             create and enqueue successors of s for every input in Σ into Q, if not
                already in S_p
25         end
26     end
27 end
28 remove entries of D \ Σ from λ
29 return H
```

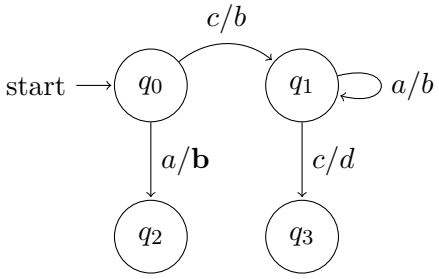
This algorithm has two points worth mentioning. First, obtaining the access sequences is a part of this algorithm. It can be obtained by building a spanning tree using any of the well-known algorithms – such as the breadth-first search (BFS) – in polynomial time. Second, this algorithm differentiates between states already existing in the initial Mealy machine and new state candidates. In case of already existing states, the process is simplified, as those states cannot and must not be merged into other states. Even if D does not contain elements that could separate their behavior – thanks to the automaton-based hypothesis – we can keep the states separate, which greatly simplifies the algorithm. Otherwise, it proceeds as normal. Possible first steps of the hypothesis construction can be seen in Figure 4.5.



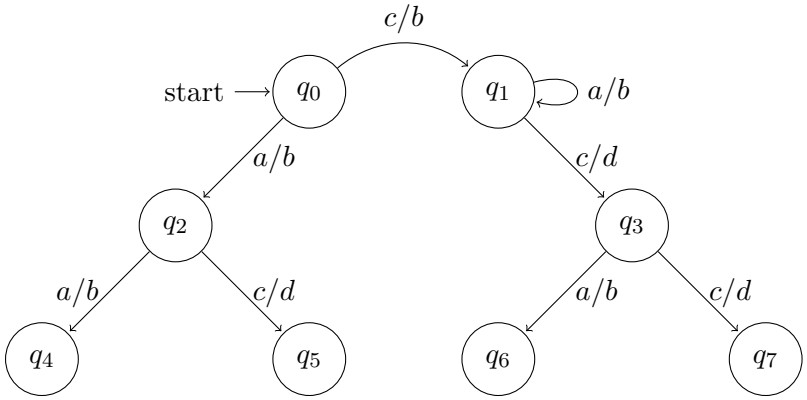
(a) Mealy automaton representing all possible automata based on an LTL formula



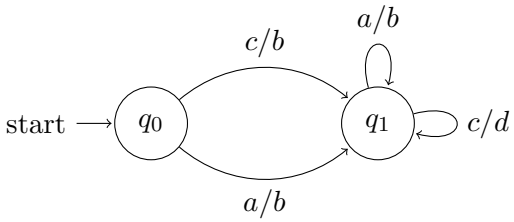
(b) Input automaton and initial hypothesis with the non-determinism eliminated



(c) q_0 and q_1 complete



(d) q_2 and q_3 complete



(e) Complete automaton after merging the appropriate states. It is one from the possible automata represented by Figure 4.5a

.

Figure 4.5: First steps of the generalized DHC hypothesis construction given the LTL formula $GF(a) \leftrightarrow GF(b)$. The input propositions are $\{a, c\}$, the output propositions are $\{b, d\}$.

Among the solutions proposed in this chapter, these algorithms have the best performance, as they do not pose any redundant interactive or automated queries – apart from the

inherent redundancy of the original algorithms, especially in case of DHC. Also, these algorithms keep the original behavior correct-by-design for all paths.

As a byproduct of this generalization, both algorithms can be used to handle any input formalism transformable to Mealy automata as a starting point – e.g. regular expressions, ω-regular expressions and also sketches of automata – and complete them with the missing information.

Although these algorithms successfully eliminate the non-determinism from automata synthesized from LTL, they do not offer solutions for most of the challenges discussed in Section 4.2. This is going to be addressed in the next section by an approach from the perspective of model refinement.

4.4 Model Refinement in Automata Learning

This section elaborates on the applicability of model refinement in automata learning – which is necessary for reasons discussed in previous sections – and how it is related to the scopes of requirements.

In Interactive Learning, the user interacts with the ILE as a black-box system. Thus, it is only possible to refine its interface – the input and output alphabets of the SUL – in order to refine the learned system. For this reason, Subsections 4.4.1 and 4.4.2 discuss different approaches to alphabet refinement. In both cases, the proper course of the refinement inside the algorithm is also discussed, from a white-box point of view.

4.4.1 Extending the Alphabets

By extending the alphabets of the hypothesis with completely new symbols, we refine the model with completely new behavior. This is useful when modeling behavior left out in earlier stages of the workflow.

Formally, this can be viewed as the partition refinement of Σ, namely the refinement of the empty set $\emptyset \subseteq \Sigma$ with other elements, as it can be seen in Figure 4.6. Following the definition of Mealy automata in Chapter 2, the transition and output functions are not defined for (elements of) $\emptyset$. Thus, there are no abstract properties defined for these elements and no assumptions can be made about the behavior or the initial LT properties. The exploration of the new behavior can be entirely left for the automata learning algorithm.

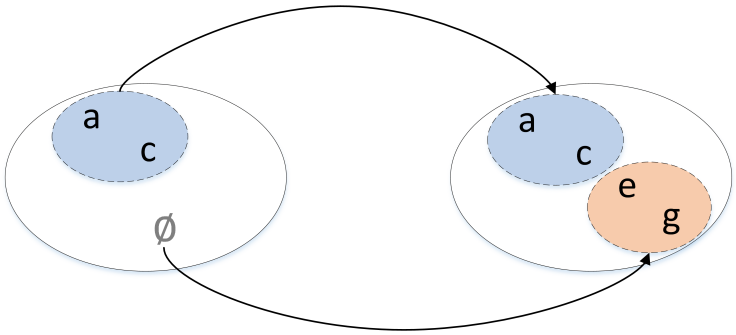


Figure 4.6: Extending the alphabet by refining $\emptyset \subseteq \Sigma$ with $\{e, g\}$.

Utilizing generalized learning algorithms such as Algorithm 4 for this type of refinement is quite simple: the initial, input Mealy machine $M(S_{M0}, s_{M0}, \Sigma_{M0}, \Omega_{M0}, \delta_{M0}, \lambda_{M0})$ should

be defined with separate input and output alphabets: Σ_{M0} and Ω_{M0}, such that $\Sigma_{M0} \subseteq \Sigma$ and $\Omega_{M0} \subseteq \Omega$, where Σ and Ω are the alphabets of the hypothesis automaton $H = (S, s_0, \Sigma, \Omega, \delta, \lambda)$. The algorithms should be initialized with complete, deterministic Mealy automata wrt. Σ_{M0} and Ω_{M0}. Naturally, this learning extends the model with new states and transitions, as in automata learning, they are the manifestation of new, thus separating behavior.

The newly completed model can be considered a refinement of the original one similarly to the alphabets: by neglecting the newly added – refined – behavior, the properties of the abstract automaton are also present in the refined one, as the two automata are isomorphic.

Example 19. *Figure 4.7 shows the refinement of an automaton with one additional input and output symbol. Notice, how the properties of the initial automaton in Figure 4.5e are reachable in the extended one, and are universally valid when removing the new elements.*

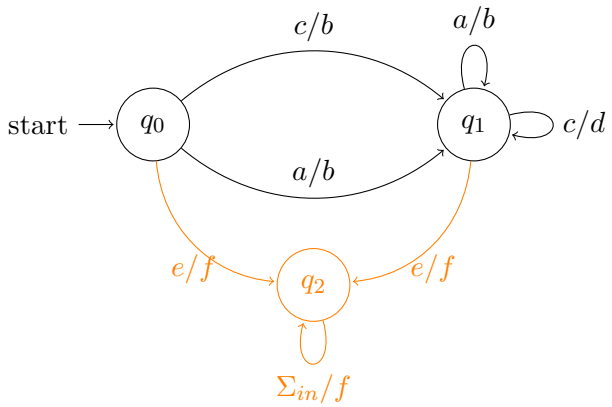


Figure 4.7: Extended automaton from the initial one in Figure 4.5e. The extended alphabets are $\Sigma_{in} = \{a, c, e\}$ and $\Sigma_{out} = \{b, d, f\}$. The LT properties present in the original one are not present in the extended one with regard to the orange elements.

The number and order of this type of refinements – even their arrangement in consecutive learning batches – do not influence the learned automaton, due to the resulting automaton being the canonical one.

4.4.2 Refinement of the Symbols

When the individual, non-empty symbols of the alphabets are refined – by the partition refinement of Σ, refining its elements as illustrated in Figure 4.8 – the possible assumptions can and should be taken into consideration. These differ for input and output symbols, as they manifest in different elements in a minimal automaton.

Refinement of Input Symbols:

Based on its definition in Chapter 2, a minimal Mealy automaton only has states that are behaviorally distinguishable. Under the assumption that the starting automaton – from which the input symbol refinement is done – is complete and minimal, the refinement of input symbols cannot introduce yet unknown behavior to the automaton, thus, new states cannot be created. Only the transitions containing the abstract input symbol can

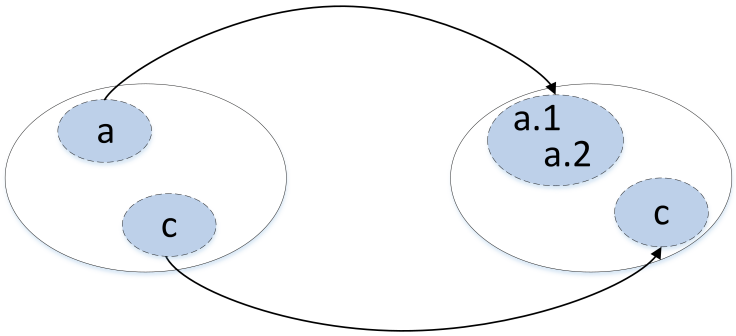


Figure 4.8: Refining the symbol $\{a\}$ with $\{a.1, a.2\}$

be refined for each refinement of its input symbol, with every other of its properties – source, target, output – having to stay the same for behavioral equivalence.

For this reason, refinement of input symbols results in trivial extensions of the model: each edge containing the abstract symbol should be refined for each refinement of the given symbol, the target states of which are one-to-one refinements of the abstract target state and the source state and the output remaining the same. This way, the existing states are reachable by (multiple) new input sequences, but their behavior changes symmetrically compared to each other, thus retaining an isomorphic structure with certain transitions multiplied.

After this kind of refinement, the model can be considered the refinement of the orignal one, as ignoring the differences between the refinements of the original symbol results in an automaton isomorphic to the original one. An example can be seen in Figure 4.9.

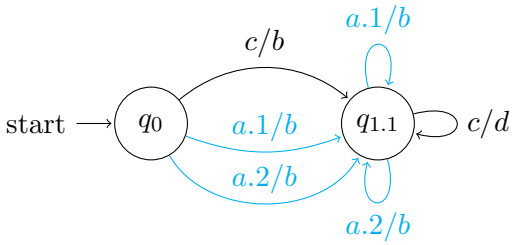


Figure 4.9: Input-refined automaton from the abstract one in Figure 4.5e. The refined alphabet is $\Sigma_{in} = \{\{a.1, a.2\}, c\}$ with the original output alphabet $\Sigma_{out} = \{b, d\}$. The transitions created due to the refinement are marked with cyan color.

This refinement preserves the LT properties for every behavior of the refined automaton: all the properties of the original automaton are present in each element of the refined one, regarding the refined symbols as the abstract ones.

Refinement of Output Symbols:

In a minimal Mealy automaton, all the states must be behaviorally distinguishable. In case of refinement of the output symbols, these signatures are directly affected by the refinement of the output alphabet. Thus, the affected behavior after the refinement must be specified, possibly using automata learning.

However, for the model to be considered a refinement of the previous one, this must follow certain rules regarding the accepted behavior. The transitions containing the abstract output must be refined one-to-one to one possible refinement of this output, along with

their target states, which may have multiple refinements. When refining the states reachable by the same access sequences in the original model, the outgoing edges of the refined state must correspond to those of the abstract one. Naturally, these edges may contain the abstract output in question, thus they may be refined too, along with their targets. As loops of the abstract states may result in some inner logic among its refinements, this refinement is a one-to-many relationship.

After this kind of refinement, the model can be considered the refinement of the original one, as ignoring the differences between the refinements of the original symbol results in an automaton isomorphic to the original one. An example can be seen in Figure 4.10.

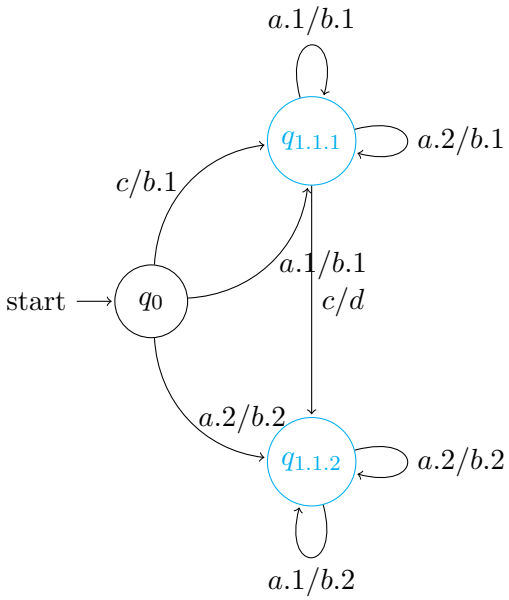


Figure 4.10: Output-refined automaton from the abstract one in Figure 4.9 with the original $\Sigma_{in} = \{\{a.1, a.2\}, c\}$ and the refined $\Sigma_{out} = \{\{b.1, b.2\}, d\}$. The states created due to the refinement are marked with cyan color.

This refinement preserves the LT properties for every behavior of the refined automaton: all the properties of the original automaton are present in each element of the refined one, regarding the refined symbols as the abstract ones.

The learning algorithm for this type of refinement can be created similarly to the Generalized DHC algorithm: these also differ from the original ones only in their hypothesis constructions. The hypothesis construction of the Refining DHC algorithm can be seen in Algorithm 5.

For learning behavior that is the refinement of an abstract one, the subroutine $cmq_{target}(sequence, from)$ was introduced. This refers to a constrained membership query, meaning that the range of the answer must be in the given subset of the output alphabet.

The expression *refinement of a state* refers to a copy of the state, for which all outgoing transitions have been copied from its abstract parent. This may result in states for which some outgoing transitions must also be refined.

Algorithm 5: Hypothesis construction of the Refining DHC Algorithm

Input: D: a set of suffixes, an input alphabet Σ,
$M_0 = (S_{M0}, s_{M0}, \Sigma, \Omega_{M0}, \delta_{M0}, \lambda_{M0})$: an initial Mealy machine, a: the output symbol to refine

Output: A Mealy machine $H = (S, s_0, \Sigma, \Omega, \delta, \lambda)$

```
1  initialize S_p, a set of access sequences for each state of M_0
2  initialize hypothesis H from M_0: copy its states and transitions
3  initialize a queue R with the states of H, such that they have at least an outgoing
     edge with a, but not its refinements
4  while R is not empty do
5      c = dequeue state from R
6      t = the not refined outgoing transitions (with output a) of c
7      remove each element of t from H
8      initialize a queue Q with the refinement states of the original targets of t
9      while Q is not empty do
10         s = dequeue state from Q
11         u = access sequence from s_0 to s
12         for d ∈ D do
13             if ud in M_0 then
14                 o = mq_hypothesis(ud)
15             else
16                 o = cmq_teacher(sequence: ud, from: refinements of a)
17             end
18             set λ(s, d) = o
19         end
20         if exists an s' ∈ S, where the output signature of s' is the same as s then
21             reroute transitions of s to s' in H
22             remove s from H
23         else
24             create and enqueue the refinements of successors of s for every input in
                 Σ into Q, if not already in S_p
25         end
26     end
27     update R
28 end
29 remove entries of D \ Σ from λ
30 return H
```

Refinements in the Interactive Workflow As opposed to extending the alphabets, symbol refinements depend greatly on the order of the refinements and the number of them completed in one step.

For the different levels of abstraction to be clearly distinguishable, a design decision is to limit the number of refinements to only one at a time. This means, that after refining one abstract element – either by alphabet extension or symbol refinement – the model must be completed before starting another refinement. This is also true in connection with

alphabet refinement: even though multiple elements may be added to the alphabets in one step, until the automaton was completed, no input or output symbols can be refined.

As a result, the order of refinements should be considered in advance: refining an input symbol then an output symbol may result in automata exhibiting different behaviors, as illustrated in Figure 4.11 (cf. Figure 4.10). These differences are not due to different answers presented to the queries of the algorithm: it even queries completely different traces in the two cases.

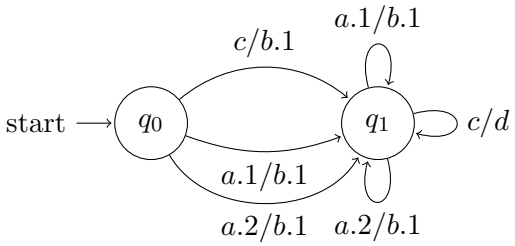


Figure 4.11: Applying the same refinement steps as in Figure 4.9 and Figure 4.10 in a different order may result in this automaton.

Both the generalized and the refining algorithms fit well into the interactive learning workflow. As they only differ in their hypothesis construction, they can be easily extended for adaptive learning and integrated into a single learning algorithm, calling each extended hypothesis construction algorithm as its hypothesis construction subroutine, whenever the corresponding refinement is requested.

Chapter 5

Implementation

In order to validate the approach, an appropriate implementation was created. Of the subsequent sections, Section 5.1 discusses the utilized tools, and Section 5.2 elaborates on the steps taken to provide an implementation of the original *Interactive Learning Entity* along with its extensions for the refinement-based workflow.

5.1 Tooling

5.1.1 Eclipse Modeling Framework

The Eclipse Modeling Framework is an Eclipse-based modeling framework and code generation facility. It defines its own structured data model – called Ecore – for describing models and providing runtime support for the models. Models are defined using the XML Metadata Interchange (XMI) format, which is supported by various Eclipse plugins developed specifically for this purpose, as EMF is fully integrated into the Eclipse platform. It provides an environment to numerous technologies, including server solutions, persistence frameworks, UI and transformation frameworks.

5.1.2 Xtext Framework

Xtext is an open-source framework for developing (mostly) domain-specific languages (DSLs). It has its own syntax for the definition of textual languages, resembling a context-free grammar extended with mappings to the in-memory representations. Unlike standard parser generators, it generates not only a parser, but also the abstract syntax tree (AST) of the grammar, and also supports several other features, such as validation rules and editing support. This is because Xtext is based on the EMF project – the metamodels of the defined languages are Ecore models –, and it is integrated into the Eclipse environment.

5.1.3 Owl

Owl [20] is a tool collection for ω-words, ω-automata and linear-time temporal logic. It provides several algorithms for automata and LTL, supporting - among others - LTL expression parsing and simplification, reading and writing ω-automata using the HOA format [5], translation of LTL formula to ω-automata with several possible acceptance conditions, and operations over ω-automata, such as product, SCC decomposition emptiness checks and acceptance-condition transformations.

Through providing these algorithms, the library supports easy development and fast prototyping in the area of LTL and automata, thus also enabling rapid concept validation.

5.1.4 Strix

Strix [22] is a tool for reactive LTL synthesis combining a direct translation of LTL formulas into deterministic parity automata (DPA) and an efficient, multi-threaded explicit state solver for parity games. In brief, Strix decomposes the given formula into simpler formulas, translates these on-the-fly into DPAs based on the queries of the parity game solver, composes the DPAs into a parity game, and at the same time already solves the intermediate games using strategy iteration, and finally translates the wining strategy, if it exists, into a Mealy machine or another appropriate formalism. It relies on Owl for the transformation of LTL to deterministic parity automata.

5.1.5 LearnLib

LearnLib[18] provides a Java framework for active and passive automata learning, with the versatile AutomataLib framework acting as a backbone of it. Learnlib provides implementations of several automata learning algorithms, as well as multiple equivalence and counterexample decomposition algorithms.

5.1.6 Automata Learning Framework

In order to give a foundation to the implementation described in this thesis, a previously created automata learning framework in [7] was extended upon. Since the framework was implemented using the Java programming language, the high-level view seen in Figure 5.1 is represented as a UML class diagram of the packages and the relations between them, essentially being an overview of the modularization of the framework.

The *Learnable* package contains the input formalisms, and the *Hypothesis* package contains the output formalisms. Both are used by the *Teacher* (package) and the learning *Algorithm* (package). The *Adapter* package is used as an abstraction layer to separate the algorithm and the teacher from the input formalism. Since automata learning algorithms have no direct access to the system under learning and generally operate in a black-box manner, the adapter package provides flexibility on what inputs can be used. As Figure 5.1 illustrates, no such adapter is used on the output layer, since Hypotheses are directly accessed by the learning algorithms, and are constructed during the learning. 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.
The advantage of such architecture is that the automata learning algorithms implemented within can be agnostic to the formalism of the input provided. This results in high reusability of the core algorithms, while being easily extensible and adaptable to arbitrary systems to infer.
The Framework includes multiple supported in- and output formalisms, and has multiple implemented active automata learning algorithms – one of which is the Direct Hypothesis Construction algorithm.

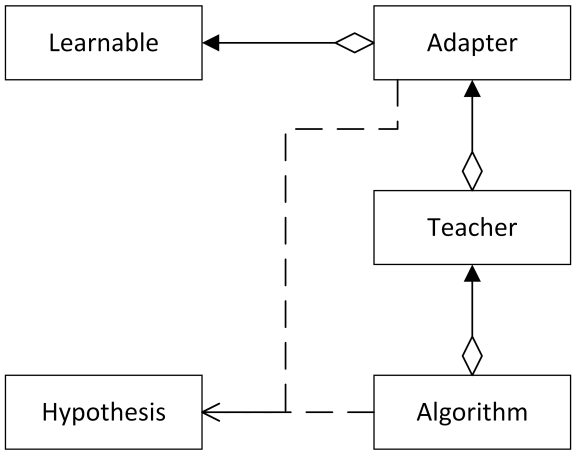


Figure 5.1: Structure and relations of the packages comprising the Automata Learning Framework. [7]

5.2 Extensions to the Framework

5.2.1 Interactive Learning

To create an interactive learning framework, some extensions were designed and implemented to the previously presented Automata Learning Framework. This utilizes automata learning algorithms, and is capable of handling a multitude of user-provided requirements as an input formalism. The designed architecture can be seen in Figure 5.2. It is important to note the extension of components shown in Figure 5.1, while still upholding the behavioral structure of the framework. As illustrated, an Oracle, a Learning Algorithm, and a Cache component outline the architecture of the ILE.

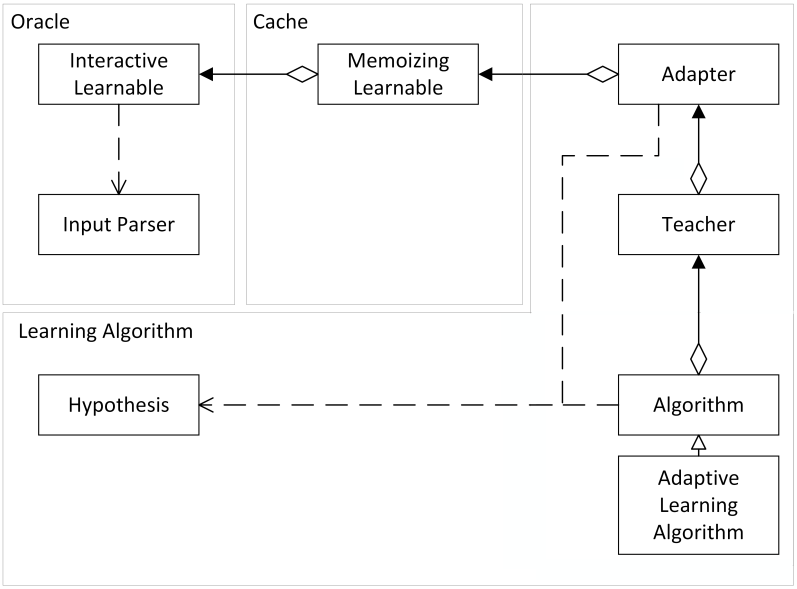


Figure 5.2: The extended Automata Learning Framework to fit the architecture of the ILE. [8]

5.2.2 Refinement-based Learning

In order to support the iterative, abstraction refinement-based component design workflow with automata learning, the architecture had to be adapted in two places:

- Propagation of *constrained queries*, *alphabet extension* and *alphabet refinement* through the teacher
- Various new functionalities in the hypothesis:
  - resetting the hypothesis to an arbitrary state
  - execution and traversal of the represented automaton
  - alphabet extension and incompleteness checking
  - alphabet refinement and refinement checking

The first one can be solved by simply extending the interface of the teacher and the corresponding pipeline. On the other hand, the second one requires further consideration.

5.2.3 Components of the Learning Algorithm

The hypothesis handed to the teacher – thus the user – only requires some basic querying capabilities and a graph structure to be visualized. However, the refinement-based algorithms need all the functionality described in the previous subsection, which is highly dependent on the different approaches of the various automata learning algorithms, resulting in different internal representations. An example for this is the automaton-based representation of the DHC algorithm, where the functionalities above can be supported by graph traversal algorithms, but they are not applicable in case of the tabular representation of L*. For this reason, the hypothesis and algorithm structure in Figure 5.3 was introduced.

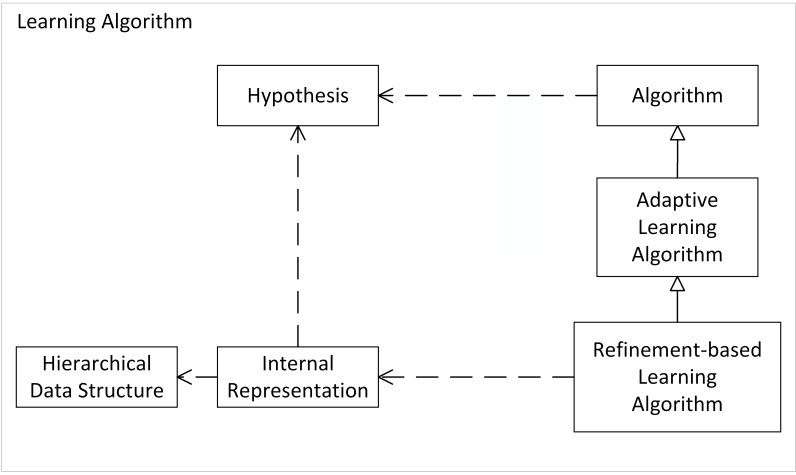


Figure 5.3: The extended Learning Algorithm, designed for refinement-based learning on complex, hierarchical data structures.

The biggest improvement in this architecture is the separation of the internal representation and the hypothesis. This is due to the fact that the components taking part in the EQ only need a simple view fit for a graph-based visualization, while the algorithm needs several additional pieces of information in order to operate correctly. Also, this

enables compatibility with components of the framework not explicitly supporting the refinement-based workflow.

Chapter 6

Case Study: Alternating Bit Protocol

This chapter demonstrates the capabilities of the framework. It presents the modeling workflow for one system component participating in a simple, yet commonly used protocol. After a short introduction on the expected functionality, an LTL specification is presented, followed by the step-by-step extension of the thus created model and a brief evaluation.

6.1 Introduction

The Alternating Bit Protocol [9] is a protocol for the correct transmission of a data stream through a so-called *faulty channel*. It consists of two components: a sender and a receiver. The sender reads data from an infinite input stream and sends it to the receiver in data packets equipped with an identifier. The receiver reads these packets from the channel and writes them to an infinite output stream. After the sender sends a packet, it waits for an appropriate acknowledgement – based on the packet identifier – from the receiver before sending the next packet. After the receiver receives a packet, it sends an appropriate acknowledgement to the sender. As there is at most one unacknowledged packet at any given time, one bit suffices for the identifier and the acknowledgement. This configuration can be seen in Figure 6.1.

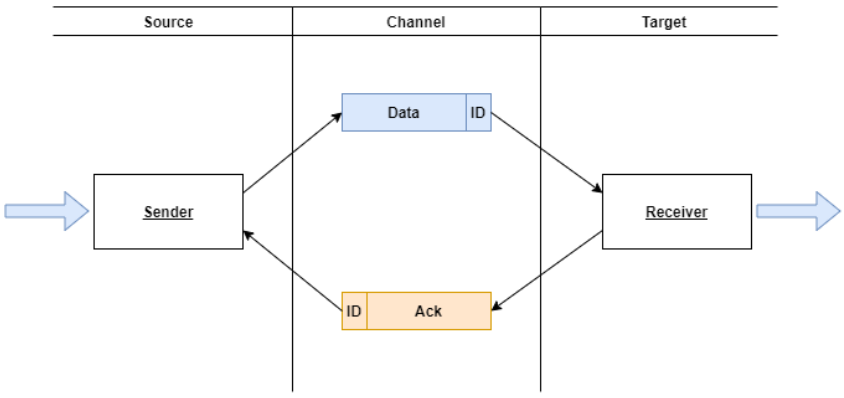


Figure 6.1: High-level overview of the alternating bit protocol.

In the following sections, we are going to synthesize a model for the sender component, using the refinement-based DHC algorithm. Adaptive learning is not used for the sake of comparison: the goal is to measure the required information, not the skills of the designing engineer.

6.2 Initial LTL Specification

In order to formulate any LTL expressions to be satisfied by the model, first, we have to identify initial input and output alphabets – in LTL terminology sets of input and output propositions, respectively – for our system component.

The initial input alphabet should consist of three different messages, corresponding to the identifier of the acknowledgement for the last message sent: {*ack0, ack1, null*}, where ack0 and ack1 represent acknowledgements with the identifier 0 or 1 respectively, and null represents no acknowledgement messages.

The initial output alphabet should consist of two different messages, corresponding to each possible data packet identifier to send: {*send0, send1*}.

According to the description of the protocol above, it is universally true for this component that after an acknowledgement with the identifier 0, it sends the next packet with the identifier 1 and re-sends it until an acknowledgement with the identifier 1 arrives. Also, in case an acknowledgement arrives with the identifier 1, it sends the next packet with the identifier 0 and re-sends it until an acknowledgement with the identifier 0 arrives. These properties can be easily translated to the (conjunction of) the following LTL formulas:

- $G(ack1 \rightarrow (send0\ W\ ack0))$
- $G(ack0 \rightarrow (send1\ W\ ack1))$

Depending on the applied LTL synthesis tool, an additional formula may be necessary, representing the event semantics of the automaton being modeled, meaning that the component can only send one message at a time:

- $G(\neg send0 \vee \neg send1)$

From these LTL formulae, the Mealy machine in Figure 6.2 can be synthesized as a starting point for refinement-based automata learning.

It is easy to see, that the Mealy machine in Figure 6.2 is a complete and minimal one. As there was no automata learning involved so far, this means, that we have completed our model for the moment. At this point, we can either extend this level of abstraction further with additional inputs and outputs, applying automata learning for continuation, or move to a lower level of abstraction, applying automata learning for refinement.

6.3 Extending the Model with Interrupts

Now, we are going to add interrupt handling to our model. The concept is quite simple: whenever the component receives an interrupt message, it answers with a handling message and continues to do so for all messages until a next interrupt message arrives.

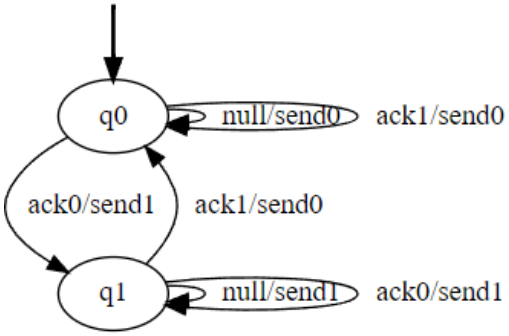


Figure 6.2: Mealy machine of the sender component of the Alternating Bit Protocol, synthesized from LTL formulae.

For this, we add an additional input symbol to the input alphabet of the automaton: *it*, representing all interrupt-related incoming messages. We also add an additional output symbol to the output alphabet of the automaton: *h*, representing all interrupt handling-related outgoing messages.

At this point, the automata learning algorithm finds both of the currently existing states to be incomplete: neither one has any outgoing edges with the input *it*, so it continues the learning process regarding both of these states as existing states to complete. A possible run can be seen below:

- ○ Query for sequence [it] in {send0, send1, h}:
- ▷ Output: h
- ○ Query for sequence [ack0, it] in {send0, send1, h}:
- ▷ Output: h
- ○ Query for sequence [it, null] in {send0, send1, h}:
- ▷ Output: h
- ○ Query for sequence [it, ack0] in {send0, send1, h}:
- ▷ Output: h
- ○ Query for sequence [it, ack1] in {send0, send1, h}:
- ▷ Output: h
- ○ Query for sequence [it, it] in {send0, send1, h}:
- ▷ Output: h
- ○ Query for sequence [ack0, it, null] in {send0, send1, h}:
- ▷ Output: h

... rest of the input alphabet after [ack0, it] ...

Note, that the information already contained in the automaton (Figure 6.2) was not queried during this phase of the learning at all. Also, the queries were not constrained – equivalently, they were constrained to the whole output alphabet – as there was no refinement involved.

This interrupt logic was really simple, so the learning completes after a few membership queries – in this case exactly 10. The resulting automaton can be seen in Figure 6.3.

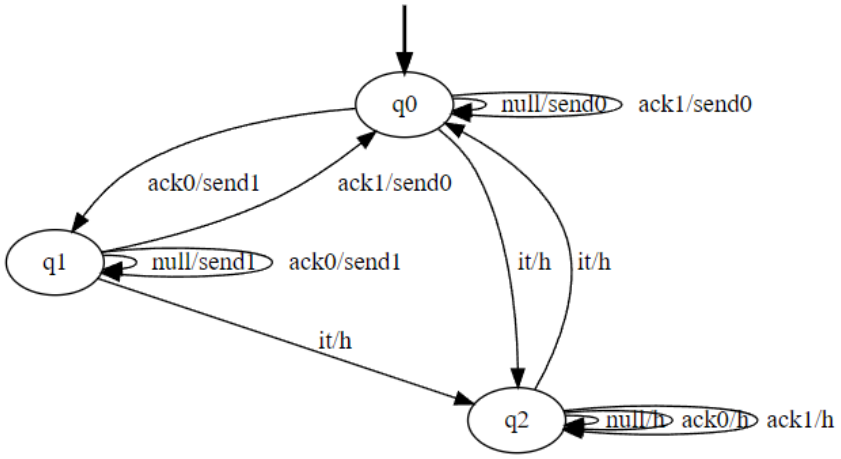


Figure 6.3: Mealy machine of the sender component of the Alternating Bit Protocol extended with interrupt handling. The initial LTL expressions do not hold for $q2$ and the related behavior.

The framework also keeps track of the states that are derived from the initial automaton, thus possess the initial temporal properties. Naturally, extending the automaton as demonstrated here results in behavior outside of the scope of the initial LTL expressions.

6.4 Refinement of the Interrupt Handling

The interrupt handling introduced in the previous section was quite abstract. In this section, we are going to introduce two different interrupt messages, and then two different interrupt handling messages. Regarding the behavior, the high-level interrupt handling stays the same, however, the exact way of handling is going to change based on the potential acknowledgements arriving during the interrupted state.

Let us refine the *it* symbol of the input alphabet with the symbols {*it1, it2*}. The automaton is also modified, as in Figure 6.4.

This step did not require any membership or equivalence queries, due to the input symbol refinement resulting in no undefined behavior in the automaton.

We should refine the output symbol h with the symbols {*h1, h2*}. At this point, the automata learning continues, as the lowest-level behavior of the system under learning is now unspecified and cannot be automatically inferred. Part of a possible run can be seen below:

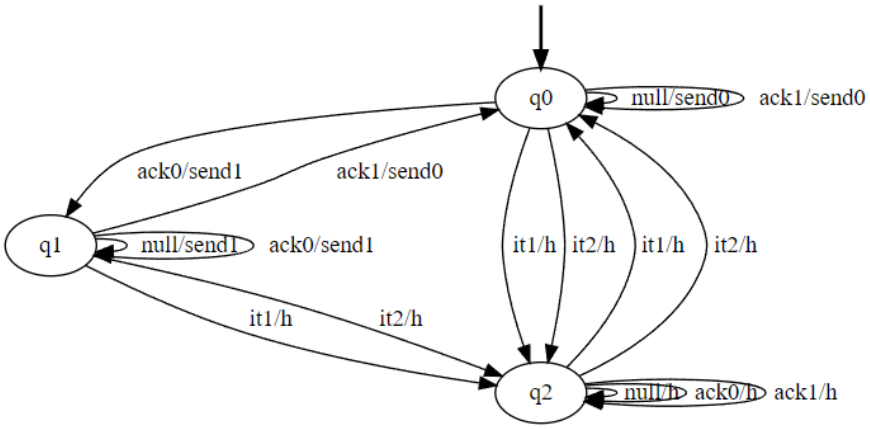


Figure 6.4: Mealy machine of the sender component of the Alternating Bit Protocol, after the refinement of the interrupt messages.

- ○ Query for sequence [it1] in {h1, h2}:
- ▷ Output: h1
- ○ Query for sequence [it2] in {h1, h2}:
- ▷ Output: h2
- ○ Query for sequence [it1, null] in {h1, h2}:
- ▷ Output: h1
- ○ Query for sequence [it1, ack0] in {h1, h2}:
- ▷ Output: h2
- ○ Query for sequence [it1, ack1] in {h1, h2}:
- ▷ Output: h1
- ○ Query for sequence [it1, it1] in {h1, h2}:
- ▷ Output: h1
- ○ Query for sequence [it1, it2] in {h1, h2}:
- ▷ Output: h1

... rest of the interrupted behavior ...

Again, the already specified behavior was not queried. Also, the specified higher-level behavior was used in constraining the queries: for behavior related to interrupts, only interrupt-handling outputs were offered and accepted.

After several membership queries, the learning results in an automaton resembling the one in Figure 6.5.

The algorithm still tracks the states, for which the LTL expressions hold. As the states *q2* and *q3* were refined from *q2* of the automaton in Figure 6.4, they are both outside the scope of the initial properties. Should we refine the elements of the initial alphabets as

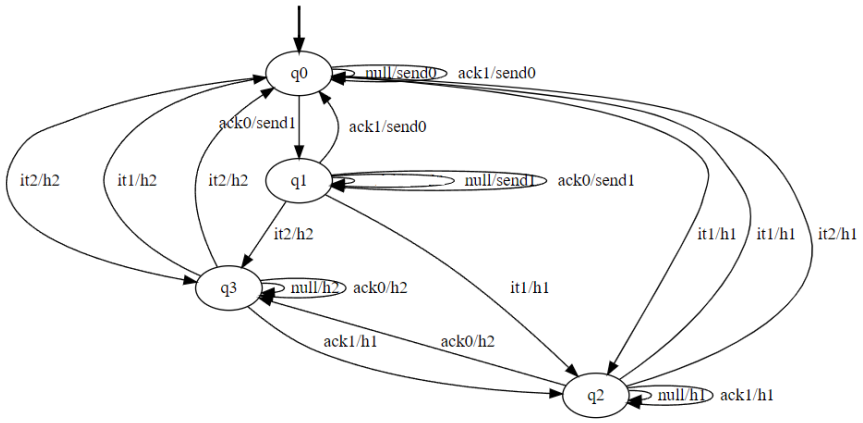


Figure 6.5: Mealy machine of the sender component of the Alternating Bit Protocol, after the refinement of the interrupt handling messages. The initial LTL expressions do not hold for $q2$, $q3$ and the related behavior.

our next steps – such as ack0 and send1 – the resulting automaton would possibly have more states with these properties.

6.5 Evaluation of the Results

The previous sections have illustrated the application of the modeling workflow on one component of a simple protocol by synthesizing a model from temporal logic expressions first, then gradually refining its behavior.

Regarding our main metric – the number of membership and equivalence queries – this workflow compares well to applying basic interactive automata learning from scratch for each step of the refinement. This can be seen in Table 6.1.

	Refinement-based Interactive Learning [MQ/EQ]	Basic Interactive Learning [MQ/EQ]
Initial (LTL) Automaton	0/0	15/1
Abstract Interrupt	10/1	46/1
Refined Interrupt	40/1	80/1
Total	50/2	141/3

Table 6.1: Comparison of refinement-based learning to basic learning from scratch for each step of the case study.

The workflow had several additional benefits. Firstly, we did not have to use automata learning for the part of the system specified by LTL expressions, thus obtaining an initial model that is correct-by-design. Although the automated synthesis from LTL formulae is more resource intensive than automata learning – still a quasi-polinomial algorithm – in practical cases it is much faster than querying an engineer. Also, when presenting membership queries to the engineer, the algorithm only posed questions regarding the newest added behavior, allowing the engineer to focus solely on that part of the behavior. In addition, several queries accepted only a constrained alphabet, providing content assist whenever possible and still ensuring correctness for that part of the system. This is due to

keeping track of refinement relationships, and as a side-effect it can also provide relevant meta-information about certain model elements – such the element of the previous step they are refinements of, or if they are in the scope of the initial p roperties.

In the end, the process resulted in the internal representation resembling that in Figure 6.6, from which only the relevant views have been shown in the previous sections.

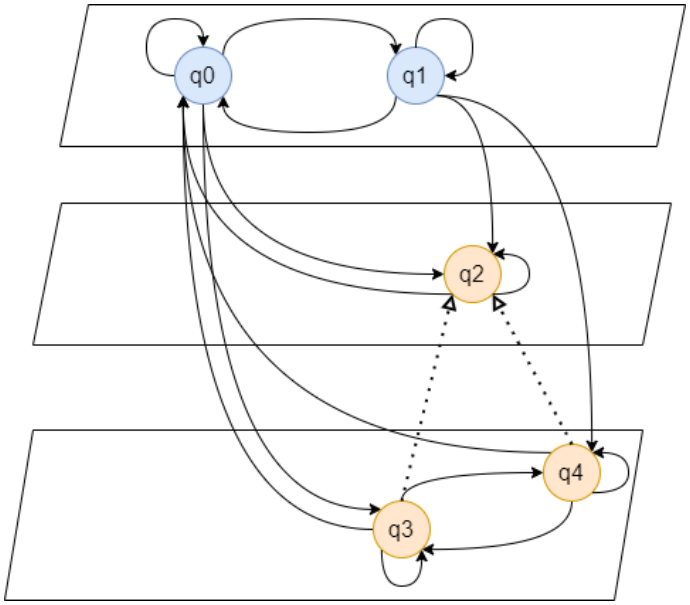


Figure 6.6: Internal representation of the ABP automaton. The labels of the transitions have been omitted for readability.

This representation has several benefits. It keeps track of the refinement relationships between elements, offering an arbitrary level of abstraction both for viewing and querying. Also, it provides information on whether the individual states (and the related edges) are in the scope of the initial properties: the blue states are either synthesized from them, or are refinements of such states, whereas the orange states are contained in the refined model but they are not in the scope of those requirements. Last but not least, it also supports transformation to simple Gamma Statecharts, enabling – among others – code generation capabilities.

Chapter 7

Evaluation

This chapter presents the applicability of the designed framework, evaluates its newly introduced learning algorithms and points out improvement possibilities. The main evaluation metric is the number of membership and equivalence queries, expressed with the size of the resulting automaton. It is important to note, that the numbers of queries regarding the algorithm, discussed in Section 7.1, are not necessarily the same as those presented to the user, discussed in Section 7.2.

7.1 Complexity of the Learning Algorithms

Let n denote the number of states in the canonical acceptor of the system under learning, k the size of the input alphabet Σ and m the length of the longest counterexample. Then, the DHC algorithm terminates after at most n equivalence queries. From the proof in [23] and [29], the number of suffixes is bound by $k + mn$, the total number of transitions to be considered in a learning round is at most nk, and the algorithm makes at most n learning rounds. The product of these factors gives the maximum number of membership queries for a run of the algorithm: $n^3mk+n^2k^2$. The framework uses an optimized version following the suffix handling proposed by Rivest and Schapir [28], which adds only one suffix from each counterexample, thus m can be regarded as 1.

As opposed to the traditional DHC, the **Adaptive DHC** algorithm can have more than n learning rounds, due to the *reset* command. However, this does not affect the maximum number of required equivalence queries. Let r denote the number of *reset* commands the algorithm receives. Then, the number of membership queries the algorithm poses is extended with an additional term to the total of $n^3k + n^2k^2 + n^2kr$ [8].

The **Generalized DHC** algorithm (not considering adaption commands) can have multiple starting points and differentiates between the new and the initial states and input symbols. Let n_{new} denote the number of additional states required to complete the initial model with $n_{initial}$ states. Similarly, let k_{new} denote the newly added input symbols and $k_{initial}$ denote the initial input symbols. Let n and k represent the total number of states and input symbols, respectively. Then, the algorithm terminates after at most n_{new} equivalence queries, as only distinguishable states can be split into separate states [29]. Also, the number of suffixes is bound by $k + n_{new}$. The number of transitions that need to be considered in iteration is $n_{new}k + n_{initial}k_{new}$, thus, the the total number of membership queries is limited by $n^3_{new}k + n^2_{new}n_{initial}k_{new} + n^2_{new}k^2 + n_{new}n_{initial}k_{new}k$.

Note, that for $n_{new} \to n$ and $k_{new} \to k$ – therefore $n_{initial} \to 1$ and $k_{initial} \to 0$– the number of equivalence and membership queries approach that of the original DHC algorithm, as being a corner-case of learning the whole automaton as an extension of the empty one. Similarly, for $n_{new} \to 0$ and $k_{new} \to 0$ – therefore $n_{initial} \to n$ and $k_{initial} \to k$ – the number of the required equivalence and membership queries also approach 0.

The **Refining DHC** algorithm (not considering adaption commands) can also have multiple starting points, but – due the specialities of model refinement – has slightly different numbers. Let $n_{additional}$ denote the additional states required for the canonical acceptor over the refined alphabet, representing the difference between the number of states of the initial automaton ($n_{initial}$) and that of the refined one (n). Then, the algorithm terminates after at most $n_{additional}$ equivalence queries. The number of suffixes is bound by $k + n_{additional}$ and the number of transitions that need to be considered in an iteration by $n_{additional}k + n_{initial}k$. Thus, the resulting limit for membership queries is $n^3_{additional}k + n^2_{additional}n_{initial}k + n_{additional}nk^2$.

Unlike in the Generalized DHC algorithm, the Refining DHC algorithm can derive information from the previous, abstract model during its run. However, the limits for the queries presented here are not lower – in case of the membership queries, they are seemingly even higher – than those of the other algorithm. This is due to the fact that as the number of possible behaviors in the abstract automaton approach 0, the number of behaviors to (re-)learn approach all the refined behaviors, resulting in the traditional DHC algorithm. When the number of additional behaviors approach 0 – represented by $n_{additional}$ – then the number of required membership queries approach 0 likewise.

It is important to note, that these theoretical limits and corner-cases are not representative for most practical applications. Even though Refining DHC can re-learn the whole automaton or run without any queries, it mostly synthesizes smaller parts of the behavior. Also, the comparisons between Generalized and Refining DHC are not meant for choosing the better one: they are and should be used to complete each other with different types of refinements, as it is done in the interactive learning framework.

Naturally, both the generalized and the refining algorithms can be extended to handle adaption commands, resulting in additional terms in the maximum number of membership queries depending on the number of *reset* commands the algorithm receives, not changing the limits in any other ways.

7.2 The Interactive Learning Entity

The number of questions posed by the Interactive Learning Entity are different from those of the learning algorithm for two reasons. On one hand, redundant queries can be easily replaced by automated queries in the *interactive oracle* or various *caching* solutions – already included in the framework, see Figure 3.7. On the other hand, the true number of these questions depend on the formalism through which the user presents them. These formalisms – discussed in Chapter 3 – may differ in the number of contained traces, therefore depend on the skills of the designing engineer. Thus, these calculations are limited to individual traces and (non-redundant) initial requirements are accounted for in the number of membership queries.

Let n denote the number of states in the canonical automaton and k denote the number of input symbols in Σ. Then, during the run of the (traditional) DHC algorithm, at most n equivalence queries and $n^2k + nk^2$ membership queries are presented to the user, while $n^3k + n^2k^2$ automated membership queries are attempted. These numbers are due to the

fact that all membership queries are attempted to be answered by automated means, then only the non-redundant ones are delegated. These non-redundant queries are independent of the number of equivalence queries, thus the missing factor n. The Generalized and Refining DHC algorithms are also similar: the number of equivalence queries remains similar to that of the algorithms – as these are always delegated – and the number of membership queries are simplified by the factor of the number of required equivalence queries.

The approach is similar in case of the adaptive algorithm too. The numbers correspond to the calculations in the previous section with one difference: the extra term is presented to the user in the form of r conflict resolutions.

Chapter 8

Conclusion

This chapter provides concluding remarks and possibilities for further improvement.

8.1 Contribution

The achieved results of this thesis can be seen in the following.

- Proposed temporal logic, and in particular LTL as a suitable formalism for specifying complex requirements in automata learning.
- Reviewed the interactive learning methodology for the extensive application of temporal logic.
  - Proposed solutions that fit into the architecture without modifications to the algorithm.
  - Extended the algorithm to use state-of-the-art approaches for model synthesis from LTL.
  - Extended the algorithm to handle the inherent abstraction as the main obstacle in the application of LTL.
- Extended the architecture of the learning algorithm to support the proposed modifications for the application of LTL.
  - Defined the required behavior to be implemented by the algorithm.
  - Defined the required extra behavior from the hypotheses of refinement-based algorithms.
  - Proposed an architecture in order to implement these modifications.
- Created an implementation that supports LTL requirements in different levels of the architecture and also the refinement-based workflow.
- Demonstrated the capabilities of the framework using LTL requirements in the refinement-based workflow through a case study.
- Evaluated the complexity of the proposed, new family of algorithms and the effects of their application on interactive learning.

8.2 Future Work

The proposed refinement-based extension to the interactive learning approach and the corresponding implementation require further analysis in practical applications. To support this, the implementation could be extended with a more user-friendly, graphical user interface, and also several other requirement formalisms for convenience. In particular, the extension of existing models using the proposed interactive and refinement-based methodology sound promising.

Non-deterministic models can be introduced to the framework in order to enable the usage of the refinement-based workflow for synthesis from general abstract models. In several cases – especially where the initial model is not synthesized from LTL – abstraction results in non-determinism, for which the refinement is not yet possible by this means.

New model synthesis approaches can be integrated through introducing extensions to the LTL formalism in order to support model quality optimization as seen in [3].

To optimize the learning and to evaluate different approaches, different, more complex approaches, such as TTT [17] could be considered for re-design to support abstraction refinement.

Bibliography

[1] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990. DOI: 10.1109/IEEESTD.1990.101064.

[2] Fides Aarts, Faranak Heidarian, Harco Kuppens, Petur Olsen, and Frits Vaandrager. Automata learning through counterexample guided abstraction refinement. In *International Symposium on Formal Methods*, pages 10–27. Springer, 2012.

[3] Shaull Almagor and Orna Kupferman. High-quality synthesis against stochastic environments. *CoRR*, abs/1608.06567, 2016. URL http://arxiv.org/abs/1608.06567.

[4] 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.

[5] Tomáš Babiak, František Blahoudek, Alexandre Duret-Lutz, Joachim Klein, Jan Křetínský, David Müller, David Parker, and Jan Strejček. The hanoi omega-automata format. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 479–486, Cham, 2015. Springer International Publishing.

[6] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*, volume 26202649. 01 2008. ISBN 978-0-262-02649-9.

[7] Aron Cs. Barcsa-Szabo. Supporting system design with automaton learning algorithms, 2019. URL https://diplomaterv.vik.bme.hu/en/Theses/Automatatanulo-algoritmusok-vizsgalata.

[8] Aron Cs. Barcsa-Szabo and Balazs Varady. Interactive learning for model-based software engineering, 2020.

[9] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Commun. ACM*, 12(5):260–261, May 1969. ISSN 0001-0782. DOI: 10.1145/362946.362970. URL https://doi.org/10.1145/362946.362970.

[10] Stephane Demri and Paul Gastin. Specification and verification using temporal logics. 2, 01 2009. DOI: 10.1142/9789814271059_0015.

[11] Volker Diekert and Paul Gastin. First-order definable languages. pages 261–306, 01 2008.

[12] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. Small-step live programming by example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, UIST '20, page 614–626, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375146. DOI: 10.1145/3379337.3415869. URL https://doi.org/10.1145/3379337.3415869.

[13] Sanford Friedenthal, Regina Griego, and Mark Sampson. Incose model based systems engineering (mbse) initiative. 01 2009.

[14] 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.

[15] 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.

[16] Falk Howar, Bernhard Steffen, and Maik Merten. Automata learning with automated alphabet abstraction refinement. In Ranjit Jhala and David Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 263–277, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-18275-4.

[17] 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.

[18] 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.

[19] 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.

[20] Jan Kretínský, Tobias Meggendorfer, and Salomon Sickert. Owl: A Library for ω-Words, Automata, and LTL. In Shuvendu K. Lahiri and Chao Wang, editors, *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, volume 11138 of *Lecture Notes in Computer Science*, pages 543–550. Springer, 2018. DOI: 10.1007/978-3-030-01090-4_34. URL https://doi.org/10.1007/978-3-030-01090-4_34.

[21] Saul A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16(1963):83–94, 1963.

[22] Michael Luttenberger, Philipp J. Meyer, and Salomon Sickert. Practical synthesis of reactive systems from ltl specifications via parity games. *Acta Informatica*, 57 (1-2):3–36, Nov 2019. ISSN 1432-0525. DOI: 10.1007/s00236-019-00349-3. URL http://dx.doi.org/10.1007/s00236-019-00349-3.

[23] 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.

[24] Thibaud Michaud and Maximilien Colange. Reactive synthesis from LTL specification with Spot. In *Proceedings of the 7th Workshop on Synthesis, SYNT@CAV 2018*, volume xx of *Electronic Proceedings in Theoretical Computer Science*, page xx, 2018.

[25] 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.

[26] 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.

[27] Roman Redziejowski. An improved construction of deterministic omega-automaton using derivatives. *Fundamenta Informaticae*, 119:393–406, 08 2012. DOI: 10.3233/FI-2012-744.

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

[29] 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.

[30] Bartha Tamás and Majzik István. *Biztonságra tervezés és biztonságigazolás formális módszerei*. Akadémiai Kiadó, 2019. ISBN 978 963 454 291 9. DOI: 10.1556/9789634542919. URL https://mersz.hu/kiadvany/534.

[31] Fujun Wang, Zining Cao, Lixing Tan, and Hui Zong. Survey on learning-based formal methods: Taxonomy, applications and possible future directions. *IEEE Access*, 8: 108561–108578, 2020. DOI: 10.1109/ACCESS.2020.3000907.

[32] Shufang Zhu, Lucas M. Tabajara, Jianwen Li, Geguang Pu, and Moshe Y. Vardi. A symbolic approach to safety LTL synthesis. *CoRR*, abs/1709.07495, 2017. URL http://arxiv.org/abs/1709.07495.

[33] Wieslaw Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1):135–183, 1998. ISSN 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(98)00009-7. URL https://www.sciencedirect.com/science/article/pii/S0304397598000097.