Learning meets Verification

Martin Leucker

Institut für Informatik
TU München, Germany

Abstract. In this paper, we give an overview on some algorithms for learning automata. Starting with Biermann's and Angluin's algorithms, we describe some of the extensions catering for specialized or richer classes of automata. Furthermore, we survey their recent application to verification problems.

1 Introduction

Recently, several verification problems have been addressed by using learning techniques. Given a system to verify, typically its essential part is learned and represented as a regular system, on which the final verification is then carried out. The aim of this paper is to present these recent developments in a coherent fashion. It should serve as an annotated list of references as well as describe the main ideas of these approaches rather than to pin down every technical detail, as these can be found in the original literature.

From the wide spectrum of learning techniques, we focus here on learning automata, or, as it is sometimes called, on *inference* of automata. We then exemplify how these learning techniques yield new verification approaches, as has recently been documented in the literature. Note, by verification we restrict to model checking [21] and testing [16] techniques.

This paper consists of two more sections: In the next section, we introduce learning techniques for automata while in Section 3, we list some of their applications in verification procedures.

In Section 2, we first recall Biermann's so-called *offline* approach and Angluin's *online* approach to learning automata. Then, we discuss variations of the setup for online learning as well as describe domain specific optimizations. Furthermore, we sketch extensions to Angluin's learning approach to so-called regular-representative systems, timed systems, and ω-regular languages. We conclude Section 2 by giving references to further extensions and implementations.

In Section 3, we show applications of learning techniques in the domain of verification. We start with the problem of minimizing automata, which follows the idea of learning a minimal automaton for a given system rather than to minimize the given system explicitly. *Black-box checking*, which renders black-box testing as learning and (white-box) model checking, is discussed next. We continue with the idea of learning assumptions in compositional verification. Next, we follow the idea that a (least) fixpoint of some functional might be learned

rather than computed iteratively. Implicitly, this idea has been followed when learning network invariants or learning representations for the set of reachable states in regular model checking [2]. We conclude Section 3 by referring to further applications.

2 Learning Algorithms for Regular Systems

The general goal of learning algorithms for *regular systems* is to identify a *machine*, usually of *minimal* size, that *conforms* with an *a priori fixed* set of strings or a (class of) machines.

Here, we take machines to be *deterministic finite automata* (DFAs) (over strings), though most approaches and results carry over directly to the setting of finite-state machines (Mealy-/Moore machines).

In general, two types of learning algorithms for DFAs can be distinguished, so-called *online* and *offline* algorithms. Offline algorithms get a fixed set of *positive* and *negative* examples, comprising strings that should be *accepted* and, respectively, strings that should be *rejected* by the automaton in question. The learning algorithm now has to provide a (minimal) automaton that accepts the positive examples and rejects the negative ones.

Gold [30] was among the first studying this problem. Typical offline algorithms are based on a characterization in terms of a constraint satisfaction problem (CSP) over the natural numbers due to Biermann [14].

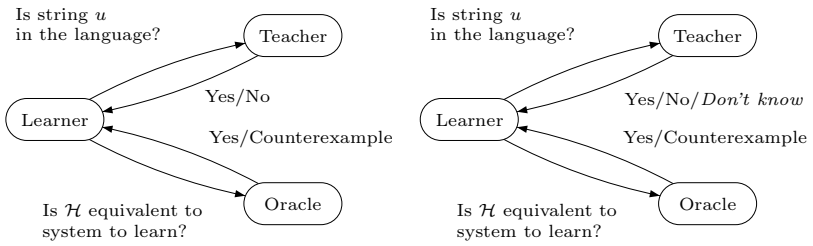
A different approach was proposed in [54], though imposing stronger assumptions (prefix-closeness) on the set of examples. We also note that there are efficient algorithms inferring a *not necessarily minimal* DFA, like [44] or [48] (known as RPNI).

Online algorithms have the possibility to ask further *queries*, whether some string is in the language of the automaton to learn or not. In this way, an online algorithm can enlarge the set of examples in a way most suitable for the algorithm and limiting the number of minimal automata in question, i.e., the ones conforming to the set of examples so far.

A popular setup for an online approach is that of Angluin's L^* algorithm [6] in which a minimal DFA is learned based on so-called *membership* and *equivalence queries*. Using a pictorial language, we have a *learner* whose job is to come up with the automaton to learn, a *teacher* who may answer whether a given string is in the language as well an *oracle* answering whether the automaton $\mathcal{H}$ currently proposed by the learner is correct or not. This setting is depicted in Figure 1(a).

Clearly, an online algorithm like Angluin's should perform better than offline algorithms like Biermann's. Indeed, Angluin's algorithm is polynomial while without the ability to ask further queries the problem of identifying a machine conforming to given examples is known to be NP-complete [31].

Note that there are slightly different approaches to query-based learning of regular languages based on *observation packs* or *discrimination trees*, which are compared to Angluin's approach in [8, 13].



(a) The setting of L^* (b) The setting of ABL*

Fig. 1. The setup for the learning algorithms

In Angluin's setting, a teacher will answer queries either positively or negatively. In many application scenarios, however, parts of the machine to learn are not completely specified or not observable. Then, queries may be answered inconclusively, by *don't know*, also denoted by ?. We term such a teacher *inexperienced*, see Figure 1(b).

In the following, we will look more closely at Biermann's approach (in Section 2.2), Angluin's algorithm (Section 2.3), as well as their combinations serving the setup with an inexperienced teacher (Section 2.4). Furthermore, we will sketch some extensions of these algorithms (Sections 2.5 – 2.9). Before, however, we look at the fundamental concept of right congruences that is the basis for the learning algorithms.

2.1 DFAs, right-congruences, and learning regular systems

Let $\mathbb{N}$ denote the natural numbers, and, for $n \in \mathbb{N}$, let $[n] := \{1, \ldots, n\}$. For the rest of this section, we fix an alphabet Σ. A *deterministic finite automaton* (DFA) $\mathcal{A} = (Q, q_0, \delta, Q^+)$ over Σ consists of a finite set of *states* Q, an *initial state* $q_0 \in Q$, a *transition function* $\delta : Q \times \Sigma \to Q$, and a set $Q^+ \subseteq Q$ of *accepting states*. A *run* of $\mathcal{A}$ is a sequence $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n$ such that $a_i \in \Sigma$, $q_i \in Q$ and $\delta(q_{i-1}, a_i) = q_i$ for all $i \in [n]$. It is called *accepting* iff $q_n \in Q^+$. The *language* accepted by $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is the set of strings $u \in \Sigma^*$ for which an accepting run exists. Since the automaton is deterministic, it is reasonable to call the states $Q \setminus Q^+$ also *rejecting states*, denoted by Q^-. We extend δ to strings as usual by $\delta(q, \epsilon) = q$ and $\delta(q, ua) = \delta((\delta, u), a)$. The size of $\mathcal{A}$, denoted by $|\mathcal{A}|$, is the number of its states Q, denoted by $|Q|$. A language is *regular* iff it is accepted by some DFA.

The basis for learning regular languages is given by their characterization in terms of *Nerode's right congruence* $\equiv_{\mathcal{L}}$: Let $\equiv_{\mathcal{L}}$ be defined by, for $u, v \in \Sigma^*$

$$u \equiv_{\mathcal{L}} v \text{ iff for all } w \in \Sigma^* : uw \in \mathcal{L} \Leftrightarrow vw \in \mathcal{L}.$$

It is folklore, that a language $\mathcal{L}$ is regular iff $\equiv_{\mathcal{L}}$ has finite index.

Intuitively, most learning algorithms estimate the equivalence classes for a language to learn. Typically, it assumes that all words considered so far are

equivalent, unless, a (perhaps empty) suffix shows that they cannot be equivalent.

Based on Nerode's right congruence, we get that, for every regular language $\mathcal{L}$, there is a *canonical* DFA $\mathcal{A}_\mathcal{L}$ that accepts $\mathcal{L}$ and has a minimal number of states: Let $\mathbf{u}_\mathcal{L}$ or shortly $\mathbf{u}$ denote the equivalence class of u wrt. $\equiv_\mathcal{L}$. Then the canonical automaton of $\mathcal{L}$ is $\mathcal{A}_\mathcal{L} = (Q_\mathcal{L}, q_{0\mathcal{L}}, \delta_\mathcal{L}, Q^+_\mathcal{L})$ defined by

- $Q_\mathcal{L} = \Sigma/\equiv_\mathcal{L}$ is the set of equivalence classes wrt. $\equiv_\mathcal{L}$,
- $q_{0\mathcal{L}} = \epsilon$,
- $\delta_\mathcal{L} : Q_\mathcal{L} \times \Sigma \to Q$ is defined by $\delta_\mathcal{L}(\mathbf{u}, a) = \mathbf{ua}$,
- $Q^+_\mathcal{L} = \{\mathbf{u} \mid u \in \mathcal{L}\}$

We omit the subscript $\mathcal{L}$ provided $\mathcal{L}$ is known from the context.

2.2 Biermann's algorithm

Biermann's learning algorithm [14] is an offline algorithm for learning a DFA $\mathcal{A}$. We are given a set of strings that are to be accepted by $\mathcal{A}$ and a set of strings that are to be rejected by $\mathcal{A}$. There is no possibility of asking queries and we have to supply a minimal DFA that accepts/rejects these strings. The set of positive and negative strings are called *sample*. We now formally describe samples and Biermann's algorithm.

A *sample* is a set of strings that, by the language in question, should either be accepted, denoted by $+$, or rejected, denoted by $-$. For technical reasons, it is convenient to work with prefix-closed samples. As the samples given to us are not necessarily prefix closed we introduce the value *maybe*, denoted by ?. Formally, a *sample* is a partial function $O : \Sigma^* \to \{+, -, ?\}$ with finite, prefix-closed domain $\mathcal{D}(O)$. That is, $O(u)$ is defined only for finitely many $u \in \Sigma^*$ and is defined for $u \in \Sigma^*$ whenever it is defined for some ua, for $a \in \Sigma$. For a string u the sample O yields whether u should be *accepted*, *rejected*, or we do not know (or do not care). For strings u and u', we say that O *disagrees* on u and u' if $O(u) \neq ?$, $O(u') \neq ?$, and $O(u) \neq O(u')$.

An automaton $\mathcal{A}$ is said to *conform* with a sample O, if whenever O is defined for u we have $O(u) = +$ implies $u \in \mathcal{L}(\mathcal{A})$ and $O(u) = -$ implies $u \notin \mathcal{L}(\mathcal{A})$.

Given a sample O and a DFA $\mathcal{A}$ that conforms with O, let S_u denote the state reached in $\mathcal{A}$ when reading u. As long as we do not have $\mathcal{A}$, we can treat S_u as a variable ranging over states and derive constraints for the assignments of such a variable. More precisely, let $\mathrm{CSP}(O)$ denote the set of equations

$$\begin{aligned} &\{S_u \neq S_{u'} && \mid O \text{ disagrees on } u \text{ and } u'\} && \text{(C1)}\\ \cup\, &\{S_u = S_{u'} \Rightarrow S_{ua} = S_{u'a} && \mid a \in \Sigma, ua, u'a \in \mathcal{D}(O)\} && \text{(C2)} \end{aligned}$$

Clearly, (C1) and (C2) reflect properties of Nerode's right congruence: (C1) tests u and u' on the empty suffix and (C2) guarantees right-congruence. Let the domain of $\mathcal{D}(\mathrm{CSP}(O))$ comprise the set of variables S_u used in the constraints.

An *assignment* of $\mathrm{CSP}(O)$ is mapping $\Gamma : \mathcal{D}(\mathrm{CSP}(O)) \to \mathbb{N}$. An assignment Γ is called a *solution* of $\mathrm{CSP}(O)$ if it fulfils the equations over the naturals,

defined in the usual manner. The set $\mathrm{CSP}(O)$ is *solvable* over $[N]$ iff there is a solution with range $[N]$. It is easy to see that every solution of the CSP problem over the natural numbers can be turned into an automaton conforming with O. We sum-up:

Lemma 1 (Learning as CSP, [14]). *For a sample O, a DFA with N states conforming to O exists iff* $\mathrm{CSP}(O)$ *is solvable over* $[N]$.

Proof. Let $\mathcal{A} = (Q, q_0, \delta, Q^+)$ be a DFA conforming with O having N states. Without loss of generality, assume that $Q = [N]$. It is easy to see that assigning the value $\delta(q_0, u)$ to each $S_u \in \mathcal{D}(\mathrm{CSP}(O))$ solves $\mathrm{CSP}(O)$.

On the other hand, given a solution Γ of $\mathrm{CSP}(O)$ with range $[N]$, define $\mathcal{A} = (Q, q_0, \delta, Q^+)$ by

- $Q = [N]$,
- $q_0 = S_\epsilon$,
- $\delta : Q \times \Sigma \to Q$ is any function satisfying $\delta(n, a) = n'$, if there is $S_u, S_{ua} \in \mathcal{D}(\mathrm{CSP}(O))$ with $S_u = n$, $S_{ua} = n'$. This is well-defined because of (C2).
- $Q^+ \subseteq Q$ is any set satisfying, for $S_u \in \mathcal{D}(\mathrm{CSP}(O))$ with $S_u = n$, $O(u) = +$ implies $n \in Q^+$, $O(u) = -$ implies $n \notin Q^+$. This is well-defined because of (C1).

Let us give three simple yet important remarks:

- Taking a different value for every S_u, trivially solves the CSP problem. Thus, a solution of $\mathrm{CSP}(O)$ within range $[|\mathcal{D}(\mathrm{CSP}(O))|]$ exists.
- Then, a solution with *minimum range* exists and yields a DFA with a minimal number of states.
- From the above proof, we see that we typically cannot expect to get a *unique* minimal automaton conforming with O—in contrast to Angluin's L^* algorithm, as we will see.

Lemma 1 together with the remarks above gives a simple non-deterministic algorithm computing in polynomial time a DFA for a given observation, measured with respect to $N := |\mathcal{D}(\mathrm{CSP}(O))|$, which is sketched Algorithm 1. Note that by results of Gold [31], the problem is NP complete and thus, the algorithm is optimal.

Algorithm 1 Pseudo code for Biermann's Learning Algorithm

```
% Input O
Guess n ∈ [N], where N := |D(CSP(O))|
Guess assignment Γ of variables in D(CSP(O))
Verify that Γ satisfies (C1) and (C2)
Construct the DFA as described in Lemma 1
```

Pruning the search space of the CSP problem While the previous algorithm is of optimal worst case complexity, let us make a simple yet important observation to simplify the CSP problem, which often pays off in practice [35]. We call a bijection $\iota : [N] \to [N]$ a *renaming* and say that assignments Γ and Γ' are *equivalent modulo* renaming iff there is a renaming ι such that $\Gamma = \iota \circ \Gamma'$.

Since names (here, numbers) of states have no influence on the accepted language of an automaton, we get

Lemma 2 (Name irrelevance). *For a sample O, $\Gamma : \mathcal{D}(\mathrm{CSP}(O)) \to [N]$ is a solution for $\mathrm{CSP}(O)$ iff for every renaming $\iota : [N] \to [N]$, $\iota \circ \Gamma$ is a solution of $\mathrm{CSP}(O)$.*

The previous lemma can be used to prune the search space for a solution: We can assign numbers to state variables, provided different numbers are used for different states.

Definition 1 (Obviously different). *S_u and $S_{u'}$ are said to be* obviously different *iff there is some $v \in \Sigma^*$ such that O disagrees on uv and $u'v$. Otherwise, we say that S_u and $S_{u'}$* look similar.

A CSP problem with M obviously different variables needs at least M different states, which gives us together with Lemma 1:

Lemma 3 (Lower bound). *Let M be the number of obviously different variables. Then $\mathrm{CSP}(O)$ is not solvable over all $[N]$ with $N < M$.*

Note that solvability over $[M]$ is not guaranteed, as can easily be seen.

As a solution to the constraint system produces an automaton and in view of Lemma 2, we can fix the values of obviously different variables.

Lemma 4 (Fix different values). *Let $S_{u_1}, \ldots, S_{u_M}$ be M obviously different variables. Then $\mathrm{CSP}(O)$ is solvable iff $\mathrm{CSP}(O) \cup \{S_{u_i} = i \mid i \in [M]\}$ is solvable.*

The simple observation stated in the previous lemma improves the solution of a corresponding SAT problem defined below significantly, as described in [35].

Solving the CSP problem It remains to come up with a procedure solving the CSP problem presented above in an reasonable manner. An explicit solution is proposed in [47]. In [35], however, an efficient encoding as a SAT problem has been given, for which one can rely on powerful SAT solvers. Actually, two different encodings have been proposed: *binary* and *unary*.

Let n be the number of strings in $\mathcal{D}(O)$ and N be the size of the automaton in question. Then $\mathrm{CSP}(O)$ has $\mathcal{O}(n^2)$ constraints. Using the binary SAT encoding yields $\mathcal{O}(n^2 N \log N)$ clauses over $\mathcal{O}(n \log N)$ variables. Totally, the unary encoding has $\mathcal{O}(n^2 N^2)$ clauses with $\mathcal{O}(nN)$ variables (see [35] for details).

While the first is more compact for representing large numbers, it turns out that the unary encoding speeds-up solving the resulting SAT problem.

2.3 Angluin's algorithm

Angluin's learning algorithm, called L^* [6], is designed for learning a regular language, $\mathcal{L} \subseteq \Sigma^*$, by constructing a minimal DFA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}$. In this algorithm a *Learner*, who initially knows nothing about $\mathcal{L}$, is trying to learn $\mathcal{L}$ by asking a *Teacher* and an *Oracle*, who know $\mathcal{L}$, respectively two kinds of queries (cf. Figure 1(a)):

- A *membership query* consists of asking whether a string $w \in \Sigma^*$ is in $\mathcal{L}$.
- An *equivalence query* consists of asking whether a hypothesized DFA $\mathcal{H}$ is correct, i.e., whether $\mathcal{L}(\mathcal{H}) = \mathcal{L}$. The *Oracle* answers *yes* if $\mathcal{H}$ is correct, or else supplies a counterexample w, either in $\mathcal{L} \setminus \mathcal{L}(\mathcal{H})$ or in $\mathcal{L}(\mathcal{H}) \setminus \mathcal{L}$.

The *Learner* maintains a prefix-closed set $U \subseteq \Sigma^*$ of prefixes, which are candidates for identifying states, and a suffix-closed set $V \subseteq \Sigma^*$ of suffixes, which are used to distinguish such states. The sets U and V are increased when needed during the algorithm. The *Learner* makes membership queries for all words in $(U \cup U\Sigma)V$, and organizes the results into a *table* T that maps each $u \in (U \cup U\Sigma)$ to a mapping $T(u) : V \mapsto \{+, -\}$ where $+$ represents accepted and $-$ not accepted. In [6], each function $T(u)$ is called a *row*. When T is

- *closed*, meaning that for each $u \in U$, $a \in \Sigma$ there is a $u' \in U$ such that $T(ua) = T(u')$, and
- *consistent*, meaning that $T(u) = T(u')$ implies $T(ua) = T(u'a)$,

the *Learner* constructs a hypothesized DFA $\mathcal{H} = (Q, q_0, \delta, Q^+)$, where

(a) $Q = \{T(u) \mid u \in U\}$ is the set of distinct rows,
(b) q_0 is the row $T(\lambda)$,
(c) δ is defined by $\delta(T(u), a) = T(ua)$, and
(d) $Q^+ = \{T(u) \mid u \in U, T(u)(\lambda) = +\}$

and submits $\mathcal{H}$ as an equivalence query. If the answer is *yes*, the learning procedure is completed, otherwise the returned counterexample u is used to extend U by adding all prefixes of u to U, and subsequent membership queries are performed in order to make the new table closed and consistent producing a new hypothesized DFA, etc. The algorithm is sketched in Algorithm 2.

Complexity It can easily be seen that the number of membership queries can be bounded by $O(kn^2m)$, where n is the number of states of the automaton to learn, k is the size of the alphabet, and m is the length of the longest counterexample. The rough idea is that for each entry in the table T a query is needed, and $O(knm)$ is the number of rows, n the number of columns. The latter is because at most n equivalence queries suffice. To see this, check that for any closed and consistent T, there is a single and therefore unique DFA conforming with T (as opposed to Biermann's approach). Thus, equivalence queries are performed with automata of strictly increasing size.

Algorithm 2 Pseudo code for Angluin's Learning Algorithm

```
1  Function Angluin()
2    initialize (U,V,T)
3    repeat
4      while not(isClosed((U,V,T)) or not(isConsistent((U,V,T))
5        if not(isConsistent((U,V,T)) then
6          find a ∈ Γ, v ∈ V and u, u' ∈ U such that
7            T(u) = T(u') and T(ua)(v) ≠ T(u'a)(v)
8          add av to V
9          for every u ∈ U ∪ UΓ
10           ask membership query for uav
11       if not(isClosed((U,V,T)) then
12         find u ∈ U, a ∈ Γ such that T(ua) ≠ T(u') for all u' ∈ U
13         move ua to U
14         for every a' ∈ Γ and v ∈ V
15           ask membership query for u'aa'v
16     construct hypothesized automaton H
17     do an equivalence query with hypothesis H
18     if the answer is a counterexample u then
19       add every prefix u' of u to U.
20       for every a ∈ Γ, v ∈ V and prefix u' of u
21         ask membership query for u'v and u'av.
22   until the answer is 'yes' to the hypothesis H
23   Output H.
```

2.4 Learning from Inexperienced Teachers

In the setting of an *inexperienced teacher* (cf. Figure 1(b)), queries are no longer answered by either *yes* or *no*, but also by *maybe*, denoted by ?. We can easily come a up with a learning algorithm in this setting, relying on Biermann's approach: First, the learner proposes the automaton consisting of one state accepting every string.[1] Then, the Learner consults the oracle, which either classifies the automaton as the one we are looking for or returns a counter example c. In the latter case, c as well as its prefixes are added to the sample O: c with $+/-$ as returned by the oracle and the prefixes with ? (unless O is already defined on the prefix), and Biermann's procedure is called to compute a minimal DFA consistent with O. Again the oracle is consulted and either the procedure stops or proceeds as before by adding the new counter example and its prefixes as before.

However, the procedure sketched above requires to create a hypothesis and to consult the oracle for every single observation and does not make use of membership queries at all.

[1] Proposing the automaton that rejects every string is equally OK.

Often, membership queries are "cheaper" than equivalence queries. Then, it might be worthwhile to "round off" the observation by consulting the teacher, as in Angluin's algorithm.

We list the necessary changes to Angluin's algorithm [35] yielding algorithm ABL*. We keep the idea of a table but now, for every $u \in (U \cup U\Sigma)$, we get a mapping $T(u) : V \to \{+, -, ?\}$. For $u, u' \in (U \cup U\Sigma)$, we say that rows $T(u)$ and $T(v)$ *look similar*, denoted by $T(u) \equiv T(u')$, iff, for all $v \in V$, $T(u)(v) \neq ?$ and $T(u')(v) \neq ?$ implies $T(u)(v) = T(u')(v)$. Otherwise, we say that $T(u)$ and $T(v)$ are *obviously different*. We call T

- *weakly closed* if for each $u \in U$, $a \in \Sigma$ there is a $u' \in U$ such that $T(ua) \equiv T(u')$, and
- *weakly consistent* if $T(u) \equiv T(u')$ implies $T(ua) \equiv T(u'a)$.

Angluin's algorithm works as before, but using the weak notions of closed and consistent. However, extracting a DFA from a weakly closed and weakly consistent table is no longer straightforward. However, we can go back to Biermann's approach now.

Clearly, Angluin's table (including entries with ?) can easily be translated to a sample, possibly by adding prefixes to $(U \cup U\Sigma)V$ with value ? to obtain a prefix-closed domain.

Catering for the optimizations listed for Biermann's algorithm, given a table $T : (U \cup U\Sigma) \times V \to \{+, -, ?\}$, we can easily approximate obviously different states: For $u, u' \in (U \cup U\Sigma)$, states S_u and $S_{u'}$ are obviously different, if the rows $T(u)$ and $T(u')$ are obviously different.

Overall, in the setting of an inexperienced teacher, we use Biermann's approach to derive a hypothesis of the automaton in question, but use the completion of Angluin's observation table as a heuristic to round the sample by means of queries.

2.5 Domain specific optimizations for Angluin's algorithm

Angluin's L^* algorithm works in the setting of arbitrary regular languages. If the class of regular languages is restricted, so-called *domain specific* optimizations may be applied to optimize the learning algorithm [38]. For example, if the language to learn is known to be *prefix closed*, a positive string ua implies u to be positive as well, preventing to consult the teacher for u [11].

Pictorially, such a setting can easily be described by adding an *assistant* between the learner and the teacher. Queries are send to the assistant who only consults the teacher in case the information cannot be deduced from context information [13]. Assistants dealing with *independent actions*, *I/O systems*, and *symmetrical systems* have been proposed in [38].

2.6 Learning of regular representative systems

Angluin's L^* algorithm identifies a DFA accepting a regular language. Such a language, however, might *represent* a more complicated object. In [15], for exam-

ple, L^* is used to infer *message-passing automata* (MPAs) accepting languages of *message sequence charts* (MSCs).

Let us work out a setup that allows to learn systems using a simple modification of Angluin's L^* algorithm. A *representation system* is a triple $(\mathcal{E}, \mathcal{O}, \mathcal{L})$, where

- $\mathcal{E}$ is a set of *elements*,
- $\mathcal{O}$ is a set of *objects*,
- and $\mathcal{L} : \mathcal{O} \to 2^{\mathcal{E}}$ is a *language function* yielding for an object o the set of elements it represents.

The objects might be classified into equivalence classes of an equivalence relation $\sim_{\mathcal{L}} \subseteq \mathcal{O} \times \mathcal{O}$ by $\mathcal{L}$: $o \sim o'$ iff $\mathcal{L}(o) = \mathcal{L}(o')$. Intuitively, objects could be understood as subsets of $\mathcal{E}$. However, like with words and automata, subsets of languages could be infinite while automata are a finite representation of such an infinite set.

A further example is where objects are MPAs, elements are MSCs, and MPA represent MSC languages. Then, two MPAs are considered to be equivalent if they recognize the same MSC language.

Our goal is now to *represent* objects (or rather their equivalence classes) by regular word languages, say over an alphabet Σ, to be able to use L^*. An almost trivial case is given when there is a bijection from regular word languages to $\mathcal{O}$. As we will see, a simple framework can also be obtained, when there is bijection of $\mathcal{O}$ to a factor of a subset of regular languages.

Let $\mathcal{D}$ be a subset of Σ^*. The motivation is that only regular word languages containing at most words from $\mathcal{D}$ are considered and learned. Furthermore, let $\approx \subseteq \mathcal{D} \times \mathcal{D}$ be an equivalence relation. We say that $L \subseteq \mathcal{D}$ is $\approx$-*closed* (or, closed under $\approx$) if, for any $w, w' \in \mathcal{D}$ with $w \approx w'$, we have $w \in L$ iff $w' \in L$.

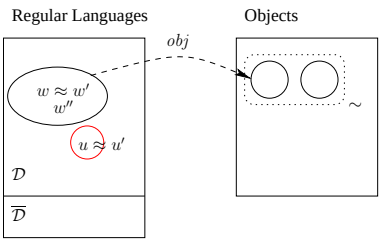


Fig. 2. Representing objects by regular languages

Naturally, $\mathcal{D}$ and $\approx$ determine the particular class $\mathfrak{R}_{\text{minDFA}}(\Sigma, \mathcal{D}, \approx) := \{L \subseteq \mathcal{D} \mid L \text{ is regular and closed under } \approx\}$ of regular word languages over Σ (where any language is understood to be given by its minimal DFA). Suppose a language of this class $\mathfrak{R}_{\text{minDFA}}(\Sigma, \mathcal{D}, \approx)$ can be learned in some sense that will be made precise. For learning elements of $\mathcal{O}$, we still need to derive an object from a language in $\mathfrak{R}_{\text{minDFA}}(\Sigma, \mathcal{D}, \approx)$. To this aim, we suppose a computable bijective mapping $obj : \mathfrak{R}_{\text{minDFA}}(\Sigma, \mathcal{D}, \approx) \to [\mathcal{O}]_\sim = \{[o]_\sim \mid o \in \mathcal{O}\}$ (where $[o]_\sim = \{o' \in \mathcal{O} \mid o' \sim o\}$). A typical situation is depicted in Fig 2, where the larger ellipse is closed under $\approx$ ($w \approx w'$), whereas the smaller circle is not, as it contains u but not u'.

As Angluin's algorithm works within the class of arbitrary DFA over Σ, its *Learner* might propose DFAs whose languages are neither a subset of $\mathcal{D}$ nor satisfy the closure properties for $\approx$. To rule out and fix such hypotheses, the language inclusion problem and the closure properties in question are required to

be *constructively decidable*, meaning that they are decidable and if the property fails, a *reason* of its failure can be computed.

Let us be more precise and define what we understand by a *learning setup*:

Definition 2. *Let* $(\mathcal{E}, \mathcal{O}, \mathcal{L})$ *be a representation system. A* learning setup *for* $(\mathcal{E}, \mathcal{O}, \mathcal{L})$ *is a quintuple* $(\Sigma, \mathcal{D}, \approx, obj, elem)$ *where*

- Σ *is an alphabet,*
- $\mathcal{D} \subseteq \Sigma^*$ *is the* domain*,*
- $\approx \subseteq \mathcal{D} \times \mathcal{D}$ *is an equivalence relation such that, for any* $w \in \mathcal{D}$, $[w]_\approx$ *is finite,*
- $obj : \mathfrak{R}_{\text{minDFA}}(\Sigma, \mathcal{D}, \approx) \to [\mathcal{O}]_{\sim_\mathcal{L}}$ *is a bijective effective mapping in the sense that, for* $\mathcal{A} \in \mathfrak{R}_{\text{minDFA}}(\Sigma, \mathcal{D}, \approx)$*, a representative of* $obj(\mathcal{A})$ *can be computed.*
- $elem : [\mathcal{D}]_\approx \to \mathcal{E}$ *is a bijective mapping such that, for any* $o \in \mathcal{O}$*,*

$$elem(\mathcal{L}(obj^{-1}([o]_{\sim_\mathcal{L}}))) = \mathcal{L}(o)$$

Furthermore, we require that the following hold for DFA $\mathcal{A}$ *over* Σ*:*

(D1) *The problem whether* $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{D}$ *is decidable. If, moreover,* $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{D}$*, one can compute* $w \in \mathcal{L}(\mathcal{A}) \setminus \mathcal{D}$*. We then say that* Inclusion$(\Sigma, \mathcal{D})$ *is* constructively decidable*.*

(D2) *If* $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{D}$*, it is decidable whether* $\mathcal{L}(\mathcal{A})$ *is* $\approx$*-closed. If not, one can compute* $w, w' \in \mathcal{D}$ *such that* $w \approx w'$*,* $w \in \mathcal{L}(\mathcal{A})$*, and* $w' \notin \mathcal{L}(\mathcal{A})$*. We then say that the problem* EqClosure$(\Sigma, \mathcal{D}, \approx)$ *is* constructively decidable*.*

Given a regular representation system $(\mathcal{E}, \mathcal{O}, \mathcal{L})$ for a which a learning setup exists, let us sketch a learning algorithm that, by using membership queries for elements $e \in \mathcal{E}$ and equivalence queries for objects $o \in \mathcal{O}$, identifies a predetermined object. Let $(\Sigma, \mathcal{D}, \approx, obj, elem)$ be a learning setup for $(\mathcal{E}, \mathcal{O}, \mathcal{L})$. To obtain this algorithm, we rely on Angluin's algorithm but modify it a little. The general idea is that the algorithms learns the regular language representing the object in question. However, membership queries for words and equivalence queries for automata are translated thanks to *elem* and respectively *obj*. However, use these functions in a meaningful manner, we modify also the processing of membership queries as well as the treatment of hypothesized DFAs:

- Once a membership query has been processed for a word $w \in \mathcal{D}$ (by querying $elem(w)$), queries $w' \in [w]_\approx$ must be answered equivalently. They are thus not forwarded to the *Teacher* anymore. Again, as in Section 2.5, we might think of an *Assistant* in between the *Learner* and the *Teacher* that checks if an equivalent query has already been performed. Membership queries for $w \notin \mathcal{D}$ are not forwarded to the *Teacher* either but answered negatively by the *Assistant*.
- When the table T is both closed and consistent, the hypothesized DFA $\mathcal{H}$ is computed as usual. After this, we proceed as follows:
  1. If $\mathcal{L}(\mathcal{H}) \not\subseteq \mathcal{D}$, compute a word $w \in \mathcal{L}(\mathcal{H}) \setminus \mathcal{D}$, declare it a counterexample, and modify the table T accordingly (possibly involving further membership queries).

2. If $\mathcal{L}(\mathcal{H}) \subseteq \mathcal{D}$ but $\mathcal{L}(\mathcal{H})$ is not $\approx$-closed, then compute $w, w' \in \mathcal{D}$ such that $w \approx w'$, $w \in \mathcal{L}(\mathcal{H})$, and $w' \notin \mathcal{L}(\mathcal{H})$; perform membership queries for $[w]_\approx$.

Actually, a hypothesized DFA $\mathcal{H}$ undergoes an equivalence test (by querying $obj(\mathcal{H})$) only if $\mathcal{L}(\mathcal{H}) \subseteq \mathcal{D}$ and $\mathcal{L}(\mathcal{H})$ is $\approx$-closed. I.e., if, in the context of the extended learning algorithm, we speak of a hypothesized DFA, we actually act on the assumption that $\mathcal{L}(\mathcal{H})$ is the union of $\approx$-equivalence classes.

Let the extension of Angluin's algorithm wrt. a learning setup as sketched above be called ExtendedAngluin. A careful analysis shows:

Theorem 1. *Let $(\Sigma, \mathcal{D}, \approx, obj, elem)$ be a learning setup for a representation system $(\mathcal{E}, \mathcal{O}, \mathcal{L})$. If $o \in \mathcal{O}$ has to be learned, then invoking*

$$\text{ExtendedAngluin}((\mathcal{E}, \mathcal{O}, \mathcal{L}), (\Sigma, \mathcal{D}, \approx, obj, elem))$$

returns, after finitely many steps, an object $o' \in \mathcal{O}$ such that $o' \sim_{\mathcal{L}} o$.

The theorem suggests the following definition:

Definition 3. *A representation system $(\mathcal{E}, \mathcal{O}, \mathcal{L})$ is* learnable *if there is some learning setup for $(\mathcal{E}, \mathcal{O}, \mathcal{L})$.*

2.7 Learning of timed systems

Angluin's algorithm has been extended to the setting of realtime systems. In [33] and [32], learning of *event-deterministic event-recording automata* and, respectively, *event-recording automata*, which both form sub-classes of timed automata [4], is described. For learning timed systems, several obstacles have to be overcome. First, timed strings range over pairs of letters (a, t) where a is from some finite alphabet while t is a real number, denoting the time when a has occurred. Thus, timed strings are sequences of letters taken from some *infinite alphabet*, while the learning algorithms deal with strings over finite alphabets. To be able to deal with strings over a finite alphabet, one joins several time points to get a *zone* [33] or *region* [32] of time points. This allows us to work over an alphabet consisting of actions and zone respectively region constraints, which, given a greatest time point K, gives rise to a finite alphabet. These strings, which are built-up from letters of actions and constraints, are sometimes also called *symbolic strings*. It has been shown that the set of symbolic strings accepted by an (event-deterministic) event-recording automaton forms a regular language [27], which we call the *symbolic regular language* of the automaton. The second obstacle to overcome is to derive a form or Nerode's right congruence for such symbolic regular languages that is consistent with a natural notion of right congruence for timed systems. This is implicitly carried out in [33, 32] by introducing *sharply-guarded* event-deterministic event-recording automata and *simple* event-recording automata, which are both unique normal forms for all automata accepting the same language.

This gives that Angluin's learning algorithm can be reused to learn (symbolic versions of) timed languages, yielding either *sharply-guarded* event-deterministic event-recording automata [33] or *simple* event-recording automata [32]. However, a direct application of Angluin's algorithm employs queries for symbolic strings that might represent complex timed behavior of the underlying system. To deal with this obstacle, an assistant can then be used to effectively bridge the level from symbolic timed strings (actions plus regions) to timed strings (action plus time value).

A different approach to learning timed systems, based on decision trees, is presented in [34].

2.8 Learning of ω-regular languages

Reactive systems, like a web server, are conceptually non-terminating systems, whose behavior is best modelled by infinite rather than finite strings. In such a setting, Angluin's learning algorithm has to be extended to learn ω-regular languages [45]. Therefore, two main obstacles have to be overcome: First, a suitable representation for infinite strings has to be found. Second, a suitable version of Nerode's right congruence has to be defined. The first obstacle is overcome by considering only so-called ultimately periodic words, which can be described by $u(v)^\omega$ for finite words $u, v \in \Sigma^*$ [53]. The second obstacle is solved by restricting the class of ω-regular languages. The given algorithm is restricted to languages L for which both L and its complement can be accepted by a deterministic ω-automaton, respectively. This class coincides with the class of *deterministic weak Büchi automata*. We are not aware of any Biermann-style or Angluin-style learning algorithm for the full class of ω-regular languages.

2.9 Further extensions

Especially in communication protocols, the input/output actions of a system can be distinguished in *control sensitive* or not. Some parameters of the system affect the protocol's state, while others are considered as *data*, just to be transmitted, for example. Optimizations of Angluin's algorithm for such a setup, in which we are given *parameterized actions*, have been proposed in [12].

Angluin's algorithm has been extended to deal with *regular tree languages* rather than word languages in [26, 18]. *Learning of strategies for games* has been considered in [25]. A symbolic version of Angluin's L^* algorithm based on BDDs [17] is presented in [5].

Learning (certain classes of) message passing automata accepting (regular) sets of message sequence charts has been studied in [15], using ideas of regular representations (cf. Section 2.6).

2.10 Implementations

Implementations of Angluin's learning algorithm have been described and analyzed in [11, 52].

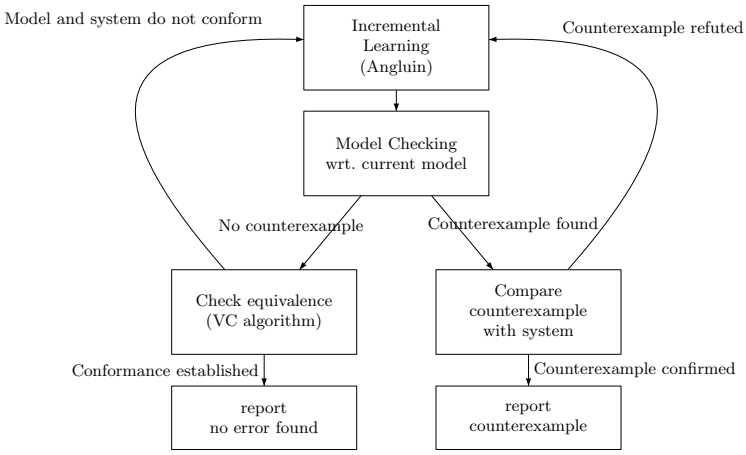


Fig. 3. Black Box Checking

3 Verification using Learning

3.1 Minimizing automata

Typical minimization algorithms for automata work on the transition graph of the given automaton. In [50], however, a minimization algorithm for incompletely specified finite-state machines was proposed, which is based on learning techniques: Instead of simplifying a given machine, a *new*, minimal machine is learned. For this, membership and equivalence queries are carried out on the given system.

Although no verification is performed in this approach, it entails one of the main motivations for using learning techniques in verification: Instead of deriving some object by modifying a large underlying system, one directly learns the small result. Clearly, it depends very much on the whole setup when this idea is fruitful.

3.2 Black-box checking

A different motivation for using learning techniques for verification purposes is when no model of the underlying system is given. Thus, we are faced with a so-called black box system i.e. a system for which no model is given but whose output can be observed for a given input. In *black box checking* or *adaptive model checking* [49, 36] these systems should be tested against a formal specification. Conceptually, the problem can be solved by learning a (white box) model of the black box, on which model checking can be performed. This can be done–under strong restrictions—for example using L^* and a conformance test like the ones in [59, 19]. In [49, 36], the tasks of learning and model checking are interweaved as explained in Figure 3, suggesting better practical performance.

First, an initial model of the system to check is learned. If model checking stops with a counter example, this might be due to the inadequacy of the current

version of the model. However, running the counter example reveals whether indeed a bug of the black-box system has been found, or, whether the counter example was spurious—and should be used to improve the model.

If model checking does not provide a counter example, we have to apply a conformance test [59, 19] to make sure that no (violating) run of the black box is missing in the model. If a missing run was detected, the model is updated, otherwise, the correctness of the black box has been proved.

3.3 Compositional verification

Compositional verification addresses the state-space explosion faced in model checking by exploiting the modular structure naturally present in system designs. One prominent technique uses the so-called *assume guarantee* rule: Take that we want to verify a property φ of the system $M_1 \parallel M_2$, consisting of two *modules* M_1 and M_2 running synchronously in parallel, denoted by $M_1 \parallel M_2 \models \varphi$. Instead of checking $M_1 \parallel M_2 \models \varphi$, one considers a module A and verifies that

1. $M_1 \parallel A \models \varphi$
2. M_2 is a refinement of A.

The rational is that A might be simpler than M_2, in the sense that both checking $M_1 \parallel A \models \varphi$ and M_2 is a refinement of A is easier than checking $M_1 \parallel M_2 \models \varphi$.

A setup, for which the assume-guarantee rule has been proven to be sound and complete [46], is when

1. modules can be represented as transition systems,
2. the parallel operator $\parallel$ satisfies $\mathcal{L}(M \parallel M') = \mathcal{L}(M) \cap \mathcal{L}(M')$, and
3. φ is a safety property and can thus be understood as a DFA $\mathcal{A}_\varphi$ accepting the allowed behavior of the system to check.

For such a setup, the assume-guarantee rule boils down to come up with some A such that

(AG1) $\mathcal{L}(M_1 \parallel A) \subseteq \mathcal{L}(\mathcal{A}_\varphi)$ and
(AG2) $\mathcal{L}(A) \supseteq \mathcal{L}(M_2)$.

In [24] and [5], it has been proposed to employ a learning algorithm to come up with such a module A. Here, we follow [5], which uses Angluin's L^* algorithm.

Whenever Angluin's algorithm proposes some hypothesis automaton A, it is easy to answer an equivalence query:

- if $\mathcal{L}(M_1 \parallel A) \not\subseteq \mathcal{L}(\mathcal{A}_\varphi)$, consider $w \in \mathcal{L}(M_1 \parallel A) \setminus \mathcal{L}(\mathcal{A}_\varphi)$. If $w \in \mathcal{L}(M_2)$, w witnesses that $M_1 \parallel M_2 \models \varphi$ does not hold. Otherwise, return w as a counterexample as result of the equivalence query.
- if $\mathcal{L}(A) \not\supseteq \mathcal{L}(M_2)$ provide a $w \in \mathcal{L}(A) \setminus \mathcal{L}(M_2)$ as a counterexample as result of the equivalence query.

If both tests succeed, we have found an A showing that $M_1 \parallel M_2 \models \varphi$ holds.

Membership queries are less obvious to handle. Clearly, when $w \in \mathcal{L}(M_2)$ then w must be in $\mathcal{L}(A)$ because of (AG2). If $w \notin \mathcal{L}(\mathcal{A}_\varphi)$ but $w \in \mathcal{L}(M_1)$, w must not be in $\mathcal{L}(A)$ since otherwise the safety property φ is not met ((AG1)).Note, if in this case also $w \in \mathcal{L}(M_2)$ (i.e., $w \in \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$, $w \notin \mathcal{L}(\mathcal{A}_\varphi)$), w is a witness that $M_1 \parallel M_2 \not\models \varphi$. In all other cases, however, it is not clear whether w should be classified as $+$ or $-$.

In [5], the following heuristic is proposed. Let $B := \overline{M_1} \cup \mathcal{A}_\varphi$, where $\overline{\mathcal{A}}$ denotes complementation. Thus, runs of B either satisfy φ or are not in the behavior of M_1 and thus not in the behavior of M_1 running in parallel with any module A or M_2. Now, if $M_1 \parallel M_2 \models \varphi$ then $\mathcal{L}(M_2) \subseteq \mathcal{L}(B)$ as M_2 may only consist of words either satisfying φ or ones that are removed when intersecting with M_1. If, on the other hand, $M_1 \parallel M_2 \not\models \varphi$, then there is a $w \in \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$ and $w \notin \mathcal{L}(\mathcal{A}_\varphi)$, meaning that $w \in \mathcal{L}(M_1) \cap \mathcal{L}(M_2) \cap \overline{\mathcal{L}(\mathcal{A}_\varphi)}$. Hence, $w \in \overline{\overline{\mathcal{L}(M_1)} \cup \mathcal{L}(\mathcal{A}_\varphi)} \cap \mathcal{L}(M_2)$. Thus, there is a w that is not in $\mathcal{L}(B)$ but in $\mathcal{L}(M_2)$.

In other words, B is a module that allows to either show or disprove that $M_1 \parallel M_2 \models \varphi$. Thus, answering membership queries according to B will eventually either show or disprove $M_1 \parallel M_2 \models \varphi$. While incrementally learning B, it is, however, expected, that one gets a hypothesis A smaller than B that satisfies (AG1) and (AG2) and thus shows that $M_1 \parallel M_2 \models \varphi$, or, that we get a word w witnessing that $M_1 \parallel M_2 \models \varphi$ does not hold.

The approach is sketched in Figure 4, where *cex* is a shorthand for *counter example*.

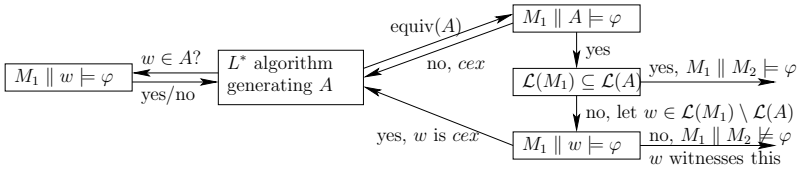


Fig. 4. Overview of compositional verification by learning assumptions

In [5], the approach has been worked out in the context of the verification tool NuSMV [20]. For this, a *symbolic* version of L^* has been developed that is based on BDDs [17] and carries out membership queries for sets of words rather than individual ones. It has been shown that the approach is beneficial for many examples. The success of the method, however, depends heavily on finding a suitable system A while learning B.

Instead of using Angluin's L^* algorithm, one could think of using an inexperienced teacher answering membership queries by ? whenever a choice is possible. While the learning algorithm is computationally more expensive, smaller invariants A can be expected. It would be interesting to compare both approaches on real world examples.

In [18], the assume-guarantee reasoning for simulation conformance between finite state systems and specifications is considered. A non-circular assume-guarantee proof rule is considered, for which a weakest assumption can be represented canonically by a deterministic tree automaton (DTA). Then, learning techniques for DTA are developed and examined by verifying non-trivial benchmarks.

In [25], game semantics, counterexample-guided abstraction refinement, assume-guarantee reasoning and Angluin's L^* algorithm are combined to yield a procedure for compositional verification of safety properties for fragments of sequential programs. For this, L^* is adapted to learn (regular) game strategies.

3.4 Learning fixpoints, regular model checking, and learning network invariants

Assume that we are given a regular set of initial states *Init* of a system to verify and a function Φ that computes for a given set W the set $\Phi(W)$ comprising of W together with successor states of W. Then, the set of reachable states is the least fixpoint $\lim_{n\to\infty} \Phi^n(W)$, where $\Phi^0(W) := Init$ and $\Phi^{n+1}(W) := \Phi(\Phi^n(W))$, for $n \geq 0$.

When proving properties for the set of reachable states, the typical approach would be to compute the (exact) set of reachable states by computing the minimal fixpoint using Φ, before starting to verify their properties. However, this approach may be problematic for two reasons:

- computing the fixpoint might be expensive, or
- the computation might not even terminate.

In regular model checking [2], for example, the set of initial states is regular (and represented by a finite automaton) and the set of successor states is computed by means of a transducer and thus is a regular set as well. There are, however, examples for which the set of reachable states is no longer regular. Thus, a straightforward fixpoint computation will not terminate.

A way out would be to consider a regular set of states over-approximating the set of reachable states.

Here, the idea of using learning techniques comes into play: Instead of computing iteratively the fixpoint starting from the initial states, one iteratively *learns* a fixpoint W. Clearly, one can stop whenever

- W is a fixpoint,
- W is a superset of *Init*, and
- W satisfies the property to verify.

For example, if one intends to verify a safety property, i.e., none of the reachable states is contained in the given bad states *Bad*, it suffices to find *any* fixpoint W that subsumes *Init* and does not intersect with *Bad*.

This general idea has been worked out in different flavors for verifying properties of infinite-state systems: Verifying safety-properties of parameterized systems by means of *network invariants* has been studied in [35]. Applications of

learning in *regular model checking* [2] for verifying safety properties [37, 56] and liveness properties [57] can also be understood as a way to find suitable fixpoints. A further application for infinite state systems is that of verifying *FIFO automata* [55]. Let us understand the gist of these approaches.

Learning network invariants One of the most challenging problems in verification is the *uniform verification of parameterized systems*. Given a parameterized system $S(n) = P[1] \parallel \cdots \parallel P[n]$ and a property φ, uniform verification attempts to verify that $S(n)$ satisfies φ for every $n > 1$. The problem is in general undecidable [7]. One possible approach is to look for restricted families of systems for which the problem is decidable (cf. [28, 22]). Another approach is to look for sound but incomplete methods (e.g., explicit induction [29], regular model checking [39, 51], or environment abstraction [23]).

Here, we consider uniform verification of parameterized systems using the heuristic of network invariants [60, 43]. In simple words, a *network invariant* for a given finite system P is a finite system I that abstracts the composition of every number of copies of P running in parallel. Thus, the network invariant contains all possible computations of every number of copies of P. If we find such a network invariant I, we can solve uniform verification with respect to the family $S(n) = P[1] \parallel \cdots \parallel P[n]$ by reasoning about I.

The general idea proposed in [60] and turned into a working method in [43], is to show by induction that I is a network invariant for P. The induction base is to prove that **(I1)** $P \sqsubseteq I$, for a suitable abstraction relation $\sqsubseteq$. The induction step is to show that **(I2)** $P \parallel I \sqsubseteq I$. After establishing that I is a network invariant we can prove **(P)** $I \models \varphi$, turning I into a *proper* network invariant with respect to φ. Then we conclude that $S(n) \models \varphi$ for every value of n.

Coming up with a proper network invariant is usually an iterative process. We start with divining a candidate for a network invariant. Then, we try to prove by induction that it is a network invariant. When the candidate system is nondeterministic this usually involves deductive proofs [42][2]. During this stage we usually need to refine the candidate until getting a network invariant. The final step is checking that this invariant is proper (by automatically model checking the system versus φ). If it is not, we have to continue refining our candidate until a proper network invariant is found. Coming up with the candidate network invariant requires great knowledge of the parameterized system in question and proving abstraction using deductive methods requires great expertise in deductive proofs and tools. Whether a network invariant exists is undecidable [60], hence all this effort can be done in vain.

In [35], a procedure searching systematically for a network invariant satisfying a given safety property is proposed. If one exists, the procedure finds a proper invariant with a minimal number of states. If no proper invariant exists, the procedure in general diverges (though in some cases it may terminate and report that no proper invariant exists). In the light of the undecidability result for the problem, this seems reasonable.

[2] For a recent attempt at mechanizing this step see [40].

Network invariants are usually explained in the setting of *transition structures* [41]. However, the learning algorithms have been given in terms of DFAs (see Section 2). Thus, we explain the approach in the setting of checking safety properties of networks that are described in terms of (the parallel product of) DFAs: We assume that P is given as a DFA and abstraction is just language inclusion: $\mathcal{A} \sqsubseteq \mathcal{B}$ iff $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$. A *safety property* is a DFA φ that accepts a *prefix-closed* language. Thus, a system $\mathcal{A}$ satisfies φ, denoted by $\mathcal{A} \models \varphi$, iff $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\varphi)$. The *parallel operator* combines two given DFAs into a new one. Finally, a *projection operator* for $\mathcal{A} \parallel \mathcal{B}$ onto $\mathcal{B}$ is a mapping $pr^{\mathcal{B}}_{\mathcal{A}\parallel\mathcal{B}} : \mathcal{L}(\mathcal{A} \parallel \mathcal{B}) \to \mathcal{L}(\mathcal{B})$ such that whenever $w \in \mathcal{L}(\mathcal{A} \parallel \mathcal{B})$ then for all $\mathcal{B}'$ with $pr^{\mathcal{B}}_{\mathcal{A}\parallel\mathcal{B}}(w) \in \mathcal{L}(\mathcal{B}')$ also $w \in \mathcal{L}(\mathcal{A} \parallel \mathcal{B}')$. In other words, (at least) the projection of w has to be removed from $\mathcal{B}$ to (eventually) remove w from the parallel product.

The careful reader observers that the proper invariant I we are looking for is indeed a fixpoint of the operator $(P \parallel \cdot)$, which subsumes the words given by P and has empty intersection with the bad words given by $\overline{\mathcal{L}(\varphi)}$. Thus, the following explanation can be understood as one way of learning fixpoints.

We now describe how to compute a proper network invariant in the case that one exists. For the rest of this section, we fix system P and a property automaton φ.

We only give an informal explanation, details can be found in [35]. We are using an unbounded number of *students* whose job it is to suggest possible invariants, one *teaching assistant* (TA) whose job is to answer queries by the students, and one *supervisor* whose job is to control the search process for a proper invariant. The search starts by the supervisor instructing one student to look for a proper invariant.

Like in Angluin's algorithm, every active student maintains a table (using $+$, $-$, and ?) and makes it weakly closed and weakly consistent by asking the TA membership queries. The TA answers with either $+$, $-$, or ?, as described below. When the table is weakly closed and consistent, the student translates the table to a sample O and this to a CSP problem. He solves the CSP problem using the SAT encoding (see Section 2.2). The solution with minimum range is used to form an automaton I that is proposed to the supervisor. The supervisor now checks whether I is indeed a proper invariant by checking (P), (I1), and (I2). If yes, the supervisor has found a proper invariant and the algorithm terminates with *proper invariant found*. If not, one of the following holds.

1. There is a string w such that $w \in \mathcal{L}(I)$ but $w \notin \mathcal{L}(\varphi)$,
2. There is a string w such that $w \in \mathcal{L}(P)$ but $w \notin \mathcal{L}(I)$,
3. There a string w such that $w \in \mathcal{L}(P \parallel I)$ but $w \notin \mathcal{L}(I)$.

In the first case, w should be removed from I. In the second case, the string w should be added to I. In these cases, the supervisor returns the appropriate string with the appropriate acceptance information to the student, who continues in the same manner as before.

In the last case, it is not clear, whether w should be added to I or removed from $P \parallel I$. For the latter, we have to remove the projection $pr^{I}_{P\parallel I}(w)$ from I. Unless w is listed negatively or $pr^{I}_{P\parallel I}(w)$ is listed positively in the table, both

possibilities are meaningful. Therefore, the supervisor has to follow both tracks. She copies the table of the current student, acquires another student, and asks the current student to continue with w in I and the new student to continue with $pr^I_{P\|I}(w)$ not in I.

In order to give answers, the teaching assistant uses the same methods as the supervisor, however, whenever a choice is possible she just says ?.

Choices can sometimes yield conflicts that are observed later in the procedure. Such a case reveals a conflicting assumption and requires the student to retire. If no working student is left, no proper invariant exists.

Clearly, the procedure sketched above finds a proper invariant if one exists. However, it consumes a lot of resources and may yield a proper invariant that is not minimal. It can, however, easily be adapted towards using only one student at a given time and stopping with a minimal proper invariant. Intuitively, the supervisor keeps track of the active students as well as the sizes of recently suggested automata. Whenever a student proposes a new automaton of size N, the supervisor computes the appropriate answer, which is either a change of the student's table or the answer *proper invariant found*. However, she postpones answering the student (or stopping the algorithm), gives the student priority N, and puts the student on hold. Then the supervisor takes a student that is on hold with minimal priority and sends the pre-computed instrumentation to the corresponding student. In case the student's instrumentation was tagged *proper invariant found* the procedure stops by printing the final proper invariant. Note that students always propose automata of at least the same size as before since the learning algorithm returns a *minimal* automaton conforming to the sample. Thus, whenever a proper invariant is found, it is guaranteed that the proper invariant is eventually reported by the algorithm, unless a smaller proper invariant is found before.

The formal details are given in [35].

Learning in regular model checking In *regular model checking* [2] we are typically faced with a finite automaton *Init* encoding the initial states of an infinite-state system and a transducer τ, which yields for an automaton $\mathcal{A}$, an automaton $\tau(\mathcal{A})$ encoding the current and successor states of states given by $\mathcal{A}$. The set of reachable states is then given by the least fixpoint of *Init* under τ. However, the set of reachable states might not be regular, implying that a simple fixpoint computation does not terminate. Thus, for example, by so-called *acceleration* techniques, supersets of fixpoints are computed [1].

The same holds for the learning approaches in [37, 56, 57], in which algorithms employing an (experienced) teacher are used, in contrast to the approach of learning network invariants.

Clearly, when learning a fixpoint $\mathcal{A}$, equivalence queries are not difficult to answer: $\mathit{Init} \subseteq \mathcal{L}(\mathcal{A})$ and $\mathcal{L}(\tau(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{A})$ can easily be answered and, if applicable, a counterexample can be computed. For membership queries, the situation is more involved. In [37, 56], the transducer is assumed to be *length preserving*, meaning that τ applied to some word w yields words w' of the same length.

Thus, the fixpoint of w under τ can be computed in finitely many steps. Then, the following heuristic is used: Given a word w' of length n, check whether there is some w of length n such that w' is in the fixpoint of w. It has been reported, that this heuristic works well in practice [37, 56]. However, it is not clear whether a regular fixpoint is found in this manner, if one exists.

In [57], verifying also *liveness* properties in the setting of regular model checking has been considered. For this, the set of reachable states *together* with information i on how many final states of a system encounter on a path of some length j leaving s is learned. It has been shown that for the function computing successor states plus this additional information, a *unique* fixpoint exists. Clearly, liveness properties can be answered when the fixpoint is given. Furthermore, for a given (s, i, j), it is easy to answer a membership query. Thus, the method works, provided a regular description for such a fixpoint exists.

Learning for verifying branching-time properties in the context of regular model checking asks for learning nested fixpoints and has been studied in [58].

3.5 Further applications

Synthesizing interface specifications Learning interface specifications for Java classes has been based on Angluin's algorithm in [3]. The problem studied is to derive (a description) of the most general way to call the methods in a Java class while maintaining some safety property. In [3], the problem is tackled by abstracting the class, giving rise to a partial-information two-player game. As analyzing such games is computationally expensive, approximative solutions based on learning are considered.

Learning versus testing In model-based testing [16], test suites are generated based on a model of the system under test (SUT). When no model is available, approximations of the SUT can be learned, the model can be analyzed, and used for test case generation. This approach, which is conceptually similar to black box checking, has been turned into a working method in [38].

In [10], the close relationship of learning techniques and test suites has been elaborated. In simple words, the following insight has been formalized and proved: if a conformance test suite is good enough to make sure that the SUT conforms to a given model, it should have enough information to identify the model. Likewise, if the observations of a system identify a single model, the observations should form a conformance test suite.

References

1. P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. Algorithmic improvements in regular model checking. In *Proc. 15^{th} Int. Conf. on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 236–248, 2003.
2. P. A. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena. A survey of regular model checking. In P. Gardner and N. Yoshida, editors, *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 35–48. Springer, 2004.

3. R. Alur, P. Cerný, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In J. Palsberg and M. Abadi, editors, *POPL*, pages 98–109. ACM, 2005.
4. R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
5. R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *17th International Conference on Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 548–562. Springer-Verlag, 2005.
6. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.
7. K. R. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.
8. J. L. Balcázar, J. Díaz, and R. Gavaldá. Algorithms for learning finite automata from queries: A unified view. In *Advances in Algorithms, Languages, and Complexity*, pages 53–72. Kluwer, 1997.
9. L. Baresi and R. Heckel, editors. *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3922 of *Lecture Notes in Computer Science*. Springer, 2006.
10. T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, and B. Steffen. On the correspondence between conformance testing and regular inference. In M. Cerioli, editor, *Fundamental Approaches to Software Engineering, FASE'05*, volume 3442 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2005.
11. T. Berg, B. Jonsson, M. Leucker, and M. Saksena. Insights to Angluin's learning. In *Proceedings of the International Workshop on Software Verification and Validation (SVV 2003)*, volume 118 of *Electronic Notes in Theoretical Computer Science*, pages 3–18, Dec. 2003.
12. T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines with parameters. In Baresi and Heckel [9], pages 107–121.
13. T. Berg and H. Raffelt. Model checking. In Broy et al. [16].
14. A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behaviour. *IEEE Transactions on Computers*, 21:592–597, 1972.
15. B. Bollig, J.-P. Katoen, C. Kern, and M. Leucker. Replaying play in and play out: Synthesis of design models from scenarios by learning. In O. Grumberg and M. Huth, editors, *Proceedings of the 13th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'07)*, volume 4424 of *Lecture Notes in Computer Science*, Braga, Portugal, Mar. 2007. Springer.
16. M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.
17. R. E. Bryant. Symbolic manipulation of boolean functions using a graphical representation. In *Proceedings of the 22nd ACM/IEEE Design Automation Conference*, pages 688–694, Los Alamitos, Ca., USA, June 1985. IEEE Computer Society Press.
18. S. Chaki, E. M. Clarke, N. Sinha, and P. Thati. Automated assume-guarrantee reasoning for simulation conformance. In *17th International Conference on Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 534–547. Springer-Verlag, 2005.

19. T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. on Software Engineering*, 4(3):178–187, May 1978. Special collection based on COMPSAC.
20. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.
21. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
22. E. M. Clarke, M. Talupur, T. Touili, and H. Veith. Verification by network decomposition. In *15th International Conference on Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 276–291. Springer-Verlag, 2004.
23. E. M. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In *7th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 126–141. Springer-Verlag, 2006.
24. J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In H. Garavel and J. Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 2003.
25. A. Dimovski and R. Lazic. Assume-guarantee software verification based on game semantics. In Z. Liu and J. He, editors, *ICFEM*, volume 4260 of *Lecture Notes in Computer Science*, pages 529–548. Springer, 2006.
26. F. Drewes and J. Högberg. Learning a regular tree language from a teacher. In Z. Ésik and Z. Fülöp, editors, *Proc. 7th Intl. Conf. Developments in Language Theory*, volume 2710 of *Lecture Notes in Computer Science*, pages 279–291, Szeged, Hungary, 2003.
27. D. D'Souza. A logical characterisation of event clock automata. *International Journal of Foundations of Computer Science (IJFCS)*, 14(4):625–639, Aug. 2003.
28. E. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *Proc. 17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Computer Science*, pages 236–254, 2000.
29. E. Emerson and K. Namjoshi. Reasoning about rings. In *Proc.* 22^{th} *ACM Symp. on Principles of Programming Languages*, 1995.
30. E. M. Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.
31. E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.
32. O. Grinchtein, B. Jonsson, and M. Leucker. Inference of timed transition systems. In *6th International Workshop on Verification of Infinite-State Systems*, volume 138/4 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2004.
33. O. Grinchtein, B. Jonsson, and M. Leucker. Learning of event-recording automata. In *Proceedings of the Joint Conferences FORMATS and FTRTFT*, volume 3253 of *Lecture Notes in Computer Science*, Sept. 2004.
34. O. Grinchtein, B. Jonsson, and P. Pettersson. Inference of event-recording automata using timed decision trees. In C. Baier and H. Hermanns, editors, *CONCUR*, volume 4137 of *Lecture Notes in Computer Science*, pages 435–449. Springer, 2006.

35. O. Grinchtein, M. Leucker, and N. Piterman. Inferring network invariants automatically. In *Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR'06)*, volume 4130 of *Lecture Notes in Artificial Itelligence*, Sept. 2006.
36. A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 357–??, 2002.
37. P. Habermehl and T. Vojnar. Regular model checking using inference of regular languages. *Electr. Notes Theor. Comput. Sci.*, 138(3):21–36, 2005.
38. H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In *Proc. 15^{th} Int. Conf. on Computer Aided Verification*, 2003.
39. B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In S. Graf and M. Schwartzbach, editors, *Proc. TACAS '00, 6^{th} Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, 2000.
40. Y. Kesten, N. Piterman, and A. Pnueli. Bridging the gap between fair simulation and trace inclusion. *Information and Computation*, 200(1):35–61, 2005.
41. Y. Kesten and A. Pnueli. Control and data abstraction: The cornerstones of practical formal verification. *Software Tools for Technology Transfer*, 2(4):328–342, 2000.
42. Y. Kesten, A. Pnueli, E. Shahar, and L. Zuck. Network invariants in action. In *Proc. CONCUR 2002, 13^{th} Int. Conf. on Concurrency Theory*, volume 2421 of *Lecture Notes in Computer Science*, 2002.
43. R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. *Information and Computation*, 117(1):1–11, 15 Feb. 1995.
44. K. J. Lang. Random dfa's can be approximately learned from sparse uniform examples. In *COLT*, pages 45–52, 1992.
45. O. Maler and A. Pnueli. On the learnability of infinitary regular sets, 1991. Esprit Basic Reasearch Action No 3096.
46. K. S. Namjoshi and R. J. Trefler. On the competeness of compositional reasoning. In E. A. Emerson and A. P. Sistla, editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 139–153. Springer, 2000.
47. A. L. Oliveira and J. P. M. Silva. Efficient algorithms for the inference of minimum size dfas. *Machine Learning*, 44(1/2):93–119, 2001.
48. J. Oncina and P. Garcia. Inferring regular languages in polynomial update time. In N. P. de la Blanca, A. Sanfeliu, and E. Vidal, editors, *Pattern Recognition and Image Analysis*, volume 1 of *Series in Machine Perception and Artificial Intelligence*, pages 49–61. World Scientific, Singapore, 1992.
49. D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. In J. Wu, S. T. Chanson, and Q. Gao, editors, *Formal Methods for Protocol Engineering and Distributed Systems, FORTE/PSTV*, pages 225–240, Beijing, China, 1999. Kluwer.
50. J. M. Pena and A. L. Oliveira. A new algorithm for the reduction of incompletely specified finite state machines. In *ICCAD*, pages 482–489, 1998.
51. A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In *Proc. 12^{th} Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 328–343. Springer Verlag, 2000.
52. H. Raffelt and B. Steffen. Learnlib: A library for automata learning and experimentation. In Baresi and Heckel [9], pages 377–380.
53. W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 4, pages 133–191. Elsevier Science Publishers B. V., 1990.

54. B. Trakhtenbrot and J. Barzdin. *Finite automata: behaviour and synthesis*. North-Holland, 1973.
55. A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Actively learning to verify safety for fifo automata. In K. Lodaya and M. Mahajan, editors, *FSTTCS*, volume 3328 of *Lecture Notes in Computer Science*, pages 494–505. Springer, 2004.
56. A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Learning to verify safety properties. In J. Davies, W. Schulte, and M. Barnett, editors, *ICFEM*, volume 3308 of *Lecture Notes in Computer Science*, pages 274–289. Springer, 2004.
57. A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Using language inference to verify omega-regular properties. In N. Halbwachs and L. D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 45–60. Springer, 2005.
58. A. Vardhan and M. Viswanathan. Learning to verify branching time properties. In D. F. Redmiles, T. Ellman, and A. Zisman, editors, *ASE*, pages 325–328. ACM, 2005.
59. M. P. Vasilevski. Failure diagnosis of automata. *Cybernetic*, 9(4):653–665, 1973.
60. P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 68–80, Grenoble, France, 1989. Springer-Verlag.