Compositional Automata Learning of Synchronous Systems

Thomas Neele[1] and Matteo Sammartino[2]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
t.s.neele@tue.nl
[2] Royal Holloway University of London, Egham, UK
University College London, London, UK
matteo.sammartino@rhul.ac.uk

Abstract. Automata learning is a technique to infer an automaton model of a black-box system via queries to the system. In recent years it has found widespread use both in industry and academia, as it enables formal verification when no model is available or it is too complex to create one manually. In this paper we consider the problem of learning the individual components of a black-box synchronous system, assuming we can only query the whole system. We introduce a *compositional* learning approach in which several learners cooperate, each aiming to learn one of the components. Our experiments show that, in many cases, our approach requires significantly fewer queries than a widely-used non-compositional algorithm such as $\mathtt{L}^*$.

1 Introduction

Automata learning is a technique for inferring an automaton from a black-box system by interacting with it and observing its responses. It can be seen as a game in which a *learner* poses queries to a *teacher* – an abstraction of the target system – with the intent of inferring a model of the system. The learner can ask two types of queries: a *membership* query, asking if a given sequence of actions is allowed in the system; and an *equivalence* query, asking if a given model is correct. The teacher must provide a counter-example in case the model is incorrect. In practice, membership queries are implemented as tests on the system, and equivalence queries as conformance test suites.

The original algorithm $\mathtt{L}^*$ proposed by Dana Angluin in 1987 [3] allowed learning DFAs; since then it has been extended to a variety of richer automata models, including symbolic [5] and register [7,26] automata, automata for ω-regular languages [4], and automata with fork-join parallelism [18], to mention recent work. Automata learning enables formal verification when no formal model is available and also reverse engineering of various systems. Automata learning has found wide application in both academia and industry. Examples are: the verification of neural networks [31], finding bugs in specific implementations of security [29,12] and network protocols [11], or refactoring legacy software [30].

In this paper we consider the case when the system to be learned consists of several concurrent components that interact in a synchronous way; the components themselves are not accessible, but their number and respective input alphabets are known. It is well-known that the composite state-space can grow exponentially with the number of components. If we use $\mathtt{L}^*$ to learn such a system as a whole, it will take a number of queries that is proportional to the whole state-space – many more than if we were able to apply $\mathtt{L}^*$ to the individual components. Since in practice queries are implemented as tests performed on the system (in the case of equivalence queries, exponentially many tests are required), learning the whole system may be impractical if tests take a non-negligible amount of time, e.g., if each test needs to be repeated to ensure accuracy of results or when each test requires physical interaction with a system.

In this work we introduce a *compositional* approach that is capable of learning models for the individual components, by interacting with an ordinary teacher for the *whole* system. This is achieved by translating queries on a single component to queries on the whole system and interpreting their results on the level of a single component. The fundamental challenge is that components are *not* independent: they interact synchronously, meaning that sequences of actions in the composite system are realised by the individual components performing their actions in a certain relative order. The implications are that: (i) the answer to some membership queries for a specific component may be *unknown* if the correct sequence of interactions with other components has not yet been discovered; and (ii) counter-examples for the global system cannot univocally be decomposed into counter-examples for individual components, therefore some of them may result in *spurious* counter-examples that need to be corrected later.

To tackle these issues, we make the following contributions:

- A *compositional learning framework*, orchestrating several instances of (an extension of) $\mathtt{L}^*$ with the purpose to learn models for the individual components from an ordinary monolithic teacher. An adapter transforms queries on single components into queries to the monolithic teacher.
- An extension of $\mathtt{L}^*$ that can deal with unknown membership query results and spurious counter-examples; when plugged into the aforementioned framework, we obtain a learning algorithm for our setting.
- An implementation of our approach as a tool Coal based on the state-of-the-art automata learning library LearnLib [22], accompanied by a comprehensive set of experiments: for some of the larger systems, our approach requires up to six orders of magnitude fewer membership queries and up to ten times fewer equivalence queries than $\mathtt{L}^*$ (applied to the monolithic system).

The rest of this paper is structured as follows. We introduce preliminary concepts and notation in Section 2. Our learning framework is presented in Section 3. Section 4 discusses the details of our implementation and the results of our experiments. Related work is highlighted in Section 5 and Section 6 concludes.

2 Preliminaries

Notation and terminology. We use Σ to denote a *finite alphabet* of action symbols, and Σ^* to denote the set of finite sequences of symbols in Σ, which we call *traces*; we use ϵ to denote the empty trace. Given two traces $s_1, s_2 \in \Sigma^*$, we denote their concatenation by $s_1 \cdot s_2$; for two sets $S_1, S_2 \subseteq \Sigma^*$, $S_1 \cdot S_2$ denotes element-wise concatenation. Given $s \in \Sigma^*$, we denote by $\mathit{Pref}(s)$ the set of prefixes of s, and by $\mathit{Suff}(s)$ the set of its suffixes; the notation lifts to sets $S \subseteq \Sigma^*$ as expected. We say that $S \subseteq \Sigma^*$ is *prefix-closed* (resp. *suffix-closed*) whenever $S = \mathit{Pref}(S)$ (resp. $S = \mathit{Suff}(S)$). The *projection* $\sigma_{\upharpoonright \Sigma'}$ of σ on an alphabet $\Sigma' \subseteq \Sigma$ is the sequence of symbols in σ that are also contained in Σ'. Finally, given a set S, we write $|S|$ for its cardinality.

In this work we represent the state-based behaviour of a system as a *labelled transition system.*

Definition 1 (Labelled Transition System). *A* labelled transition system *(LTS) is a four-tuple $L = (S, \rightarrow, \hat{s}, \Sigma)$, where*

- *S is a set of states, which we refer to as the* state space*;*
- *$\rightarrow \subseteq S \times \Sigma \times S$ is a transition relation, which we write in infix notation as $s \xrightarrow{a} t$, for $(s, a, t) \in \rightarrow$.*
- *$\hat{s} \in S$ is an initial state; and*
- *Σ is a finite set of actions, called the* alphabet*.*

We say that L is deterministic *whenever for each $s \in S$, $a \in \Sigma$ there is at most one transition from s labelled by a.*

Some actions in Σ may not be allowed from a given state. We say that an action a is *enabled* in s, written $s \xrightarrow{a}$, if there is t such that $s \xrightarrow{a} t$. This notation is also extended to traces $\sigma \in \Sigma^*$, yielding $s \xrightarrow{\sigma} t$ and $s \xrightarrow{\sigma}$. The language of L is the set of traces enabled from the starting state, formally:

$$\mathcal{L}(L) = \{\sigma \in \Sigma^* \mid \hat{s} \xrightarrow{\sigma}\} \ .$$

From here on, we only consider deterministic LTSs. Note that this does not reduce the expressivity, in terms of the languages that can be encoded.

Remark 1. Languages of LTSs are always prefix-closed, because every prefix of an enabled trace is necessarily enabled. Prefix-closed languages are accepted by a special class of deterministic finite automata (DFA), where all states are final except for a sink state, from which all transitions are self-loops. Our implementation (see Section 4) uses these models as underlying representation of LTSs.

We now introduce a notion of parallel composition of LTSs, which must synchronise on shared actions.

Definition 2. *Given n LTSs where $L_i = (S_i, \rightarrow_i, \hat{s}_i, \Sigma_i)$ for $1 \leq i \leq n$, their* parallel composition*, notation $\Vert_{i=1}^{n} L_i$, is an LTS over the alphabet $\bigcup_{i=1}^{n} \Sigma_i$, defined as follows:*

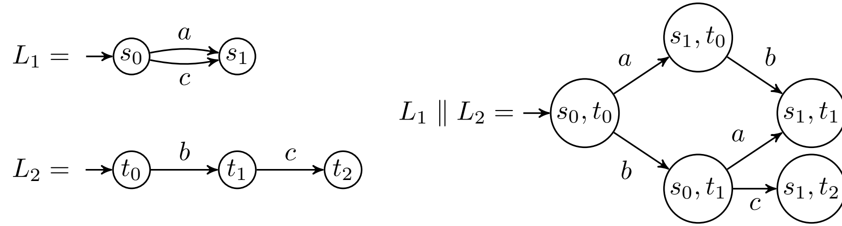
- *the state space is* $S_1 \times S_2 \times \cdots \times S_n$*;*
- *the transition relation is given by the following rule*

$$\frac{s_i \xrightarrow{a}_i t_i \quad \text{for all } i \text{ such that } a \in \Sigma_i \qquad s_j = t_j \quad \text{for all } j \text{ such that } a \notin \Sigma_j}{(s_1, \ldots, s_n) \xrightarrow{a} (t_1, \ldots, t_n)}$$

- *the initial state is* $(\hat{s}_1, \ldots, \hat{s}_n)$*.*

Intuitively, a certain action a can be performed from $(s_1, \ldots, s_n)$ only if it can be performed by all component LTSs that have a in their alphabet; all other LTSs must stay idle. We say that an action a is *local* if there is exactly one i such that $a \in \Sigma_i$, otherwise it is called *synchronising*. The parallel composition of LTSs thus forces individual LTSs to cooperate on synchronising actions; local actions can be performed independently. We typically refer to the LTSs that make up a composite LTS as *components*. Synchronisation of components corresponds to communication between components in real-world settings.

Example 1. Consider the left two LTSs below with the respective alphabets $\{a, c\}$ and $\{b, c\}$. Their parallel composition is depicted on the right.

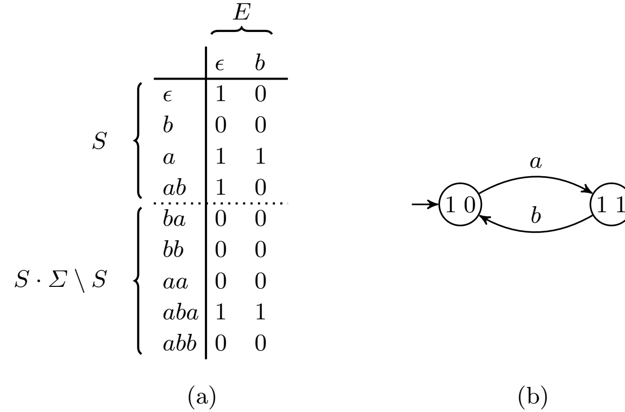


Here a and b are local actions, whereas c is synchronising. Note that, despite L_1 being able to perform c from its initial state s_0, there is no c transition from (s_0, t_0), because c is not initially enabled in L_2. First L_2 will have to perform b to reach t_1, where c is enabled, which will allow $L_1 \parallel L_2$ to perform c. □

We sometimes also apply parallel composition to sets of traces: $\Vert_i S_i$ is equivalent to $\Vert T_i$, where each T_i is a tree-shaped LTS that accepts exactly S_i, *i.e.*, $\mathcal{L}(T_i) = S_i$. In such cases, we will explicitly mention the alphabet each T_i is assigned. This notation furthermore applies to single traces: $\Vert_i \sigma_i = \Vert_i \{\sigma_i\}$.

2.1 L* algorithm

We now recall the basic L* algorithm. Although the algorithm targets DFAs, we will present it in terms of deterministic LTSs, which we use in this paper (these are a sub-class of DFAs, see Remark 1). The algorithm can be seen as a game in which a *learner* poses queries to a *teacher* about a target language $\mathcal{L}$ that only the teacher knows. The goal of the learner is to learn a *minimal* deterministic LTS with language $\mathcal{L}$. In practical scenarios, the teacher is an abstraction of the target system we wish to learn a model of. The learner can ask two types of queries:



		ϵ	b
S	ϵ	1	0
	b	0	0
	a	1	1
	ab	1	0
$S \cdot \Sigma \setminus S$	ba	0	0
	bb	0	0
	aa	0	0
	aba	1	1
	abb	0	0

(a) (b)

Fig. 1: A closed and consistent observation table and the LTS that can be constructed from it.

- **Membership query:** is a trace s in the target language $\mathcal{L}$? The teacher will return a Yes/No answer.
- **Equivalence query:** does a given *hypothesis* LTS H accept $\mathcal{L}$? The teacher will return Yes/No; a No answer comes with a *counter-example*, i.e., a trace in $\mathcal{L}(H) \mathbin{\Delta} \mathcal{L}$, where Δ denotes the symmetric difference.

The learner organises the information received in response to queries in an *observation table*, which is a triple (S, E, T), consisting of a finite, prefix-closed set $S \subseteq \Sigma^*$, a finite, suffix-closed set $E \subseteq \Sigma^*$, and a function $T : (S \cup S \cdot \Sigma) \cdot E \to \{0, 1\}$. The function T can be seen as a table in which rows are labelled by traces in $S \cup S \cdot \Sigma$, columns by traces in E, and cells $T(s \cdot e)$ contain 1 if $s \cdot e \in \mathcal{L}$ and 0 otherwise.

Example 2. Consider the prefix-closed language $\mathcal{L}$ over the alphabet $\Sigma = \{a, b\}$ consisting of traces where a and b alternate, starting with a; for instance $aba \in \mathcal{L}$ but $abb \notin \mathcal{L}$. An observation table generated by a run of `L*` targeting this language is shown in Figure 1a. □

Let $row_T \colon S \cup S \cdot \Sigma \to (E \to \{0, 1\})$ denote the function $row_T(s)(e) = T(s \cdot e)$ mapping each row of T to its content (we omit the subscript T when clear from the context). The crucial observation is that T approximates the Nerode congruence [28] for $\mathcal{L}$ as follows: s_1 and s_2 are in the same congruence class only if $row(s_1) = row(s_2)$, for $s_1, s_2 \in S$. Based on this fact, the learner can construct a hypothesis LTS from the table, in the same way the minimal DFA accepting a given language is built via its Nerode congruence:[3]

- the set of states is $\{row(s) \mid s \in S, row(s)(\epsilon) = 1\}$;

[3] For the minimal DFA, the set of states is $\{row(s) \mid s \in S\}$; here we only take accepting states as we are building an LTS.

- the initial state is $row(\epsilon)$;
- the transition relation is given by $row(s) \xrightarrow{a} row(s \cdot a)$, for all $s \in S$ and $a \in \Sigma$.

In order for the transition relation to be well-defined, the table has to satisfy the following conditions:

- **Closedness:** for all $s \in S, a \in \Sigma$, there is $s' \in S$ such that $row_T(s') = row_T(s \cdot a)$.
- **Consistency:** for all $s_1, s_2 \in S$ such that $row_t(s_1) = row_t(s_2)$, we have $row_T(s_1 \cdot a) = row_T(s_2 \cdot a)$, for all $a \in \Sigma$.

Example 3. The table of Example 2 is closed and consistent. The corresponding hypothesis LTS, which is also the minimal LTS accepting $\mathcal{L}$, is shown in Figure 1b. □

The algorithm works in an iterative fashion: starting from the empty table, where S and E only contain ϵ, the learner extends the table via membership queries until it is closed and consistent, at which point it builds a hypothesis and submits it to the teacher in an equivalence query. If a counter-example is received, it is incorporated in the observation table by adding its prefixes to S, and the updated table is again checked for closedness and consistency. The algorithm is guaranteed to eventually produce a hypothesis H such that $\mathcal{L}(H) = \mathcal{L}$, for which an equivalence query will be answered positively, causing the algorithm to terminate.

3 Learning Synchronous Components Compositionally

In this section, we show how to compositionally learn an unknown system $M = M_1 \parallel \cdots \parallel M_n$ consisting of n parallel LTSs. To achieve this, we assume that we are given: (i) a teacher for M; and (ii) the respective alphabets $\Sigma_1, \ldots, \Sigma_n$ of $M_1, \ldots, M_n$. To achieve this, we propose the architecture in Figure 2. We have n leaners, which are instances of (an extension of) the $\mathtt{L}^*$ algorithm, one for each component M_i. The instance $\mathtt{L}^*_i$ can pose queries for M_i to an *adapter*, which converts them to queries on M. The resulting yes/no answer (and possibly counter-example) is translated back to information about M_i, which is returned to leaner $\mathtt{L}^*_i$. To achieve this, the adapter moreover choreographs the learners to some extent: before an equivalence query $H \stackrel{?}{=} M$ can be sent to the teacher, the adapter must first receive equivalence queries $H_i \stackrel{?}{=} M_i$ from each learner.

We first discuss the implementation of the adapter and show its limitations. To deal with these limitations, we next propose a couple of extensions to $\mathtt{L}^*$ (Section 3.2). Completeness claims are stated in Section 3.3. Several optimisations are discussed in Section 3.4.

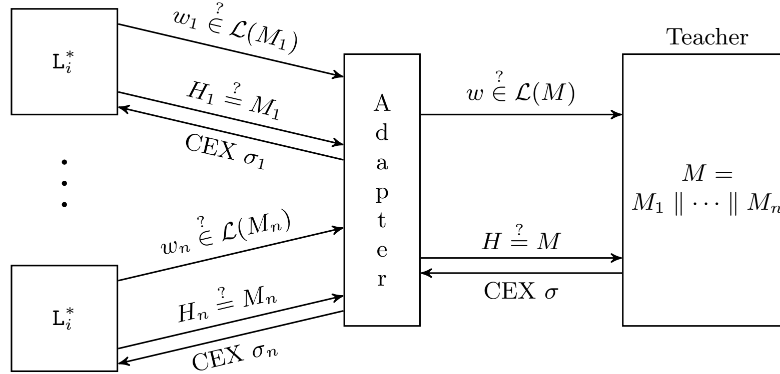


Fig. 2: Architecture for learning LTS M consisting of components $M_1 \parallel \cdots \parallel M_n$.

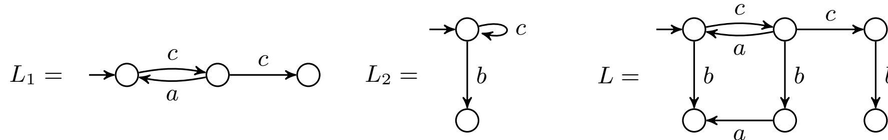


Fig. 3: Running example consisting of two LTSs L_1 and L_2 and their parallel composition L. The respective alphabets are $\{a, c\}$, $\{b, c\}$ and $\{a, b, c\}$.

3.1 Query Adapter

As sketched above, our adapter answers queries on each of the LTSs M_i, based on information obtained from queries on M. However, the application of the parallel operator causes loss of information, as the following example illustrates. We will use the LTSs below as a running example throughout this section.

Example 4. Consider the LTSs L_1, L_2 and $L = L_1 \parallel L_2$ depicted in Figure 3. Their alphabets are $\{a, c\}$, $\{b, c\}$ and $\{a, b, c\}$, respectively.

Suppose we sent a membership query bc to the teacher and we receive as answer that $bc \notin \mathcal{L}(L)$. At this point, we do not have sufficient information to deduce about the respective projections whether $bc_{\upharpoonright\{a,c\}} = c \notin \mathcal{L}(L_1)$ or $bc_{\upharpoonright\{b,c\}} = bc \notin \mathcal{L}(L_2)$ (or both). In this case, only the latter holds. Similarly, if a composite hypothesis $H = H_1 \parallel H_2$ is rejected with a negative counter-example $ccc \notin \mathcal{L}(L)$, we cannot deduce whether this is because $ccc \notin \mathcal{L}(L_1)$ or $ccc \notin \mathcal{L}(L_2)$ (or both). Here, however, the former is true but the latter is not, *i.e.*, ccc is not a counter-example for H_2 at all. □

Generally, given negative information on the composite level ($\sigma \notin \mathcal{L}(M)$), it is hard to infer information for a single component M_i, whereas positive information ($\sigma \in \mathcal{L}(M)$) easily translates back to the level of individual components.

We thus need to relax the guarantees on the answers given by the adapter in the following way:

1. Not all membership queries can be answered, the adapter may return the answer 'unknown'.
2. An equivalence query for component i can be answered with a *spurious* counter-example $\sigma_i \in \mathcal{L}(H_i) \cap \mathcal{L}(M_i)$.

The procedures that implement the adapter are stated in Listing 1. For each $1 \leq i \leq n$, we have one instance of each of the functions Member_i and Equiv_i, used by the ith learner to pose its queries. Here, we assume that for each component i, a copy of the latest hypothesis H_i is stored, as well as a set P_i which contains traces that are certainly in $\mathcal{L}(M_i)$. Membership and equivalence queries on M will be forwarded to the teacher via the functions $\mathit{Member}(\sigma)$ and $\mathit{Equiv}(H)$, respectively.

Membership Queries A membership query $\sigma \in \mathcal{L}(M_i)$ can be answered directly by posing $\sigma \in \mathcal{L}(M)$ to the teacher if σ contains only actions local to M_i. However, in the case where σ contains synchronising actions, cooperation from other components M_j is required. So, during the runtime of the program, for each i we collect traces in a set P_i, for which it is certain that $P_i \subseteq \mathcal{L}(M_i)$. That is, P_i contains traces which were returned as positive counter-examples (line 16) or membership queries (line 5). Recall from Section 2 that we can construct tree-LTSs to compute $\|_{j \neq i} P_j$, where each P_i has alphabet Σ_i. By construction, we have $\mathcal{L}(\|_{j \neq i} P_j) \subseteq \mathcal{L}(\|_{j \neq i} M_j)$, and so we have an under-approximation of the behaviour of other components, possibly including some synchronising actions they can perform. If we find in $\mathcal{L}(\|_{j \neq i} P_j)$ a trace σ' such that σ and σ' contain the same sequence of synchronising actions (line 2, stored in set Π), we construct an arbitrary interleaving (respecting synchronising actions) of σ and σ' and forward it to the teacher (line 4). Such an interleaving is a trace $\sigma_{int} \in \mathcal{L}(\sigma \parallel \sigma')$ of maximal length. Note that a $\sigma' \in \Pi$ trivially exists if σ does not contain synchronising actions. If, on the other hand, no such σ' exists, we do not have sufficient information on how other LTSs M_j can cooperate, and we return 'unknown' (line 7).

Example 5. Refer to the running example in Figure 3. Suppose that the current knowledge about L_2 is $H_2 = \{\epsilon, b\}$. When $\mathit{Member}_1(c)$ is called, $\Pi = \emptyset$, because there is no trace $\sigma' \in P_2$ that is equal to c when restricted to $\{a, c\}$, therefore *unknown* is returned. Intuitively, since the second learner has not yet discovered that c or bc (or some other trace containing a c) is in its language, the adapter is unable to turn the query c on L_1 into a query for the composite system. □

Example 6. Suppose now that $cac \in P_1$, *i.e.*, we already learned that $cac \in \mathcal{L}(L_1)$. When posing the membership query $cbc \in \mathcal{L}(L_2)$, the adapter finds that cac and cbc contain the same synchronising actions (*viz.* cc) and constructs an interleaving, for example $cabc$. The teacher answers negatively to the query $cabc \in \mathcal{L}(L)$, and thus we learn that $cbc \notin \mathcal{L}(L_2)$. □

Listing 1: Membership and equivalence query procedures for component i.

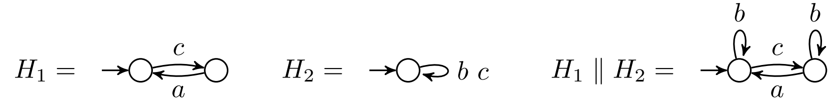
Input: Alphabets $\Sigma_1, \ldots, \Sigma_n$ of the components
Data: for each i, the latest hypothesis H_i and a set P_i of traces, initially $\{\epsilon\}$.

1 **Function** $Member_i(\sigma)$
2 $\quad \Pi := \{\sigma' \in \mathcal{L}(\|_{j \neq i} P_j) \mid \sigma'_{\restriction \Sigma_i} = \sigma_{\restriction \Sigma_{other}}\}$ where $\Sigma_{other} = \bigcup_{j \neq i} \Sigma_j$;
3 $\quad$ **if** $\Pi \neq \emptyset$ **then**
4 $\quad\quad$ $answer := Member(\sigma_{int})$ for some $\sigma' \in \Pi$ and some maximal $\sigma_{int} \in \mathcal{L}(\sigma \| \sigma')$; `/* construct interleaving */`
5 $\quad\quad$ **if** $answer = yes$ **then** $P_i := P_i \cup \{\sigma\}$;
6 $\quad\quad$ **return** $answer$
7 $\quad$ **else return** $unknown$;
8 **Function** $Equiv_i(H')$
9 $\quad H_i := H'$;
10 $\quad$ **while** $true$ **do**
11 $\quad\quad$ barrier(n) ; `/* wait until this point is reached for every i */`
12 $\quad\quad$ construct $H = \|_i H_i$;
13 $\quad\quad$ **switch** $Equiv(H)$ **do**
14 $\quad\quad\quad$ **case** yes **do return** yes;
15 $\quad\quad\quad$ **case** (no, σ) **do**
16 $\quad\quad\quad\quad$ **if** $\sigma \notin \mathcal{L}(H)$ **then** $P_i := P_i \cup \{\sigma_{\restriction \Sigma_i}\}$;
17 $\quad\quad\quad\quad$ **if** $a \in \Sigma_i$, where $\sigma = \sigma' a$, and $\sigma \in \mathcal{L}(H) \Leftrightarrow \sigma_{\restriction \Sigma_i} \in \mathcal{L}(H_i)$ **then**
18 $\quad\quad\quad\quad\quad$ **return** $(no, \sigma_{\restriction \Sigma_i})$

Equivalence Queries For equivalence queries, the adapter offers functions $Equiv_i$. To construct a corresponding query on the composite level, we first need to gather a hypothesis H_i for each i. Thus, we synchronise all learners in a barrier (line 11), after which a composite hypothesis can be constructed and forwarded to the teacher (lines 12, 13). An affirmative answer can be returned directly, while in the negative case we investigate the returned counter-example σ. If σ is a positive counter-example, we add its projection to P_i (line 16). By the assumption that σ is shortest[4], H and M agree on all $\sigma' \in Pref(\sigma) \setminus \{\sigma\}$. Thus, σ only concerns H_i if the last action in σ is contained in Σ_i. Furthermore, we need to check whether H and H_i agree on σ: it can happen that $\sigma_{\restriction \Sigma_i} \in \mathcal{L}(H_i)$ but $\sigma \notin \mathcal{L}(H)$ due to other hypotheses not providing the necessary communication opportunities. If both conditions are satisfied (line 17), we return the projection of σ on Σ_i (line 18). Otherwise, we cannot conclude anything about H_i at this moment and we iterate (line 10). In that case, we effectively wait for other hypotheses H_j, with $j \neq i$, to be updated before trying again. A termination argument is provided later in this section.

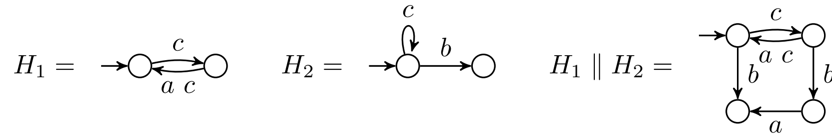
[4] This assumption can be satisfied in practice by using a lexicographical ordering on the conformance test suite the teacher generates to decide equivalence.

Example 7. Again considering our running example (Figure 3), suppose the two learners call in parallel the functions $Equiv_1(H_1)$ and $Equiv_2(H_2)$. The provided hypotheses and their parallel composition are as follows:



The adapter forwards $H = H_1 \parallel H_2$ to the teacher, which returns the counter-example cc. The last symbol, c, occurs in both alphabets, but $cc \in \mathcal{L}(H)$ does not hold and $cc_{\restriction \Sigma_2} \in \mathcal{L}(H_2)$ does, so only $Equiv_1(H_1)$ returns (no, cc). The call to $Equiv_2(H_2)$ hangs in the while loop of line 10 until $Equiv_1$ is invoked with a different hypothesis. □

Example 8. Suppose now that the hypotheses and their composition are:



When we submit $Equiv(H_1 \parallel H_2)$, we may receive the negative counter-example ccc, which is a shortest counter-example. This counter-example does not contain any information to suggest that it only applies to H_1. It is a spurious counter-example for H_2, since that *should* contain the trace ccc. □

3.2 L* extensions

As explained in the previous section, the capabilities of our adapter are limited compared to an ordinary teacher. We thus extend L* to deal with the answer 'unknown' to membership queries and to deal with spurious counter-examples.

Answer 'unknown'. The setting of receiving incomplete information through membership queries first occurred in [15], and is also discussed in [24]. Here we briefly recall the ideas of [15]. To deal with partial information from membership queries, the concept of an observation table is generalised such that the function $T : (S \cup S \cdot \Sigma) \cdot E \to \{0, 1\}$ is a partial function, that is, for some cells we have no information. Based on T, we now define the function $row : S \cup S \cdot \Sigma \to E \to \{0, 1, ?\}$ to fill the cells of the table: $row_T(s)(e) = T(se)$ if $T(se)$ is defined and ? otherwise. We refer to '?' as a *wildcard*; its actual value is currently unknown and might be learned at a later time or never at all. To deal with the uncertain nature of wildcards, we introduce a relation $\approx$ on rows, where $row(s_1) \approx row(s_2)$ iff for every $e \in E$, $row(s_1)(e) \neq row(s_2)(e)$ implies that $row(s_1)(e) = ?$ or $row(s_2)(e) = ?$. Note that $\approx$ is not an equivalence relation since it is not transitive. Closedness and consistency are defined as before, but

now use the new relation $\approx$. We say an LTS M is *consistent* with T iff for all $s \in \Sigma^*$ such that $T(s)$ is defined, we have $T(s) = 1$ iff $s \in \mathcal{L}(M)$.

As discussed earlier, Angluin's original $\mathtt{L}^*$ algorithm relies on the fact that, for a closed and consistent table, there is a unique minimal DFA (or, in our case, LTS) that is consistent with T. However, the occurrence of wildcards in the observation table may allow multiple minimal LTSs that are consistent with T. Such a minimal consistent LTS can be obtained with a SAT solver, as described in [19].

Similar to Angluin's original algorithm, this extension comes with some correctness theorems. First of all, it terminates outputting the minimal LTS for the target language. Furthermore, each hypothesis is consistent with all membership queries and counter-examples that were provided so far. Lastly, each subsequent hypothesis has at least as many states as the previous one, but never more than the minimal LTS for the target language.

Spurious Counter-Examples. We now extend this algorithm with the ability to deal with spurious counter-examples. Any *negative* counter-example $\sigma \in \mathcal{L}(H_i)$ might be spurious, *i.e.*, it is actually the case that $\sigma \in \mathcal{L}(M_i)$. Since $\mathtt{L}^*$ excludes σ from the language of all subsequent hypotheses, we might later get the same trace σ, but now as a *positive* counter-example. In that case, the initial negative judgment from the equivalence teacher was spurious.

One possible way of dealing with spurious counter-examples, is adding to $\mathtt{L}^*$ the ability to *overwrite* entries in the observation table in case a spurious counter-example is corrected. However, this may cause the learner to diverge if infinitely many spurious counter-examples are returned. Therefore, we instead choose to add a backtracking mechanism to ensure our search will converge. The pseudo code is listed in Listing 2; we refer to this as $\mathtt{L}^*_{?,\mathtt{b}}$ ($\mathtt{L}^*$ with wildcards and backtracking).

We have a mapping BT that stores backtracking points; BT is initialised to the empty mapping (line 1). Lines 5-11 ensure the observation table is closed and consistent in the same way as $\mathtt{L}^*$, but use the relation $\approx$ on rows instead. Next, we construct a minimal hypothesis that is consistent with the observations in T (line 12). This hypothesis is posed as an equivalence query. If the teacher replies with a counter-example σ for which $T(\sigma) = 0$, then σ was a spurious counter-example, so we backtrack and restore the observation table from just before $T(\sigma)$ was introduced (line 15). Otherwise, we store a backtracking point for when σ later turns out to be spurious (line 17); this is only necessary if σ is a negative counter-example. Note that not all information is lost when backtracking: the set P_i stored in the adapter is unaffected, so some positive traces are carried over after backtracking. Finally, we incorporate σ into the observation table (line 18). When the teacher accepts our hypothesis, we terminate.

We finish this section with an example that shows how spurious counter-examples may be resolved.

Listing 2: Learning with wildcards and backtracking.

```
1  Set BT to ∅;
2  Initialise S and E to {ϵ};
3  Extend T to S ∪ S · Σ_i by calling Member_i;
4  repeat
5      while (S, E, T) is not closed and consistent do
6          if (S, E, T) is not consistent then
7              Find s_1, s_2 ∈ S, a ∈ Σ_i, e ∈ E such that row_T(s_1) ≈ row_T(s_2) and
                 T(s_1 · a · e) ≉ T(s_2 · a · e);
8              Add a · e to S and extend T by calling Member_i;
9          if (S, E, T) is not closed then
10             Find s_1 ∈ S, a ∈ Σ_i such that row_T(s_1 · a) ≉ row_T(s) for all s ∈ S;
11             Add s_1 · a to S and extend T by calling Member_i;
12     Call Equiv_i(H) for some minimal LTS H consistent with T;
13     if Teacher replies with counter-example σ then
14         if T(σ) = 0 then        /* σ corrects an earlier spurious CEX */
15             (S, E, T) := BT(σ);
16         else if σ ∈ L(H) then                    /* σ might be spurious */
17             BT(σ) := (S, E, T);
18         Add σ and all its prefixes to S and extend T by calling Member_i;
19 until Teacher replies yes to conjecture H;
20 return H;
```

Example 9. Refer again to the LTSs of our running example in Figure 3. Consider the situation after proposing the hypotheses of Example 8 and receiving the counter-example *ccc*, which is spurious for the second learner.

In the next iteration, $Member_2$ can answer some membership queries, such as *cbc*, necessary to expand the table of the second learner. This is enabled by the fact that P_1 contains *cc* from the positive counter-example of Example 7 (line 2 of Listing 1). The resulting updated hypotheses are as follows.

Now the counter-example to composite hypothesis $H'_1 \parallel H'_2$ is *cacc*. The projection on Σ_2 is *ccc*, which directly contradicts the counter-example received in the previous iteration. This spurious counter-example is thus repaired by backtracking in the second learner. The invocation of $Equiv_1(H'_1)$ by the first learner does not return this counter-example, since $H'_1 \parallel H'_2$ and H'_1 do not agree on *cacc*, so the check on line 17 of Listing 1 fails.

Finally, in the next iteration, the respective hypotheses coincide with L_1 and L_2 and both learners terminate. □

3.3 Correctness

As a first result, we show that our adapter provides correct information on each of the components when asking membership queries. This is required to ensure that information obtained by membership queries does not conflict with counter-examples. Proofs are omitted for space reasons.

Theorem 1. *Answers from* $Member_i$ *are consistent with* $\mathcal{L}(M_i)$.

Before presenting the main theorem on correctness of our learning framework, we first introduce several auxiliary lemmas. In the following, we assume n instances of $\mathtt{L}^*_{?,\mathtt{b}}$ run concurrently and each queries the corresponding functions $Member_i$ and $Equiv_i$, as per our architecture (Figure 2). First, a counter-example cannot be spurious for all learners; thus at least one learner obtains valid information to progress its learning.

Lemma 1. *Every counter-example obtained from* $Equiv(H)$ *is valid for at least one learner.*

The next lemma shows that even if a spurious counter-example occurs, this does not induce divergence, since it is always repaired by a corresponding positive counter-example in finite time.

Lemma 2. *If* $Equiv(H)$ *always returns a shortest counter-example, then each spurious counter-example is repaired by another counter-example within a finite number of invocations of* $Equiv(H)$*, the monolithic teacher.*

Our main theorem states that a composite system is learned by n copies of $\mathtt{L}^*_{?,\mathtt{b}}$ that each call our adapter (see Figure 2).

Theorem 2. *Running* n *instances of* $\mathtt{L}^*_{?,\mathtt{b}}$ *terminates, and on termination we have* $H_1 \parallel \cdots \parallel H_n = M_1 \parallel \cdots \parallel M_n$.

Remark 2. We cannot claim the stronger result that $H_i = M_i$ for all i, since different component LTSs can result in the same parallel composition. For example, consider the below LTSs, both with alphabet $\{a\}$:

$H_1 = \rightarrow\bigcirc \qquad H_2 = \rightarrow\bigcirc\circlearrowleft a$

Here we have $H_1 \parallel H_2 = H_1 \parallel H_1$. The equivalence oracle thus may also return *yes* even when the component LTSs differ slightly.

3.4 Optimisations

There are a number of optimisations that can dramatically improve the practical performance of our learning framework. We briefly discuss them here.

First, finding whether there is a trace $\sigma' \in \Pi$ (line 2 of Listing 1) can quickly become expensive once the sets P_i grow larger. We thus try to limit the size of each P_i without impacting the amount of information it provides on the

synchronisation opportunities offered by component M_i. Therefore, when we derive that $\sigma \in \mathcal{L}(M_i)$, we only store the shortest prefix ρ of σ such that ρ and σ contain the same synchronising actions. That is, $\sigma = \rho \cdot \rho'$ and ρ' contains only actions local to M_i. Furthermore, we construct $\|_{j \neq i} P_j$ only once after each call to $Equiv_i$ and we cache accesses to $\|_{j \neq i} P_j$, such that it is only traversed once when performing multiple queries σ^1, σ^2 for which it holds that $\sigma^1_{\restriction \Sigma_{other}} = \sigma^2_{\restriction \Sigma_{other}}$. A possibility that we have not explored is applying *partial-order reduction* to eliminate redundant interleavings in $\|_{j \neq i} P_j$.

Since the language of an LTS is prefix-closed, we can – in some cases – extend the function T that is part of the observation table without performing membership queries. Concretely, if $T(\sigma) = 0$ then we can set $T(\sigma \cdot \sigma') = 0$ for any trace σ'. Dually, if $T(\sigma \cdot \sigma') = 1$ then we set $T(\sigma) = 1$.

4 Experiments

We created an experimental implementation of our algorithms in a tool called Coal (COmpositional Automata Learner) [27], implemented in Java. It relies on LearnLib [22], a library for automata learning, which allows us to re-use standard data structures, such as observation tables, and compare our framework to a state-of-the-art implementation of $\mathtt{L}^*$. To extract a minimal LTS from an observation table, we first attempt the inexact blue-fringe variant of RPNI [20] (as implemented in LearnLib). If this does not result in an LTS that is minimal, we resort to an exact procedure based on a SAT translation; we use the Z3 solver [10].

Our experiments are run on a machine with an Intel Core i3 3.6GHz, with 16GB of RAM, running Ubuntu 20.04. For each experiment, we use a time-out of 30 minutes.

4.1 Random Systems

We first experiment with a large number of composite systems where each of the component LTSs is randomly generated. This yields an accurate reflection of actual behavioural transition systems [16]. Each component LTS has a random number of states between 5 and 9 (inclusive, uniformly distributed) and a maximum number of outgoing edges per state between 2 and 4 (inclusive, uniformly distributed).

We assign alphabets to the components LTSs in five different ways that reflect real-world communication structures, see Figure 4. Here, each edge represents a communication channel that consists of two synchronising actions; each component LTS furthermore has two local actions. The hyperedge in *multiparty* indicates multiparty communication: the two synchronising actions in such a system are shared by all component LTSs. The graph that represents the *bipartite* communication structure is always complete, and the components are evenly distributed between both sides. *Random* is slightly different: it contains $2(n-1)$

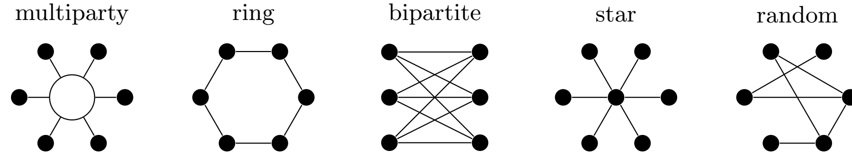


Fig. 4: Communication structure of the randomly generated systems. Dots represent components LTSs; edges represent shared synchronising actions.

edges, where n is the number of components, each consisting of one action; we furthermore ensure the random graph is connected.

For our five communication structures, we create ten instances for each number of components between 4 and 9; this leads to a total benchmark set of 300 LTSs. Out of these, 47 have more than 10,000 states, including 12 LTSs of more than 100,000 states. The largest LTS contains 379,034 states. *Bipartite* often leads to relatively small LTSs, due to its high number of synchronising actions.

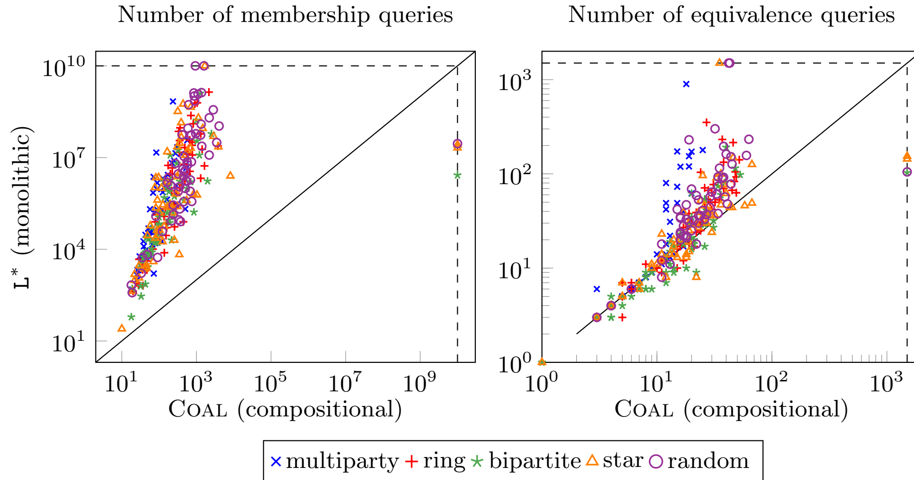


Fig. 5: Performance of $\mathtt{L}^*$ and compositional learning on random models.

On each LTS, we run the classic $\mathtt{L}^*$ algorithm and Coal, and record the number of queries posed to the teacher.[5] The result is plotted in Figure 5; note the log scale. Here, marks that lie on the dashed line indicate a time-out or out-of-memory for one of the two algorithms.

Coal outperforms the monolithic $\mathtt{L}^*$ algorithm in the number of membership queries for all cases (unless it fails). In more than half of the cases, the

[5] The number of queries is the standard performance measure for query learning algorithms; runtime is less reliable, as it depends on the specific teacher implementation.

Table 1: Performance of COAL and L* for realistic composite systems.

model	scaling	comp	states	COAL time(s)	COAL memQ	COAL eqQ	COAL spCE	L* time(s)	L* memQ	L* eqQ
CloudOps	$W = 1, C = 1, N = 3$	5	690	1.06	957	24	0	1.85	2,740,128	88
CloudOps	$W = 1, C = 1, N = 4$	6	1,932	1.13	1,004	26	0	16.99	22,252,120	216
CloudOps	$W = 2, C = 1, N = 3$	5	3,858	47.13	8,897	41	3	8.94	12,574,560	99
CloudOps	$W = 2, C = 1, N = 4$	6	10,824	48.20	8,811	36	3	84.02	91,178,900	227
ProdCons	$K = 5, P = 1, C = 1$	3	246	0.51	285	13	3	0.34	160,126	30
ProdCons	$K = 5, P = 2, C = 1$	4	962	0.54	401	13	0	2.35	2,523,625	91
ProdCons	$K = 5, P = 3, C = 2$	6	13,001	1.65	1,239	16	0	65.42	60,186,235	187
ProdCons	$K = 5, P = 3, C = 3$	7	45,302	3.21	2,276	16	0	241.37	222,567,729	193
ProdCons	$K = 3, P = 2, C = 2$	5	2,273	0.61	456	13	0	1.66	2,141,165	43
ProdCons	$K = 5, P = 2, C = 2$	5	3,329	1.29	596	15	1	6.36	6,984,705	93
ProdCons	$K = 7, P = 2, C = 2$	5	4,385	0.97	799	15	0	17.56	15,792,997	135

difference is at least three orders of magnitude; it can even reach six orders of magnitude. For equivalence queries, the difference is less obvious, but our compositional approach scales better for larger systems. This is especially relevant, because in practice implementations equivalence queries may require a number of membership queries that is exponential in the size of the system. Multiparty communication systems benefit most from compositional learning. The number of spurious counter-examples that occurs for these models is limited: about one on average. Only twelve models require more than five spurious counter-examples; the maximum number required is thirteen. This is encouraging, since even for this varied set of LTSs the amount of duplicate work performed by COAL is limited.

4.2 Realistic Systems

Next, we investigate the performance of COAL on two realistic systems that were originally modelled as a Petri net. These Petri nets can be scaled according to some parameters to yield various instances. The *ProdCons* system models a buffer of size K that is accessed by P producers and C consumers; it is described in [32, Fig. 8]. The *CloudOpsManagement* net is obtained from the 2019 Model Checking Contest [2], and describes the operation of C containers and operating systems and W application runtimes in a cloud environment. Furthermore, we scale the number N of application runtime components. We generate the LTS that represents the marking graph of these nets and run L* and COAL; the results are listed in Table 1. For each system, we list the values of scaling parameters, the number of components and the number of states of the LTS. For COAL and L*, we list the runtime and the number of membership and equivalence queries; for COAL we also list the number of spurious counter-examples (column spCE).

The results are comparable to our random experiments: COAL outperforms L* in number of queries, especially for larger systems. For the two larger CloudOps-

Management instances, the increasing runtime of COAL is due to the fact that two of the components grow as the parameter W increases. The larger number of states causes a higher runtime of the SAT procedure for constructing a minimal LTS.

We remark that in our experiments, the teacher has direct access to the LTS we aim to learn, leading to cheap membership and equivalence queries. Thus, in this idealised setting, $\mathtt{L}^*$ incurs barely any runtime penalty for the large number of queries it requires. Using a realistic teacher implementation would quickly cause time-outs for $\mathtt{L}^*$, making the results of our experiments less insightful.

5 Related Work

Finding ways of projecting a known concurrent system down into its components is the subject of several works, e.g., [8,17]. In principle, it would be possible to learn the system monolithically and use the aforementioned results. However, as shown in Section 4, this may result in a substantial query blow-up.

Learning approach targeting various concurrent systems exist in the literature. As an example of the monolithic approach above, the approach of [6] learns asynchronously-communicating finite state machines via queries in the form of message sequence charts. The result is a monolithic DFA that is later broken down into components via an additional synthesis procedure. This approach thus does not avoid the exponential blow-up in queries. Another difference with our work is that we consider synchronous communication.

Another monolithic approach is [18], which provides an extension of $\mathtt{L}^*$ to *pomset automata*. These automata are acceptors of partially-ordered multisets, which model concurrent computations. Accordingly, this relies on an oracle capable of processing pomset-shaped queries; adapting the approach to an ordinary sequential oracle – as in our setting – may cause a query blow-up.

A severely restricted variant of our setting is considered in [13], which introduces an approach to learn *Systems of Procedural Automata*. Here, DFAs representing procedures are learned independently. The constrained interaction of such DFAs allows for deterministically translating between component-level and system-level queries, and for univocally determining the target of a counterexample. Our setting is more general – arbitrary (not just pair-wise) synchronisations are allowed at any time – hence these abilities are lost.

Two works that do not allow synchronisation at all are [23,25]. In [23] individual components are learned without any knowledge of the component number and their individual alphabets, however components cannot synchronise (alphabets are assumed to be disjoint). This is a crucial difference with our approach, which instead has to deal with unknown query results and spurious counterexamples precisely due to the presence of synchronising actions. An algorithm for learning Moore machines with decomposable outputs is propose in [25]. This algorithm spawns several copies of $\mathtt{L}^*$, one per component. This approach is not applicable to our setting, as we do not assume decomposable output and allow dependencies between components.

Other approaches consider teachers that are unable to reply to membership queries [1,14,15,24]; they all use SAT-based techniques to construct automata. The closest works to ours are: [24], considering the problem of compositionally learning a property of a concurrent system with full knowledge of the components; and [1], learning an unknown component of the *serial* composition of two automata. In none of these works spurious counter-examples arise.

6 Conclusion

We have shown how to learn component systems with synchronous communication in a compositional way. Our framework uses an adapter and a number of concurrent learners. Several extensions to $\mathtt{L}^*$ were necessary to circumvent the fundamental limitations of the adapter. Experiments with our tool Coal show that our compositional approach offers much better scalability than a standard monolithic approach.

In future work, we aim to build on our framework in a couple of ways. First, we want to apply these ideas to all kinds of extensions of $\mathtt{L}^*$ such as TTT [21] (for reducing the number of queries) and algorithms for learning extended finite state machines [7]. Our expectation is that the underlying learning algorithm can be replaced with little effort. Next, we want to eliminate the assumption that the alphabets of individual components are known a priori. We envisage this can be achieved by combining our work and [23].

We also would like to explore the integration of learning and model-checking. A promising direction is *learning-based assume-guarantee reasoning*, originally introduced by Cobleigh et. al. in [9]. This approach assumes that models for the individual components are available. Using our approach, we may be able to drop this assumption, and enable a fully black-box compositional verification approach.

Acknowledgements. We thank the anonymous reviewers for their useful comments, and Tobias Kappé for suggesting several improvements. This research was partially supported by the EPSRC Standard Grant *CLeVer* (EP/S028641/1).

References

1. Abel, A., Reineke, J.: Gray-Box Learning of Serial Compositions of Mealy Machines. In: NFM. pp. 272–287 (2016). https://doi.org/10.1007/978-3-319-40648-0_21
2. Amparore, E., et al.: Presentation of the 9th Edition of the Model Checking Contest. In: TACAS2019. LNCS, vol. 11429, pp. 50–68 (2019). https://doi.org/10.1007/978-3-030-17502-3_4
3. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6
4. Angluin, D., Fisman, D.: Learning regular omega languages. Theor. Comput. Sci. **650**, 57–72 (2016). https://doi.org/10.1016/j.tcs.2016.07.031

5. Argyros, G., D'Antoni, L.: The Learnability of Symbolic Automata. In: CAV. pp. 427–445 (2018). https://doi.org/10.1007/978-3-319-96145-3_23
6. Bollig, B., Katoen, J., Kern, C., Leucker, M.: Learning Communicating Automata from MSCs. IEEE Trans. Software Eng. **36**(3), 390–408 (2010). https://doi.org/10.1109/TSE.2009.89
7. Cassel, S., Howar, F., Jonsson, B., Steffen, B.: Active learning for extended finite state machines. Formal Aspects Comput. **28**(2), 233–263 (2016). https://doi.org/10.1007/s00165-016-0355-5
8. Castellani, I., Mukund, M., Thiagarajan, P.S.: Synthesizing Distributed Transition Systems from Global Specifications. In: FSTTCS. LNCS, vol. 1738, pp. 219–231 (1999). https://doi.org/10.1007/3-540-46691-6_17
9. Cobleigh, J.M., Giannakopoulou, D., Pasareanu, C.S.: Learning Assumptions for Compositional Verification. In: TACAS. Lecture Notes in Computer Science, vol. 2619, pp. 331–346. Springer (2003). https://doi.org/10.1007/3-540-36577-X_24
10. De Moura, L., Bjørner, N.: Z3: An efficient SMT Solver. In: TACAS 2008. LNCS, vol. 4963, pp. 337–340 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
11. Fiterau-Brostean, P., Janssen, R., Vaandrager, F.W.: Combining Model Learning and Model Checking to Analyze TCP Implementations. In: CAV2016. LNCS, vol. 9780, pp. 454–471 (2016). https://doi.org/10.1007/978-3-319-41540-6_25
12. Fiterau-Brostean, P., Jonsson, B., Merget, R., de Ruiter, J., Sagonas, K., Somorovsky, J.: Analysis of DTLS Implementations Using Protocol State Fuzzing. In: USENIX (2020), `https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean`
13. Frohme, M., Steffen, B.: Compositional learning of mutually recursive procedural systems. Int J Softw Tools Technol Transfer **23**, 521–543 (2021). https://doi.org/10.1007/s10009-021-00634-y
14. Grinchtein, O., Leucker, M.: Learning Finite-State Machines from Inexperienced Teachers. In: ICGI. pp. 344–345 (2006). https://doi.org/10.1007/11872436_30
15. Grinchtein, O., Leucker, M., Piterman, N.: Inferring Network Invariants Automatically. In: IJCAR. pp. 483–497 (2006). https://doi.org/10.1007/11814771_40
16. Groote, J.F., van der Hofstad, R., Raffelsieper, M.: On the random structure of behavioural transition systems. Science of Computer Programming **128**, 51–67 (2016). https://doi.org/10.1016/j.scico.2016.02.006
17. Groote, J.F., Moller, F.: Verification of parallel systems via decomposition. In: CONCUR 1992. LNCS, vol. 630, pp. 62–76 (1992). https://doi.org/10.1007/BFb0084783
18. van Heerdt, G., Kappé, T., Rot, J., Silva, A.: Learning Pomset Automata. In: FoSSaCS2021. LNCS, vol. 12650, pp. 510–530 (2021). https://doi.org/10.1007/978-3-030-71995-1_26
19. Heule, M.J.H., Verwer, S.: Exact DFA Identification Using SAT Solvers. In: ICGI 2010. LNCS, vol. 6339, pp. 66–79 (2010). https://doi.org/10.1007/978-3-642-15488-1_7
20. de la Higuera, C.: Grammatical Inference: Learning Automata and Grammars. Cambridge University Press, USA (2010)
21. Isberner, M., Howar, F., Steffen, B.: The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning. In: RV2014. LNCS, vol. 8734, pp. 307–322 (2014). https://doi.org/10.1007/978-3-319-11164-3_26
22. Isberner, M., Howar, F., Steffen, B.: The Open-Source LearnLib: A Framework for Active Automata Learning. In: CAV2015. LNCS, vol. 9206, pp. 487–495 (2015). https://doi.org/10.1007/978-3-319-21690-4_32

23. Labbaf, F., Groote, J.F., Hojjat, H., Mousavi, M.R.: Compositional Learning for Interleaving Parallel Automata. In: FoSSaCS 2023. LNCS, Springer (2023)
24. Leucker, M., Neider, D.: Learning Minimal Deterministic Automata from Inexperienced Teachers. In: ISoLA. pp. 524–538 (2012). https://doi.org/10.1007/978-3-642-34026-0_39
25. Moerman, J.: Learning Product Automata. In: ICGI. vol. 93, pp. 54–66. PMLR (2018), http://proceedings.mlr.press/v93/moerman19a.html
26. Moerman, J., Sammartino, M., Silva, A., Klin, B., Szynwelski, M.: Learning nominal automata. In: POPL. pp. 613–625 (2017). https://doi.org/10.1145/3009837.3009879
27. Neele, T., Sammartino, M.: Replication package for the paper "Compositional Automata Learning of Synchronous Systems" (2023). https://doi.org/10.5281/zenodo.7503397
28. Nerode, A.: Linear automaton transformations. Proceedings of the American Mathematical Society **9**(4), 541–544. (1958). https://doi.org/10.2307/2033204
29. de Ruiter, J., Poll, E.: Protocol State Fuzzing of TLS Implementations. In: USENIX. pp. 193–206 (2015), https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter
30. Schuts, M., Hooman, J., Vaandrager, F.W.: Refactoring of Legacy Software Using Model Learning and Equivalence Checking: An Industrial Experience Report. In: IFM. LNCS, vol. 9681, pp. 311–325 (2016). https://doi.org/10.1007/978-3-319-33693-0_20
31. Shih, A., Darwiche, A., Choi, A.: Verifying Binarized Neural Networks by Angluin-Style Learning. In: SAT. vol. 11628, pp. 354–370 (2019). https://doi.org/10.1007/978-3-030-24258-9_25
32. Zuberek, W.: Petri net models of process synchronization mechanisms. In: SMC1999. vol. 1, pp. 841–847. IEEE (1999). https://doi.org/10.1109/ICSMC.1999.814201