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

Interactive Learning for Model-Based Software Engineering

Scientific Students' Association Report

Authors:

Áron Barcsa-Szabó
Balázs Várady

Advisors:

Rebeka Farkas
dr. Vince Molnár
dr. András Vörös

2020

Contents

Abstract

Model-based technologies improve the efficiency of designing and developing IT systems by making it possible to automate verification, code generation and system analysis based on a formal model. A simple way of describing the behavior of systems is state-based modeling, which - due to the advancements of formal analysis techniques in recent years - can be widely and effectively utilized when analyzing systems. A possible way of synthesizing such models is to apply active automata learning algorithms.

Acquiring a correct formal model of a system can be challenging. On one hand, it is difficult for the designing engineer to keep every property of the envisioned system in mind at a given time, partly because of the complexity of the system, and because of possible hidden implications and contradictions. On the other hand, there are fully automated solutions, for instance, active automata learning, where the model construction is characterised by a teacher component - which is familiar with the extensive behavior of the system under learning - and a learner component - which synthesises the model via queries to the teacher component. However, such solutions have practical boundaries when validating the inferred behavior of the system. We propose a semi-automated solution, that applies automata learning to provide an interactive environment for model development.

The objective of this work is to support the design of systems and components from the ground up through InterActive automata learning. It utilizes the frequent input of designing engineers - who themselves are regarded as the teaching component of the algorithm - along with automated techniques, resolving the infeasibility of automated equivalence validation. The resulting semi-automated concept allows the engineer to focus on the expected behavior of the system, specifying its behavioral requirements in a declarative way and evaluating the model proposed by the algorithm.

This thesis presents an adaptive state-based modeling framework, into which we designed and integrated the interactive algorithm. The thus created framework combines the advantages of manual and automated solutions. Additionally, we extended the framework to handle and reconcile different formalisms, allowing the analysis and development of interactive automata learning algorithms to support model-driven development with an extended scope.

Chapter 1

Introduction

Context Model-based engineering is the formalized application of modeling during system design and development. It improves the efficiency of designing and developing IT systems by formalizing verification, code generation and system analysis, and in certain cases enables their automation as well. Such models can be designed both manually and in automated ways – by applying various model synthesis techniques. In case of behavioral models, a straightforward way of representation is state-based modeling, which - due to the advancements of formal analysis techniques in recent years - can be widely and effectively utilized when analyzing systems. A possible way of synthesizing such models is to apply active automata learning algorithms.

Problem Statement Acquiring a correct model of a system can be challenging. On one hand, it is difficult for the designing engineer to keep every property of the envisioned system in mind at a given time, partly because of the complexity of the system, and because of possible hidden implications and contradictions. Additionally, to conveniently specify requirements, different scopes, abstractions and formalisms may be applied, which may be difficult to reconcile. On the other hand, there are fully automated solutions – usually stricter in these aspects –, for instance, active automata learning, where the model construction is characterised by a teacher component - which is familiar with the extensive behavior of the system under learning - and a learner component - which synthesises the model via queries to the teacher component. However, such solutions have practical boundaries when validating the inferred behavior of the system.

Objective The objective of our work is to support the design of systems and components from the ground up through a semi-automated solution – *InterActive* automata learning – which utilizes both the frequent input of the designing engineers and automated techniques. As a result of this approach, the users themselves are regarded as the teacher component of the algorithm, resolving the infeasibility of automated equivalence validation. This results in a semi-automated solution driven by declarative behavioral requirement specification, which allows the designing engineers to focus on the exptected behavior of the system and on evaluating the model proposed by the algorithm.

Contribution The thesis presents an adaptive state-based modeling framework combining the advantages of manual and automated solutions, into which we designed and integrated the interactive algorithm. We created a proof of concept implementation of the approach, allowing system design through different formalisms, and the analysis and development of interactive automata learning algorithms to support model-driven development with an extended scope.

Related Work There are multiple automata learning frameworks in the literature, including *LearnLib*[19] that provides a Java framework for active and passive automata learning, *libalf*, which provides learning techniques for finite automata implemented in *C++* and *Tomte*[3], a framework utilizing LearnLib for counterexample driven abstraction refinement of real software components modeled in a restricted class of finite automata.

In [6] Dana Angluin et al. predict strongly unambiguous Büchi-automata using automata learning. Cobleigh et al. use active automata learning through multiple oracles to learn assumptions for compositional verification[10]. Groce et al. utilize model checking as an equivalence oracle of active automata learning to automatically handle inconsistent models through Adaptive Model Checking[14]. Giannakopoulou et al. utilize active automata learning and symbolic execution to learn temporal component interfaces[13].

In [20], Kahani et al. synthesize real-time embedded systems by taking certain properties of the system to reduce the synthesis of models to the solution of quantifier-free first-order logic formulas, generate appropriate solutions using the LTS formalism, extend and minimize it and transform it to UML-RT state machines. Kupferman et al. introduce extensions of the LTL formalism to transform the model synthesis problem into an optimization problem based on various quality metrics. They assume a stochastic setting and propose a solution which is 2EXPTIME-complete in [4].

Molnar et al. introduce the Gamma Statechart Composition Framework to facilitate the design, verification, validation and code generation for component-based reactive systems[26].

Outline The thesis is organized as follows. Chapter 2 provides an outline of the necessary theoretical background. Chapter 3 gives an overview of our approach, first in methodology, then in architecture. Chapter 4 describes the tools and steps taken to create a proof of concept implementation to validate our approach. Chapter 5 presents a case study to demonstrate the capabilities and limitations of the implementation and the approach. Chapter 6 evaluates the components of the implementation. Chapter 7 provides concluding remarks and possibilities for further improvement.

Chapter 2

Background

This chapter provides the necessary theoretical background of the thesis. First, we introduce model-based engineering, then discuss the foundations of automata theory and automata learning, finally we elaborate upon specifying requirements.

2.1 Model-Based Engineering

Due to the application of the modeling concept in several completely different domains, first of all, we need to define the meaning of *model*.

Definition 1 (Model). A model is the simplified image 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 Section 2.2.

The process of deriving design artifacts is called *model transformation*.

Definition 3 (Model Transformation). Model transformation is the process of generating the target model from the source model. This process is described by by a transformation definition consisting of transformation rules, and a transformation tool that executes them. A transformation rule is the mapping of elements of the source model to the elements of the target model. [21] ▪

Model transformations can be categorized based on the types of the source and target models: model-to-model (M2M), model-to-text (M2T), text-to-model (T2M) and text-to-text (T2T). These categories fundamentally define the tools required and usable for handling the different models.

There are also two important factors to consider when designing a model transformation:

- *Consistency*: the same structure or behavior is described by the source and the target models (in their respective domains).
- *Traceability*: the images of the original elements of the source model can be traced back to the original elements, from which they were generated.

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[12]. 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 automaton theory is based.

2.2.1 Fundamentals of Formal Language Theory

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

Definition 4 (Alphabet). Let Σ be a finite, non-empty set. Σ is an alphabet, its elements are symbols or characters. ▪

Definition 5 (Word). If Σ is an alphabet, then any finite sequence comprised of the symbols of Σ are words (or Strings). Σ^n represents the set of every n length word consisting

of symbols in Σ: $\Sigma^n : w_1 w_2 \dots w_n$, where $\forall 0 \leq i \leq n : w_i \in \Sigma$. The set of every word under an alphabet, formally $\bigcup_{n>0} \Sigma^n$ is denoted by Σ^*. The empty word is denoted by ϵ. ▪

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

Definition 6 (Prefixes, Substrings and Suffixes). Let an arbitrary $w = uvs$, where $w, u, v, s \in \Sigma^*$. u is the prefix, v is the substring, and s is the suffix of w. Formally:

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

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

Definition 7 (Formal Language). An arbitrary set of words under an Alphabet Σ is a Language. Formally: $L \subseteq \Sigma^*$. ▪

Definition 8 (Prefix-closure). Let $L \subseteq \Sigma^*$ and $L' = \{u \in \Sigma^*, v \in \Sigma^* : uv \in L\}$. In other words, L' is the set containing all the prefixes of every word of L. L is prefix-closed if $L = L'$. ▪

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

2.2.2 Properties of Deterministic Automata

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

One of the simplest automata is the Deterministic Finite Automaton.

Definition 9 (Deterministic Finite Automaton). A Deterministic Finite Automaton is a Tuple of $DFA = (S, s_0, \Sigma, \delta, F)$, where:

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

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

An example of a DFA (Deterministic Finite Automaton) from[30] 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 $4i + 3a$.*

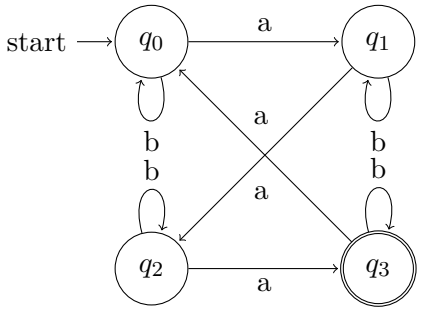


Figure 2.1. A simple DFA from [18].

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 10 (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)*.*

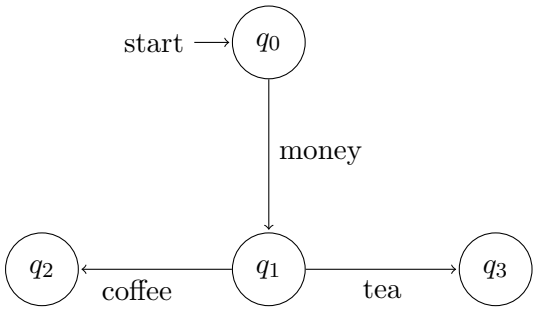


Figure 2.2. A simple LTS

DFAs 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 automata designed to communicate with output symbols instead of accepting and rejecting states.

Definition 11 (Mealy machine). A Mealy machine or Mealy automaton is a Tuple of $M = (S, s_0, \Sigma, \Omega, \delta, \lambda)$, where:

- S is a finite, non-empty set containing the states of the automaton,
- $s_0 \in S$ is the initial state,
- Σ is the input alphabet of the automaton,
- Ω is the output alphabet of the automaton,
- $\delta : Q \times \Sigma \rightarrow Q$ 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.[30]

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.*

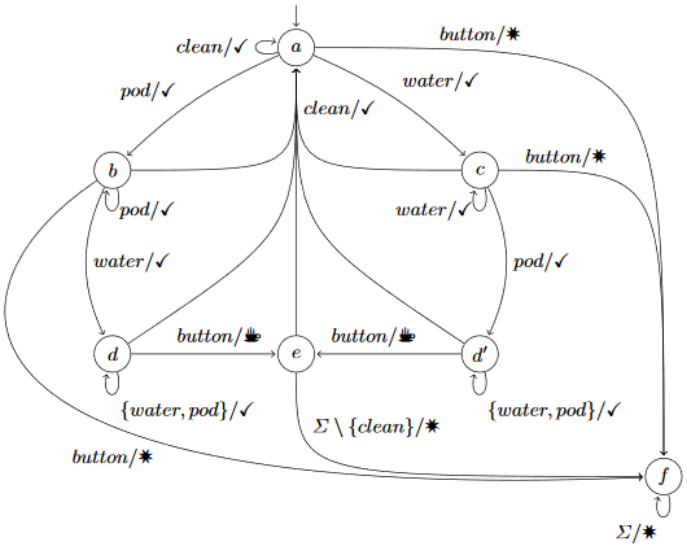


Figure 2.3. Mealy machine representing the functionality of a coffee machine.[30]

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

2.2.3 Relations of Formal Languages and Automata

Definition 12 (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 13 (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 14 (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 $\equiv_M$.[22]. ▪

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[22], 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[22][28]). 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 [22][28]. ▪

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

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

Example 5. *Given the Mealy machine $M_{coffeemachine}$ in Figure 2.3, the runs:*
<clean, ✓>,
<pod water button, ☕>
are in $[\![M_{coffeemachine}]\!]$, since the given input words cause the corresponding outputs, while the runs
<clean, ☕> and
<water button button, ✓>
are not, since these input sequences do not produce those outputs.

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

Definition 15 (Equivalence of words wrt. $\equiv_P$[30]). 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 6. *Taking Figure 2.3 as an example, the following words are equivalent wrt. $\equiv_{[\![M]\!]}$:*

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

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

Theorem 2 (Characterization theorem[30]). Iff mapping $P : \Sigma^* \to \Omega$ $\equiv_P$ has finitely many equivalence classes, there exists a Mealy machine M, for which P is a semantic functional.

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

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

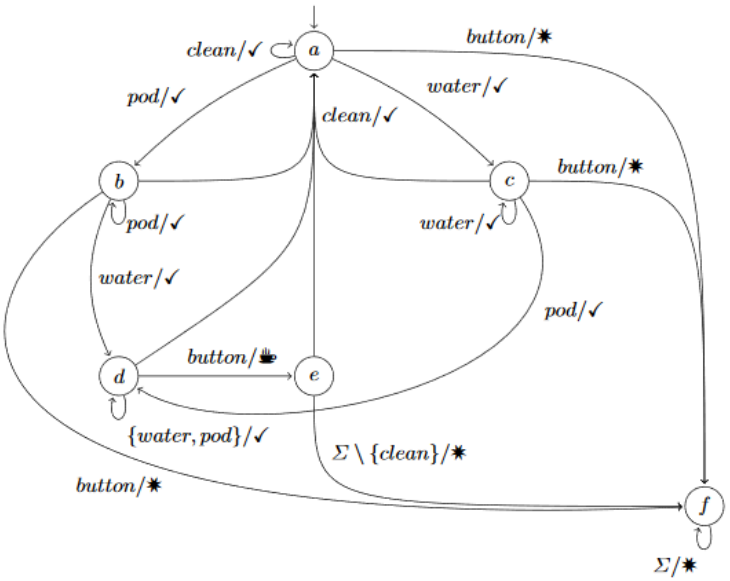


Figure 2.4. Minimal version of the Mealy machine seen in 2.3

2.2.4 Minimization of Automata

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

Definition 16 (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 2.4.

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[30]): 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 17 (k-distinguishability[30]). 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 18 (exact k-distinguishability). Two states, $s, s' \in S$ are exact k-distinguishable, denoted by $k^=$ iff s and s' are k-distinguishable, but not (k-1)-distinguishable ▪

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

- no states are 0-distinguishable and
- two states s_1 and s2 are (k+1)-distinguishable iff there exists an input symbol $\alpha \in \Sigma$, such that $\lambda(s_1, \alpha) \neq \lambda(s_2, \alpha)$ or $\delta(s_1, \alpha)$ and $\delta(s_2, \alpha)$ are k-distinguishable.[30]

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 7. *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 2.4.

The process explained in Example 7 is partition refinement, the exact algorithm and proof of its validity can be seen in [30]. Partition refinement is a version of the minimization algorithm for DFAs proposed by Hopcroft[16].

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

Definition 19 (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.[30] Essentially this establishes P_1 and P_2 from Example 7 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.

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[17].

In order to monitor a system, access to its behavioral information is required. There are two approaches, which separate 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 to it. 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[5]. It defines the separation of the algorithm to a teacher and a learner component in a way, where the teacher can only answer the minimally adequate questions needed to learn the system. These two questions, or queries are are follows:

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

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

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

As seen on Figure 2.5, 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[30]:

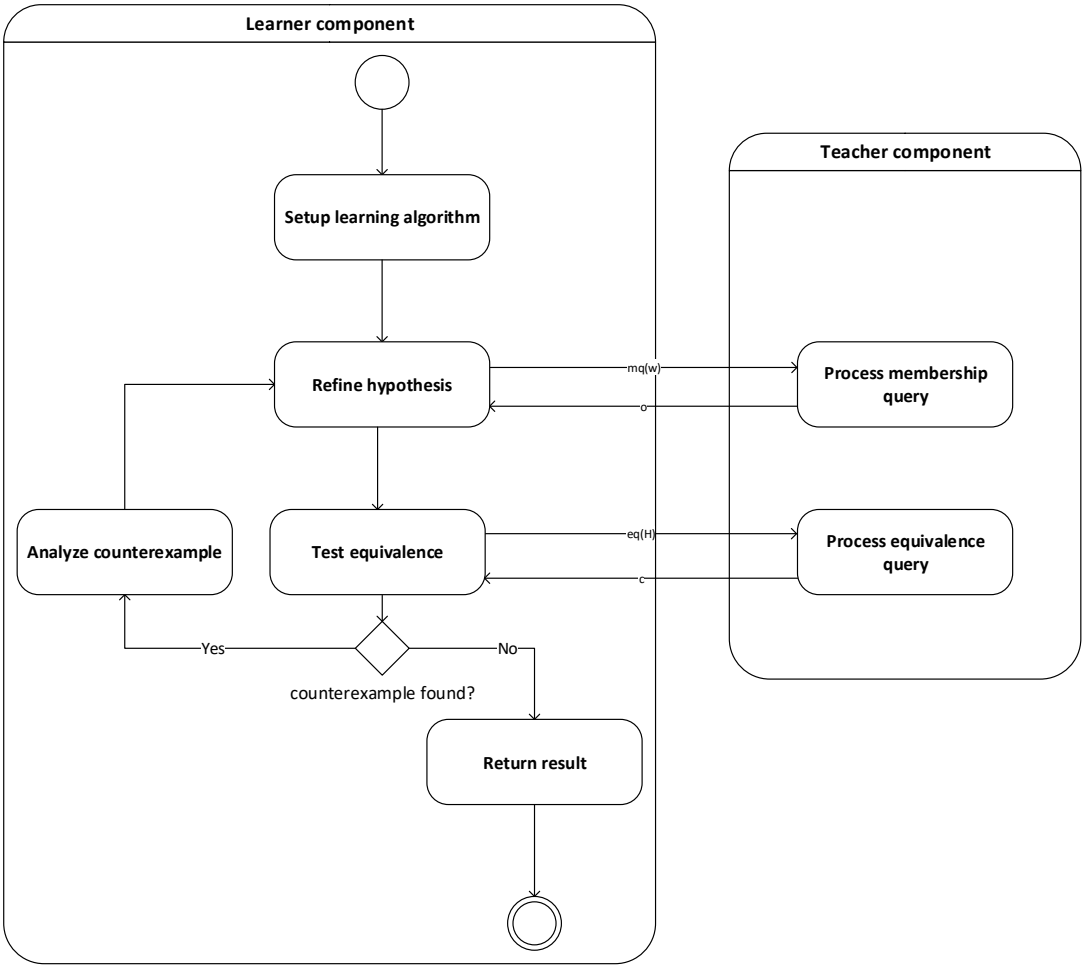


Figure 2.5. Active automata learning

- 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 20 (D-epimorphism). Let $D \subseteq \Sigma^*$. We call a surjective function $f_D : S \to S'$ existential D-epimorphism (surjective homomorphism) between M and M' if, for all $s' \in S'$ there exists an $s \in S$ with $f_D(s) = s'$ such that for all $\alpha \in \Sigma$ and all $d \in D$: $f_D(\delta(s, \alpha)) = \delta'(s', \alpha)$, and $\lambda^*(s, d) = \lambda^*(s', d)$. ▪

Note, that active learning deals with canonical Mealy machines, in other words, the canonical form of SUL, and not, the perhaps much larger Mealy machine of SUL itself.

Since active learning algorithms maintain an incrementally growing extended spanning tree for $H = (S_H, h_0, \Sigma, \Omega, \delta_H, \lambda_H)$, i.e., a prefix-closed set of words reaching all its states and covering all transitions, it is straightforward to establish that these hypothesis models are images of the canonical version of SUL under a canonical existential D-epimorphism, where D is the set of distinctive futures underlying the hypothesis construction[30]

- 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.[30]

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

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

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

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

2.3.1 Direct Hypothesis Construction[25]

The Direct Hypothesis Construction algorithm, which hypothesis construction can be seen in Algorithm 1 follows the idea of the breath-first search of graph theory. It constructs the hypothesis using a queue of states, which is initialized with the states of the spanning tree to be maintained. Explored states are removed from this queue, while the discovered successors are enqueued, if they are provably new states. The algorithm starts with a one-state hypothesis, including only the initial state, reached by ϵ and D = Σ. It then tries to complete the hypothesis: for every state, 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. [30][25]

Algorithm 1: Hypothesis construction of the Direct Hypothesis Construction algorithm as seen in [30].

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

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

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

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[25].

2.4 Specifying 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

Throughout this thesis, the concept of requirements is used widely, therefore, it is essential to define it precisely.

Definition 21 (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). ▪

Requirements are important, as the specification is present at both the beginning and the end of the software development process: the design can only start, if there are some requirements formulated, and acceptance tests are only possible in the presence of requirements.

Requirements can be specified in many different ways, the most common being textual requirements in traditional feature lists. This method is an informal way of requirements 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 requirements by defining patterns and mapping the individual patterns to formal semantics, however, there are also more abstract approaches, such as *temporal logics*.

The rationale behind the precise formalization of requirements is the wide range of automated applications, especially in *formal methods* – such as validation, formal verification, test oracle generation and requirement documentation generation.

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. There exist also definitions using Kripke-structures [24] as base models. The syntax of LTL expressions over paths of LTSs is defined as follows:

Definition 22 (Syntax of LTL Expressions [9]). 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 $\neg F(\neg p)$,
- $pWBq$ as $\neg((\neg p)Uq)$,
- pBq as $\neg((\neg p)Uq) \wedge Fq$

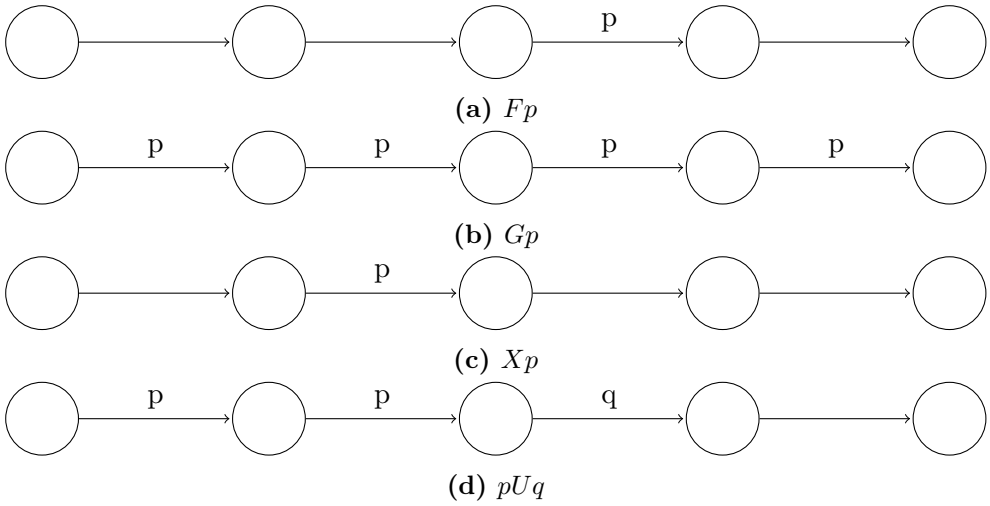
The semantics of LTL expressions are defined as follows:

Definition 23 (Semantics of LTL Expressions[9]). Let $\pi = (s_0, a_1, s_1, a_2, ...)$ be a path of an LTS model M. Then the formal semantics to the LTL expression P 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 te logical entailment operator, $M, \pi \vDash q$ denoting: for path π of model M, q holds. ▪

Figure 2.6 shows examples for the intuitive meanings of different LTL operators.



(a) Fp

(b) Gp

(c) Xp

(d) pUq

Figure 2.6. Intuitive examples for the meanings of different LTL operators

LTL expressions over paths of LTS models can be used to formulate requirements for systems interpretable as LTSs 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 Büchi-automata (or in general ω-automata), which then can be executed parallel with the modeled system to verify its behavior.

Büchi-automata are the nondeterministic extension of the already introduced finite automata to infinite input words, which can be defined in the following way.

Definition 24 (Büchi-automaton [9]). A Büchi-automaton is a tuple $A = (S, s_0, \Sigma, \delta, F)$, where:

- S is 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 accepting states of the automaton.

A run of the Büchi-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: $lim(r) = \{s | \nexists j \geq 0 : \forall k > j : s \neq s_k\}$.

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. ▪

Example 8. *Figure 2.7 shows a Büchi automaton generated from the expression* $(GFa) \to (GFb)$*. Notice the nondeterminism and how the automaton accepts (only) infinite runs which satisfy the original LTL expression.*

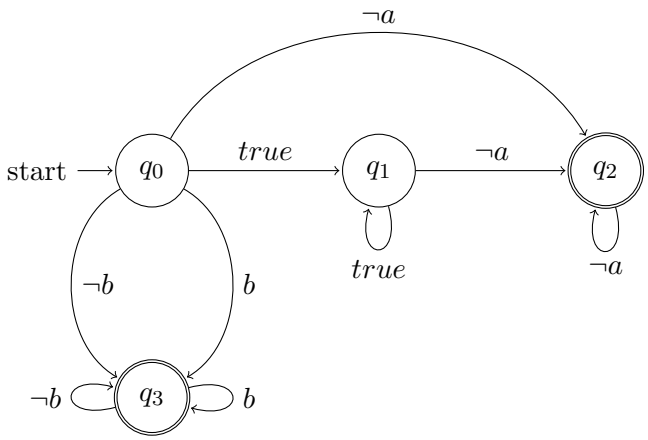


Figure 2.7. A Büchi automaton for the LTL expression $(GFa) \to (GFb)$

Chapter 3

Overview of the Approach

In this chapter, the various aspects of the proposed approach are detailed. In Section 3.1, the application of this methodology is presented from the users' point of view: how to use the interactive automata learning framework – also called *Interactive Learning Entity* or *ILE* - and how they can utilize it to design reactive systems in a declarative way. Then, in Section 3.2, the applied software architecture, software components, algorithms and data structures are presented: first, the components concerned with the automata learning algorithm, then those responsible for its interaction with the oracle component, then the possible interactions of the oracle with the engineer.

3.1 Overview of the Methodology

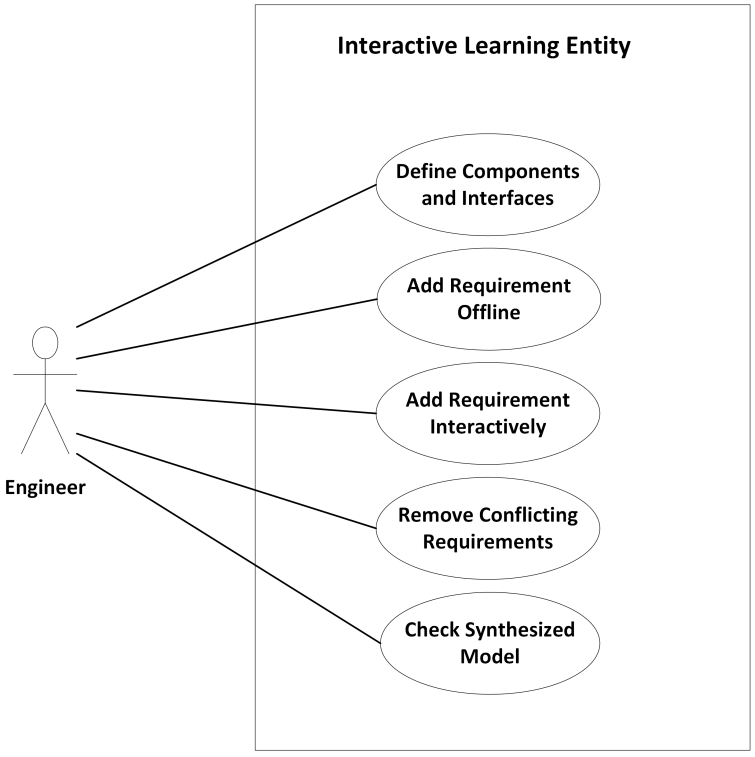


Figure 3.1. Interaction types between the engineer and the ILE

Our methodology is heavily based on the interaction of the user with the ILE. The different types of interacitons are summarized on Figure 3.1 and are elaborated on in Subsection 3.1.2. These interactions take place in a predefined order – the *proposed workflow*, illustrated on Figure 3.2, the individual steps of which are explained in detail in the following subsections. 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 input formalisms of individual steps in both the offline and the online phases have a predefined syntax with the corresponding, precisely defined semantics. Their common feature is the declarative way of describing the system components, which allows the engineer to focus solely on the expected behavior and acquire a minimal model exhibiting the specified functionality.

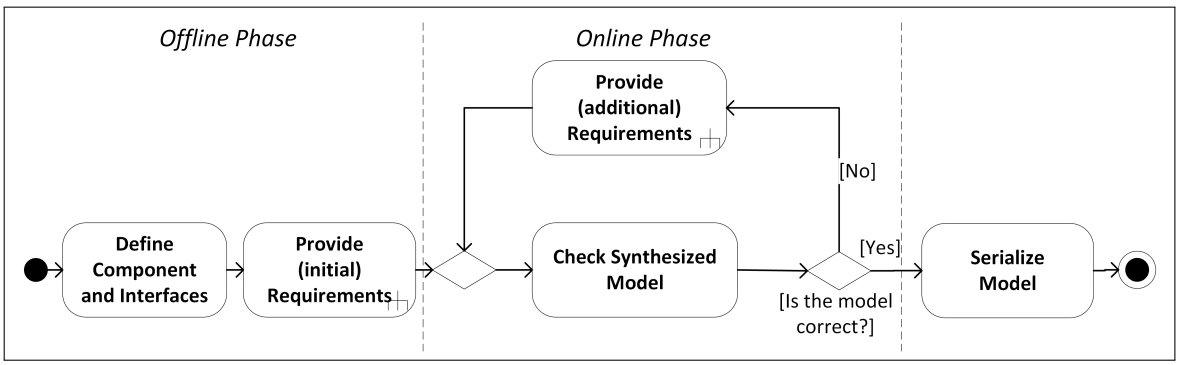


Figure 3.2. The Proposed Workflow

3.1.1 Component and Interface Definition

The first step of the workflow is the definition of the system components. This happens in the offline phase, as the determination of the system components, their exact boundaries and interfaces is part of the architecture, not the behavior. The engineer must provide the names of the system components, along with their interfaces – in other words their input- and output alphabets – before the workflow can proceed to the next step.

Users are encouraged to specify input and output characters qualified with port names in the format '`Port.character`', as this supplies the subsequent steps with essential information about the connections of the individual system components.

The components are handled as independent systems in every other aspect. 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). The syntax of component and interface declarations is quite simple, as illustrated in Listing 3.1.

```
Please provide the system components (space-separated):
>TrafficLight
Please provide the input alphabet for component TrafficLight (space-separated):
>TrafficControl.interrupt TrafficControl.toggle
Please provide the output alphabet for component TrafficLight (space-separated):
>TrafficDisplay.red TrafficDisplay.yellow TrafficDisplay.green TrafficDisplay.blinkingYellow
```

Listing 3.1. Example of a component declaration along with its interfaces

3.1.2 Requirement Types

During the workflow, the engineers can provide requirements in both phases. These requirements can vary greatly in their scope – from being specific to individual runs to being generally valid for the whole component – in addition to the differences in the formalism the user defines them through.

In the offline phase, this means that the users add requirements they have formulated in advance. This is useful for more general requirements, with the scope of the whole component, easily formulated as program logic expressions, or long and complex traces.

In the online phase, adding requirements means answering the questions formulated by the algorithm about a yet unspecified behavior at a specific place in the trace currently being examined. This too can be answered through program logic - e.g. when the engineer realizes a general property during the model construction - but also through traces and through giving the corresponding output directly.

The currently supported requirement types can be seen on Figure 3.3.

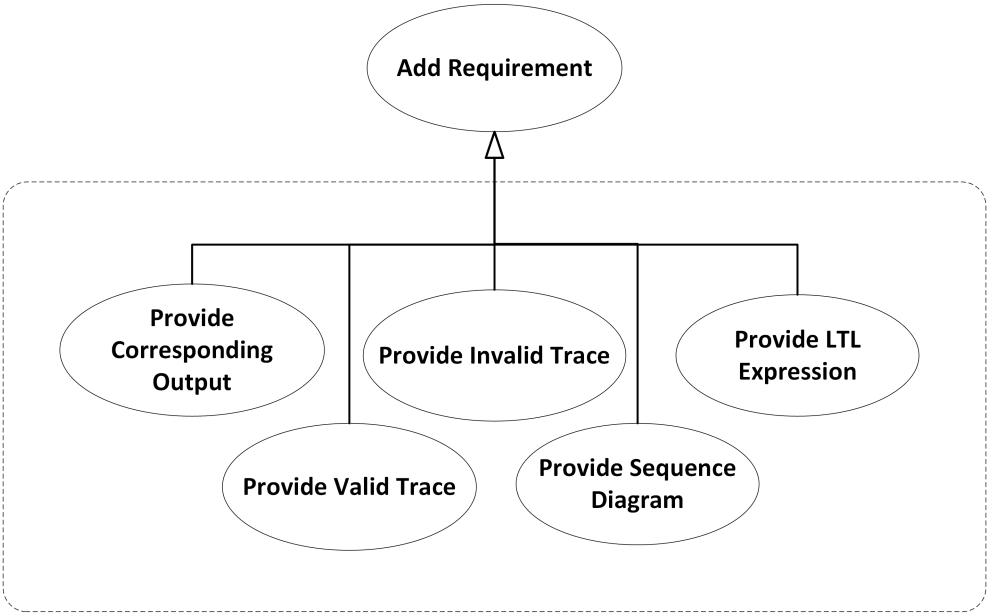


Figure 3.3. The supported requirement types

Corresponding Output

This is the simplest way of specifying the behavior of the system, also containing the least amount of information among the different model types. To put simply, this means giving the output for a given input sequence, without any additional information. This supposedly answers the question of the ILE at *one* given point, and that is the end of its scope.

Examples of corresponding output specification can be seen in Listing 3.2. The alphabets of the component are the ones defined in in Listing 3.1.

```
Offline:
// choosing the type of the requirement omitted
>TrafficControl.toggle TrafficControl.toggle TrafficControl.toggle/TrafficDisplay.yellow

Online:
Unknown output for input sequence [TrafficControl.toggle TrafficControl.toggle TrafficControl.
   toggle]:
```

```
// choosing the type of the requirement omitted
>TrafficDisplay.yellow
```

Listing 3.2. Examples of corresponding output specification

Valid Trace

Valid traces contain information about multiple related input sequences, as they provide the corresponding output for any prefix of the contained input sequence. This can be useful, as the engineers often take the whole output sequence into consideration when determining the output for some inputs. Thus, the ILE can obtain *multiple* answers concerning the behaviors in question by automated means, saving on the number of required interactions with the user.

Examples can be seen in Listing 3.3, assuming the previously used alphabets.

```
Offline:
// choosing the type of the requirement omitted
>TrafficControl.toggle/TrafficDisplay.red TrafficControl.toggle/TrafficDisplay.green TrafficControl
   .toggle/TrafficDisplay.yellow

Online:
Unknown output for input sequence [TrafficControl.interrupt]:
// choosing the type of the requirement omitted
>TrafficControl.interrupt/TrafficDisplay.blinkingYellow TrafficControl.interrupt/TrafficDisplay.red
    TrafficControl.interrupt/TrafficDisplay.blinkingYellow
```

Listing 3.3. Examples of a valid trace specifications

Invalid Trace

Invalid traces are similar to valid traces, with the difference that the contained behavior must not appear in the resulting model – thus defining traces to exclude. They are most useful for small output alphabets, or when the range of possible behaviors is otherwise contained - e.g. through program logic expressions or several other excluded traces.

They can also be used to check the hidden implications of other requirements: trace-based models are easy to construct and the ILE will signal any conflicts with other, more complex requirements of which the engineer may not see the hidden implications.

Examples for invalid traces can be seen in Listing 3.4. Notice, that invalid traces have the same syntax as valid traces, the difference is in their semantics.

```
Offline:
// choosing the type of the requirement omitted
>TrafficControl.interrupt/TrafficDisplay.green TrafficControl.interrupt/TrafficDisplay.yellow

Online:
Unknown output for input sequence [TrafficControl.interrupt]:
// choosing the type of the requirement omitted
>TrafficControl.interrupt/TrafficDisplay.green TrafficControl.interrupt/TrafficDisplay.yellow
   TrafficControl.interrupt/TrafficDisplay.red
```

Listing 3.4. Example of a trace to exclude

Sequence Diagram

UML-like sequence diagrams are trace-based models that can contain multiple traces, due to them having various combined fragments for branching the behavior - like *alt* and *opt* - and for referencing behaviors specified elsewhere - like *ref*.

Sequence diagrams can also be used to model arbitrarily long, possibly looping behavior, thereby containing plenty of information, which results in possibly answering *numerous* questions formulated by the ILE.

We introduce our own sequence diagram formalism specifically designed to model system components. An example for their syntax can be seen on Figure 3.4. The '*systemComponent*' is the component the behavior of which is being modeled, the '*inputComponent*' and '*outputComponent*' are symbolizing the sources and targets of the inputs and outputs. The textboxes marked with stars are the port qualifications.

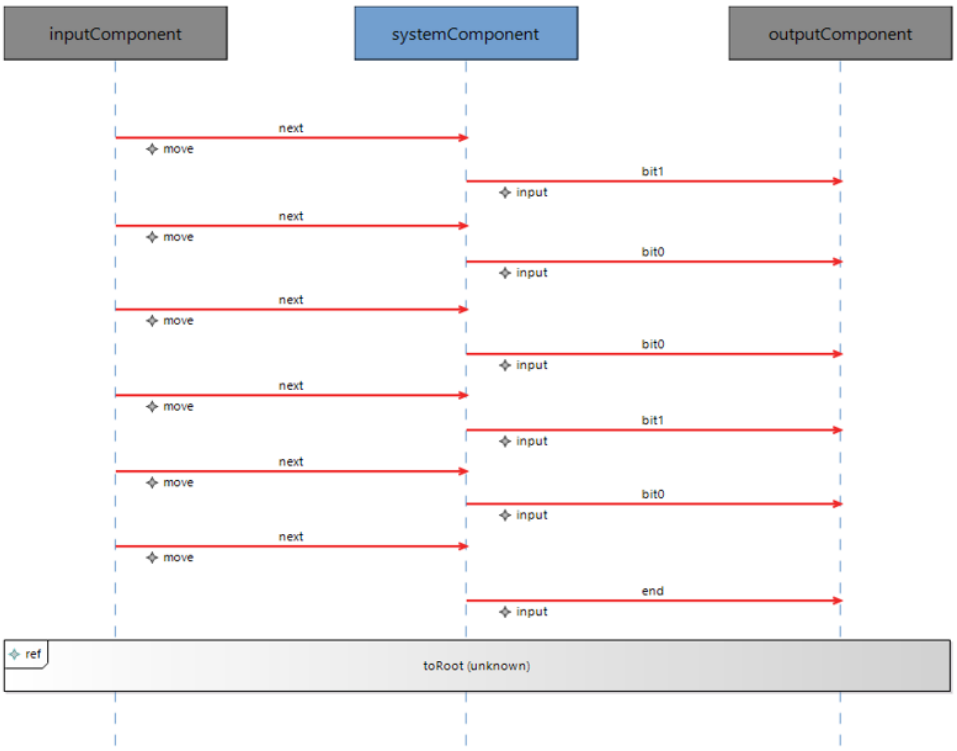


Figure 3.4. Example for the syntax of the sequence diagrams

It is important to note, that the specification and integration of sequence diagrams into this framework is not yet complete.

LTL Expression

LTL expressions are able to contain infinitely many traces through program logic-based requirement specification: they can be used to formulate propositional logic expressions with temporal connectives over *paths* of a *base model*, as described in Subsection 2.4.2.

They can be used to formulate requirements that must hold for the whole component, during the whole execution. Thus – depending on their interpretation – they contain lots of information, which may result in answering *numerous* questions posed by the ILE.

For our application, we introduce our own LTL expression language with its own syntax and semantics - although attempting to keep it similar to other generally known variants, especially that of SPOT [11]. The full syntax of the LTL expressions – also determining the operator precedence – can be seen in Listing A.1 in the Appendix.

The base model of the LTL expressions is the LTS interpretation of the component under learning. This means, that the set of atomic propositions that can be used in these expressions are 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 event happening at any given step during the execution of the system. We introduced these semantics to the LTL expressions:

the conjunction of exactly one input and one output character must hold at any given point for it to be considered correct – and every other character must be negated at the same point. This also entails, that given another character not explicitly negated at that point, it is automatically negated, and in case that no proposition is declared explicitly, either one of the non-explicitly negated characters hold.

The semantics of the supported temporal connectives, and other aspects of the LTL semantics in general, are similar to those described in Subsection 2.4.2.

Examples for LTL expressions can be seen in Listing 3.5.

```
Offline:
// choosing the type of the requirement omitted
>F(TrafficControl.interrupt -> X(G(TrafficControl.toggle) -> G(TrafficDisplay.blinkingYellow)))

Online:
Unknown output for input sequence [TrafficControl.interrupt TrafficControl.toggle]:
// choosing the type of the requirement omitted
>F(TrafficControl.interrupt -> X(G(TrafficControl.toggle) -> G(TrafficDisplay.blinkingYellow)))

Equivalent as a result of the event semantics (omitting port qualifications for simplicity):
>F(interrupt&!toggle -> X(G(toggle&!interrupt) -> G(blinkingYellow&!red&!green&!yellow)))
```

Listing 3.5. Examples of LTL expressions

It is important to note, that the specification of our LTL variant is not yet finished.

3.1.3 Conflicting Requirements

The requirements provided by the designing engineer to the ILE may easily be conflicting, especially in case of LTL expressions and invalid traces that describe arbitrarily long sets of behaviors. This is expected, as during system design, when the engineer refines the models and reaches lower levels of abstraction, certain scenarios may conflict with some oversimplified conditions. At that point, those too must be refined, thus replaced. This is why it is essential for the system to provide some kind of conflict handling within the practical boundaries of the available resources.

This problem is a difficult and resource intensive task for algorithmic reasons elaborated later. Consequently, the ILE only guarantees to handle the conflict, when it also interferes with the model synthesis, in which case, the user is asked to remove one of the conflicting models, before the analysis of the behavior can proceed – as shown on Figure 3.5.

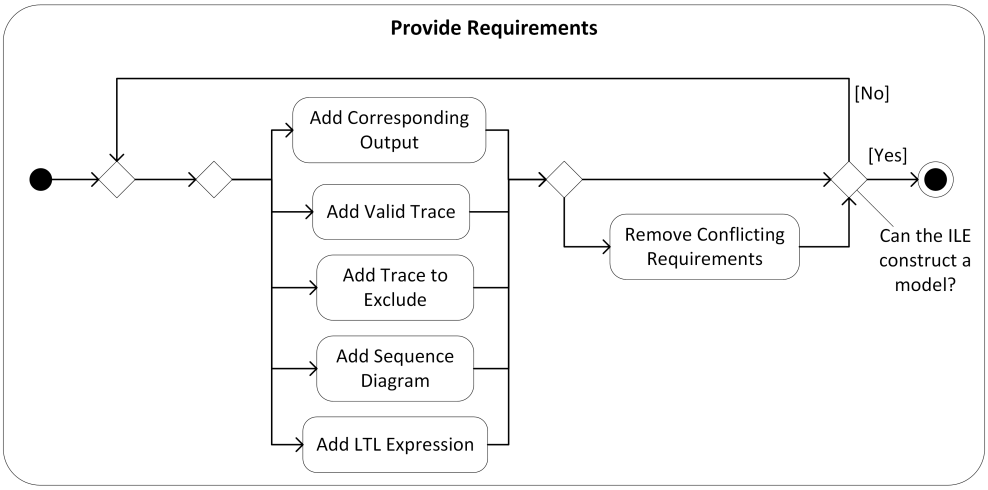


Figure 3.5. The process of adding a requirement

As conflicts only become apparent during the online phase of the workflow, that is where the conflicts have to be handled. However, conflicts that were introduced earlier are also discovered and resolved in that phase. An example of a requirement conflict handling can be seen in Listing 3.6.

```
Models 1) IO Pair Model: [TraffiControl.interrupt]/TrafficDisplay.red and 0) Invalid Trace Model:
   TrafficControl.interrupt/TrafficDisplay.red are conflicting.
Please choose which model to remove:
>1      //this removes the I/O pair model
```

Listing 3.6. Example of a requirement conflict handling

3.1.4 Checking the Correctness of 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 model representing the current state of the model synthesis – the equivalent of an equivalence query in automata learning algorithms. The user can either approve this model – in which case the automata learning and therefore the designing of the behaviour is complete – 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, conflicting requirement can be provided to guide the learning algorithm through the process described in Subsection 3.1.3.

If the model is accepted, the design phase is complete and the model is serialized. If a counterexample is provided, the online phase resumes and the system design continues until the next possible model is reached.

Examples of models offered in equivalence queries can be seen on Figure 3.6.

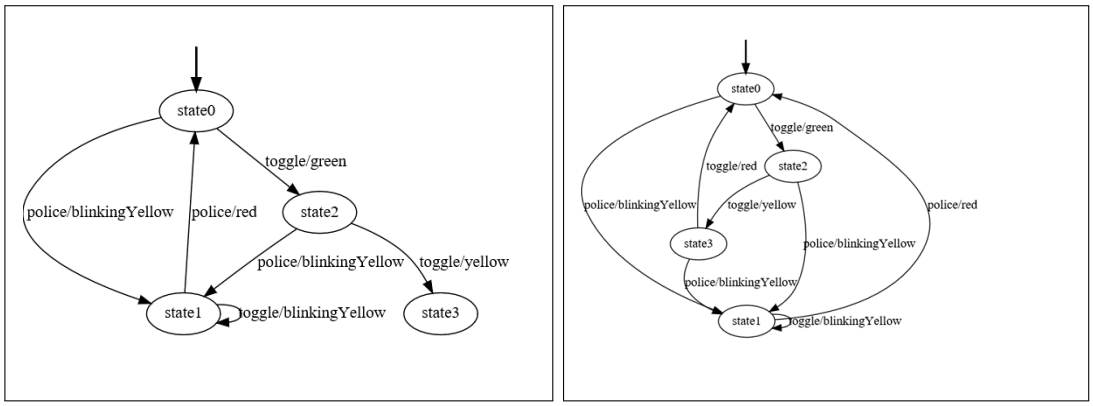


Figure 3.6. Equivalence query for an incomplete model (*left*) and the final model (*right*)

3.1.5 The Resulting System Model

When each of the component models declared during the first step of the workflow are completed, the resulting system model can be serialized and handed over to the engineer for further extensions or usage, e.g. for code generation. The serialization can happen in various formalisms.

A possible formalism is the Gamma statechart, introduced in [26]. Gamma statecharts are high-level state-based models, to which every functionality offered by the Gamma Statechart Composition Framework can be applied. Our framework offers full-scale Gamma serialization: when choosing this formalism, a whole project will be created, along with interface definitions, component definitions - for each of the previously declared components, with the synthesized behavior - and a composite system definition, connecting the components based on the names of their ports.

Another possibility is the serialization to the Mealy machine formalism of the framework - as presented when checking the correctness of the model. This results in a lower-level set of independent models, with a completely different set of applications.

3.2 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. A functional overview of this architecture is depicted on Figure 3.7. The following subsections elaborate on the connections and details of these components.

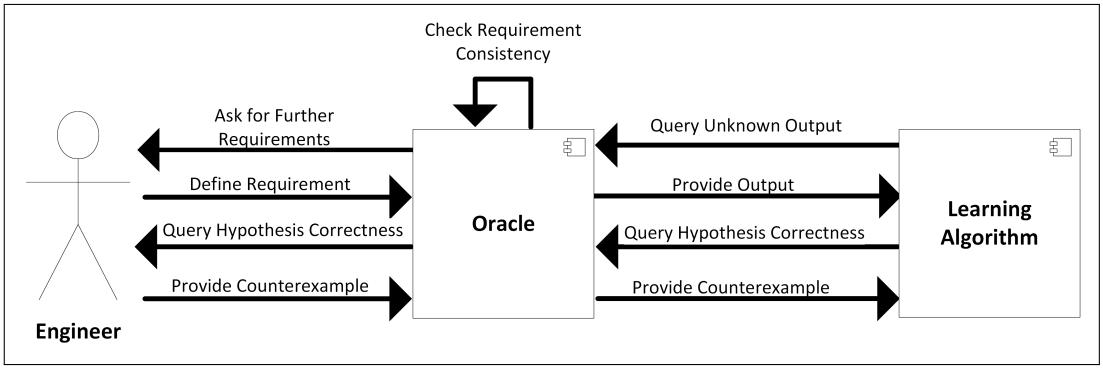


Figure 3.7. High-level architecture of the components of the ILE

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 3.7 shows the Learning Algorithm delegating its queries through the oracle, which delegates the questions to the user. The abstract approach presented in Figure 3.7 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 optimizations 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.

3.2.1 The Cost of Interaction

As discussed in the previous subsection, one of the most important cost metrics of the presented interactive learning architecture is the number of questions that reach the user. In order to minimize this, we propose a heuristic by which a decision can be made regarding which queries provide valuable information – based on the currently defined requirements. Since active automata learning algorithms aren't equipped for such *adaptive learning*, we created a new approach.

Active learning algorithms generally assume that the information they require is readily available, and thus follow a "greedy" approach of querying. While this is appropriate for

fully automated solutions, greedily exploring the design space can result in several magnitudes larger amount of previously unexplored queries, which do not necessarily provide new information, resulting in longer learning rounds (with more membership queries) exploring a larger amount of the behavior, and conversely less equivalence queries. To allow the designing engineer to validate the hypothesized model more frequently, and control of which unexplored behavior should be explored, a less greedy approach is required.

To solve the above issue, we introduce the concept of adaptive active automata learning, which uses types of behaviors - as illustrated in Figure 3.8 - as a heuristic to adaptively decide if and where a greedy approach should be taken. Already explored behaviors (e.g. previously queried, cached) as well as behaviors contained in the defined requirements can be answered in an automated way, allowing greediness. On the other hand, not specified behaviors should be explored in a more reserved manner, controlled by the user - not the automata learning algorithm. Based on the requirements outlined above, we defined three commands to control the adaption of such an algorithm.

- *OPTIMISTIC* (greedy) heuristic is used if the algorithm should follow a greedy approach in the next steps.
- *PESSIMISTIC* (reserved) heuristic is used if the algorithm should not query the investigated behavior further.
- *RESET* is used to re-start the learning if necessary.

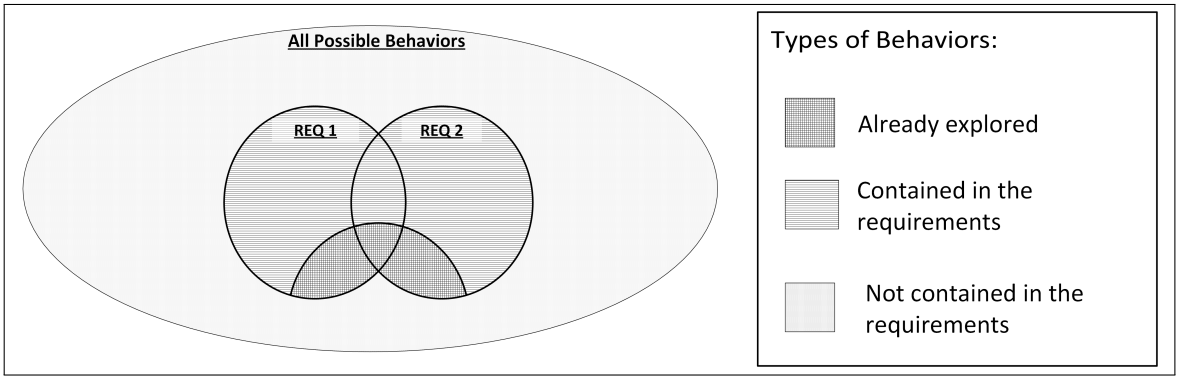


Figure 3.8. Types of behaviors during the learning process with two defined requirements

3.2.2 The Oracle

The architecture of the oracle component can be seen on Figure 3.9.

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. It consists of two main components: the parser and the interactive learnable.

The parser is responsible for handling the input of the user and transforming it to a formalism interpretable by the interactive learnable. This is necessary for enabling event semantics in requirements, separating the input formalism from that of the possible dependencies, and enabling feedback on the input of the user. This is realized through the conventional architecture of compilers: creating a language for the requirements, generating a parser which creates an abstract syntax tree, then a DOM (Document Object

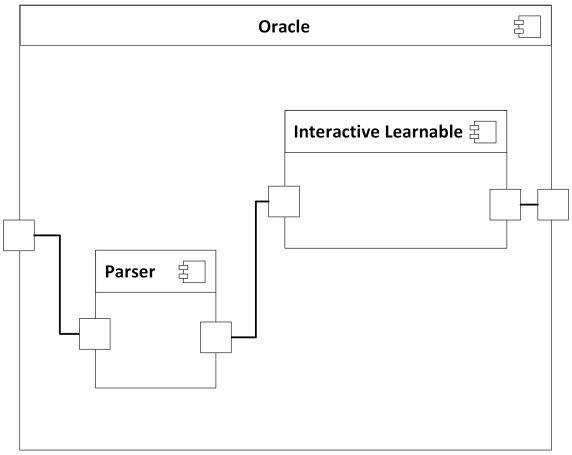


Figure 3.9. Architecture of the Oracle

Model) from the input and provides feedback, applying M2M or M2T transformations to the desired formalisms.

The interactive learnable stores the requirements received from the parser in the form of *partial models* and answers the queries of the learning algorithm. The architecture of of the interactive learnable can be seen on Figure 3.10.

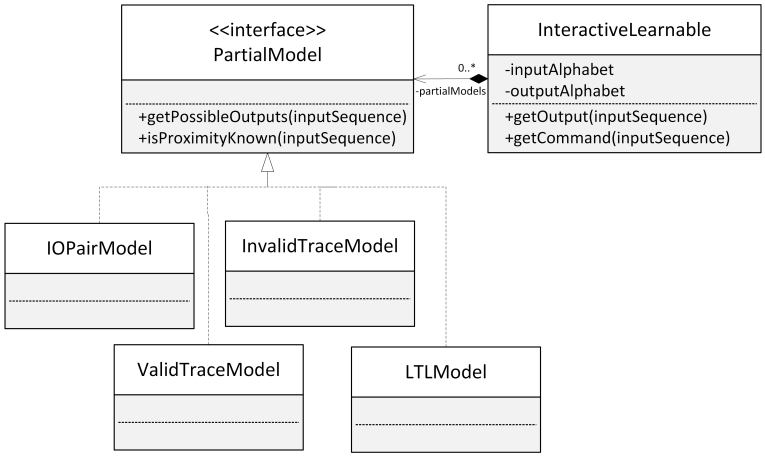


Figure 3.10. Architecture of the interactive learnable component

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, creating the loop on Figure 3.5.

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 one of the proposed adaption commands 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 is a frequent operation 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 Subsection 3.1.3 – introduces two serious problems: 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 our 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.

Currently, there are two main types of partial models - corresponding to the two main types of requirements: trace-based models, which store valid or invalid scenarios for an execution of a model, and LTL models, containing high-level behavioral properties which must be fulfilled by the resulting model. The following two subsections elaborate on these models.

Trace-based models correspond to trace-based requirements and store a arbitrarily long finite sequence of input-output pairs. The corresponding requirement types include: corresponding outputs to input sequences, valid and invalid traces and sequence diagrams.

The common property of these models is that they can be represented via conventional incomplete finite automata. The automata contain information about a given input sequence if they reach an accepting state at the end of the word. They contain more information when other accepting states are reachable from that point.

Trace-based requirement types are mapped to a specific kind of partial model representing the corresponding automaton. Although each of these automata is a finite automaton, they vary greatly in their complexity of execution, thus the complexity of their provided behavior. The number of these models can be huge with very frequent execution.

For instance, in case of IOPairModels (corresponding output-type requirements), it is enough to check if the accepting state can be reached and they surely contain no additional information. However, sequence diagrams can contain branching and loops in the behavior, thus require more complex algorithms.

LTL models on the other hand correspond to LTL requirements: program logic based requirements that describe the behavior for infinite input/output sequences.

They correspond to nondeterministic ω-automata. As the acceptance condition of Büchi-automata is closest to that of finite automata – as shown in Chapter 2 –, we chose them for the target of the mapping of the corresponding requirements. It should be noted, that any of the automata classes recognizing the ω-regular languages could have been chosen with an appropriate interpretation.

As Büchi automata accept only infinitely long words over the given input alphabet, the process of the determination of the possible outputs - the condition of a character being marked a possible output - was adopted accordingly.

In our interpretation, a Büchi automaton has information about a given finite input sequence, if at the end of the input the automaton reaches an accepting state. Then the possible outputs are the elements of the output alphabet on the last edge taken. As the automaton is nondeterministic (and cannot be determinized), there may be multiple accepting states, thus multiple edges, and even one edge might contain multiple possible outputs. In this case the conjunction of the possible outputs on the corresponding edges is the possible output of the automaton.

There may be other interpretations of Büchi automata in this context. For instance, it would be enough for an input sequence to reach a state from which an accepting state is reachable. This would provide an answer in more cases, than our interpretation, but also introduce faulty behavior which would have to be handled in the workflow. Also, it would be possible to track impossible outputs when no accepting state can be reached.

A summarizing example of the different automaton types can be seen on figure 3.11.

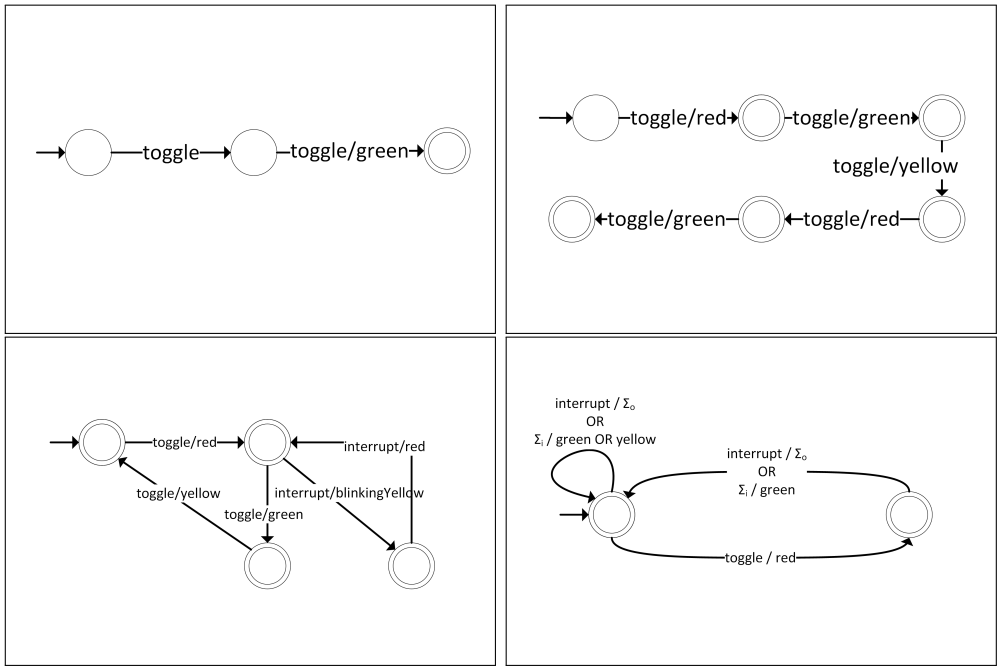


Figure 3.11. Examples for attempting to model the requirement 'after toggle/red, for toggle the output is green' through I/O pair automaton (*upper left*), valid trace automaton (*upper right*), sequence diagram automaton (*lower left*) and a Büchi automaton (*lower right*).

3.2.3 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, we built upon and designed a variant of the Direct Hypothesis Construction algorithm. 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. The Direct Hypothesis Construction algorithm is also a straightforward and easily modifiable algorithm, making the adaption to specific heuristics simple to design and implement. In order to make adaption possible, the teacher component of automata learning needs to handle not only the output of a query, but also the adaption heuristic the algorithm should adapt to. This enables an adaptive learning algorithm, which receives both the desired output and the learning heuristic to utilize from its teacher component - keeping an identical, but extended automata learning architecture, as shown in Figure 3.12. The resulting algorithm can be seen in Algorithm 2. 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.

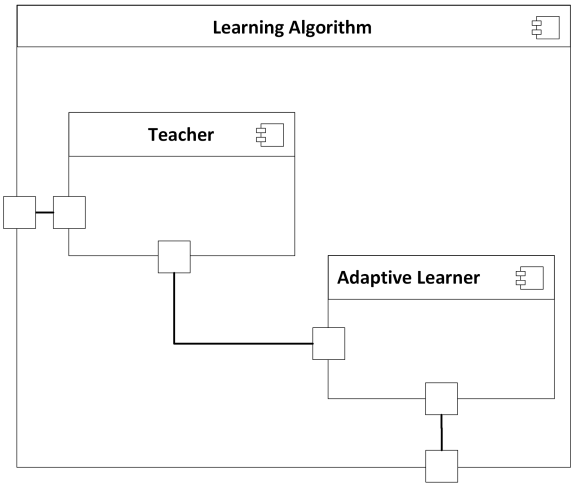


Figure 3.12. Architecture of the Learning Algorithm

Algorithm 2: 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 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
```

3.2.4 Caching

Caching the previous answers to queries can be a straightforward way of reducing the number of questions the oracle has to answer, specifically in the case of the DHC algorithm, where each learning round makes every single query that the previous did. Thus we introduced a caching layer between the oracle and the learning algorithm, which controls which questions are forwarded to the oracle, and conversely, which can be automatically answered.

In case of the user providing conflicting requirements, as proposed in Subsection 3.1.3, the user needs to remove one or more requirements, some of which might already have corresponding entries in the cache. Since query responses inferred through some models (such as LTL expressions) are not backwards traceable, there might not be a way to specify exactly which cached entries became outdated. In order to solve this issue, while still enabling the user to provide conflicting requirements, the cache is reset every time a conflict arises, as illustrated on Figure 3.13. Since the oracle stores the requirements, this does not create extra questions to the user and in most cases can happen in the background through communication of the cache and the oracle. The overview of the resulting framework can be seen in Figure 3.14.

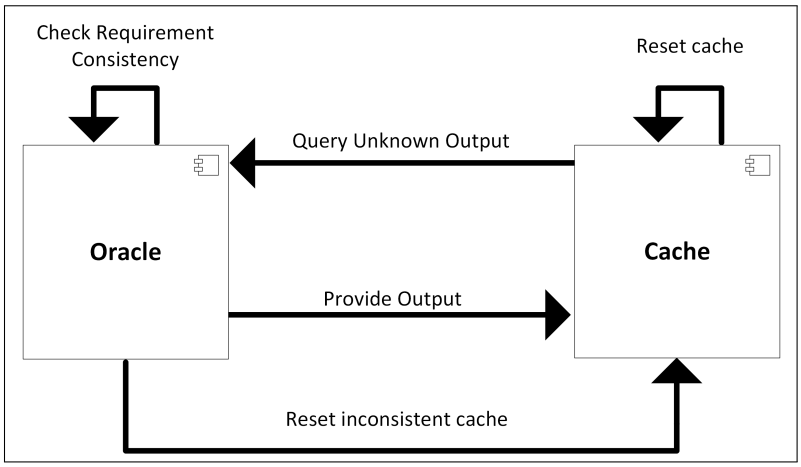


Figure 3.13. Overview of cache reset logic

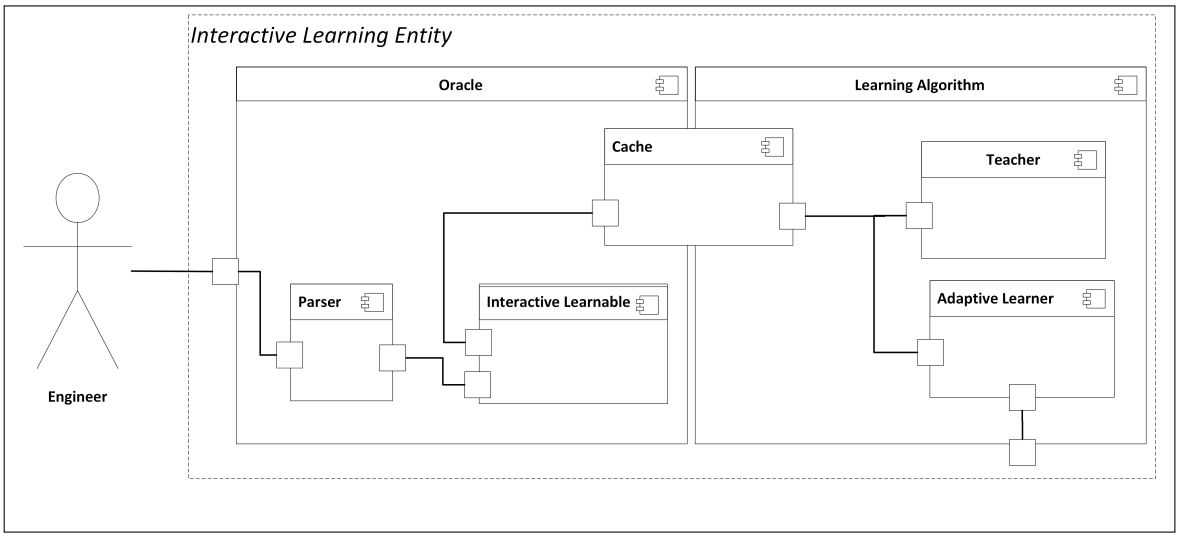


Figure 3.14. Architectural overview

Chapter 4

Implementation

In order to validate our approach, an implementation was created. Of the subsequent sections, Section 4.1 discusses the utilized tools, and Section 4.2 elaborates on the steps taken to provide a proof of concept implementation of the proposed *Interactive Learning Entity*.

4.1 Tooling

4.1.1 Eclipse Environment

Eclipse is a popular, open-source integrated development environment (IDE). It is mainly used for Java-related application development, but also supports several other programming languages. It consists of a base workspace and an extensible plug-in system. Using this plug-in system, the develpment environment is easily customizable for different purposes, such as programming in different programming lanugages, modeling (using the Gamma Framework or Yakindu), or testing.

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.

4.1.2 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.

Xtend

Xtend is a general-purpose, high-level programming language based on Java. It is statically typed, object-oriented and uses the type system of Java. Xtend programs are compiled to Java code, thus allowing seamless integration with existing Java libraries. It provides numerous convenient extensions to Java, such as dispatch methods, type inference, operator overloading and extension methods.

4.1.3 Sirius

Sirius is an open-source project for developing graphical modeling languages. It is integrated into the Eclipse environment, enabing the specification of viewpoints for EMF models, thus the creation of graphical views. In a Sirius workbench (editor), the elements of the viewpoint specification models are mapped to individual EMF model elements, thus allowing their graphical interpretation and editing. The whole viewpoint definition procedure is declarative, using OCL [15] (or Acceleo Query Language, AQL) expressions for the traversal of the diagram elements when needed.

Sirius supports various representation types. Traditional Sirius diagrams consist mainly of nodes and edges between nodes, suitable for models in which the position of the diagram elements carries no meaning - like several structural modeling languages. It also supports table and tree representations, and also *sequence diagrams* for modeling behavior - in which the position on the diagram is also part of the semantics.

4.1.4 Owl

Owl [23] is a tool collection for ω-words, ω-automata and linear 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 [7], 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.

4.1.5 LearnLib

LearnLib[19] 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.

4.1.6 Automata Learning Framework

In order to give a foundation to the implementation described in this thesis, a previously created automata learning framework in [8] was extended upon. Since the framework was implemented using the Java programming language, the high-level view seen in Figure 4.1 is represented as an 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 (*Teacher* package) and the learner (*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 way, the adapter package provides flexibility on what inputs can be used. As Figure 4.1 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 re-usability 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.

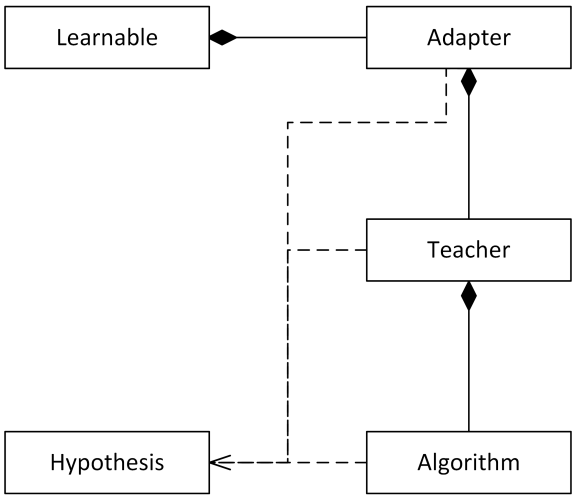


Figure 4.1. Structure and relations of the packages comprising the Automata Learning Framework[8]

4.2 Interactive Learning Entity

To achieve our goal, we designed and implemented extensions to the previously presented Automata Learning Framework resulting in a proof of concept implementation of the proposed Interactive Learning Entity, which 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 on Figure 4.2. It is important to note the extension of components shown in Figure 4.1, 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. The following subsections elaborate upon these components.

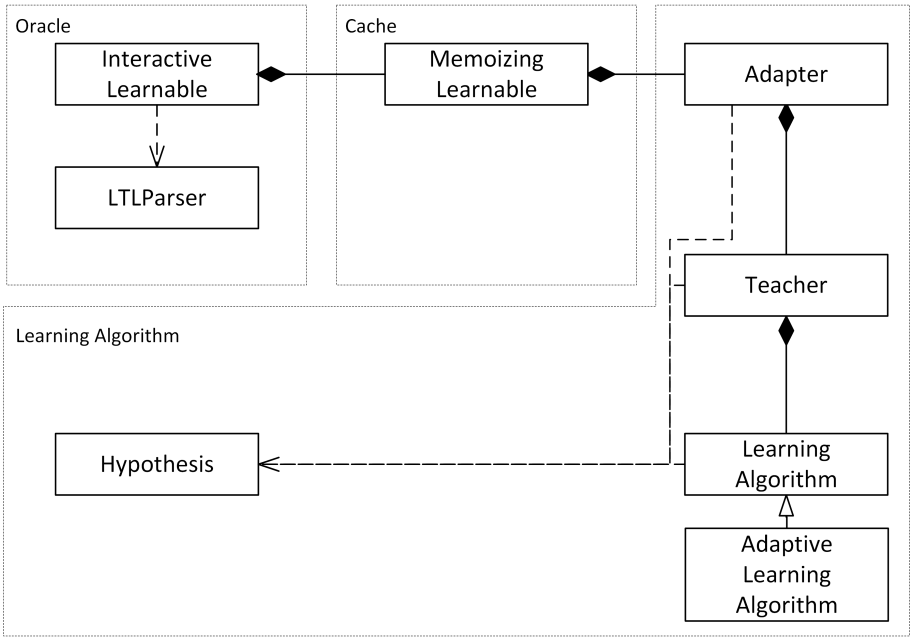


Figure 4.2. The architecture of the ILE.

4.2.1 The Oracle

The **interactive learnable** is the main part of the oracle component, responsible for storing models created from the provided reqirements and answering questions of the learning algorithm. In case of adaptive learning algorithms, it also provides information about its ability to automatically explore the design space, thus enabling the optimization of the number of questions the user has to answer.

The current implementation provides three commands: *OPTIMISTIC*, *PESSIMISTIC* and *RESET*. In the current implementation, *RESET* command is only given, when a partial model (or the corresponding requirement) is removed from the oracle. The current heuristics for the individual partial model types are summarized in Table 4.1, with the oracle issuing the *OPTIMISTIC* command if and only if at least one partial model returns *OPTIMISTIC*, otherwise it returns *PESSIMISTIC*.

Partial Model Type	Applied Heuristics
I/O Pair	Always *PESSIMISTIC*, as per its definition it cannot contain more information than the current answer.
Valid Trace	*OPTIMISTIC* whenever the input in question is is applicable and the trace is longer than the given input.
Invalid Trace	Always *PESSIMISTIC*, as this kind of model (in general) carries no exact information
LTL Expression (Büchi automaton)	Always PESSIMISTIC, as the information content of Büchi-automata is a non-trivial question.

Table 4.1. The current heuristics of the individual partial models for the *OPTIMISTIC* and *PESSIMISTIC* adaption commands

The **LTL parser** is a simple parser implemented using the Xtext framework and an Ecore metamodel. It is responsible for the introduction of the event semantics into the provided expressions, as described in Subsection 3.1.2. It is also able to serialize these expressions to different formats using Xtend-based M2T transformations for reasons such as possible

feedback to the user – especially operator precedence clarification – and operator reordering – for compatibility with other LTL variants, especially that of Owl [23].

4.2.2 The Learning Algorithm

The **Adaptive Learning Algorithm** is an extension of an active automata learning algorithm with the presented adaption commands. In our implementation, we extended upon the framework's DHC algorithm (working analogously to the one presented in Chapter 2), and created the algorithm seen in Algorithm 2.

The **Teacher** and **Adapter** components were only extended so they can handle not only outputs and counterexamples, but adaption commands as well, allowing the formalism-agnostic interchange of information between the Oracle and the Learning Algorithm.

The implemented *Adaptive Learning Algorithm* depends on LearnLib[19] to decompose the counter examples provided by the *Teacher* component utilizing the approach proposed by Rivest and Schapire in [29].

The current implementation of the *Adaptive Learning Algorithm* infers a Mealy machine as the hypothesized model, utilizing an Ecore metamodel of Mealy machines shown in Figure B.1 and the Xtext grammar shown in in Listing B.1 in the Appendix.

4.2.3 Caching

The **Memoizing Learnable** is such a *Learnable*, that wraps around another *Learnable* and memoizes its behavior. The caching is done via a radix tree, an example of which is shown on Figure 4.3. The *Memoizing Learnable* monitors the adaption command given by the *Interactive Learnable*, and on $RESET$ deletes the current cache, re-iterates every query to the oracle – except for the one causing the $RESET$ – re-building the radix tree so no conflicting information remains. It passes every command it receives through the *Adapter* (and the *Teacher*) component to the *Learning Algorithm*.

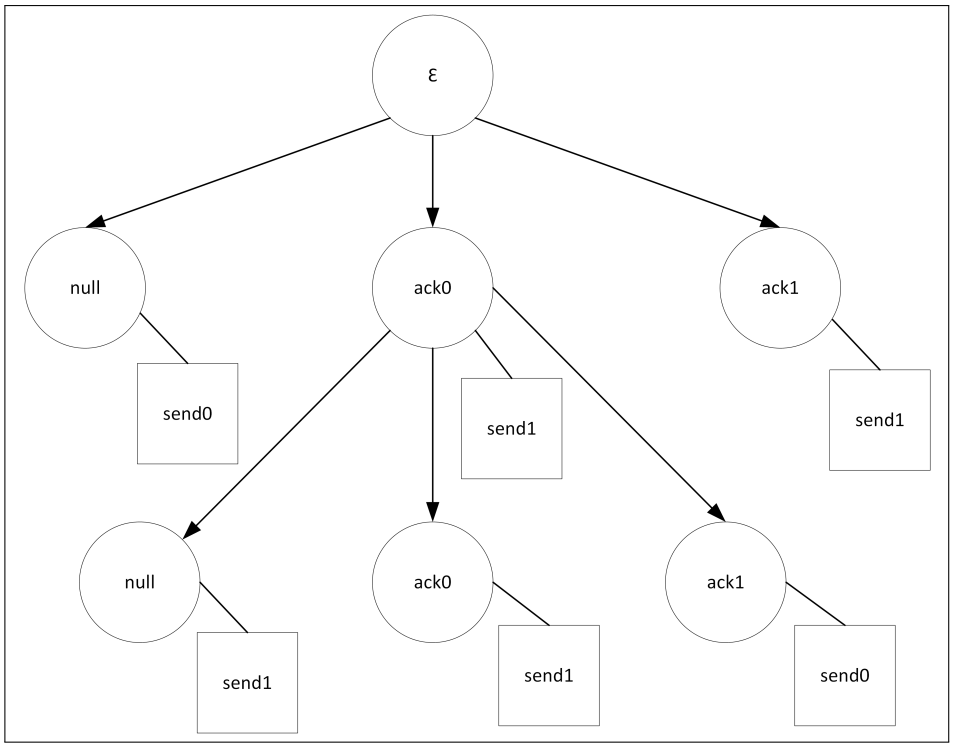


Figure 4.3. Radix tree containing the input/output sequences of the Mealy machine shown in Figure B.2 in the Appendix.

Chapter 5

Case Study: Pedestrian Crossing

This section demonstrates the capabilities and limitations of the framework. It presents a problem commonly modeled using state-based models, which is complex enough to demonstrate all aspects of the designed Interactive Learning Entity, but also simple enough to solve - thus verify - only using some background knowledge and common sense. The case study was inspired by the example seen in [27].

5.1 Introduction

The problem to solve is modeling a pedestrian crossing with a standard traffic light and a pedestrian light as illustrated on Figre 5.1. As the traffic lights and the pedestrian lights on the opposite sides of the crossing behave identically, we are going to model only one instance of each device.

The traffic light is looping through the red-green-yellow-red sequence. As an extra, there is an interrupted mode that may be triggered by the police, which results in blinking yellow light. The pedestrian light loops through the red-green-red sequence, and turns black when an interrupt arrives. A subsequent interrupt turns the lights back on, also considering that the sytem must always be in a safe state - i.e. the lights must not allow passage for both the pedestrians and the road vehicles at the same time.

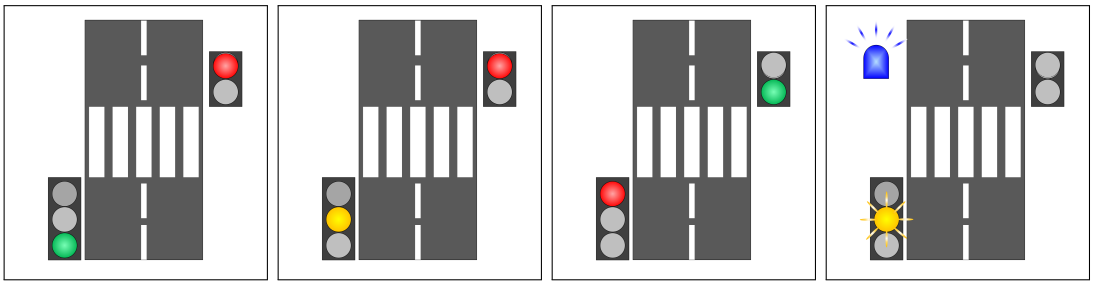


Figure 5.1. Possible states of the system: normal operation (*three from the left*) and the interrupted state *(right)*

5.2 Component Design

The previous subsection mentioned two components of the composite system: a traffic light and a pedestrian light. To realize the safe state of the system, the components must synchronize their behavior, justificating the existence of a third, controller component. The traffic and pedestrian light components should have one input and one output port – they are relatively simple – and the controller should have an input port for the police and two output ports for the components.

The traffic light component has two possible inputs on its input port (TrafficControl) - toggle and interrupt - and four outputs on its output port (TrafficDisplay) - red, green, yellow and blinking yellow - as it appeared in the problem description.

The pedestrian light component has the same two inputs on its input port (PedestrianControl) - toggle and interrupt - and three outputs on its output port (PedestrianDisplay) - red, green, and black - as it appeared in the problem description.

The controller component controls the rhythm of the change of states and also interrupts the other components when the police interrupt arrives. It has an input port ('*Police*') for the police interrupt and two output ports ('*TrafficControl*' and '*PedestrianControl*') - matching the input ports of the other components.

The described components and their connections are illustrated on Figure 5.2.

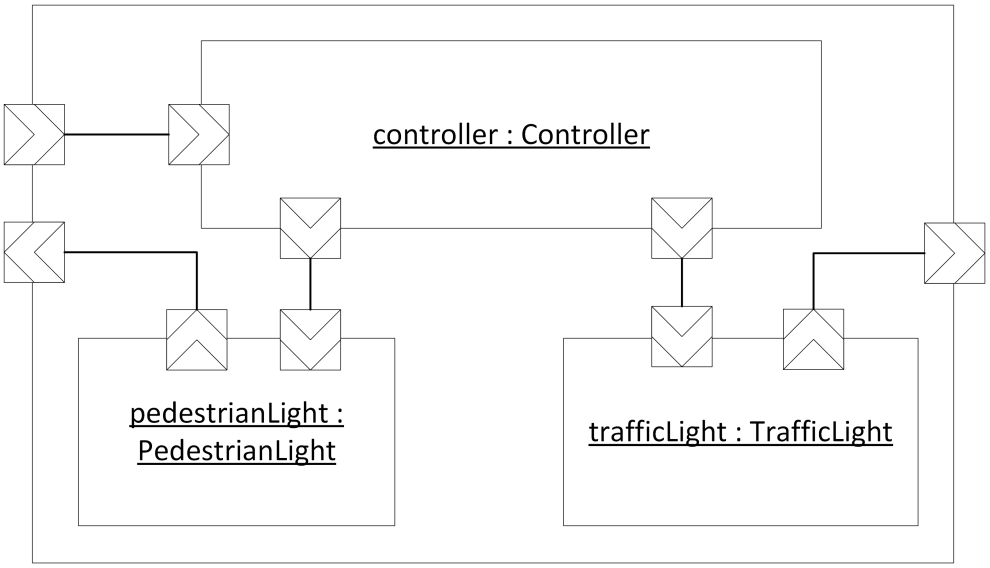


Figure 5.2. Components of the modeled system and their connections

The Expected Behavior of the Components

The components have been separated and their interfaces precisely defined, thus, we can proceed to formulating behavior-related requirements. For instance, the traffic light component must conform to the following – not exhaustive – list:

- The traffic light must loop through the sequence 'toggle/red toggle/green toggle/yellow toggle/red' during normal operation.
- The traffic light must display blinking yellow signal when a police interrupt arrives during normal operation.
- The traffic light must return to normal operation when a second interrupt arrives.

- The traffic light must display red signal when returning to normal operation.
- The traffic light must always display blinking yellow when interrupted.

A very similar list can be constructed for the pedestrian light component. The controller component is slightly more complicated, as it may toggle or interrupt the other components at the same time, so its alphabet shall contain separate elements for these cases.

5.3 Synthesizing the Components

Now we add the previously formulated requirements to the ILE, in formalisms that it is able to interpret. In the following examples for the interaction of the user with the ILE, '∘' symbolizes the ILE and '▷' symbolizes the user (interface qualifications are omitted for shorter and simpler expressions).

∘ Provide the requirements for component 'TrafficLight':

▷ Valid Trace: toggle/red toggle/green toggle/yellow toggle/red

▷ LTL Expression: F(interrupt -> X(G(toggle) -> G(blinkingYellow)))

▷ Invalid Trace: interrupt/red interrupt/blinkingYellow

▷ LTL Expression: F(G(interrupt -> (blinkingYellow | red)))

... other components ...

After adding these requirements during the offline phase, the synthesis of the components can proceed to the online, interactive phase. A possible run of the learning process can be seen below.

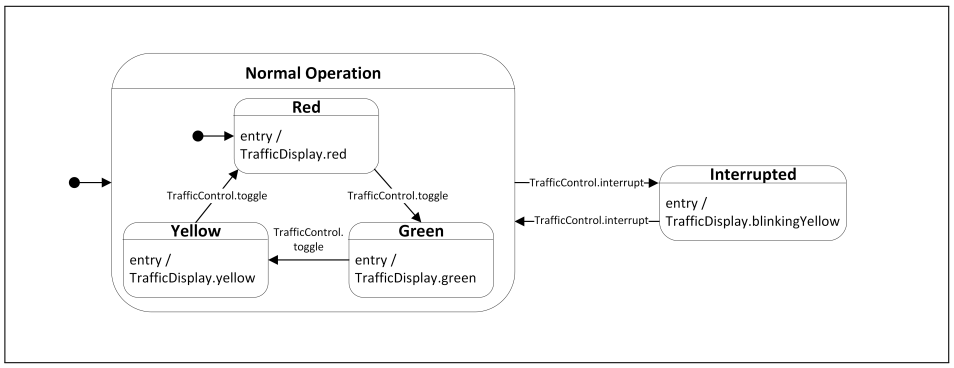
- Learning component 'TrafficLight'
- Provide the output for sequence [toggle, interrupt]:
  - Corresponding Output: blinkingYellow
- Provide the output for sequence [toggle, interrupt, interrupt]:
  - Corresponding Output: red
- Provide the output for sequence [toggle, toggle, interrupt]:
  - Corresponding Output: blinkingYellow
- Provide the output for sequence [toggle, toggle, interrupt, interrupt]:
  - Corresponding Output: red
- Provide the output for sequence [toggle, toggle, toggle, interrupt]:
  - Corresponding Output: blinkingYellow
- Equivalence Query (Figure C.1 in the Appendix)
  - Counterexample: interrupt interrupt
- Provide the output for sequence [interrupt, interrupt, interrupt]:
  - Valid Trace: interrupt/blinkingYellow interrupt/red interrupt/blinkingYellow interrupt/red
- Provide the output for sequence [interrupt, interrupt, toggle]:
  - Corresponding Output: green
- Provide the output for sequence [interrupt, toggle, interrupt]:
  - Corresponding Output: red
- Equivalence Query (Figure 5.3b)
  - Approved.

... other components ...

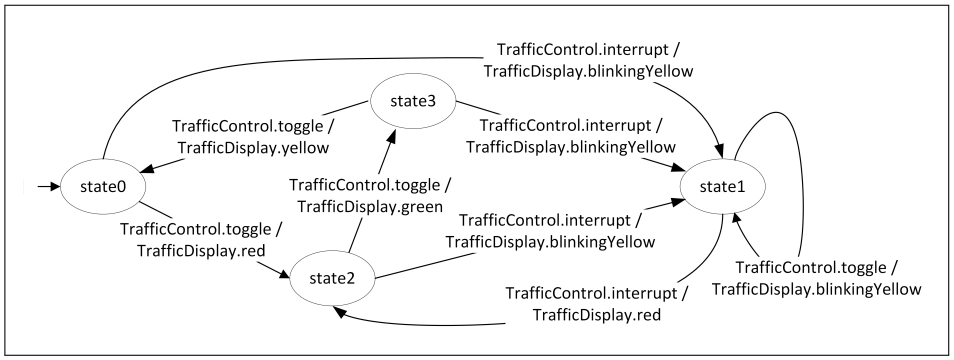
The same learning process can be seen on Listing C.1 in the Appendix, which presents the inputs and outputs as they appeara on the command line interface.

5.4 The Learned Models

Now we can examine any differences between the expected and the learned models. Figure 5.3 presents the expected and the synthesized traffic light models.



(a) Expected



(b) Learned

Figure 5.3. The expected and the learned traffic light models

The structure of the models is clearly different, notably:

- The expected models contain hierarchical states, while the learned ones are 'flat'.
- The expected models also contain entry and exit actions, while the learned models only contain actions on transitions.
- The expected models contain meaningful state names, while the learned ones only have generated ones.

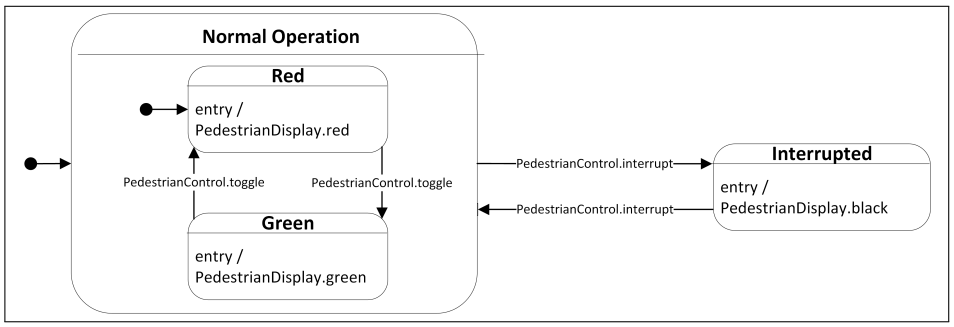
However, the models are behaviorally equivalent. Both models meet the requirements stated in Subsection 5.2, and after careful examination, it is obvious that the only initial states are different – as no entry actions are used in the learned models – and the transition starting from the hierarchical state is separated into three different transitions.

The differences between the expected and learned pedestrian light components can be seen on Figure 5.4. The differences are the similar to those of the traffic light models, as the modeled behavior is also very similar.

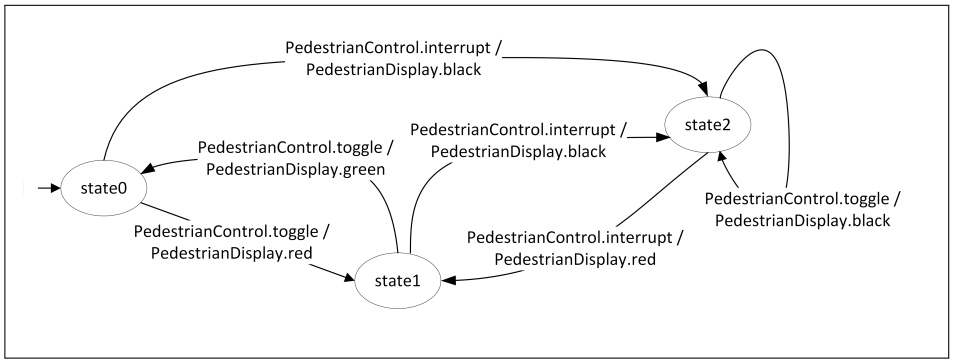
The behavior of the controller component can be learned similarly. The expected and the synthesized models can be seen on Figure 5.5. Note, that in addition to '`Police.interrupt`', the component also has an unqualified t input word. This represents a timeout event the engineer must extend the serialized model with – but acts as a regular input event during the learning.

After learning every component, the collection of the models is serialized as a Gamma project, with the following files and contents:

- TrafficLight.gcd (the learned traffic light component)



(a) Expected

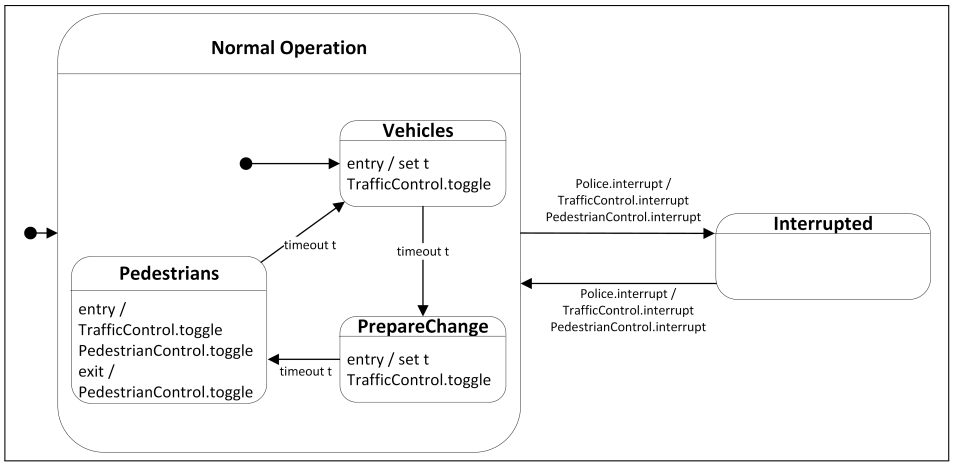


(b) Learned

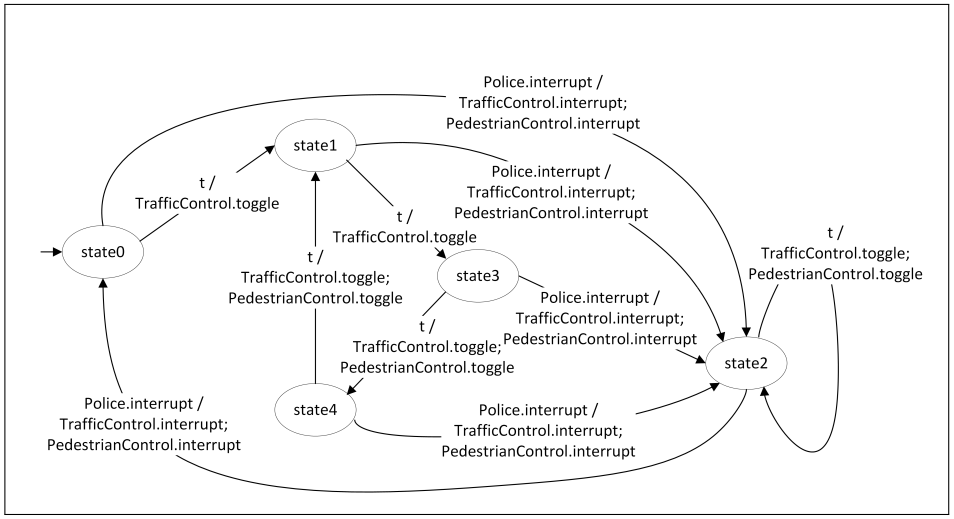
Figure 5.4. The expected and the learned pedestrian light models

- PedestrianLight.gcd (the learned pedestrian light component)
- Controller.gcd (the learned controller component)
- CompositeSystem.gcd (connections between the component ports, as illustrated on Figure 5.2)
- Interfaces.gcd (the interface definitions based on the ports of the components)

After extending the controller with statechart-specific elements (namely the timeout event), the engineer can use the Gamma Framework[26] to check the correctness of the system model or even generate implementation code.



(a) Expected



(b) Learned

Figure 5.5. The expected and the learned controller models

Chapter 6

Evaluation

This chapter presents the applicability of the designed framework, evaluates its current capabilities and points out improvement possibilities. The main evaluation metric is the number of queries each component makes - and how many of these reaches the user.

6.1 The Oracle

The main metric of the oracle is the number of behavior-related questions the engineer has to answer during the course of the model synthesis (including the offline phase). This is really difficult to measure, as the exact number of these questions depends on several parameters: the complexity of the desired model, as well as the order, formalism, complexity and skillful construction of the requirements formulated by the designing engineer. Some of these parameters are difficult to measure by themselves, thus, the following comparison of the supported formalisms is rather an illustration of the current capabilities of the framework through a realistic example.

For this demonstration, the traffic light component from the case study in Chapter 5 is going to be used. We assume, that the user only adds requirements he perceives conducive to the model synthesis and tries to formulate realistic – not unnecessarily complex – requirements. We also assume, that the requirements the user provides cannot be conflicting.

The baseline of this experiment is the number of questions the user has to answer by always providing the corresponding outputs to the questions of the ILE, as higher numbers are the result of redundant requirements. We examine the cases when valid traces are also allowed (and at least one must be used), then add (and require the use of) LTL expressions and finally invalid traces. Sequence diagrams are excluded from this comparison. The results can be seen on Figure 6.1.

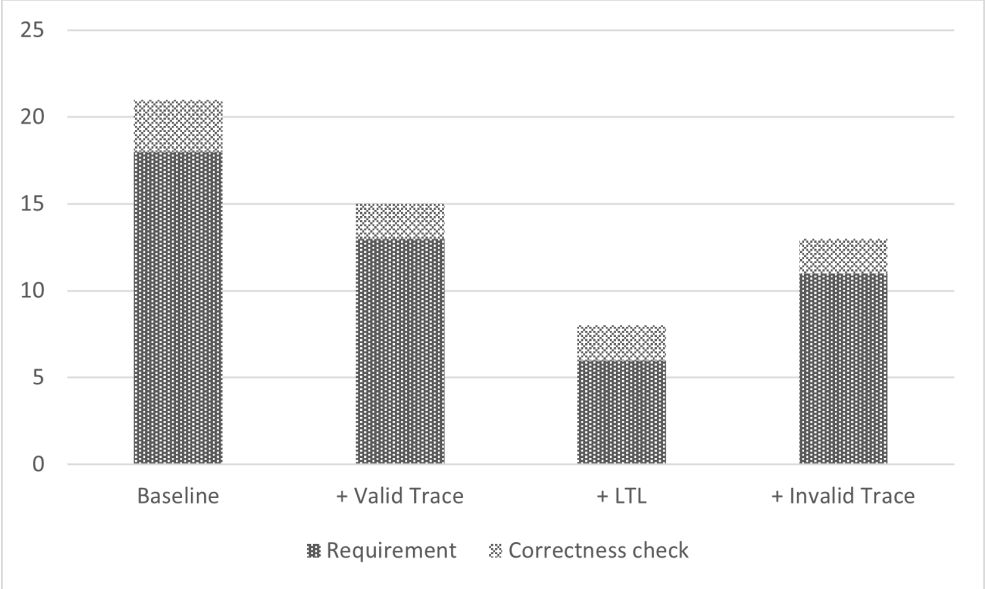


Figure 6.1. Number of required interactions for modelling the traffic light component gradually extending the set of allowed formalisms

The smallest number of queries was measured when both valid traces and LTL expressions were used. The addition of invalid traces resulted in a slightly higher number, demonstrating that this formalism is difficult to apply and useful mostly for its other benefits.

6.2 The Learning Algorithm

The runtime of the *Adaptive Learning Algorithm* depends on the adaption commands provided.

The Direct Hypothesis Construction algorithm, which, as theorized and proved in [30] and [25], terminates after at most $n^3mk + n^2k^2$ membership, and n equivalence queries, where n is the number of states of the canonical acceptor of the system under learning, k is the size of the input alphabet and m is the longest counterexample. The utilized suffix handling proposed by Rivest and Schapir[29], adds only one suffix to the set of suffixes in each learning round, so $m = 1$[30]. As opposed to the traditional DHC, the Adaptive DHC can have more than n learning rounds (but not more than n equivalence queries) as a result of the *RESET* command. Let r be the total number of reset commands the algorithm receives. As seen in [30], the set of suffixes is bound by $k + mn$, or in this case, $k+1*(n+r)$. Since the number of transitions needed to consider never exceed nk[30], and the maximum number of equivalence queries is n, the maximum number of membership queries of the *Adaptive Learning Algorithm* is $(k + n + r) * nk * n = n^2k^2 + n^3k + n^2kr$.

When *PESSIMISTIC* commands are used, the number of equivalence queries get closer to n, but the number of *new* membership queries only increase by k. Conversely, *OPTIMISTIC* commands reduce the number of equivalence queries needed, but increase the membership queries in a single learning round.

6.3 Caching

The implemented memoization prevents the redundant queries – seen in Section 6.2 – reach the oracle by caching each new query in a radix tree. If the cache contains the outputs received for c number of input sequences, the cache reset induced by the $RESET$ command creates $c - 1$ queries to the oracle (one for every sequence except for the one that induced the $RESET$).

Chapter 7

Conclusion

This chapter provides concluding remarks and possibilities for further improvement.

7.1 Contribution

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

- We designed a new, semi-automated methodology to support system design.
  - We defined a multi-phase workflow to implement this methodology.
  - We defined formalisms for declarative requirement types: corresponding outputs, valid and invalid traces, sequence diagrams and LTL expressions.
  - We enabled refinement-based requirement specification by introducing conflict handling among conflicting requirements.
  - We proposed a solution to the infeasibility of equivalence validation.
- We designed an architecture to support the proposed methodology.
  - We created the approach of adaptive learning using heuristics to determine inferable information.
  - We reconciled different modeling formalisms using an automata theory based approach.
  - We designed an adaptive variant of the Direct Hypothesis Construction algorithm.
  - We introduced caching to tackle the ineffectiveness of automatized information extraction.
- We created a proof of concept implementation of the proposed architecture in order to validate our approach.
- We demonstrated the capabilities and limitations of the implementation and the approach through a case study.
- We evaluated the components of the implementation.

7.2 Future Work

The possible future work opportunities are discussed in the following.

The proposed interactive learning approach and its proof of concept implementation requires further analysis in practical model-driven applications. To support further analysis a graphical user interface could also be implemented to enable the convenient design of systems and system components.

Additional features can be introduced to the model synthesis, such as onboarding further requirement formalisms to the framework, such as CTL expressions and invalid corresponding outputs for extended flexibility in design, introducing high-level statechart elements, such as timeout events, hierarchical states etc. into the learning, allowing the specification of initial models and patterns to guide the result of the model synthesis and adding priorities and scopes to requirements to facilitate refinement-based modeling.

New model synthesis approaches can be integrated through introducing extensions to the LTL formalism in order to support model quality optimization as seen in [4].

To optimize the learning and to evaluate different approaches, different automata learning algorithms, such as L*[5] and TTT[18] could be re-designed to support adaption.

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] ISO/IEC 14977:1996 Information Technology - Syntactic Metalanguage - Extended BNF, 1996.

[3] 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.

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

[5] 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.

[6] Dana Angluin, Timos Antonopoulos, and Dana Fisman. Strongly unambiguous büchi automata are polynomially predictable with membership queries. In *28th EACSL Annual Conference on Computer Science Logic (CSL 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[7] 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.

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

[9] Majzik István Bartha Tamás. *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.

[10] Jamieson M Cobleigh, Dimitra Giannakopoulou, and Corina S Păsăreanu. Learning assumptions for compositional verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 331–346. Springer, 2003.

[11] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, volume 9938 of

Lecture Notes in Computer Science, pages 122–129. Springer, October 2016. DOI: 10.1007/978-3-319-46520-3_8.

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

[13] Dimitra Giannakopoulou, Zvonimir Rakamarić, and Vishwanath Raman. Symbolic learning of component interfaces. In *International Static Analysis Symposium*, pages 248–264. Springer, 2012.

[14] Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive model checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 357–370. Springer, 2002.

[15] Object Management Group. Object Constraint Language – Version 2.4. Technical report, Object Management Group, 2014.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] Nafiseh Kahani, Mojtaba Bagherzadeh, and James R. Cordy. Synthesis of state machine models. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '20, page 274284, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370196. DOI: 10.1145/3365438.3410936. URL https://doi.org/10.1145/3365438.3410936.

[21] A.G. Kleppe, J. Warmer, J.B. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture : Practice and Promise*. Object technology. Addison-Wesley, 2003. ISBN 9780321194428. URL https://books.google.hu/books?id=nC6oS5xQGukC.

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

[27] 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 *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 113–116. IEEE, 2018.

[28] 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.

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

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

Appendix A

LTL Expressions

A.1 The Syntax of the LTL Expressions

```
LTLExpression = ArrowExpression;
ArrowExpression = (OrExpression '->' ArrowExpression) |
                  (OrExpression '<->' ArrowExpression) |
                   OrExpression;
OrExpression = (OrExpression '|' AndExpression) |
                AndExpression;
AndExpression = (AndExpression '&' UntilExpression) |
                 UntilExpression;
UntilExpression = (FutureGloballyExpression 'U' UntilExpression) |
                   FutureGloballyExpression;
FutureGloballyExpression = ('F' NextExpression) |
                           ('G' NextExpression) |
                            NextExpression;
NextExpression = ('X' PrimaryExpression) |
                 PrimaryExpression;
PrimaryExpression = ('(' LTLExpression ')') |
                    ('!' PrimaryExpression) |
                     LiteralExpression;
LiteralExpression = AtomicProposition |
                    'true' |
                    'false';
AtomicProposition = '^'?('a-z'|'A-Z'|'_') ('a-z'|'A-Z'|'_'|'.'|'0-9')*;
```

Listing A.1. Full syntax of the LTL expressions using the EBNF notation [2]

Appendix B

Implementation Details

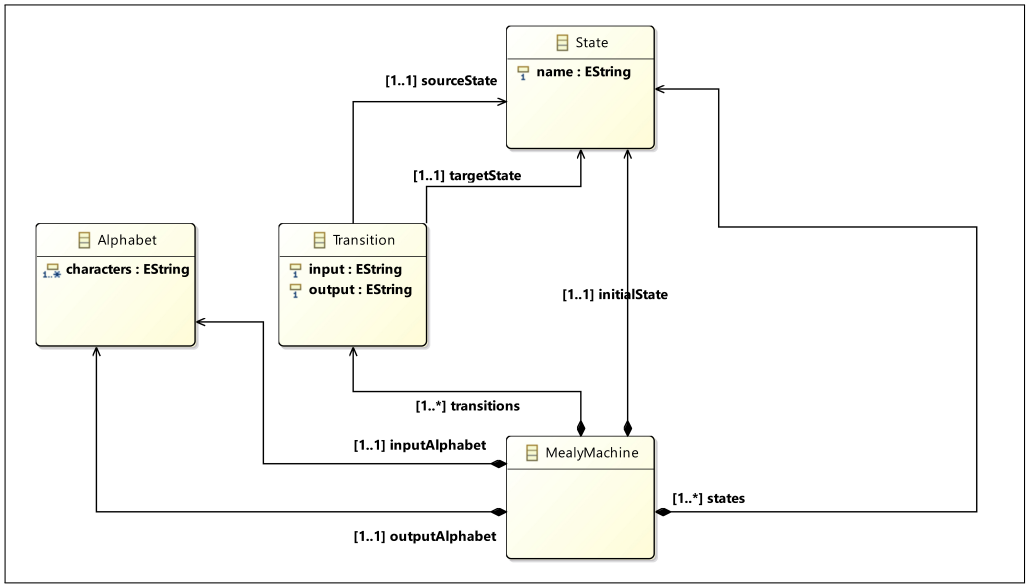


Figure B.1. Ecore metamodel describing Mealy machines

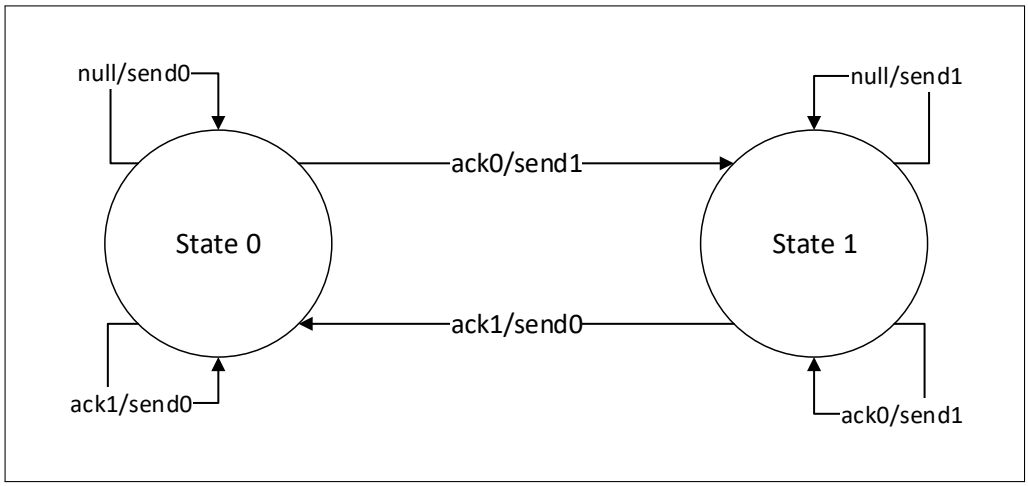


Figure B.2. Mealy machine describing the behavior of the alternating bit protocol

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

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

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

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

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

Listing B.1. Xtext grammar describing Mealy machines.

Appendix C

Pedestrian Crossing

```
Unknown output for input sequence: [control_toggle, control_interrupt]
Ambiguous output: [trafficDisplay_blinkingYellow, trafficDisplay_red]
Would you like to specify the output through an (I)O pair, an (L)TL expression, a (V)alid Trace or
   an I(N)valid Trace?
>I
Please provide the expected output:
>TrafficDisplay.blinkingYellow

Unknown output for input sequence: [control_toggle, control_interrupt, control_interrupt]
Ambiguous output: [trafficDisplay_blinkingYellow, trafficDisplay_red]
Would you like to specify the output through an (I)O pair, an (L)TL expression, a (V)alid Trace or
   an I(N)valid Trace?
>I
Please provide the expected output:
>TrafficDisplay.red

Unknown output for input sequence: [control_toggle, control_toggle, control_interrupt]
Ambiguous output: [trafficDisplay_blinkingYellow, trafficDisplay_red]
Would you like to specify the output through an (I)O pair, an (L)TL expression, a (V)alid Trace or
   an I(N)valid Trace?
>I
Please provide the expected output:
>TrafficDisplay.blinkingYellow

Unknown output for input sequence: [control_toggle, control_toggle, control_interrupt, control_
   interrupt]
Ambiguous output: [trafficDisplay_blinkingYellow, trafficDisplay_red]
Would you like to specify the output through an (I)O pair, an (L)TL expression, a (V)alid Trace or
   an I(N)valid Trace?
>I
Please provide the expected output:
>TrafficDisplay.red

Unknown output for input sequence: [control_toggle, control_toggle, control_toggle, control_
   interrupt]
Ambiguous output: [trafficDisplay_blinkingYellow, trafficDisplay_red]
Would you like to specify the output through an (I)O pair, an (L)TL expression, a (V)alid Trace or
   an I(N)valid Trace?
>I
Please provide the expected output:
>TrafficDisplay.blinkingYellow

Equivalence Query. Please provide a counterexamle:
Control.interrupt Control.interrupt

Unknown output for input sequence: [control_interrupt, control_interrupt, control_interrupt]
Ambiguous output: [trafficDisplay_blinkingYellow, trafficDisplay_red]
Would you like to specify the output through an (I)O pair, an (L)TL expression, a (V)alid Trace or
   an I(N)valid Trace?
>V
Please provide a valid trace:
```

```
>Control.interrupt/TrafficDisplay.blinkingYellow Control.interrupt/TrafficDisplay.red Control.
   interrupt/TrafficDisplay.blinkingYellow Control.interrupt/TrafficDisplay.red

Unknown output for input sequence: [control_interrupt, control_interrupt, control_toggle]
Ambiguous output: [trafficDisplay_blinkingYellow, trafficDisplay_red, trafficDisplay_yellow,
   trafficDisplay_green]
Would you like to specify the output through an (I)O pair, an (L)TL expression, a (V)alid Trace or
   an I(N)valid Trace?
>I
Please provide the expected output:
>TrafficDisplay.green

Unknown output for input sequence: [control_interrupt, control_toggle, control_interrupt]
Ambiguous output: [trafficDisplay_blinkingYellow, trafficDisplay_red]
Would you like to specify the output through an (I)O pair, an (L)TL expression, a (V)alid Trace or
   an I(N)valid Trace?
>I
Please provide the expected output:
>TrafficDisplay.red

Equivalence Query. Please provide a counterexamle:
>
```

Listing C.1. A possible run of the learning process

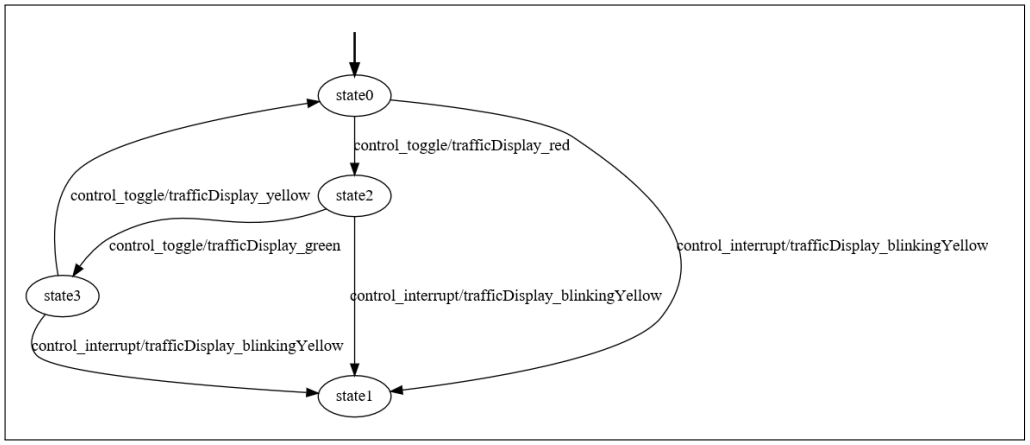


Figure C.1. Equivalence query of the incomplete traffic light component