Proceedings of Machine Learning Research 93:54–66, 2019 International Conference on Grammatical Inference

Learning Product Automata

Joshua Moerman JOSHUA.MOERMAN@CS.RU.NL
Institute for Computing and Information Sciences
Radboud University
Nijmegen, the Netherlands

Editors: Olgierd Unold, Witold Dyrka, and Wojciech Wieczorek

Abstract

We give an optimisation for active learning algorithms, applicable to learning Moore machines with decomposable outputs. These machines can be decomposed themselves by projecting on each output. This results in smaller components that can then be learnt with fewer queries. We give experimental evidence that this is a useful technique which can reduce the number of queries substantially. Only in some cases the performance is worsened by the slight overhead. Compositional methods are widely used throughout engineering, and the decomposition presented in this article promises to be particularly interesting for learning hardware systems.

Keywords: query learning, product automata, composition

1. Introduction

Query learning (or, active learning) is becoming a valuable tool in engineering of both hardware and software systems (Vaandrager, 2017). Indeed, applications can be found in a broad range of applications: finding bugs in network protocols as shown by Fiterău-Broştean et al. (2016, 2017), assisting with refactoring legacy software as shown by Schuts et al. (2016), and reverse engineering bank cards by Chalupar et al. (2014).

These learning techniques originate from the field of grammatical inference. One of the crucial steps for applying these to black box systems was to move from deterministic finite automata to deterministic Moore or Mealy machines, capturing reactive systems with any kind of output. With little adaptations, the algorithms work well, as shown by the many applications. This is remarkable, since little specific knowledge is used beside the input alphabet of actions.

Realising that composition techniques are ubiquitous in engineering, we aim to use more structure of the system during learning. In the present paper we use the simplest type of composition; we learn product automata, where outputs of several components are simply paired. Other types of compositions, such as sequential composition of Mealy machines, are discussed in Section 5.

To the best of the author's knowledge, this has not been done before explicitly. Furthermore, libraries such as LearnLib (see Isberner et al., 2015) and libalf (see Bollig et al., 2010b) do not include such functionality out of the box. *Implicitly*, however, it has been done before. Rivest and Schapire (1994) use two tricks to reduce the size of some automata in their paper "Diversity-based inference of finite automata". The first trick is to look at

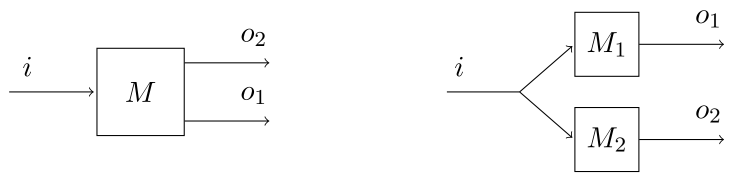

Figure 1: A Moore machine with two outputs (left) can be equivalently seen as two (potentially smaller) Moore machines with a single output each (right).

the *reversed automaton* (in their terminology, the *diversity-based automaton*). The second trick (which is not explicitly mentioned, unfortunately) is to have a different automaton for each observable (i.e., output). In one of their examples the two tricks combined give a reduction from $\pm 10^{19}$ states to just 54 states.

In this paper, we isolate this trick and use it in query learning. We give an extension of `L*` which handles products directly and we give a second algorithm which simply runs two learners simultaneously. Furthermore, we argue that this is particularly interesting in the context of model learning of hardware, as systems are commonly engineered in a compositional way. We give preliminary experimental evidence that the technique works and improves the learning process. As benchmarks, we learn (simulated) circuits which provide several output bits.

2. Preliminaries

We use the formalism of Moore machines to describe our algorithms. Nonetheless, the results can also be phrased in terms of Mealy machines.

Definition 1 *A* Moore machine *is a tuple $M = (Q, I, O, \delta, o, q_0)$ where Q, I and O are finite sets of states, inputs and outputs respectively, $\delta \colon Q \times I \to Q$ is the transition function, $o \colon Q \to O$ is the output function, and $q_0 \in Q$ is the initial state. We define the size of the machine, $|M|$, to be the cardinality of Q.*

We extend the definition of the transition function to words as $\delta \colon Q \times I^* \to Q$. The *behaviour* of a state q is the map $[\![q]\!] \colon I^* \to O$ defined by $[\![q]\!](w) = o(\delta(q, w))$. Two states q, q' are *equivalent* if $[\![q]\!] = [\![q']\!]$. A machine is *minimal* if all states have different behaviour and all states are reachable. We will often write $[\![M]\!]$ to mean $[\![q_0]\!]$ and say that machines are equivalent if their initial states are equivalent.

Definition 2 *Given two Moore machines with equal input sets, $M_1 = (Q_1, I, O_1, \delta_1, o_1, q_{01})$ and $M_2 = (Q_2, I, O_2, \delta_2, o_2, q_{02})$, we define their* product *$M_1 \times M_2$ by:*

$$M_1 \times M_2 = (Q_1 \times Q_2, I, O_1 \times O_2, \delta, o, (q_{01}, q_{02})),$$

where $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ and $o((q_1, q_2)) = (o_1(q_1), o_2(q_2))$.

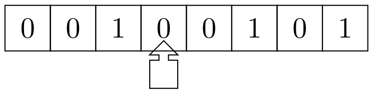

Figure 2: A state of the 8-bit register machine.

The product is formed by running both machines in parallel and letting I act on both machine synchronously. The output of both machines is observed. Note that the product Moore machine might have unreachable states, even if the components are reachable. The product of more than two machines is defined by induction.

Let M be a machine with outputs in $O_1 \times O_2$. By post-composing the output function with projection functions we get two machines, called *components*, M_1 and M_2 with outputs in O_1 and O_2 respectively. This is depicted in Figure 1. Note that M is equivalent to $M_1 \times M_2$. If M and its components M_i are taken to be minimal, then we have $|M| \leq |M_1| \cdot |M_2|$ and $|M_i| \leq |M|$. In the best case we have $|M_i| = \sqrt{|M|}$ and so the behaviour of M can be described using only $2\sqrt{|M|}$ states, which is less than $|M|$ (if $|M| > 4$). With iterated products the reduction can be more substantial as shown in the following example. This reduction in state space is beneficial for learning algorithms.

We introduce basic notation: $\pi_i \colon A_1 \times A_2 \to A_i$ are the usual projection functions. On a function $f \colon X \to A_1 \times A_2$ we use the shorthand $\pi_i f$ to denote $\pi_i \circ f$. As usual, uv denotes concatenation of string u and v, and this is lifted to sets of strings $UV = \{uv \mid u \in U, v \in V\}$. We define the set $[n] = \{1, \dots, n\}$ and the set of Boolean values $\mathbb{B} = \{0, 1\}$.

2.1. Example

We take the *n-bit register machine* example from Rivest and Schapire (1994). The state space of the n-bit register machine M_n is given by n bits and a position of the reading/writing head, see Figure 2. The inputs are commands to control the position of the head and to flip the current bit. The output is the current bit vector. Formally it is defined as $M_n = (\mathbb{B}^n \times [n], \{L, R, F\}, \mathbb{B}^n, \delta, o, i)$, where the initial state is $i = ((0, \dots, 0), 1)$ and the output is $o(((b_1, \dots, b_n), k)) = (b_1, \dots, b_n)$. The transition function is defined such that L moves the head to the left, R moves the head to the right (and wraps around on either ends), and F flips the current bit. Formally,

$$\begin{aligned}
\delta(((b_1, \dots, b_n), k), L) &= \begin{cases} ((b_1, \dots, b_n), k-1) & \text{if } k > 1, \\ ((b_1, \dots, b_n), n) & \text{if } k = 1, \end{cases} \\
\delta(((b_1, \dots, b_n), k), R) &= \begin{cases} ((b_1, \dots, b_n), k+1) & \text{if } k < n, \\ ((b_1, \dots, b_n), 1) & \text{if } k = n, \end{cases} \\
\delta(((b_1, \dots, b_n), k), F) &= ((b_1, \dots, \neg b_k, \dots, b_n), k).
\end{aligned}$$

The machine M_n is minimal and has $n \cdot 2^n$ states. So although this machine has very simple behaviour, learning it will require a lot of queries because of its size. Luckily, the machine can be decomposed into smaller components. For each bit l, we define a component $M_n^l = (\mathbb{B} \times [n], \{L, R, F\}, \mathbb{B}, \delta^l, \pi_1, (0, 1))$ which only stores one bit and the head position. The transition function δ^l is defined similarly as before on L and R, but only flips the bit on L if the head is on position l (i.e., $\delta^l((b, l), F) = (\neg b, l)$ and $\delta^l((b, k)) = (b, k)$ if $k \neq l$).

The product $M_n^1 \times \cdots \times M_n^n$ is equivalent to M_n. Each of the components M_n^l is minimal and has only $2n$ states. So by this decomposition, we only need $2 \cdot n^2$ states to describe the whole behaviour of M_n. Note, however, that the product $M_n^1 \times \cdots \times M_n^n$ is not minimal; many states are unreachable.

3. Learning

We describe two approaches for active learning of product machines. One is a direct extension of the well-known `L*` algorithm. The other reduces the problem to any active learning algorithm, so that one can use more optimised algorithms.

We fix an unknown target machine M with a known input alphabet I and output alphabet $O = O_1 \times O_2$. The goal of the learning algorithm is to infer a machine equivalent to M, given access to a *minimally adequate teacher* as introduced by Angluin (1987). The teacher will answer the following two types of queries.

- *Membership queries* (MQs): The query consists of a word $w \in I^*$ and the teacher will answer with the output $[\![M]\!](w) \in O$.
- *Equivalence queries* (EQs): The query consists of a Moore machine H, the *hypothesis*, and the teacher will answer with `YES` if M and H are equivalent and she will answer with a word w such that $[\![M]\!](w) \neq [\![H]\!](w)$ otherwise.

3.1. Learning product automata with an `L*` extension

We can use the general framework for automata learning as set up by van Heerdt et al. (2017). The general account does not directly give concrete algorithms, but it does give generalised definitions for *closedness* and *consistency*. The main data structure for the algorithm is an observation table.

Definition 3 *An* observation table *is a triple (S, E, T) where $S, E \subseteq I^*$ are finite sets of words and $T \colon S \cup SI \to O^E$ is defined by $T(s)(e) = [\![M]\!](se)$.*

During the `L*` algorithm the sets S, E grow and T encodes the knowledge of $[\![M]\!]$ so far.

Definition 4 *Let (S, E, T) be an observation table.*

- *The table is* product-closed *if for all $t \in SI$ there exist $s_1, s_2 \in S$ such that $\pi_i T(t) = \pi_i T(s_i)$ for $i = 1, 2$.*
- *The table is* product-consistent *if for $i = 1, 2$ and for all $s, s' \in S$ we have $\pi_i T(s) = \pi_i T(s')$ implies $\pi_i T(sa) = \pi_i T(s'a)$ for all $a \in I$.*

These definitions are related to the classical definitions of closedness and consistency as shown in the following lemma. The converses of the first two points do not necessarily hold. We also proof that if a observation table is product-closed and product-consistent, then a well-defined product machine can be constructed which is consistent with the table.

Algorithm 1 The product-L* algorithm.

```
 1: Initialise $S$ and $E$ to $\{\epsilon\}$
 2: Initialise $T$ with MQs
 3: repeat
 4:    while $(S, E, T)$ is not product-closed or -consistent do
 5:       if $(S, E, T)$ not product-closed then
 6:          find $t \in SI$ such that there is no $s \in S$ with $\pi_i T(t) = \pi_i T(s)$ for some $i$
 7:          add $t$ to $S$ and fill the new row using MQs
 8:       if $(S, E, T)$ not product-consistent then
 9:          find $s, s' \in S$, $a \in I$ and $e \in E$ such that $\pi_i T(s) = \pi_i T(s')$ but $\pi_i T(sa)(e) \neq$
             $\pi_i T(s'a)(e)$ for some $i$
10:          add $ae$ to $E$ and fill the new column using MQs
11:    Construct $H$ (by Lemma 6)
12:    if $EQ(H)$ gives a counterexample $w$ then
13:       add $w$ and all its prefixes to $S$
14:       fill the new rows with MQs
15: until $EQ(H) =$ YES
16: return $H$
```

Lemma 5 *Let $OT = (S, E, T)$ be an observation table and let $\pi_i OT = (S, E, \pi_i T)$ be a component. The following implications hold.*

1. *OT is closed $\implies$ OT is product-closed.*
2. *OT is consistent $\impliedby$ OT is product-consistent.*
3. *OT is product-closed $\iff$ $\pi_i OT$ is closed for each i.*
4. *OT is product-consistent $\iff$ $\pi_i OT$ is consistent for each i.*

Proof (1) If OT is closed, then each $t \in SI$ has a $s \in S$ such that $T(t) = T(s)$. This implies in particular that $\pi_i T(t) = \pi_i T(s)$, as required. (In terms of the definition, this means we can take $s_1 = s_2 = s$.)

(2) Let OT be product-consistent and $s, s' \in S$ such that $T(s) = T(s')$. We then know that $\pi_i T(s) = \pi_i T(s')$ for each i and hence $\pi_i T(sa) = \pi_i T(s'a)$ for each i and a. This means that $T(sa) = T(s'a)$ as required.

Statements (3) and (4) just rephrase the definitions. ■

Lemma 6 *Given a product-closed and -consistent table we can define a product Moore machine consistent with the table, where each component is minimal.*

Proof If the table OT is product-closed and -consistent, then by the previous lemma, the tables $\pi_i OT$ are closed and consistent in the usual way. For these tables we can use the construction of Angluin (1987). As a result we get a minimal machine H_i which is consistent with table $\pi_i OT$. Taking the product of these gives a machine which is consistent with OT. (Beware that this product is not necessarily the minimal machine consistent with OT.) ■

Algorithm 2 Learning product machines with other learners.

```
1: Initialise two learners $L_1$ and $L_2$
2: repeat
3:   while $L_i$ queries $MQ(w)$ do
4:     forward $MQ(w)$ to the teacher and get output $o$
5:     return $\pi_i o$ to $L_i$
   {at this point both learners constructed a hypothesis}
6:   Let $H_i$ be the hypothesis of $L_i$
7:   Construct $H = H_1 \times H_2$
8:   if $EQ(H)$ returns a counterexample $w$ then
9:     if $[\![H_1]\!](w) \neq \pi_1 [\![M]\!](w)$ then
10:      return $w$ to $L_1$
11:    if $[\![H_2]\!](w) \neq \pi_2 [\![M]\!](w)$ then
12:      return $w$ to $L_2$
13: until $EQ(H) =$ YES
14: return YES to both learners
15: return $H$
```

The product-`L*` algorithm (Algorithm 1) resembles the original `L*` algorithm, but uses the new notions of closed and consistent. Its termination follows from the fact that `L*` terminates on both components.

By Lemma 5 (1) we note that the algorithm does not need more rows than we would need by running `L*` on M. By point (4) of the same lemma, we find that it does not need more columns than `L*` would need on each component combined. This means that in the worst case, the table is twice as big as the original `L*` would do. However, in good cases (such as the running example), the table is much smaller, as the number of rows is less for each component and the columns needed for each component may be similar.

3.2. Learning product automata via a reduction

The previous algorithm constructs two machines from a single table. This suggests that we can also run two learning algorithms to construct two machines. We lose the fact that the data structure is shared between the learners, but we gain that we can use more efficient algorithms than `L*` without any effort.

Algorithm 2 is the algorithm for learning product automata via this reduction. It runs two learning algorithms at the same time. All membership queries are passed directly to the teacher and only the relevant output is passed back to the learner. (In the implementation, the query is cached, so that if the other learner poses the same query, it can be immediately answered.) If both learners are done posing membership queries, they will pose an equivalence query at which point the algorithm constructs the product automaton. If the equivalence query returns a counterexample, the algorithm forwards it to the learners.

The crucial observation is that a counterexample is necessarily a counterexample for at least one of the two learners. (If at a certain stage only one learner makes an error, we keep the other learner suspended, as we may obtain a counterexample for that one later on.)

This observation means that at least one of the learners makes progress and will eventually converge. Hence, the whole algorithm will converge.

In the worst case, twice as many queries will be posed, compared to learning the whole machine at once. (This is because learning the full machine also learns its components.) In good cases, such as the running example, it requires much less queries. Typical learning algorithms require roughly $\mathcal{O}(n^2)$ membership queries, where n is the number of states in the minimal machine. For the example M_n this bound gives $\mathcal{O}((n \cdot 2^n)^2) = \mathcal{O}(n^2 \cdot 2^{2n})$ queries. When learning the components M_n^l with the above algorithm, the bound gives just $\mathcal{O}((2n)^2 + \cdots + (2n)^2) = \mathcal{O}(n^3)$ queries.

4. Experiments

We have implemented the algorithm via reduction in LearnLib.[1] As we expect the reduction algorithm to be the most efficient and simpler, we leave an implementation of the direct extension of `L*` as future work. The implementation handles products of any size (as opposed to only products of two machines). Additionally, the implementation also works on Mealy machines and this is used for some of the benchmarks.

In this section, we compare the product learner with a regular learning algorithm. We use the `TTT` algorithm by Isberner et al. (2014) for the comparison and also as the learners used in Algorithm 2. We measure the number of membership and equivalence queries. The results can be found in Table 1.

The equivalence queries are implemented by random sampling so as to imitate the intended application of learning black-box systems. This way, an exact learning algorithm turns into a PAC (probably approximately correct) algorithm. Efficiency is typically measured by the total number of input actions which also accounts for the length of the membership queries (including the resets). This is a natural measure in the context of learning black box systems, as each action requires some amount of time to perform.

We evaluated the product learning algorithm on the following two classes of machines.

n-bit register machine The machines M_n are as described before. We note that the product learner is much more efficient, as expected.

Circuits In addition to the (somewhat artificial) examples M_n, we use circuits which appeared in the logic synthesis workshops (LGSynth89/91/93), part of the ACM/SIGDA benchmarks.[2] These models have been used as benchmarks before for FSM-based testing methods by Hierons and Türker (2015) and describe the behaviour of real-world circuits. The circuits have bit vectors as outputs, and can hence be naturally be decomposed by taking each bit individually. As an example, Figure 3 depicts one of the circuits (`bbara`). The behaviour of this particular circuit can be modelled with seven states, but when restricting to each individual output bit, we obtain two machines of just four states. For the circuits `bbsse` and `mark1`, we additionally regrouped bit together in order to see how the performance changes when we decompose differently.

1. The implementation and models can be found on-line at https://gitlab.science.ru.nl/moerman/learning-product-automata.
2. The original files describing these circuits can be found at https://people.engr.ncsu.edu/brglez/CBL/benchmarks/.

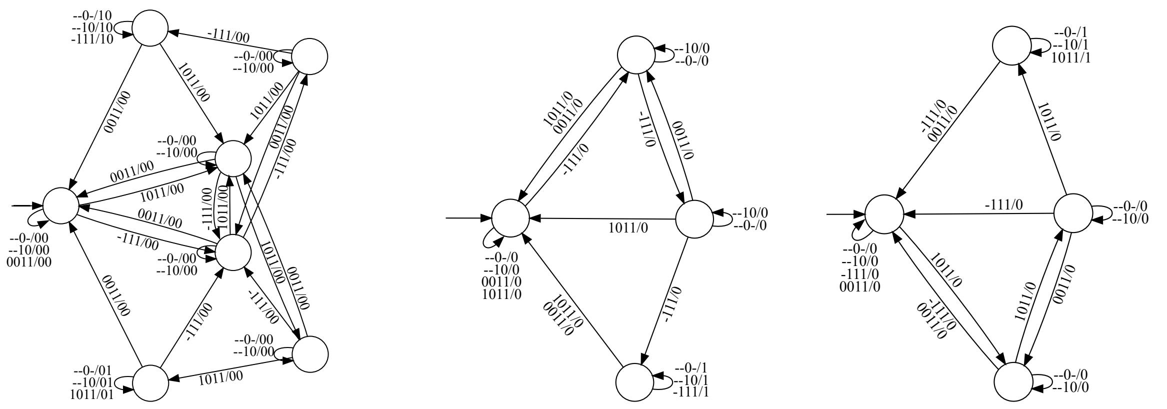

Figure 3: The `bbara` circuit (left) has two output bits. This can be decomposed into two smaller circuits with a single output bit (middle and right).

For some circuits the number of membership queries is reduced compared to a regular learner. Unfortunately, the results are not as impressive as for the n-bit register machine. An interesting case is `ex3` where the number of queries is slightly increased, but the total amount of actions performed is substantially reduced. The number of actions needed in total is actually reduced in all cases, except for `bbsse`. This exception can be explained by the fact that the biggest component of `bbsse` still has 25 states, which is close to the original 31 states. We also note that the choice of decomposition matters, for both `mark1` and `bbsse` it was beneficial to regroup components.

In Figure 4, we look at the size of each hypothesis generated during the learning process. We note that, although each component grows monotonically, the number of reachable states in the product does not grow monotonically. In this particular instance where we learn `mark1` there was a hypothesis of 58 128 states, much bigger than the target machine of 202 states. This is not an issue, as the teacher will allow it and answer the query regardless. Even in the PAC model with membership queries, this poses no problem as we can still efficiently determine membership. However, in some applications the equivalence queries are implemented with a model checker (e.g., in the work by Fiterău-Broştean et al., 2016) or a sophisticated test generation tool. In these cases, the increased size of intermediate hypotheses may be undesirable.

5. Discussion

We have shown two query learning algorithms which exploit a decomposable output. If the output can be split, then also the machine itself can be decomposed in components. As the preliminary experiments show, this can be a very effective optimization for learning black box reactive systems. It should be stressed that the improvement of the optimization depends on the independence of the components. For example, the n-bit register machine has nearly independent components and the reduction in the number of queries is big. The more realistic circuits did not show such drastic improvements in terms of queries. When

			Product learner			TTT learner		
Machine	States	Components	EQs	MQs	Actions	EQs	MQs	Actions
M_2	8	2	3	100	621	5	115	869
M_3	24	3	3	252	1 855	5	347	2946
M_4	64	4	8	456	3 025	6	1 058	13 824
M_5	160	5	6	869	7 665	17	2 723	34 657
M_6	384	6	11	1 383	12 870	25	6 250	90 370
M_7	896	7	11	2 087	24 156	52	14 627	226 114
M_8	2048	8	13	3 289	41 732	160	34 024	651 678
bbara	7	2	3	167	1 049	3	216	1 535
keyb	41	2	25	12 464	153 809	24	6024	265 805
ex3	28	2	24	1 133	9 042	18	878	91 494
bbsse	31	7	20	14 239	111 791	8	4 872	35 469
mark1	202	16	30	16 712	145 656	67	15 192	252 874
bbsse*	31	4	19	11 648	89 935	8	4 872	35 469
mark1*	202	8	22	13 027	117 735	67	15 192	252 874

Table 1: Comparison of the product learner with an ordinary learner.

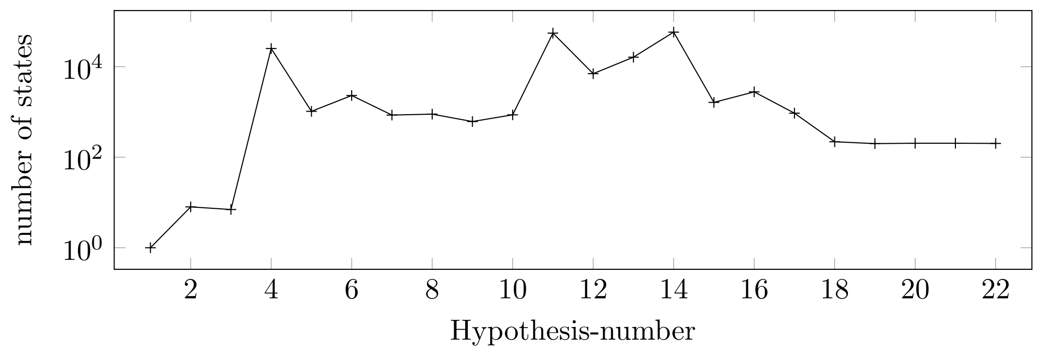

Figure 4: The number of states for each hypothesis while learning mark1.

taking the length of the queries in account as well (i.e., counting all actions performan on the system), we see an improvement for most of the test cases.

In the remainder of this section we discuss related ideas and future work.

5.1. Measuring independence

As the results show, the proposed technique is often beneficial but not always. It would be interesting to know when to use decomposition. It is an interesting question how to (quantitatively) measure the independence. Such a measure can potentially be used by the learning algorithm to determine whether to decompose or not.

5.2. Generalisation to subsets of products

In some cases, we might know even more about our output alphabet. The output set O may be a proper subset of $O_1 \times O_2$, indicating that some outputs can only occur "synchronised". For example, we might have $O = \{(0,0)\} \cup \{(a,b) \mid a,b \in [3]\}$, that is, the output 0 for either component can only occur if the other component is also 0.

In such cases we can use the above algorithm still, but we may insist that the teacher only accepts machines with output in O for the equivalence queries (as opposed to outputs in $\{0,1,2,3\}^2$). When constructing $H = H_1 \times H_2$ in line 7 of Algorithm 2, we can do a reachability analysis on H to check for non-allowed outputs. If such traces exist, we know it is a counterexample for at least one of the two learners. With such traces we can fix the defect ourselves, without having to rely on the teacher.

5.3. Product DFAs

For two DFAs $(Q_1, \delta_1, F_1, q_{01})$ and $(Q_2, \delta_2, F_2, q_{02})$, a state in the product automaton is accepting if both components are accepting. In the formalism of Moore machines, the finals states are determined by their characteristic function and this means that the output is given by $o(q_1, q_2) = o_1(q_1) \wedge o_2(q_2)$. Again, the components may be much smaller than the product and this motivated Heinz and Rogers (2013) to learn (a subclass of) product DFAs. This type of product is more difficult to learn as the two components are not directly observable. Such automata are also relevant in model checking and some of the (open) problems are discussed by Kupferman and Mosheiff (2015).

5.4. Learning automata in reverse

The main result of Rivest and Schapire (1994) was to exploit the structure of the so-called "diversity-based" automaton. This automaton may also be called the reversed Moore machine. Reversing provides a duality between reachability and equivalence. This duality is theoretically explored by Rot (2016) and Bonchi et al. (2014) in the context of Brzozowski's minimization algorithm.

Let M^R denote the reverse of M, then we have $[\![M^R]\!](w) = [\![M]\!](w^R)$. This allows us to give an `L*` algorithm which learns M^R by posing membership queries with the words reversed. We computed M^R for the circuit models and all but one of them was much larger than the original. This suggests that it might not be useful as an optimisation in learning hardware or software systems. However, a more thorough investigation is desired.

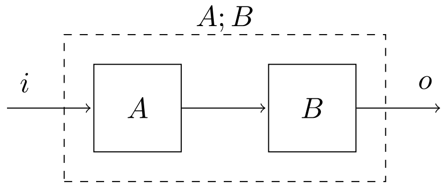

Figure 5: The sequential compostion $A;B$ of two Mealy machines A and B.

5.5. Other types of composition

The case of learning a sequential composition is investigated by Abel and Reineke (2016). In their work, there are two Mealy machines, A and B, and the output of A is fed into B, see Figure 5. The goal is to learn a machine for B, assuming that A is known (i.e., white box). The oracle only answers queries for the sequential composition, which is defined formally as $[\![A;B]\!](w) = [\![B]\!]([\![A]\!](w))$. Since B can only be interacted with through A, we cannot use `L*` directly. The authors show how to learn B using a combination of `L*` and SAT solvers. Moreover, they give evidence that this is more efficient than learning $A;B$ as a whole.

An interesting generalisation of the above is to consider A as an unknown as well. The goal is to learn A and B simultaneously, while observing the outputs of B and the communication between the components. The authors conjecture that this would indeed be possible and result in a learning algorithm which is more efficient than learning $A;B$ (private communication).

Another type of composition is used by Bollig et al. (2010a). Here, several automata are put in parallel and communicate with each other. The goal is not to learn a black box system, but to use learning when designing such a system. Instead of words, the teacher (i.e., designer in this case) receives message sequence charts which encode the processes and actions. Furthermore, they exploit partial order reduction in the learning algorithm.

We believe that a combination of our and the above compositional techniques can improve the scalability of learning black box systems. Especially in the domain of software and hardware we expect such techniques to be important, since the systems themselves are often designed in a modular way.

Acknowledgments

We would like to thank Nathanaël Fijalkow, Ramon Janssen, Gerco van Heerdt, Harco Kuppens, Alexis Linard, Alexandra Silva, Rick Smetsers, and Frits Vaandrager for proofreading this paper and providing useful feedback. Thanks to Andreas Abel for discussing the case of learning a sequential composition of two black box systems. Also thanks to anonymous reviewers for interesting references and comments.

References

Andreas Abel and Jan Reineke. Gray-box learning of serial compositions of mealy machines. In Sanjai Rayadurgam and Oksana Tkachuk, editors, *NASA Formal Methods - 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings,* volume 9690 of *Lecture Notes in Computer Science*, pages 272–287. Springer, 2016. ISBN 978-3-319-40647-3. URL https://doi.org/10.1007/978-3-319-40648-0_21.

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.

Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, and Martin Leucker. Learning communicating automata from mscs. *IEEE Trans. Software Eng.*, 36(3):390–408, 2010a. URL https://doi.org/10.1109/TSE.2009.89.

Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and David R. Piegdon. libalf: The automata learning framework. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 360–364. Springer, 2010b. ISBN 978-3-642-14294-9. URL https://doi.org/10.1007/978-3-642-14295-6_32.

Filippo Bonchi, Marcello M. Bonsangue, Helle Hvid Hansen, Prakash Panangaden, Jan J. M. M. Rutten, and Alexandra Silva. Algebra-coalgebra duality in Brzozowski's minimization algorithm. *ACM Trans. Comput. Log.*, 15(1):3:1–3:29, 2014. URL http://doi.acm.org/10.1145/2490818.

Georg Chalupar, Stefan Peherstorfer, Erik Poll, and Joeri de Ruiter. Automated reverse engineering using Lego®. In Sergey Bratus and Felix "FX" Lindner, editors, *8th USENIX Workshop on Offensive Technologies, WOOT '14, San Diego, CA, USA, August 19, 2014*. USENIX Association, 2014. URL https://www.usenix.org/conference/woot14/workshop-program/presentation/chalupar.

Paul Fiterău-Broştean, Ramon Janssen, and Frits W. Vaandrager. Combining model learning and model checking to analyze TCP implementations. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 454–471. Springer, 2016. ISBN 978-3-319-41539-0. URL https://doi.org/10.1007/978-3-319-41540-6_25.

Paul Fiterău-Broştean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits W. Vaandrager, and Patrick Verleg. Model learning and model checking of SSH implementations. In Hakan Erdogmus and Klaus Havelund, editors, *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 10-14, 2017*, pages 142–151. ACM, 2017. ISBN 978-1-4503-5077-8. URL http://doi.acm.org/10.1145/3092282.3092289.

Jeffrey Heinz and James Rogers. Learning subregular classes of languages with factored deterministic automata. In Andras Kornai and Marco Kuhlmann, editors, *Proceedings of the 13th Meeting on the Mathematics of Language (MoL 13)*, pages 64–71, Sofia, Bulgaria, August 2013. Association for Computational Linguistics.

Robert M. Hierons and Uraz Cengiz Türker. Incomplete distinguishing sequences for finite state machines. *Comput. J.*, 58(11):3089–3113, 2015. URL https://doi.org/10.1093/comjnl/bxv041.

Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, volume 8734 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 2014. ISBN 978-3-319-11163-6. 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 Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 487–495. Springer, 2015. ISBN 978-3-319-21689-8. 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. URL https://doi.org/10.1016/j.ic.2014.09.010.

Ronald L. Rivest and Robert E. Schapire. Diversity-based inference of finite automata. *J. ACM*, 41(3):555–589, 1994. URL http://doi.acm.org/10.1145/176584.176589.

Jurriaan Rot. Coalgebraic minimization of automata by initiality and finality. *Electr. Notes Theor. Comput. Sci.*, 325:253–276, 2016. URL https://doi.org/10.1016/j.entcs.2016.09.042.

Mathijs Schuts, Jozef Hooman, and Frits W. Vaandrager. Refactoring of legacy software using model learning and equivalence checking: An industrial experience report. In Erika Ábrahám and Marieke Huisman, editors, *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*, volume 9681 of *Lecture Notes in Computer Science*, pages 311–325. Springer, 2016. ISBN 978-3-319-33692-3. URL https://doi.org/10.1007/978-3-319-33693-0_20.

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

Gerco van Heerdt, Matteo Sammartino, and Alexandra Silva. CALF: Categorical automata learning framework. In Valentin Goranko and Mads Dam, editors, *26th EACSL Annual Conference on Computer Science Logic, CSL 2017, August 20-24, 2017, Stockholm, Sweden*, volume 82 of *LIPIcs*, pages 29:1–29:24. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. ISBN 978-3-95977-045-3. URL https://doi.org/10.4230/LIPIcs.CSL.2017.29.