Compositional Learning of Mealy Machines with Projections

Rick Koenders RICK.KOENDER@OU.NL and **Joshua Moerman** JOSHUA.MOERMAN@OU.NL
Open Universiteit, the Netherlands

Editor: Editor's name

Abstract

We consider parallel composition in the context of active automata learning. The problem we solve is that automata learning typically learns a single finite state machine for a system, even when the system is composed of smaller subsystems. We assume a closed-box setting, where the compositional structure of the system is not known and has to be inferred during learning. We develop a new algorithm, CoALP, which, like L* or TTT, learns the system but simultaneously learns a decomposition of the system. Depending on the structure of the system, this reduces the number of queries drastically, as shown by an evaluation of the algorithm. The algorithm CoALP is generic in the sense that it can use any learning algorithm for the subsystems.

Keywords: Active Automata Learning, Model Learning, Compositionality, Finite State Machines.

1. Introduction

Automata learning algorithms like L* by Angluin (1987) and TTT by Isberner et al. (2014) are very capable of inferring the state machine from observed inputs and outputs, even with thousands of states. They are successfully applied to software systems in order to, e.g., find security flaws in DTLS (Fiterau-Brostean et al., 2023), or to understand legacy systems at ASML (Sanchez et al., 2019). Many more applications of model learning are listed in a survey article by Vaandrager (2017). However, oftentimes, systems are not *just* a finite state machine; they have additional structure because these systems are engineered in a structured way. For instance, a system may be composed from smaller components.

In this paper we tackle the problem of learning a model of composed systems. The aim is to create learning algorithms which, first of all, scale to larger (composed) systems, and, second, to obtain models which are easier to analyse, as they are decomposed into smaller components. The type of composition we use in this paper is parallel composition: inputs to the system are given to each component, and each component will react to it. The resulting outputs are assembled into a combined output. Our learning algorithm CoALP can learn a decomposition on-the-fly and it assumes no additional information (i.e., it is fully closed-box). In particular, the (number of) components have to be inferred.

Recently new learning algorithms are described which incorporate some sort of *compositionality* into model learning (Frohme and Steffen, 2021; Labbaf et al., 2023; Neele and Sammartino, 2023; Samadi et al., 2025). The most recent, by Henry et al. (2025), is the most general so far and reports efficient learning, with up to 5 times fewer queries needed. All these compositional learning techniques decompose the input alphabet, whereas our proposed learning algorithm decomposes on the output. As a consequence, the results are

© 2026 R. Koenders & J. Moerman.

incomparable. For instance, the decomposition in fig. 1 cannot be expressed in the mentioned papers. We discuss the related work in more detail in section 6.1.

This work generalises the compositional method of Moerman (2018), by dropping the assumption that the output has to be a pair (or tuple) of outputs.

2. Preliminaries

We recall common definitions and notation on sets and words (or sequences). The set of all words with symbols in A is denoted by A^*, which includes the empty word $\varepsilon \in A^*$. Concatenation of two words u, v is written by juxtaposition uv. Given a function $f\colon A \to B$, we get an induced function $f^*\colon A^* \to B^*$, where f acts on each symbol in a word.

2.1. Finite State Machines

We focus on finite state machines with outputs on their transitions which are deterministic and fully specified.

Definition 1 (Mealy machine) *A* Mealy machine *is a tuple $M = (Q, X, Y, q_0, \delta, \lambda)$ where Q is a finite set of states; X is a finite set of inputs; Y is a finite set of outputs; $q_0 \in Q$ is an initial state; $\delta\colon Q \times X \to Q$ is a transition function; and $\lambda\colon Q \times X \to Y$ is an output function.*

In figures we depict transitions as $q \xrightarrow{a/y} q'$ whenever $\delta(q, a) = q'$ and $\lambda(q, a) = y$. We extend δ and λ to words in I^* in the usual inductive way:

$$\begin{aligned} \delta^*\colon Q \times X^* &\to Q & \qquad \lambda^*\colon Q \times X^* &\to Y^* \\ \delta^*(q, \varepsilon) &= q & \lambda^*(q, \varepsilon) &= \varepsilon \\ \delta^*(q, aw) &= \delta^*(\delta(q, a), w) & \lambda^*(q, aw) &= \lambda(q, a) \cdot \lambda^*(\delta(q, a), w) \end{aligned}$$

Definition 2 (Behaviour) *The* behaviour *of a Mealy machine $M = (Q, X, Y, q_0, \delta, \lambda)$ is defined as the function $[\![M]\!]\colon X^* \to Y^*$, assigning to each input word an output word:*

$$[\![M]\!](w) = \lambda^*(q_0, w) \quad \textit{for all } w \in X^*$$

Definition 3 (Equivalence) *Two Mealy machines M_1 and M_2 are* equivalent *if*

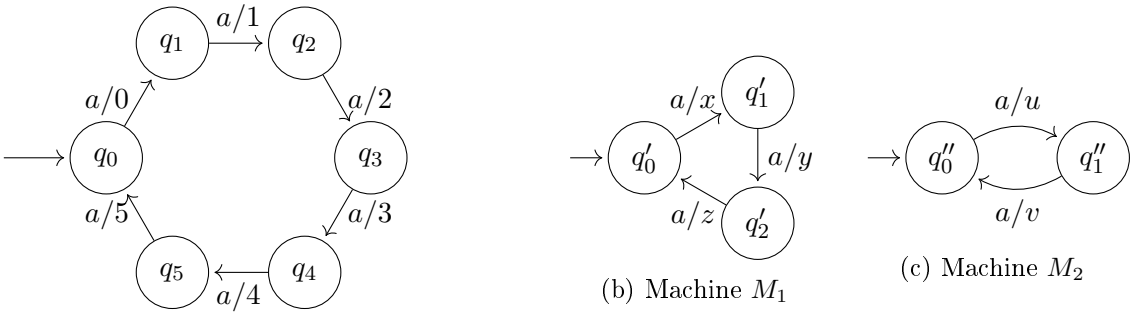
$$[\![M_1]\!](w) = [\![M_2]\!](w) \quad \textit{for all } w \in X^*\,.$$

2.2. Parallel Composition of Mealy machines

The parallel composition executes both machines in parallel, feeding the input to both machines and combining the output into a pair. This intuition is made precise in the following definition and lemma.

Definition 4 (Parallel composition) *Given two Mealy machines M_1 and M_2 with $M_i = (Q_i, X, Y_i, q_{0,i}, \delta_i, \lambda_i)$ we define the* parallel composition *as the Mealy machine $M_1 \times M_2 = (Q_1 \times Q_2, X, Y_1 \times Y_2, (q_{0,1}, q_{0,2}), \delta, \lambda)$ where*

$$\delta((q_1, q_2), x) = (\delta_1(q_1, x), \delta_2(q_2, x)) \qquad \lambda((q_1, q_2), x) = (\lambda_1(q_1, x), \lambda_2(q_2, x))\,.$$



(a) Machine N

(b) Machine M_1

(c) Machine M_2

Figure 1: Machine N (example 1) with 6 states can be decomposed as $M_1 \times M_2$, which has only 5 states.

The behaviour of a composition $M_1 \times M_2$ outputs sequences of type $(Y_1 \times Y_2)^*$. By projection we can recover one component: $\pi_i^* [\![M_1 \times M_2]\!](w) = [\![M_i]\!](w)$ for all $w \in X^*$. The parallel composition can be generalised to k Mealy machines M_i by iteratively performing the binary operation. The composition is associative (up to isomorphism).

The parallel composition is related to the *synchronous product* of processes. We make this relation explicit in the related work section (section 6.1).

3. Decomposition

In this section we present what it means to "undo" a composition. That is, we assume a finite state machine model N with inputs in X and outputs in Y, and we ask whether there are models $M_1, \ldots, M_k$ such that N is equivalent to $M_1 \times \cdots \times M_k$.

In the context of model learning, the system is observed and outputs are often given as opaque objects (e.g., byte strings). In particular, it is unreasonable to expect that the system is *exactly equal* to a composition with outputs in $Y_1 \times Y_2$. To accommodate this, we introduce a notion of equivalence which allows outputs to be relabelled.

Definition 5 *Two machines M, N with the same input set X and output sets Y_1, Y_2 respectively are* relabel-equivalent *if there are two injective maps $f_i \colon Y_i \to Z$ such that*

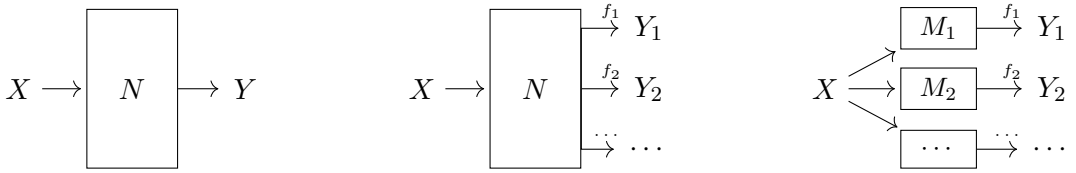
$$f_1^* [\![M]\!](w) = f_2^* [\![N]\!](w) \quad \text{for all } w \in X^* .$$

Example 1 *In fig. 1 an example is given of a decomposition. The original system N has 6 states in a cycle. The output cycles through the numbers 0 to 5.*

This system can be decomposed into two smaller cycles, with 3 and 2 states respectively. We relabel the outputs of N by

$$f_1(0) = (x,u) \quad f_1(1) = (y,v) \quad f_1(2) = (z,u) \quad f_1(3) = (x,v) \quad f_1(4) = (y,u) \quad f_1(5) = (z,v),$$

and keep the labels of $M_1 \times M_2$ fixed, i.e., $f_2 = \mathsf{id}$. Indeed, both maps are injective, which shows that N is relabel-equivalent to $M_1 \times M_2$.



(a) A state machine with an inputs in X and outputs in Y.

(b) A state machine producing outputs in separate parts.

(c) Multiple smaller state machines.

Figure 2: The main idea of decomposition using projections.

The above example is typical in our context: we want to describe the system under learning, N, as a relabel-equivalent composition $M_1 \times \cdots \times M_k$. In such cases, it is enough to relabel with $f : Y \to Y_1 \times \cdots \times Y_k$, where each Y_i is the output set of M_i, and take $g = \mathsf{id}$.

In good cases, the decomposition can be considerably smaller, and we distinguish two notions of "smaller".

Definition 6 (Size) *The size of a Mealy machine M, denoted by $|M|$, is the number of states $|M| = |Q|$.*[1] *The size of a composition $M_1 \times \cdots \times M_k$ is $\sum_{i=1}^{k} |M_i|$.*

Definition 7 *If N is relabel-equivalent to $M_1 \times \cdots \times M_k$, then we call N*

- strongly decomposable *if $\sum_{i=1}^{k} |M_i| < |N|$, and*
- weakly decomposable *if for all i we have $|M_i| < |N|$.*

A strong decomposition means that the model is strictly smaller when represented as a composition. A weak decomposition only means each individual component is smaller.

3.1. How to obtain a decomposition

We now describe a general recipe to obtain a decomposition. Suppose that N is relabel-equivalent to $M_1 \times \cdots \times M_k$, with corresponding injections $f : Y \to Z$ and $g : Y_1 \times \cdots \times Y_k \to Z$. Then, because the Mealy machines produce the same output after f and g, we know that $\operatorname{im} f = \operatorname{im} g$, and we can invert g to obtain an injective map $Y \to Y_1 \times \cdots \times Y_k$. Now, this map can be described by its components $Y \to Y_i$. This shows that every decomposition gives rise to a set of projection functions $f_i : Y \to Y_i$.

The converse also holds: If we have a set of functions $\{f_i : Y \to Y_i\}$, we can construct separate components of N by projecting the output on f_i, provided that the set of functions is jointly injective. This process of projecting into components is illustrated in fig. 2.

Definition 8 *A set of functions $\{f_1, \ldots, f_k\}$ with $f_i : Y \to Y_i$ is* jointly injective *if the product map $\langle f_i \rangle : Y \to \Pi_{i \in \{1,\ldots,k\}} Y_i$ given by $\langle f_i \rangle(y) = (f_1(y), \ldots, f_k(y))$ is injective.*

So in order to find a decomposition, it suffices to find a jointly injective set of functions. In certain situations, domain knowledge may be enough to determine a set which leads to small decompositions, but in a general closed-box setting, such domain knowledge is not present. In the following section we present an optimal decomposition using SAT solvers.

1. In our work, we keep the input set fixed, and for this reason we only care about the number of states.

3.1.1. Optimal decomposition.

By using a SAT solver, we can obtain an optimal decomposition, meaning that the number of states in each component is minimal.[2] For this optimal decomposition, the number of components k has to be fixed as part of the input. (This k can be automatically searched for, as it is bounded by the number of outputs.) We briefly discuss the reduction to SAT. Variables will be denoted by fraktur letters: $\mathfrak{a}, \mathfrak{b}, \mathfrak{c}$. The general idea is to guess an equivalence relation on outputs and a bisimulation on states for every component $i \in \{1, \ldots, k\}$.

Given a Mealy machine M with inputs in X and outputs in Y, we first guess the functions $f_1, \ldots, f_k$, as those suffice to find a decomposition. As it is not known how big the codomain of f_i is, we do not guess the function directly. Instead, the kernel $\ker(f) = \{(y_1, y_2) \in Y \times Y \mid f(y_1) = f(y_2)\}$ is guessed. The kernel for f_i is represented by a set of binary variables: $\{\mathfrak{a}_{i,y_1,y_2} \mid y_1, y_2 \in Y\}$. We require the following constraints:

- Reflexivity: $\mathfrak{a}_{i,y,y}$ holds for each $y \in Y$.
- Symmetry: $\mathfrak{a}_{i,y_1,y_2} \to \mathfrak{a}_{i,y_2,y_1}$ holds for each $y_1, y_2 \in Y$.
- Transitivity: $\mathfrak{a}_{i,y_1,y_2} \wedge \mathfrak{a}_{i,y_2,y_3} \to \mathfrak{a}_{i,y_1,y_3}$ holds for each $y_1, y_2, y_3 \in Y$.
- Jointly injective: for each pair (y_1, y_2) with $y_1 \neq y_2$, we require that at least one relation does not equate them: $\neg\mathfrak{a}_{1,y_1,y_2} \vee \neg\mathfrak{a}_{2,y_1,y_2} \vee \cdots \vee \neg\mathfrak{a}_{k,y_1,y_2}$.

Next, we guess relations $R_i \subseteq Q \times Q$ on the state space. These relations are encoded by the variables $\{\mathfrak{b}_{i,q,q'} \mid i \in \{1, \ldots, k\}, q, q' \in Q\}$. We require that each relation R_i is a bisimulation for component i. Here, we use the structure of the model.

$$\mathfrak{b}_{i,q,q'} \to \mathfrak{b}_{i,\delta(q,a),\delta(q',a)} \quad \text{for all } a \in X,\ q, q' \in Q,\ i \in \{1, \ldots, k\}$$

And we require that if the states are deemed equivalent in component i, then their outputs must be related by the kernel of f_i:

$$\mathfrak{b}_{i,q,q'} \to \mathfrak{a}_{i,\lambda(q,a),\lambda(q',a)} \quad \text{for all } a \in X,\ q, q' \in Q,\ i \in \{1, \ldots, k\}$$

Last, we pick a unique representative for each class of the equivalence relation R_i. For each state we introduce a variable $\mathfrak{c}_{r,q}$ denoting that q is the representative. To encode that the representatives are unique, we use any order on the states: $Q = \{q_1, \ldots, q_n\}$.

$$\mathfrak{c}_{i,q_l} \vee \mathfrak{b}_{i,q_l,q_1} \vee \cdots \vee \mathfrak{b}_{i,q_l,q_{l-1}} \quad \text{for all } l \in \{1, \ldots, n\}$$
$$\mathfrak{c}_{i,q} \wedge \mathfrak{c}_{i,q'} \to \neg\mathfrak{b}_{i,q,q'} \quad \text{for all } q, q' \in Q$$

With these variables, we can count the exact number of states of the decomposition. We use incremental SAT solving to find the smallest number of representatives which satisfy all the constraints. For SAT solving, we use PySAT by Ignatiev et al. (2024) and SAT4J by Berre and Parrain (2010).

4. Compositional Learning with Parallel Compositions

In this section, we explain our CoALP algorithm. This algorithm can learn the components of a Mealy machine while (optionally) simultaneously learning the projections. To learn the components themselves, we use *sub-learners*, which can be instantiated with any active automata learning algorithm, like L* or TTT.

2. Alternatively, for a strong decomposition we minimise $\sum_{i=1}^{k} |M_i|$.

4.1. The Minimally Adequate Teacher Framework

We briefly explain the basics of the *Minimally Adequate Teacher* framework, as introduced by Angluin (1987). The aim is to construct a Mealy machine M from a given function $\mathcal{L}\colon X^* \to Y^*$ such that $[\![M]\!] = \mathcal{L}$. The function $\mathcal{L}$ is "closed-box" and can only be interacted with as an oracle. The Minimally Adequate Teacher framework defines which queries the learner can ask the oracle, also called the teacher. The CoALP algorithm uses sub-learners for each projection function f, so we will also explain how we can adapt the queries for the sub-learners.

Firstly, a learner may ask the teacher for the output $\mathcal{L}(w)$ for each individual input word w. This is called an *output query (OQ)* (for DFAs, this is usually called a *membership query*). When a sub-learner i, which learns the function $f_i^* \circ \mathcal{L}$, asks for the output of w, our algorithm will forward the same query to the teacher. Given the response $\mathcal{L}(w)$ from the teacher, our algorithm provides $f_i^*(\mathcal{L}(w))$ to the sub-learner.

Additionally, we can ask *equivalence queries (EQ)* to check whether a hypothesised machine M is correct. If the hypothesis is incorrect, i.e., $[\![M]\!] \neq \mathcal{L}$, the teacher provides the learner with a counterexample $w \in X^*$. In our algorithm, we answer output queries until all sub-learners ask an equivalence query. As we will explain in section section 4.2, we will first check for consistency. Then CoALP asks an equivalence query itself by constructing the product of all hypotheses of the sub-learners. If the hypothesis is incorrect, we forward the counterexample w to the sub-learners.[3]

4.2. Consistency

Before describing our CoALP algorithm, we explain the two types of consistency we require the hypotheses of the sub-learners to have.

While executing CoALP, we may sometimes need to restart the sub-learners if we switch to a new set of projections. To prevent the loss of progress, we require the hypotheses to be consistent with a cache. In our case, our cache will consist of the inputs (and the corresponding cached outputs) for all output queries that have been asked so far.

Definition 9 (Cache consistency) *Given a set $C \subset X^*$ we say that a Mealy machine M is* cache consistent *(with C) if $[\![M]\!](w) = \mathcal{L}(w)$ for all $w \in C$. We call a set of machines $\{M_i\}$ with the projections $\{f_i\}$ cache consistent with C if each M_i is cache consistent with C for the function $f_i^* \circ \mathcal{L}$.*

Secondly, the hypothesis machines M_i may be in conflict with each other. The output of the composition would then not be (uniquely) defined. Therefore, we require that the components are consistent with each other.

Definition 10 (Component consistent) *Given projection functions $f_1, \ldots, f_k$, we say that Mealy machines $M_1, \ldots, M_k$ are* component consistent *if for every input word $w \in X^*$ there is exactly one output $y \in Y^*$ such that $f_i^*(y) = [\![M_i]\!](w)$ for every i.*

3. This assumes that the sub-learners can handle spurious counterexamples. If this assumption does not hold, we can first determine for which sub-learners $[\![H_i]\!](w) \neq f_i^*(\mathcal{L}(w))$ and only forward the counterexample to those sub-learners.

Algorithm 1: CoALP

Input: Input alphabet X, learning algorithm L, teacher providing OQs and EQs
Output: Decomposition equivalent to the target M
Create a cache of answered OQs;
repeat
  Determine jointly injective set of functions $\{f_1, \ldots, f_k\}$;
  if *the new functions yield a decomposition into smaller components than the previous functions (or this is the first loop)* **then**
    start a sub-learner L_i for each function f_i;
  end
  repeat
    When L_i asks an OQ for input w, ask the OQ to the teacher and add the response y to the cache; return $f_i^*(y)$ to L_i;
  until *all learners ask an EQ with hypothesis* H_i;
  if $H_1, \ldots, H_k$ *are not cache consistent or component consistent* **then**
    Find input w for which the inconsistency occurs and return w to the sub-learners;
  else
    Construct $H = H_1 \times \cdots \times H_k$;
    if $EQ(\langle f_i \rangle^{-1} H) = w$ **then**
      Return w to the sub-learners;
    end
  end
until $EQ(\langle f_i \rangle^{-1} H) = YES$;
return $H_1, \ldots, H_k$ and $f_1, \ldots, f_k$;

If $\{M_i\}$ is component consistent, $\langle f_i \rangle^{-1}(M_1 \times \ldots \times M_k)$ denotes the (uniquely defined) Mealy machine such that $f_i(\langle f_i \rangle^{-1}(M_1 \times \ldots \times M_k)) = M_i$ for all i.

4.3. The CoALP Algorithm

Algorithm 1 describes the CoALP algorithm in detail. Note that the algorithm is generic in the way the projection functions f_i are chosen and it is generic in the underlying learning algorithm for the sub-learners. We explain four ways to choose f_i's at the end of this section. The choice of f_i is allowed to change during the learning process.

Correctness of the CoALP algorithm follows from termination, since it only terminates if the teacher replies that the hypothesis is equivalent to the target. Theorem 11 says that CoALP terminates.

Theorem 11 *Given an algorithm to calculate a jointly injective set of functions* $\{f_i \colon Y \to Y_i\}_{i \in I}$ *that always terminates, the CoALP algorithm terminates and is correct.*

4.4. Choice of the projection functions

The CoALP algorithm can be instantiated in many different ways, by changing the way in which we determine the functions f_i. These fall into two broad categories:

1. Predetermine a good set of functions $\{f_1, \ldots, f_k\}$ with $f_i \colon Y \to Y_i$ and use those throughout the learning process.

(a) Using the single projection $\{\mathsf{id}\colon Y \to Y\}$ yields the underlying learning algorithm (such as L^*, TTT, $\mathsf{L}^\#$, ...).

(b) If we know that the output set is a product $Y_1 \times Y_2$, one can use the projections $\{\pi_i\colon Y_1 \times Y_2 \to Y_i \mid i \in \{1,2\}\}$. This yields Algorithm 2 from Moerman (2018).

(c) Another option is to compute an optimal decomposition (from section 3.1.1) on a model which is believed to be similar to the one being learnt.

2. Compute the functions $\{f_1, \ldots, f_k\}$ *on-the-fly*. For example, we can use the optimal decomposition method described in section 3.1.1 *during learning*. In this case, we compute the optimal decomposition on the hypothesis from the previous loop and use the result during the current loop. Then, the set of projection functions may change each time an equivalence query is posed. To determine the number of components, we try to decompose the machine into two components first and retry with one more component each time we find a decomposition into smaller components.

Though theorem 11 says that all instantiations of CoALP will eventually learn the correct machine, the efficiency by which they do so may differ greatly. In the evaluation, we compare the underlying algorithm (1a), the case of a pre-computed optimal decomposition (1c), and the optimal on-the-fly decomposition (2).

5. Empirical Evaluation

In this section, we will compare the efficiency of learning in a compositional manner with learning in the standard non-compositional manner.

5.1. Models and decomposability

We use the models from the AutomataWiki by Neider et al. (2018) to evaluate the applicability of compositionality in model learning. The wiki consists of many finite state machine models obtained from real-world case studies. Besides these real-world models we added models constructed specifically as parallel compositions (see section 5.1.1).

An overview of the models is given in table 1. The table also reports whether the models have a decomposition or not. Some of the composed models constructed for this paper do not have a decomposition. This is because there are a few small instances in which the random generation happens to generate trivial components.

		has strong decomp.		has weak decomp.	
Benchmark	# of models	#	pct.	#	pct.
Parallel compositions	88	85	97%	85	97%
AutomataWiki (Protocols)	102	3	3%	52	51%
AutomataWiki (Other)	289	42	15%	121	42%
Labbaf et al.	197	178	90%	185	94%

Table 1: Finite state machine models and their decomposability. The first two classes are used in the evaluation of CoALP.

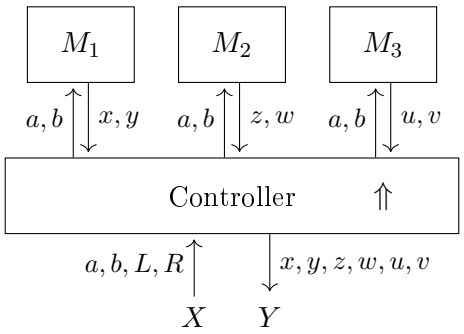


Figure 3: Schematic of our constructed Mealy machine with $k = 3$ components. Each M_i is randomly generated and a controller determines the active component (indicated by ⇑).

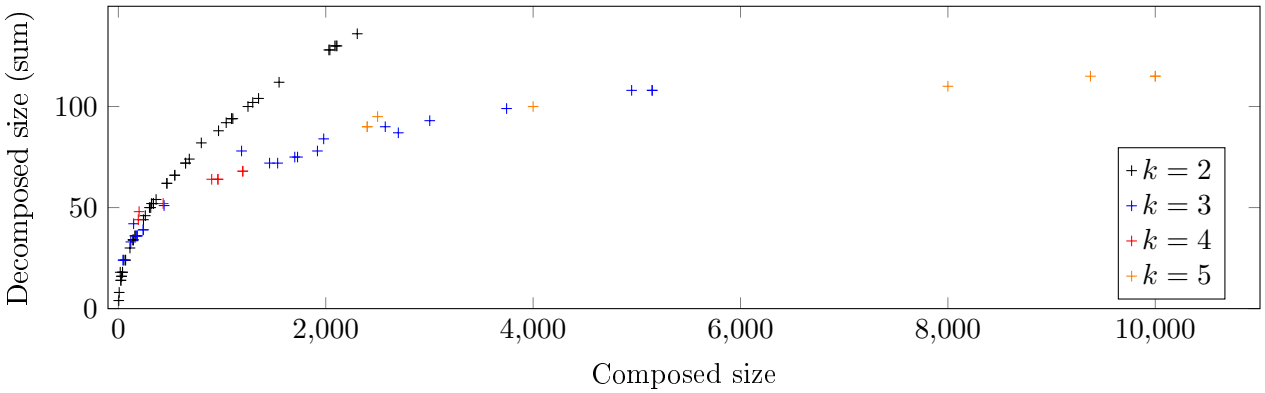


Figure 4: The size (number of states) of each constructed benchmark with k components.

We analysed models from Labbaf et al. (2023) on decomposability. Surprisingly, these models are almost always decomposable, although they are constructed as *interleaving composition* (where the *input* set is decomposed into subsets). More recent work by Samadi et al. (2025) uses similar models. The models from the AutomataWiki often have no decomposition, because these applications focussed on a smaller part of the system.

5.1.1. Parallel composition models

We briefly explain how we generated models with a parallel composition. First, multiple randomly generated components are generated with a common input alphabet $\{a, b\}$, but all with unique outputs, see fig. 3. A controller keeps track of which component is active, and uses special inputs L and R for switching between which component is active. The inputs $\{a, b\}$ are only sent to the active component; an inactive component remains in its state. The benchmark consists of machines with 2 to 5 components, with each component having 5 to 40 states. The biggest Mealy machine in this set, when composed, has 10000 states. The sizes, together with the size of the decomposition, are plotted in fig. 4.

5.2. Experimental set-up

Now we present the experiments we carried out to evaluate our CoALP. We use TTT as a baseline and also as the underlying algorithm in our new algorithm. We evaluate two instantiations of CoALP based on weak decompositions: 1) with optimal, precomputed projections, and 2) with projections computed on-the-fly. The experiments were run using our implementation[4] of the algorithm using the LearnLib framework by Isberner et al. (2015).

The number of components used by the instantiations of CoALP using a weak decomposition for this second benchmark varies from 1 to 3, as using more than 3 components yielded no further reduction in the size of the largest component.

As the equivalence oracle, we used randomised testing based on the Wp-method by Fujiwara et al. (1991) as implemented in LearnLib. Tests required to find counterexamples are included in the number of queries. This is done to mimic a real-world scenario where no equivalence oracle (other than testing) exists. To reduce the impact of the randomisation on our results, each instance was repeated 20 times and we take the median. Queries are cached, so that duplicate tests are not counted twice. A deterministic equivalence check is done before the call to the equivalence oracle, ensuring exact equivalence of the learned model and eliminating the final call to the equivalence oracle completely.

5.3. Results

Figure 5 shows the number of queries required to learn a model, comparing TTT with the two versions of CoALP. As expected, CoALP performs very well on our constructed benchmark. In the bigger models (with $k = 5$ components), the number of queries is reduced by three orders of magnitude.

The on-the-fly variant performs worse (but better than TTT), as it learns the decomposition as well. In some bigger cases the SAT-solving produces an out-of-memory error.

On the realistic benchmark with protocols, CoALP shows little improvement over TTT, except for a few exceptionally good cases. This can be explained by the fact that many of these models do not have a decomposition. In those cases, we see that CoALP with the precomputed projections performs exactly the same as TTT. There is a little bit of noise, because the equivalence queries are implemented by random testing.

For the on-the-fly decomposition of protocols, we see that there are instances where it performs worse than TTT. So our proposed algorithm sometimes comes at a cost.

6. Conclusion and Discussion

We presented CoALP, a compositional learning algorithm which learns a decomposition of a Mealy machine. The decomposition is based on projections of the output, and the learning algorithm can learn the projection on-the-fly without any domain-specific knowledge. Currently, the algorithm searches for an optimal decomposition (minimal number of states), by using a SAT solver.

The performance of CoALP is evaluated on models from the AutomataWiki and new composed models. Clearly, CoALP works well on the composed models, even if it has to learn the decomposition on-the-fly. The number of queries can be reduced by a few orders of

4. The code is available at https://github.com/SCRK16/OLStar.

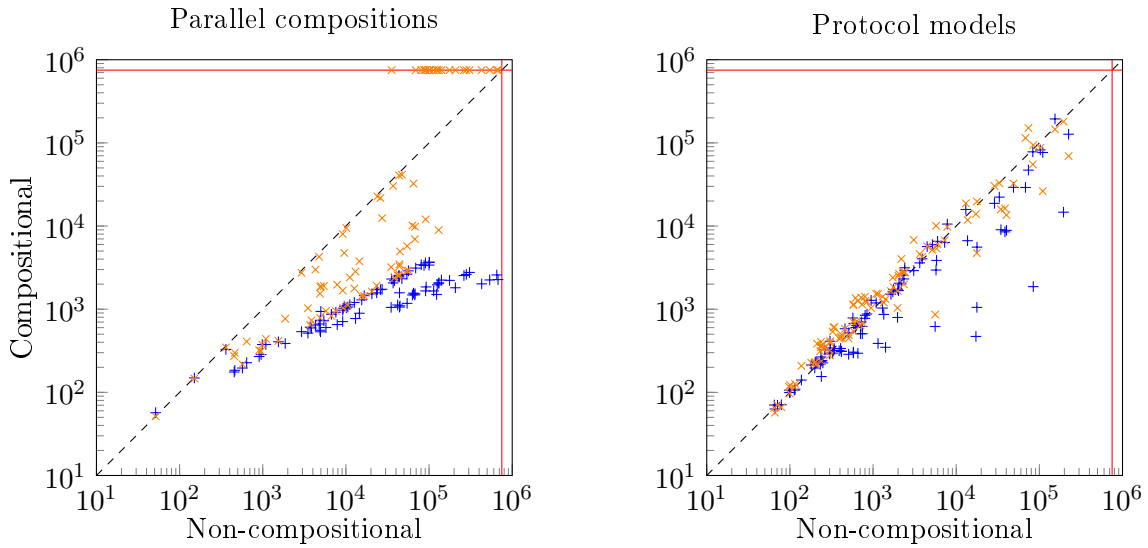


Figure 5: Performance of our compositional learning algorithm (y-coordinate) compared to TTT (x-coordinate). Performance is measured by the number of queries. Instances below the dashed $y = x$ line are in favour of CoALP. In blue (+) the results with precomputed decomposition and in orange (×) the results with the on-the-fly algorithm. Instances which result in an out-of-memory are plotted on the red lines.

magnitude. For the realistic models which are not decomposable, we observe an additional cost for learning a decomposition on-the-fly.

Last, we remark that table 1 shows that many models are decomposable, even if they are not intentionally built that way. For instance, models in the AutomataWiki are often learnt from small sub-systems, because the algorithms could not scale to the full system. Yet, some of these models are decomposable.

We conclude the paper by comparing our results with related work, and by mentioning further research.

6.1. Related Work

Closest to our goals is the work of Henry et al. (2025), a continuation of Neele and Sammartino (2023). They also report a great reduction in the number of queries, and they also use SAT solvers. However, they use labelled transition systems (LTSs) to describe the learnt model(s) and this cannot express the type of (de)composition we consider. Concretely, none of the papers can deal with the decomposition as shown in fig. 1. For this example, it is crucial to consider a decomposition on outputs, as opposed to decomposing the input set.

In these papers, the composition is called *synchronous product*, as the components synchronise on the input. This is also the case in our composition. The difference is that we allow the corresponding outputs to be different and combined in an arbitrary way. For this reason, we do not use the term synchronous product.

When using LTSs to describe the model, no distinction is made between inputs and outputs; there is a single set of actions. So, another difference compared to our learning

algorithm is that their algorithm assumes control over all actions. In our work, we assume control only over the input actions, and the system under learning has control over the output. For that reason, we cannot directly use their models for evaluation.

The work of Labbaf et al. (2023) and Samadi et al. (2025) *do* use Mealy machines as their model, but they still only decompose the input set (similar to the work on LTSs described above). Although their type of (de)composition is very different compared to ours, their models also decompose as a parallel composition. We hope to combine the two approaches (decomposition on inputs vs. outputs), as discussed in section 6.2.

Parallel composition as we consider it, has been considered by Moerman (2018). Our work is a direct generalisation of this earlier work, and does not assume the compositional structure on the output set is known in advance.

Another type of composition, which is specific to Mealy machines, is the work of Abel and Reineke (2016). Here, two Mealy machines are put in sequence, feeding the output of one machine as input into a second machine. This connection is assumed to be internal and not observable. This is in contrast with the algorithms of Neele and Sammartino (2023) and Samadi et al. (2025) where actions on interfaces are observable. In a functional composition, it is not possible to directly interact with the second machine, and a SAT solver is used to construct a model which is consistent.

We also want to mention the work of Frohme and Steffen (2021) which learns a context-free grammar, modelled as a *system of procedural automata*. In a way, each automaton in their model is a component, and their result can be seen as compositional learning. The type of composition is sequential, as opposed to parallel (or interleaving). Another difference is that they require the input set to contain special symbols, which show when another component is called.

6.2. Future work

Finding a good decomposition during learning is difficult and now solved by a SAT solver. However, this gets infeasible for bigger models. Other ways of computing a set of jointly injective functions should be investigated, possibly heuristically. Alternatively, one could try to reduce the number of calls to the SAT solver during learning. The current algorithm optimises the decomposition at every equivalence query.

We do not know whether finding the optimal decomposition is NP-hard. It is similar to the open problem of checking whether a regular language is composite (Kupferman and Mosheiff, 2015). The decomposition considered in our work relates to *simple decompositions* of *op. cit.*. However, unlike their setting in which components are always combined with boolean conjunction, our setting does not fix the functions used in composition.

Another possibility is to combine decomposition based on outputs with other forms of decomposition to reduce the size of components even further. Decomposition based on inputs, as discussed in related work (section 6.1), is dual to our approach and we think that a combination of the methods is possible, giving the best of both worlds.

Finally, we realise that the equivalence oracle does not leverage compositionality; it only checks whether the composed hypothesis is correct. It would be interesting to make the Minimally Adequate Teacher framework flexible and allow queries about the (de)composition

structure. Concretely, the equivalence oracle could be improved by incorporating the compositional structure to generate tests more cleverly.

Acknowledgments

An earlier version of this paper appeared at the LearnAut'24 workshop. We have since generalised the type of decomposition and moved from an extension of L* to a generalised algorithm. We would like to thank the anonymous reviewers and the participants of the workshop for their useful comments and discussions. We also thank Hossein Hojjat and Thomas Neele for discussions on their approaches to compositional learning.

References

Andreas Abel and Jan Reineke. Gray-box learning of serial compositions of mealy machines. In *NFM*, volume 9690 of *Lecture Notes in Computer Science*, pages 272–287. Springer, 2016.

Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75 (2):87–106, 1987. URL https://doi.org/10.1016/0890-5401(87)90052-6.

Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *J. Satisf. Boolean Model. Comput.*, 7(2-3):59–6, 2010.

Paul Fiterau-Brostean, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. Automata-based automated detection of state machine bugs in protocol implementations. In *NDSS*. The Internet Society, 2023. URL https://www.ndss-symposium.org/ndss-paper/automata-based-automated-detection/.

Markus Frohme and Bernhard Steffen. Compositional learning of mutually recursive procedural systems. *Int. J. Softw. Tools Technol. Transf.*, 23(4):521–543, 2021. URL https://doi.org/10.1007/s10009-021-00634-y.

Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Trans. Software Eng.*, 17(6):591–603, 1991. URL https://doi.org/10.1109/32.87284.

Léo Henry, Mohammad Reza Mousavi, Thomas Neele, and Matteo Sammartino. Compositional active learning of synchronizing systems through automated alphabet refinement. In *CONCUR*, volume 348 of *LIPIcs*, pages 20:1–20:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025.

Alexey Ignatiev, Zi Li Tan, and Christos Karamanos. Towards universally accessible SAT technology. In *SAT*, pages 4:1–4:11, 2024. doi: 10.4230/LIPICS.SAT.2024.16. URL https://doi.org/10.4230/LIPIcs.SAT.2024.16.

Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In *RV*, volume 8734 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 2014. URL https://doi.org/10.1007/978-3-319-11164-3_26.

Malte Isberner, Falk Howar, and Bernhard Steffen. The open-source learnlib - A framework for active automata learning. In *CAV (1)*, volume 9206 of *Lecture Notes in Computer Science*, pages 487–495. Springer, 2015. URL https://doi.org/10.1007/978-3-319-21690-4_32.

Orna Kupferman and Jonathan Mosheiff. Prime languages. *Inf. Comput.*, 240:90–107, 2015.

Faezeh Labbaf, Jan Friso Groote, Hossein Hojjat, and Mohammad Reza Mousavi. Compositional learning for interleaving parallel automata. In *FoSSaCS*, volume 13992 of *Lecture Notes in Computer Science*, pages 413–435. Springer, 2023. URL https://doi.org/10.1007/978-3-031-30829-1_20.

Joshua Moerman. Learning product automata. In *ICGI*, volume 93 of *Proceedings of Machine Learning Research*, pages 54–66. PMLR, 2018. URL http://proceedings.mlr.press/v93/moerman19a.html.

Thomas Neele and Matteo Sammartino. Compositional automata learning of synchronous systems. In *FASE*, volume 13991 of *Lecture Notes in Computer Science*, pages 47–66. Springer, 2023. URL https://doi.org/10.1007/978-3-031-30826-0_3.

Daniel Neider, Rick Smetsers, Frits W. Vaandrager, and Harco Kuppens. Benchmarks for automata learning and conformance testing. In *Models, Mindsets, Meta*, volume 11200 of *Lecture Notes in Computer Science*, pages 390–416. Springer, 2018.

Mahboubeh Samadi, Aryan Bastany, and Hossein Hojjat. Compositional learning for synchronous parallel automata. *Fundamental Approaches to Software Engineering LNCS 15693*, page 101, 2025.

Lisette Sanchez, Jan Friso Groote, and Ramon R. H. Schiffelers. Active learning of industrial software with data. In *FSEN*, volume 11761 of *Lecture Notes in Computer Science*, pages 95–110. Springer, 2019. URL https://doi.org/10.1007/978-3-030-31517-7_7.

Frits W. Vaandrager. Model learning. *Commun. ACM*, 60(2):86–95, 2017. URL https://doi.org/10.1145/2967606.