Learning Transparent Data Automata[⋆]

Normann Decker[1], Peter Habermehl[2], Martin Leucker[1], and Daniel Thoma[1]

[1] ISP, University of Lübeck, Germany
{decker, leucker, thoma}@isp.uni-luebeck.de

[2] Univ Paris Diderot, Sorbonne Paris Cité, LIAFA, CNRS, France
peter.habermehl@liafa.univ-paris-diderot.fr

Abstract. This paper studies the problem of learning data automata (DA), a recently introduced model for defining languages of data words which are finite sequences of pairs of letters from a finite and, respectively, infinite alphabet. The model of DA is closely related to general Petri nets, for which no active learning algorithms have been introduced so far. This paper defines transparent data automata (tDA) as a strict subclass of deterministic DA. Yet, it is shown that every language accepted by DA can be obtained as the projection of the language of some tDA. The model of class memory automata (CMA) is known to be equally expressive as DA. However deterministic DA are shown to be strictly less expressive than deterministic CMA. For the latter, and hence for tDA, equivalence is shown to be decidable. On these grounds, in the spirit of Angluin's L* algorithm we develop an active learning algorithm for tDA. They are incomparable to register automata and variants, for which learning algorithms were given recently.

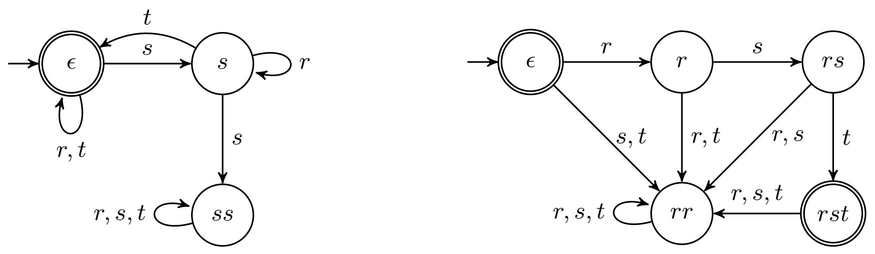
1 Introduction

Learning of formal languages is a fundamental problem in computer science. In the setting of active learning, where a *learner* can ask *membership queries* (i.e. is a given word in the language to be learned) and *equivalence queries* (i.e. is a learning hypothesis equivalent to the language to be learned) to an *oracle (teacher)*, it has been shown [1] that regular languages over a finite alphabet can be learned using the L* algorithm. However, in several application areas like program verification and database management it is important to be able to reason about data coming from an infinite domain. For that purpose *data words*, i.e. sequences of pairs (a, d) of letters a from a finite alphabet and data values d from an infinite alphabet, are used. The data values can contain for example process identifiers from an infinite domain allowing to model naturally parameterized systems with an unbounded number of components. For *data languages*, i.e. sets of data words, *data automata* have been introduced recently [2] as a computational model. A data automaton is a tuple $(\mathcal{A}, \mathcal{B})$ composed of a transducer $\mathcal{A}$, the *base automaton*, and a finite state automaton $\mathcal{B}$, the *class*

[⋆] This work is partially supported by EGIDE/DAAD-Procope (LeMon).

automaton. A data word w is handled in two phases. First the transducer $\mathcal{A}$ reads w without data values and possibly modifies the individual (finite-domain) letters. This results in a data word w' where the data values have not changed. Then, the so called *class strings* of w' are individually checked by the class automaton $\mathcal{B}$. The class strings of a data word are the sub-strings of finite-domain letters carrying the same data value. Roughly, the base automaton enforces a *global property* whereas the class automaton enforces a *local property.* In [2] it is shown that emptiness of data automata and reachability in general Petri nets are polynomially equivalent.

An example of a data automaton, taken with minor changes from [3], is given in Figure 1. It will be used throughout this paper and corresponds to a system where a printer is shared by processes. Data values correspond here to process identifiers and each process can request (r), start (s) and terminate (t). The global property is that each started job must be terminated before the next one can be started and the local property is that each process can invoke one job ($\epsilon + rst$). An accepted data word is for example $\binom{rrstst}{121122}$. Notice that the global property can be characterized by a transducer not modifying the letters but just copying them.



(a) Base automaton $\mathcal{A}$ (b) Class automaton $\mathcal{B}$

Fig. 1: Transparent data automaton $\mathcal{D} = (\mathcal{A}, \mathcal{B})$

The example illustrates two characteristics of many systems, namely the fact that (1) the local behaviour is not constrained by the global one. This means that a process can run on its own. Furthermore, (2) the global behaviour is just a filter, i.e. the transducer is not changing any letters.

Turning to the problem of learning of data automata which we tackle in this paper, we notice that learning the full class of data automata is difficult, as they are closely related to general unbounded Petri nets, a powerful model of computation. Furthermore in active learning it is desirable to have at least theoretically the possibility of answering equivalence queries. However, the equivalence problem for data automata is undecidable. Therefore, we have to look for a simpler but still expressive sub-class of data automata.

In this paper we introduce *transparent data automata* (tDA), which correspond to data automata with the conditions (1) and (2), i.e. $\mathcal{A}$ and $\mathcal{B}$ are finite state automata over the same input alphabet and $\mathcal{L}(\mathcal{B}) \subseteq \mathcal{L}(\mathcal{A})$. We show that tDA have a decidable equivalence problem in Section 2.

More precisely, we obtain decidability even for deterministic *class memory automata* (dCMA), a closely related automaton model for data words introduced in [3]. While general class memory automata (CMA) are expressively equivalent to DA, we show dCMA are strictly more expressive than deterministic DA that in turn are strictly more expressive than tDA. However, tDA are still quite powerful as we show that each DA can be encoded into a tDA that accepts the same language up to projection. This in turn induces the fact that a variant of the emptiness problem is still as hard as Petri net reachability. Furthermore, the class of complements of tDA exceeds the class of dCMA.

For tDA, we introduce several learning algorithms in Section 3. We first consider the case where the global behaviour $\mathcal{A}$ is known and the local behaviour $\mathcal{B}$ is to be learned. We handle that case by adapting Angluin's L^* algorithm. The case where $\mathcal{B}$ is known and $\mathcal{A}$ is to be learned is more difficult since membership queries cannot always be answered conclusively. We therefore adopt the approach of learning from inexperienced teachers [4,5] that may additionally answer membership queries with *don't know*. Finally, we combine the two algorithms in the case where neither $\mathcal{A}$ nor $\mathcal{B}$ are directly accessible.

Another well studied automaton model for data words are *register automata* (RA) [6]. They have a finite control and can additionally store data values in a finite set of registers. Transitions can depend on equality checks between the current data value and values from the registers. DA are strictly more expressive than RA which are strictly more expressive than their deterministic variant. RA and deterministic RA are both incomparable to tDA: The printer example (Figure 1) cannot be accepted by any register automaton and, on the other hand, dCMA are neither more nor equally expressive then deterministic RA [3]. This transfers to tDA as we show that dCMA subsume them.

There are a number of works on learning register automata and its variants [7–9] based all on extensions or adaptations of L*. For Workflow Petri Nets a learning algorithm has been given in [10]. However unlike our models these Petri Nets are bounded and for unbounded Petri Nets we are not aware of any active learning algorithm.

2 Transparent Data Automata

In this section we consider deterministic automata on data words. In particular, we introduce and study transparent data automata that we build on in the subsequent Section 3.

Data words and data languages. Let Σ be a finite alphabet and Δ an infinite set of *data values*. A *data word* is a finite sequence $w = w_1 w_n \in (\Sigma \times \Delta)^*$ of pairs $w_i = (a_i, d_i)$ of letters and data values. We call $\mathsf{str}(w) = a_1 a_n \in \Sigma^*$ the *string projection* of w. The *class string* of w for a data value $d \in \Delta$ is the maximal projected subsequence $w{\downharpoonleft}_d := a_{i_1} a_{i_m} \in \Sigma^*$ of w with data value d, i. e., for all

$1 \leq j \leq m$ we have $1 \leq i_j \leq n$, $d_{i_j} = d$, $i_j < i_{j+1}$ (for $j < m$) and for each $1 \leq k \leq n$ with $d_k = d$ there is some $i_j = k$.

We refer to the set of all (non-empty) class strings of w as $w\lfloor$. We use the data values $1, 2, 3,$ as representatives for arbitrary data values. A data word where the sequence of data values is $d_1 d_n$ and with string projection u is written as $\binom{u}{d_1 d_n}$. If all data values are 1 we may abbreviate that by $\binom{u}{\mathbf{1}}$.

For any automaton $\mathcal{A}$, $\mathcal{L}(\mathcal{A})$ denotes the set of all accepted (data) words but we may also write $w \in \mathcal{A}$ and $L \subseteq \mathcal{A}$ for $w \in \mathcal{L}(\mathcal{A})$ and $L \subseteq \mathcal{L}(\mathcal{A})$, respectively. We use $\overline{L}$ to denote the complement of a language L. A *(deterministic) letter-to-letter transducer* $\mathcal{T}$ is a (deterministic) finite state automaton over a finite input alphabet Σ that additionally outputs a letter from some finite output alphabet Γ for every letter it reads. For $u \in \Sigma^*$ we denote $\mathcal{T}(u)$ the set of possible outputs of $\mathcal{T}$ when reading u.

Data automata. A *data automaton* [2] (DA) is a tuple $\mathcal{D} = (\mathcal{A}, \mathcal{B})$. The *base automaton* $\mathcal{A}$ is a letter-to-letter transducer with input alphabet Σ and output alphabet Γ. The *class automaton* $\mathcal{B}$ is a finite state automaton with input alphabet Γ. A data word $w = \binom{u}{d_1 d_n}$ is accepted by $\mathcal{D}$ if its string projection $u \in \mathcal{A}$ and there is an output $u' \in \mathcal{A}(u)$ of $\mathcal{A}$ s.t. every class string of $\binom{u'}{d_1 d_n}$ is accepted by $\mathcal{B}$. We call a data automaton *deterministic* (dDA), if the base automaton $\mathcal{A}$ is deterministic.

Class Memory Automata. A *class memory automaton* [3] (CMA) over Σ is a tuple $\mathcal{C} = (Q, \Sigma, \delta, q_0, F_{\mathrm{l}}, F_{\mathrm{g}})$ where Q is a finite set of states, $\delta : Q \times (Q \cup \{\bot\}) \times \Sigma \to 2^Q$ is the transition function, $q_0 \in Q$ is the initial state and $F_{\mathrm{l}} \subseteq Q$ and $F_{\mathrm{g}} \subseteq Q$ are the locally and globally accepting states, respectively. A configuration of $\mathcal{C}$ is a pair (q, f) where $q \in Q$ is a state and $f : D \to Q \times \{\bot\}$ is a *memory function* storing the state in which some data value has last been read. If d has not been read before $f(d) = \bot$. The initial configuration is (q_0, f_0) with $\forall_{x \in D} f_0(x) = \bot$. When reading a pair (a, d), the automaton can change from a configuration (q, f) to a configuration (q', f') if $q' \in \delta(q, f(d), a)$, $f'(d) = q'$ and $\forall_{x \in D \setminus \{d\}} f'(x) = f(x)$. A configuration (q, f) is accepting, if $q \in F_{\mathrm{g}}$ and $\forall_{d \in D} f(d) \in (F_{\mathrm{l}} \cup \{\bot\})$. For a configuration (q, f) we call q the global state and all states referred to by f the local states. $\mathcal{C}(w)$ denotes the set of configurations that $\mathcal{C}$ can reach reading a data word w and w is accepted by $\mathcal{C}$ if there is an accepting configuration in $\mathcal{C}(w)$. We call a CMA deterministic (dCMA) if $|\delta(q_{\mathrm{g}}, q_{\mathrm{l}}, a)| \leq 1$ for all $q_{\mathrm{g}} \in Q$, $q_{\mathrm{l}} \in Q \cup \{\bot\}$ and $a \in \Sigma$.

Expressiveness. As is shown in [3], DA and CMA are expressively equivalent. Also, the classes are effectively closed under intersection and union [2, 3]. The emptiness problem for the automata models is shown to be equivalent to reachability in Petri nets and therefore decidable. However the classes are not closed under complementation. For the deterministic case, the classes are not equivalent anymore. Intuitively, DA can globally recognize data values only by means of non-deterministic guessing while CMA do not always rely on that as the present data value affects the transition function.

Lemma 1. *dDA $\subsetneq$ dCMA*

Proof. The inclusion follows from the construction in [3] which translates DA into CMA and preserves determinism. The automaton classes are separated by the language $L \subseteq (\Sigma \times \Delta)^*$ containing the data words over $\Sigma = \{a\}$ with at least two different data values. L is accepted by the dCMA in Figure 2, whereas there is no dDA accepting L: As contradiction, assume some dDA $\mathcal{D} = (\mathcal{A}, \mathcal{B})$ accepts L and consider the words $w_1 = \binom{aa}{12}$, $w_2 = \binom{aaa}{112} \in L$. From w_1 we see that $\mathcal{A}$ accepts aa and from w_2 it follows that $\mathcal{B}$ accepts the corresponding projection $\mathcal{A}(aa)$. Then, however, $\mathcal{D}$ also accepts $\binom{aa}{11} \notin L$ as $\mathcal{A}$ is deterministic. □

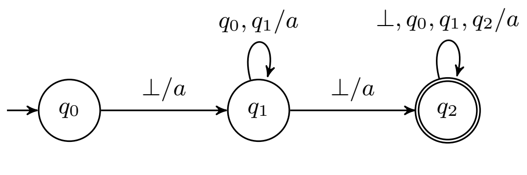


Fig. 2: A dCMA accepting data words with at least two data values. All states are accepting locally and q_2 is also accepting globally.

We now define transparent data automata. They form a sub-class of data automata that reflect the intuition that local behaviour is not constrained by the global one, i.e. any process can run on its own. Technically, we require, that the global language contains the local language as a subset. The condition will later in Section 3 allow us to use global observations for deducing information on the local automaton.

Definition 1 (Transparent Data Automaton). *A* transparent data automaton *(tDA) is a tuple $\mathcal{D} = (\mathcal{A}, \mathcal{B})$ where $\mathcal{A}$ and $\mathcal{B}$ are finite state automata over the same input alphabet Σ and $\mathcal{L}(\mathcal{B}) \subseteq \mathcal{L}(\mathcal{A})$.*

Note that, in fact, a tDA is a DA in the sense that the global automaton $\mathcal{A}$ can be interpreted as letter-to-letter transducer with $\Gamma = \Sigma$ that just outputs accepted input words unchanged.

2.1 Expressiveness of tDA

The transparency condition is a restriction designed to allow for active learning as will be discussed in Section 3. However, we remark in this section that tDA are nevertheless complex. But first, by extending Lemma 1, we obtain a strict hierarchy between tDA, dDA and dCMA.

Theorem 1. *tDA $\subsetneq$ dDA $\subsetneq$ dCMA*

Proof. Note that for every tDA there is an equivalent dDA as the base automaton is a finite automaton which can be determinized. To verify that tDA are strictly less expressive than dDA, consider the language of data words over $\Sigma = \{a\}$ where every second position carries a new data value. A dDA with internal alphabet $\Gamma = \{a, \hat{a}\}$ can accept that language by marking every other position using the base automaton whereas the class automaton checks that the first letter is marked. On the contrary the language is not accepted by any tDA since the base automaton is neither aware of the data values itself nor can it transmit any positioning information to the class automaton. □

Checking tDA for emptiness boils down to checking the class automaton and is thus in NLogSpace. However, the slight modification of checking if there is an accepted word with at least two data values is still as hard as emptiness of DA. This follows from the possibility to encode any DA $\mathcal{D}$ into a tDA $\mathcal{D}'$ in such a way that $\mathcal{D}'$ accepts the same language as $\mathcal{D}$ up to some projection Π. We outline this connection more precisely in the following.

For the DA $\mathcal{D} = (\mathcal{A}, \mathcal{B})$, let Σ, Γ be the input and internal alphabet of the base automaton $\mathcal{A}$, respectively. $\mathcal{D}'$ employs an extended alphabet Σ' that is built from the product of Σ and Γ, an additional flag and a new symbol \$. Formally, we let $\Sigma' := (\Sigma \times \Gamma \times \{0,1\}) \cup \{\$\}$. Recall, that the input and internal alphabets are required to be equal for tDA.

The projection $\Pi : 2^{\Sigma' \times \Delta} \to 2^{\Sigma \times \Delta}$ basically removes all additional information stored in Σ'. For a data language $L \subseteq \Sigma' \times \Delta$ we let $\Pi(L) := \left\{ \binom{\pi(w)}{d_2 d_n} \middle| \binom{\$\$w}{d_0 d_1 d_2 d_n} \in L \right\}$ where $\pi : \Sigma'^* \to \Sigma^*$ is the letter-wise projection to the first component, i.e. $\pi(a, g, x) = a$ for $(a, g, x) \in \Sigma'$ and $\pi(a_0 a_1 a_n) = \pi(a_0)\pi(a_1)\pi(a_n)$ for $a_i \in \Sigma'$.

Theorem 2. *For all DA $\mathcal{D}$ there exists (constructively) a tDA $\mathcal{D}'$ such that $\mathcal{L}(\mathcal{D}) = \Pi(\mathcal{L}(\mathcal{D}'))$.*

Proof. First we note that we can deafen the base automaton $\mathcal{A}$ by simply considering the output of $\mathcal{A}$ as additional input, i.e. we interpret $\mathcal{A}$ over the input alphabet $\Sigma \times \Gamma$. This yields a finite automaton $\hat{\mathcal{A}}$ over $\Sigma \times \Gamma$ accepting exactly the correct input/output combinations $(w, \mathcal{A}(w)) \in \Sigma \times \Gamma$ of $\mathcal{A}$. Now, we can define from the class automaton $\mathcal{B}$ a class automaton $\hat{\mathcal{B}}$ over the alphabet $\Sigma \times \Gamma$ which does the same on Γ as $\mathcal{B}$ while ignoring Σ. Then, $(\hat{\mathcal{A}}, \hat{\mathcal{B}})$ is a transparent data automaton in the sense of Definition 1.

From that we can construct $\mathcal{D}' = (\mathcal{A}', \mathcal{B}')$ as follows. Let $\$ \notin (\Sigma \times \Gamma)$ be a new symbol. For $\mathcal{A}'$ we take the deafened base automaton $\hat{\mathcal{A}}$ but extend the input alphabet by a new flag, 0 or 1, and the special symbol \$, i.e. $\Sigma' := (\Sigma \times \Gamma \times \{0,1\}) \cup \{\$\}$. $\mathcal{A}'$ checks that the input starts with \$\$ and then behaves as $\hat{\mathcal{A}}$ on the rest of the input ignoring the flag.

The class automaton $\mathcal{B}'$ behaves just like $\hat{\mathcal{B}}$ but checks that the first input symbol carries flag 1 and all following symbols carry flag 0. Additionally, it accepts a single \$ as input word.

In combination, this ensures that $\mathcal{D}'$ accepts only data words where exactly the first occurrence of every data value is marked by flag 1 (when considering the symbol \$ as carrying flag 1). If the original automaton $\mathcal{D}$ accepts some word w then the modified automaton $\mathcal{D}'$ accepts some word with at least two data values, namely $\binom{\$\$}{d_0 d_1} w$ where the data values d_0, d_1 do not occur in w and the first occurrences of data values in w are marked by flag 1. On the other hand, if the modified automaton $\mathcal{D}'$ accepts some word w' then w' has the form $\binom{\$\$}{d_0 d_1}\binom{w}{d_2 d_n}$ and the original automaton $\mathcal{D}$ accepts $\binom{\pi(w)}{d_2 d_n}$.

It remains to ensure the transparency condition $\mathcal{L}(\mathcal{B}') \subseteq \mathcal{L}(\mathcal{A}')$. This is done by letting the base automaton $\mathcal{A}'$ accept any word accepted by the new class automaton $\mathcal{B}'$. This potentially adds new data words to the represented language, however, they all do *not* start with \$\$. Thus these are excluded by the projection Π defined above. □

Despite Theorem 2 does not directly yield a hardness result, it provides evidence that the transparency restriction does not make the model trivial. The construction used in the proof yields a tDA that is only polynomial in the size of the DA. Using this technique we can proof the following corollary.

Corollary 1. *Given a tDA, deciding whether there is an accepted word containing at least two different data values is at least as hard as Petri net reachability.*

2.2 Complementation and Equivalence of tDA

As said earlier, DA and CMA are not closed under complementation. This is also the case for the deterministic versions. Moreover, even complements of the smallest class tDA may exceed the largest deterministic class dCMA.

Theorem 3. *dCMA do not capture the complements of tDA.*

Proof (Theorem 3). Let L be the language of data words over $\Sigma = \{a\}$ for which every class string is of even length. L is accepted by a tDA where the base automaton is universal and the class automaton just counts modulo two. For accepting the complement $\overline{L}$ of L, however, an automaton has to check for an input word, that there is a data value for which the class string is of odd length.

Assume there is a dCMA $\mathcal{C}$ that accepts $\overline{L}$. Since $\mathcal{C}$ has finitely many states, there are positions $m < n$ s.t. $\mathcal{C}$ is in the same state q after reading $u = \binom{aa}{1m}$ and after reading $uv = \binom{a\ \ a}{1mn}$. Because $u \in \overline{L}$, all local states as well as the global state in the configuration of $\mathcal{C}$ after reading u must be accepting. On the contrary, $uu \notin \overline{L}$ and so continuing by reading u again, $\mathcal{C}$ must change to some non-accepting state, either globally or locally. As the automaton is deterministic, appending u to uv must effect the same change as the data values in v are not present in u and the local states w.r.t. v do thus not influence the transitions taken by $\mathcal{C}$ when reading u. Therefore, after reading uvu, the configuration of $\mathcal{C}$ must contain some non-accepting local or global state which contradicts $uvu \in \overline{L}$. □

Even though neither the deterministic nor the non-deterministic class of data automata are closed under complementation, the complements of the deterministic classes can still be constructed in terms of CMA and thus allow for algorithmic analysis, in particular for emptiness checking.

Theorem 4. *For a deterministic CMA $\mathcal{C}$ we can construct a CMA accepting exactly the complement $\overline{\mathcal{L}(\mathcal{C})}$.*

Note that this result does not follow from [3]. There, a complementable variant called Presburger CMA is introduced and claimed to subsume dCMA but in fact it does not. Presburger CMA replace the acceptance condition of CMA by a *limited* Presburger formula, allowing essentially modulo constraints over the local states of a configuration. While this allows for complementation by complementing the Presburger formula, the original acceptance condition can no longer be encoded. Adding the original condition to Presburger CMA breaks closure under complementation.

Proof. Let $\mathcal{C} = (Q, \Sigma, \delta, Q_0, F_\mathrm{l}, F_\mathrm{g})$ be a dCMA accepting the data language $L = \mathcal{L}(\mathcal{C})$. For the complement $\overline{L}$ of L, we observe that for some data word $w \in (\Sigma \times \Delta)^*$ and the configuration $\mathcal{C}(w)$ of $\mathcal{C}$ after reading w we have $w \in \overline{L}$ iff (1) all local states in $\mathcal{C}(w)$ are accepting and the global state is rejecting or (2) there is some local state in $\mathcal{C}(w)$ that is rejecting.

Hence, we construct two CMA $\tilde{\mathcal{C}}$ and $\hat{\mathcal{C}}$ that accept all words obeying conditions (1) and (2), respectively. Then, the automaton $\overline{\mathcal{C}} := \tilde{\mathcal{C}} \cup \hat{\mathcal{C}}$ accepts exactly $\overline{L}$ and is a CMA since this class is effectively closed under union.

To recognize data words satisfying the first condition, we simply complement the set of global accepting states: $\tilde{\mathcal{C}} := (Q, \Sigma, \delta, q_0, F_\mathrm{l}, \overline{F}_\mathrm{g})$.

To recognize the second condition let $\hat{\mathcal{C}} = (\hat{Q}, \Sigma, \hat{\delta}, \{(q_0, 0, 0)\}, \hat{F_\mathrm{l}}, \hat{F_\mathrm{g}})$ where the state space $\hat{Q} := Q \times B \times B$ is that of $\mathcal{C}$ enriched by two boolean flags from $B = \{0, 1\}$. The idea is that, whenever observing a new data value, the automaton guesses whether this is the data value for which the corresponding class causes the rejection in the original automaton $\mathcal{C}$. The automaton then sets both flags to 1. The first flag is propagated globally, thereby keeping track of the fact that a data value has been guessed. The second flag is propagated locally, thereby marking the chosen data value. Except for the described guessing step and propagation of both flags, $\hat{\mathcal{C}}$ just simulates $\mathcal{C}$. The automaton accepts globally, once a data value has been chosen and thus the first flag is 1, i.e. for $\hat{F_\mathrm{g}} = Q \times \{1\} \times B$. The automaton accepts locally if the local state for the chosen data value would have been locally rejecting, i.e. if the second flag is 1 and the original state is in $\overline{F}_\mathrm{l}$. Formally, if $\hat{F_\mathrm{l}} = (Q \times B \times \{0\}) \cup (\overline{F_\mathrm{l}} \times B \times \{1\})$.

The transition function $\hat{\delta} : \hat{Q} \times (\hat{Q} \cup \{\bot\}) \times \Sigma \to 2^{\hat{Q}}$ is defined as follows. We use _ as placeholder for an arbitrary, irrelevant value.

$$\hat{\delta}((q, 0, \_), (q', \_, \_), a) := \{(\delta(q, q', a), 0, 0)\} \tag{1}$$

$$\hat{\delta}((q, 0, \_), \bot, a) := \{(\delta(q, \bot, a), 0, 0), (\delta(q, \bot, a), 1, 1)\} \tag{2}$$

$$\hat{\delta}((q, 1, \_), (q', \_, x), a) := \{(\delta(q, q', a), 1, x)\} \tag{3}$$

$$\hat{\delta}((q, 1, \_), \bot, a) := \{(\delta(q, \bot, a), 1, 0)\} \tag{4}$$

Equations 1 and 2 capture the case, that no data value has been guessed so far. In case of Equation 1 the present data value is not new. In case of Equation 2 the present data value is new. Thus, the automaton can progress just as for Equation 1 or choose to guess the present data value. Equations 3 and 4 capture that a data value has already been guessed. In case of Equation 3 the present data value is not new and the first flag in the global and the second flag in the local state have to be propagated. Thereby the information, that a value has been guessed is propagated globally, and the information, which value has been guessed is propagated locally. In case of Equation 4 the present data value is new and the first flag in the global state has to be propagated. As the value is new, it can not be the chosen value and thus the second flag is set to 0. $\square$

With the emptiness check for CMA, their closure under union and intersection [2, 3] and the complement construction above, we can decide language inclusion and equivalence of dCMA and its subclasses.

Corollary 2. *Language inclusion and equivalence of dCMA, dDA and tDA is decidable.*

3 Learning transparent data automata

In this section we develop learning algorithms for tDA. We first recall the classical active learning procedure, and a variation with inconclusive answers to membership queries along the lines of [5]. Based on these we show how the class automaton of some tDA can be learned assuming the base automaton is known. Then, we provide an algorithm for learning the base automaton assuming the class automaton is known. Finally, we combine the two developed approaches to obtain a learning procedure for a completely unknown tDA.

3.1 Learning of finite automata

The learning algorithm L*. Angluin's learning algorithm, called L* [1], is designed for learning a regular language, $L \subseteq \Sigma^*$, by constructing a minimal DFA $\mathcal{A}$ accepting precisely L. In this algorithm a *learner*, who initially knows nothing about L, is trying to learn L by asking an *oracle* (also called the *teacher*), that knows L, two kinds of queries: A *membership query* for a word $u \in \Sigma^*$ is the question whether u is in L. Given a DFA $\tilde{\mathcal{A}}$ as a *hypothesis*, an *equivalence query* is the question whether $\tilde{\mathcal{A}}$ is correct, i.e. $\mathcal{L}(\tilde{\mathcal{A}}) = L$. The oracle answers *yes* if $\tilde{\mathcal{A}}$ is correct, or else provides a *positive* or *negative counter-example* u from $L \setminus \mathcal{L}(\tilde{\mathcal{A}})$ or $\mathcal{L}(\tilde{\mathcal{A}}) \setminus L$, respectively.

The learner maintains a prefix-closed set $U \subseteq \Sigma^*$ of prefixes, which are candidates for identifying states, and a suffix-closed set $V \subseteq \Sigma^*$ of suffixes, which are used to distinguish such states. The sets U and V are extended when needed during the algorithm. The learner poses membership queries for all words in $(U \cup U\Sigma)V$, and organizes the results into a *table* $T : (U \cup U\Sigma) \times V \to \{✓, ✗\}$ where $(U \cup U\Sigma)$ are the row and V the column labels, respectively, and ✓ represents *accepted* and ✗ *not accepted*. Where convenient, we write $T(u)$ for the complete row in T indexed by $u \in U \cup U\Sigma$.

When T is *closed*, i.e. for each $u \in U$, $a \in \Sigma$ there is a $u' \in U$ such that $T(ua) = T(u')$, and *consistent*, i.e. $T(u) = T(u')$ implies $T(ua) = T(u'a)$, the learner constructs a hypothesis $\tilde{\mathcal{A}} = (Q, q_0, \delta, F)$, where $Q = \{T(u) \mid u \in U\}$ is the set of distinct rows, q_0 is the row $T(\epsilon)$, δ is defined by $\delta(T(u), a) = T(ua)$, and $F = \{T(u) \mid u \in U, T(u)(\epsilon) = \checkmark\}$. The hypothesis is posed as an equivalence query to the oracle. If the answer is *yes*, the learning procedure is completed, otherwise the returned counter-example $c \in \Sigma^*$ is used to extend U by adding all prefixes of c to U, and subsequent membership queries are performed in order to make the new table closed and consistent producing a new hypothesis.

Concerning complexity, it can easily be seen that the number of membership queries can be bounded by $O(kn^2m)$, where n is the number of states of the automaton to learn, k is the size of the alphabet, and m is the length of the longest counter-example.

Learning from inexperienced teacher. In the setting of an *inexperienced teacher*, membership queries are no longer answered only by *yes* or *no*, but also by *don't know*, denoted **?**. Angluin's algorithm can easily be adapted to work with an inexperienced teacher and we list the necessary changes [5]. The table can now also contain **?**, i.e., $T : (U \cup U\Sigma) \times V \to \{\checkmark, \times, ?\}$. For $u, u' \in (U \cup U\Sigma)$, we say that rows $T(u)$ and $T(v)$ *look similar*, denoted by $T(u) \equiv T(u')$, iff, for all $v \in V$, $T(u)(v) \neq \mathbf{?}$ and $T(u')(v) \neq \mathbf{?}$ implies $T(u)(v) = T(u')(v)$. Otherwise, we say that $T(u)$ and $T(v)$ are *obviously different*. We call T *weakly closed* if for each $u \in U$, $a \in \Sigma$ there is a $u' \in U$ such that $T(ua) \equiv T(u')$, and *weakly consistent* if $T(u) \equiv T(u')$ implies $T(ua) \equiv T(u'a)$. Angluin's algorithm works as before, but using the weak notions of closed and consistent. While extracting a DFA from a weakly closed and weakly consistent table is no longer straightforward, it is possible using techniques developed by Biermann and Feldman [11] that infer an automaton from a sample set $S = (S^{\checkmark}, S^{\times})$ consisting of positive and negative examples $S^{\checkmark}$ and $S^{\times}$, respectively. As there is no longer a unique automaton for a weakly closed and weakly consistent table, the overall complexity of identifying a given automaton is no longer polynomial in the number of equivalence queries but may be exponential (see [5] for details).

3.2 Learning the class automaton

Let the base automaton $\mathcal{A}$ of some tDA $\mathcal{D} = (\mathcal{A}, \mathcal{B})$ over an alphabet Σ be known. For learning the class automaton $\mathcal{B}$ we can employ the classical L* algorithm as presented above. The algorithm, however, assumes direct access to an oracle for $\mathcal{B}$ that answers membership and equivalence queries. Given such an oracle for $\mathcal{D}$, we can answer queries for $\mathcal{B}$ using queries for $\mathcal{D}$ as follows.

Membership. To answer a membership query $u \overset{?}{\in} \mathcal{B}$ for $u \in \Sigma^*$, we can directly reuse the answer of the oracle for the query $\binom{u}{\mathbf{1}} \overset{?}{\in} \mathcal{D}$. $\binom{u}{\mathbf{1}} \in \mathcal{D}$ implies $u \in \mathcal{B}$ since u is a class string of $\binom{u}{\mathbf{1}}$. Further, if $\binom{u}{\mathbf{1}} \notin \mathcal{D}$ then $u \notin \mathcal{B}$ due to the transparency property $\mathcal{B} \subseteq \mathcal{A}$ meaning that $\mathcal{A}$ cannot reject u while $\mathcal{B}$ accepts it.

Algorithm 1 Learning the class automaton

```
function LEARNB(Automaton $\mathcal{A}$)
    function ISMEMBERB(Word $u$)
        return ISMEMBERD($\binom{u}{11}$)
    function ISEQUIVALENTB(Automaton $\tilde{\mathcal{B}}$)
        result := ISEQUIVALENTD(($\mathcal{A}, \tilde{\mathcal{B}}$))
        if ISPOSITIVECE(result) then
            for all c $\in$ CLASSSTRINGS(result) do
                if $c \notin \tilde{\mathcal{B}}$ then return ASPOSITIVECE(c)
        else if ISNEGATIVECE(result) then
            for all c $\in$ CLASSSTRINGS(result) do
                if $\neg$ISMEMBERB(c) then return ASNEGATIVECE(c)
        else return true
    return ANGLUIN(ISMEMBERB, ISEQUIVALENTB)
```

Equivalence. To obtain the answer to an equivalence query $\tilde{\mathcal{B}} \overset{?}{\equiv} \mathcal{B}$ for some hypothesis $\tilde{\mathcal{B}}$ we can pose the query $(\mathcal{A}, \tilde{\mathcal{B}}) \overset{?}{\equiv} (\mathcal{A}, \mathcal{B})$ to the oracle for $\mathcal{D}$. A positive answer confirms that $\tilde{\mathcal{B}}$ is correct. A negative counter-example $c \in (\mathcal{A}, \tilde{\mathcal{B}})$ is accepted by the hypothesis but should be rejected. Thus, at least one of its class strings must be wrongly accepted by $\tilde{\mathcal{B}}$ since $\mathcal{A}$ is correct. To find it, we consider all class strings $u \in c\!\downharpoonleft \cap \tilde{\mathcal{B}}$ accepted by the hypothesis and use membership queries $u \overset{?}{\in} \mathcal{B}$ for identifying one that must be rejected. This is returned as counter-example $\tilde{\mathcal{B}}$.

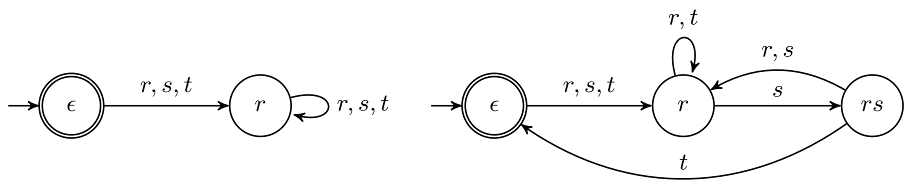
If $c \notin (\mathcal{A}, \tilde{\mathcal{B}})$ is a positive counter-example there must be some class string of c rejected by $\tilde{\mathcal{B}}$ whereas all should be accepted. Hence, we check all class strings $u \in c\!\downharpoonleft$ for one being rejected by $\tilde{\mathcal{B}}$ to find the counter-example that is returned.

Algorithm 1 takes a base automaton as input and calls Angluin's L*, providing the functions for membership and equivalence queries. We assume the functions ISMEMBERD and ISEQUIVALENTD to be globally defined and represent the query to the oracle for $\mathcal{D}$. We suppose that ISEQUIVALENTD returns yes or either a positive or negative counter-example which can be checked by the functions ISPOSITIVECE and ISNEGATIVECE. The function CLASSSTRINGS takes a data word and returns all class strings of it.

Example. As an example, consider the transparent data automata $\mathcal{D} = (\mathcal{A}, \mathcal{B})$ for the printer example from Section 1 and assume $\mathcal{B}$ is unknown. L* first asks queries $\mathcal{B}$ for the empty word ϵ which is translated to the empty data word ϵ. Since $\mathcal{D}$ accepts, the answer is positive. For closing the table, we need to answer queries for all words in $\{\epsilon\} \cdot \Sigma$. Since $\binom{r}{1}, \binom{s}{1}, \binom{t}{1} \notin \mathcal{D}$ the answer is negative for all of them and we obtain a new state for which we choose e.g. r as representative. To close the table, L* asks for the words in $r \cdot \Sigma$ and we answer them by posing queries $\binom{rr}{1}, \binom{rs}{1}, \binom{rt}{1} \overset{?}{\in} \mathcal{D}$ to the oracle which are all negative. The resulting table consists of $U = \{\epsilon, r\}$ and $V = \{\epsilon\}$ with $T(\epsilon) = ✓$ and $T(r) = ✗$. It is closed and consistent and yields the hypothesis $\tilde{\mathcal{B}}_1$ shown in Figure 3a.

For the equivalence query, we ask $(\mathcal{A}, \tilde{\mathcal{B}}_1) \overset{?}{\equiv} \mathcal{D}$ and obtain a counter-example, say $c = \binom{rrststst}{122211} \in \mathcal{D}$. From the class strings $c\downharpoonleft = \{rst\}$ we obtain rst as a positive counter-example as it is rejected by $\tilde{\mathcal{B}}_1$ and return that to the L* instance. The prefixes rst, rs are added to U and the table is filled using queries for rst, rs as well as all words from $\{rst, rs\} \cdot \Sigma$. Apart from rst we answer all of them negatively using the oracle for $\mathcal{D}$ as before. We have now $T(rs) = T(r)$ but $T(rs \cdot t) \neq T(r \cdot t)$ and resolve this inconsistency by adding t to V and fill the table using membership queries. The table is now closed and consistent, the hypothesis $\tilde{\mathcal{B}}_2$ is shown in Figure 3b.

Asking $(\mathcal{A}, \tilde{\mathcal{B}}_2) \overset{?}{\equiv} \mathcal{D}$ we obtain, e. g., the positive counter-example $\binom{rrrsstt}{1211212} \in \mathcal{D}$. Membership queries for all class strings $\{rrst, rst\}$ yield that $rrst$ is not in $\mathcal{B}$ but in $\tilde{\mathcal{B}}_2$ and we can provide $rrst$ as negative counter-example for $\tilde{\mathcal{B}}_2$. Consequently, $rrst$ and its prefixes are added to U, the table is filled and we observe first the inconsistency $T(r) = T(rr)/T(r \cdot st) \neq T(rr \cdot st)$. After adding st to V and filling the table we handle the second inconsistency $T(\epsilon) = T(rst)/T(\epsilon \cdot rst) \neq T(rst \cdot rst)$ similarly adding rst to V. We finally have $U = \{\epsilon, r, rst, rs, rrst, rrs, rr\}$ and $V = \{\epsilon, t, st, rst\}$ leading to the correct hypothesis (Figure 1b). Note that rr, rrs and $rrst$ have equal rows in T and are thus represented by the same state. The final table is presented as Table 1.



(a) Hypothesis $\tilde{\mathcal{B}}_1$.

(b) Hypothesis $\tilde{\mathcal{B}}_2$.

Fig. 3: Hypotheses while learning the class automaton $\mathcal{B}$ of the printer example

3.3 Learning the base automaton

In the previous section we assumed the base automaton $\mathcal{A}$ to be given and we now show how $\mathcal{A}$ can be learned when the class automaton $\mathcal{B}$ is known. The major difference is that we can not always answer membership queries for $\mathcal{A}$ conclusively. If some word is rejected due to the class automaton $\mathcal{B}$, the oracle for $\mathcal{D}$ will answer negatively but we do not gain any information on whether $\mathcal{A}$ also rejects. The approach (Algorithm 2) is therefore based on the learning procedure allowing for the additional answer *don't know* (**?**) [4, 5].

Membership. To answer a membership query $u \overset{?}{\in} \mathcal{A}$ for a word $u \in \Sigma^*$ of length n, we consider the set $\mathsf{dw}(u)$ of all data words $\binom{u}{d_1 d_n}$ up to isomorphism on the data domain Δ. If there exists a data word $w \in \mathsf{dw}(u)$ where all class strings are accepted by $\mathcal{B}$, i.e. $w\!\downharpoonleft\, \subseteq \mathcal{B}$, we can directly use the answer of the oracle for $\mathcal{D}$ to the query $w \overset{?}{\in} \mathcal{D}$ since acceptance of w, and therefore u, only depends on $\mathcal{A}$. If there is no such data word, we have to answer inconclusively (**?**).

Note that an inconclusive answer does not imply that the value is arbitrary. The answer of the base automaton does not matter *for acceptance* of a data word when the class automaton would always reject. However, choosing arbitrary answers, e.g., always no, can make the language that is factually learned non-regular. Hence, there would be no guarantee anymore that the learning procedure for the base automaton terminates as there might not exist an automaton with a finite state space.

Equivalence. To check a hypothesis $\tilde{\mathcal{A}}$ for equivalence we pose the query $(\tilde{\mathcal{A}}, \mathcal{B}) \overset{?}{\equiv} \mathcal{D}$ to the oracle. A positive answer confirms that $\tilde{\mathcal{A}}$ is correct. A positive or negative counter-example $w = \binom{u}{d_1 d_n}$ directly yields the respective counter-example u for $\tilde{\mathcal{A}}$. If w is rejected by $(\tilde{\mathcal{A}}, \mathcal{B})$ but should be accepted, $\mathcal{B}$ has to accept all class strings and this is the case as $\mathcal{B}$ is correct. Thus $\tilde{\mathcal{A}}$ wrongly rejects u. Otherwise, if w is accepted by the hypothesis but should be rejected, all class strings of w are accepted by $\mathcal{B}$. Hence, it must be $\mathcal{A}$ that causes the rejection of w by rejecting its string projection u. The function STRINGPROJECTION takes a data word and returns the string projection of it.

Note that we stop learning the base automaton as soon as the hypothesis $\tilde{\mathcal{A}}$ is equivalent to $\mathcal{A}$ in the context of $\mathcal{B}$. That is, $(\tilde{\mathcal{A}}, \mathcal{B})$ is a DA equivalent to $\mathcal{D}$ even though $\tilde{\mathcal{A}}$ and $\mathcal{A}$ are not necessarily language-equivalent.

L* with guided search. The setting of learning an automaton with potentially inconclusive membership queries does, in principle, work independently of the amount of conclusive answers. However, if many membership queries are answered inconclusively , i.e., the table is mostly filled by **?**, the algorithm essentially learns from counter-examples.

In our setting, we can improve the procedure using $\mathcal{B}$. L* explores the state space of the unknown automaton by following single edges: Inconsistencies in the table are generated by appending single letters to the access strings in U, i.e. asking additional membership queries for $U\Sigma$. A conclusive answer to these queries, however, is only possible if there is (by chance) an assignment of data values to the generated prefix s.t. all class strings are accepted by $\mathcal{B}$.

Hence it is reasonable to extend the prefixes $u \in U$ not only by the single letters from Σ but also by possibly longer suffixes that guarantee that there is indeed an assignment of data values s.t. all class strings are accepted by $\mathcal{B}$. For $u \in \Sigma^*$ we therefore let $\mathsf{ext}_{\mathcal{B}}(u)$ be the set of shortest suffixes $v \in \Sigma^*$ that can be appended to u s.t. there is a data word $\binom{uv}{d_1 d_n}$ that is not rejected locally, i.e. $\binom{wv}{d_1 d_n}\!\downharpoonleft\, \subseteq \mathcal{B}$. By short we mean that the extensions of the single class strings are as short as possible to be accepted by $\mathcal{B}$. Formally, for $c \in \Sigma^*$, let $\mathsf{short}_{\mathcal{B}}(c) =$

Algorithm 2 Learning the base automaton

```
1:  function learnA(Automaton $\mathcal{B}$)
2:      function isMemberA(Word $u$)
3:          for all $w \in \mathsf{dw}(u)$ do
4:              if $w\downharpoonleft \subseteq \mathcal{B}$ then return isMemberD($w$)
5:          return ?
6:      function isEquivalentA(Automaton $\tilde{\mathcal{A}}$)
7:          result := isEquivalentD(($\tilde{\mathcal{A}}, \mathcal{B}$))
8:          if isPositiveCE(result) then
9:              return asPositiveCE(stringProjection(result))
10:         else if isNegativeCE(result) then
11:             return asNegativeCE(stringProjection(result))
12:         else return true
13:     return angluinInexp(isMemberA, isEquivalentA)
```

$\{v \in \Sigma^* \mid cv \in \mathcal{B} \text{ and } \forall_{v' \in \Sigma^*, |v'|<|v|} : cv' \notin \mathcal{B}\}$ be the shortest extensions of c accepted by $\mathcal{B}$ and $\bigsqcup\!\bigsqcup_i L_i$ be the shuffle of (finitely many) languages $L_i \subseteq \Sigma^*$. Then $\mathsf{ext}_{\mathcal{B}}(u) := \bigcup_{w \in \mathsf{dw}(u)} \bigsqcup\!\bigsqcup_{c \in w\downharpoonleft} \mathsf{short}_{\mathcal{B}}(c)$ and additionally to filling the table for $U \cdot \Sigma$, as standard L* does, we also consider $U \cdot \mathsf{ext}_{\mathcal{B}}(u)$ for all $u \in U\Sigma$.

Example. We consider again the printer example (Section 1) and apply Algorithm 2 for learning the base automaton $\mathcal{A}$. The first query for the empty word ϵ gives a positive answer. The table is closed with $\{\epsilon\}\Sigma$. We must answer the respective queries with **?** since the only data words are $\binom{r}{1}$, $\binom{s}{1}$, $\binom{t}{1}$ for which the class strings are rejected by $\mathcal{B}$.

In the guided search step we do not find shortest extensions of s, t yielding definite results. For r we find st since $\binom{rst}{111}\downharpoonleft = \{rst\} \subseteq \mathcal{B}$ which is added to V. For filling the table queries $st, sst, tst, rst \overset{?}{\in} \mathcal{A}$ have to be answered. We answer with **?** for st, sst, tst since no data values can be found to produce a query for $\mathcal{D}$ but posing $\binom{rst}{111} \overset{?}{\in} \mathcal{D}$ yields that $rst \in \mathcal{A}$.

The table is now weakly closed and weakly consistent giving $S_1 = (\{\epsilon, rst\}, \{\})$ as sample set and thereby the trivial hypothesis $\tilde{\mathcal{A}}_1$ accepting Σ^*. For the equivalence check, we pose $(\tilde{\mathcal{A}}_1, \mathcal{B}) \overset{?}{\equiv} \mathcal{D}$ to the oracle that returns a negative counter-example, e.g. $\binom{rrsstt}{111111} \notin \mathcal{D}$. Adding the counter-example to U and proceeding until the table is again weakly closed and weakly consistent yields Table 2 and the sample set $S_2 = (S_2^{\checkmark}, S_2^{\text{✗}})$ with

$$\begin{aligned} S_2^{\checkmark} &= \{\epsilon, rst, rrstst, rrsrtstst, rrrststst\}, \\ S_2^{\text{✗}} &= \{rrsstt, rrssttrst, rrsstrtst, rrsstrstt, rrssrttst, \\ &\quad rrssrtstt, rrssrsttt, rrsrsttst, rrsrssttt, rrsrststt, \\ &\quad rrsrtsstt, rrrssttst, rrrstsstt, rrrsssttt\}. \end{aligned}$$

The hypothesis $\tilde{\mathcal{A}}_2$ must now have at least three states and could thus already be correct.

3.4 Complete learning of tDA

We have seen so far how both components, the base and the class automaton can be learned using the respective other. Now the approaches can be interleaved to obtain a complete learning procedure for transparent data automata as follows.

The two learning procedures for the base and the class automaton are started independently. Both processes synchronize only on equivalence queries. After providing a hypothesis, the process has to wait for the other process to come up with a hypothesis as well. These two form the query $(\tilde{\mathcal{A}}, \tilde{\mathcal{B}}) = \tilde{\mathcal{D}} \overset{?}{\equiv} \mathcal{D}$ which is posed to the oracle for $\mathcal{D}$. From a counter-example for $\tilde{\mathcal{D}}$ we can always generate a counter-example for at least one of the processes. This ensures that the hypothesis $\tilde{\mathcal{D}}$ improves in every step. If a counter-example is derived for only one of the processes, the respective other has to wait until a new hypothesis is provided.

Algorithm 3 presents the handling of membership and equivalence queries. The coordination is done by the function ANGLUINPARALLEL that takes the functions for handling both types of membership queries and the equivalence query for the complete hypothesis $\tilde{\mathcal{D}}$. For better readability, Algorithm 3 is simplified in the sense that it generates a counter-example for exactly one of the learning tasks, even though it may be possible to obtain one for each. Such parallelization and also the synchronization part that is omitted here can be implemented in a straight forward manner using, e. g., threads and message channels or an actor model.

Membership. Membership queries for the class automaton $\mathcal{B}$ can be handled as before since the base automaton $\mathcal{A}$ is not involved. For answering membership queries for $\mathcal{A}$ Algorithm 2 looked up a set of words in $\mathcal{B}$ which is now simulated using membership queries for $\mathcal{B}$ instead of a direct lookup.

Equivalence. The individual equivalence queries are handled together forming a hypothesis $\tilde{\mathcal{D}} = (\tilde{\mathcal{A}}, \tilde{\mathcal{B}})$ for $\mathcal{D}$. A positive answer confirms that the single hypotheses are correct. When the oracle provides a counter-example $w \in (\Sigma \times \Delta)^*$ we consider the following two cases.

If $w \in \tilde{\mathcal{D}}$ is a negative counter-example we check each class string $c \in w\!\downharpoonleft$ of w that is rejected by the hypothesis $\tilde{\mathcal{B}}$ for membership in $\mathcal{B}$. That way we either find a counter-example for $\tilde{\mathcal{B}}$ and can proceed with learning $\mathcal{B}$ or $\tilde{\mathcal{B}}$ behaves correctly on w and its string projection $\mathsf{str}(w)$ must then be a counter-example for $\tilde{\mathcal{A}}$ that is used to continue learning $\mathcal{A}$.

A positive counter-example $w \notin \tilde{\mathcal{D}}$ is wrongly rejected by the hypothesis and we check if some class string $c \in w\!\downharpoonleft$ of w is rejected by $\tilde{\mathcal{B}}$. If there is such c, it is supposed to be accepted and returned as counter-example for $\tilde{\mathcal{B}}$. Otherwise $\tilde{\mathcal{B}}$ correctly accepts all class strings of w and thus $\tilde{\mathcal{A}}$ must have wrongly rejected the string projection $\mathsf{str}(w)$ which we return as counter-example for $\tilde{\mathcal{A}}$. Note that with every equivalence query to the oracle we gain a counter-example for at least one of the sub-processes and as soon as one of the sub-processes learned the correct automaton, it does not change anymore until the other process finishes as well.

Algorithm 3 Learning transparent data automata

```
 1: function learnAB( )
 2:     function isMemberA(Word $u$)
 3:         for all $w \in$ dataWords($u$) do
 4:             if $\forall_{c \in w\downharpoonleft}$ : isMemberB($c$) then return isMemberD($w$)
 5:         return ?
 6:     function isMemberB(Word $u$)
 7:         return isMemberD($\binom{u}{11}$)
 8:     function isEquivalentAB(Automaton $\tilde{\mathcal{A}}$, Automaton $\tilde{\mathcal{B}}$)
 9:         result := isEquivalentD($(\tilde{\mathcal{A}}, \tilde{\mathcal{B}})$)
10:         if isPositiveCE(result) then
11:             for all $u \in$ result$\downharpoonleft$ do
12:                 if $u \notin \tilde{\mathcal{B}}$ then return asPositiveCEForB($u$)
13:             return asPositiveCEForA(stringProjection(result))
14:         else if isNegativeCE(result) then
15:             for all $u \in$classStrings(result) do
16:                 if $\neg$isMemberB($u$) then return asNegativeCEForB($u$)
17:             return asNegativeCEForA(stringProjection(result))
18:         else return true
19:     return angluinParallel(isMemberA, isMemberB, isEquivalentAB)
```

Example. We illustrate the complete learning procedure using again the printer example as above. The process for learning $\mathcal{B}$ is started and proceeds as described in Section 3.2 until the first hypothesis $\tilde{\mathcal{B}}_1$ is constructed. Next, the learning procedure for $\mathcal{A}$ starts as described in Section 3.3 except that for checking $r, s, t \in \mathcal{B}$, membership queries for $\mathcal{B}$ are used.

For the guided search, we use $\tilde{\mathcal{B}}_1$ to find suffixes for r, s, t. While knowing $\mathcal{B}$ led to the suffix *st* for r, we now obtain no suitable suffix at all as $\tilde{\mathcal{B}}_1$ does not accept any continuation. We still have $U = V = \{\epsilon\}$ and $T(u) = \textbf{?}$ for all $u \in U\Sigma$. The sample sets are now $S_1 = (\{\epsilon\}, \emptyset)$ but that generates the same hypothesis $\tilde{\mathcal{A}}_1$ accepting Σ^*.

As $\tilde{\mathcal{B}}_1$ rejects any class string, the equivalence query $(\tilde{\mathcal{A}}_1, \tilde{\mathcal{B}}_1) \overset{?}{\equiv} \mathcal{D}$ returns a positive counter-example, e.g. $\binom{rst}{111} \in \mathcal{D}$. The string projection rst is a positive counter-example for $\tilde{\mathcal{B}}_1$ and the process proceeds as before finally coming up with hypothesis $\tilde{\mathcal{B}}_2$.

As rst is already accepted by $\tilde{\mathcal{A}}_1$, it is no counter-example for learning $\mathcal{A}$. The next equivalence query is $(\tilde{\mathcal{A}}_1, \tilde{\mathcal{B}}_2) \overset{?}{\equiv} \mathcal{D}$ yielding a negative counter-example, e.g. $\binom{rrsstt}{121212} \notin \mathcal{D}$. The only class string is rst which is accepted by $\mathcal{B}$ and so we do not obtain a counter-example for $\tilde{\mathcal{B}}_2$ but keep the hypothesis.

For $\tilde{\mathcal{A}}$, $rrsstt \in \tilde{\mathcal{A}}_1$ must be a negative counter-example because $\tilde{\mathcal{B}}_2$ behaved correctly on all class strings. The process for learning $\mathcal{A}$ therefore adds $rrsstt$ and all prefixes to the table (U). In our case, the additional suffixes found by using $\tilde{\mathcal{B}}_2$ for the guided search are the same as we got using $\mathcal{B}$ in Section 3.3.

The obtained table is thus also the same and we obtain a hypothesis with at least three states for which we assume to choose the correct one $\tilde{\mathcal{A}}_2 \equiv \mathcal{A}$

The following equivalence query $(\tilde{\mathcal{A}}_2, \tilde{\mathcal{B}}_2) \overset{?}{\equiv} \mathcal{D}$ yields a negative counter-example, e.g. $\binom{rrrstst}{1212211} \notin \mathcal{D}$. The class strings are $rrst$ and rst and querying them for membership in $\mathcal{B}$ yields $rrst \notin \mathcal{B}$ as negative counter-example for $\tilde{\mathcal{B}}_2$. As before, this leads to the correct hypothesis $\tilde{\mathcal{B}}_3 = \mathcal{B}$.

Theorem 5 (Termination). *The learning procedures (Algorithms 1, 2 and 3) terminate.*

The algorithms employ Angluin's learning algorithm L* which terminates for regular languages [1], also in the setting of possibly inconclusive answers by the oracle [5, 11]. These results apply in our setting as the base and the class automata are finite automata and the algorithms simulate an oracle for the respective regular languages.

4 Conclusion

In this paper, we have presented an active learning algorithm for a subclass of deterministic data automata, which we called transparent data automata. To put this class into a general picture, we have shown that despite data automata being equally expressive to class memory automata, deterministic data automata are a strict subclass of deterministic class memory automata. For the latter, we have shown that their complement is within the class of (non-deterministic) data automata, which comes with a decidable emptiness problem. Thus, equivalence of deterministic class memory automata and thus transparent data automata is decidable, which guarantees that the oracle used within the active learning algorithm is in principle realizable. The transparency condition for data automata intuitively states that local behavior may not be restricted globally, following the idea that a single process should be able to operate especially when no further process is around.

Note that one could consider the case in which global behaviour is part of the local behaviour. Then, one would obtain a similar learning procedure without *don't know*s for the base automaton as for the local automaton in the transparent case. As we do not see any valuable practical setting in which this is satisfied, we do not list the results here.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2) (1987) 87–106
2. Bojanczyk, M., David, C., Muscholl, A., Schwentick, T., Segoufin, L.: Two-variable logic on data words. ACM Trans. Comput. Log. **12**(4) (2011) 27
3. Björklund, H., Schwentick, T.: On notions of regularity for data languages. Theor. Comput. Sci. **411**(4-5) (2010) 702–715

4. Grinchtein, O., Leucker, M., Piterman, N.: Inferring network invariants automatically. In Furbach, U., Shankar, N., eds.: IJCAR. Volume 4130 of Lecture Notes in Computer Science., Springer (2006) 483–497
5. Leucker, M., Neider, D.: Learning minimal deterministic automata from inexperienced teachers. In Margaria, T., Steffen, B., eds.: ISoLA (1). Volume 7609 of Lecture Notes in Computer Science., Springer (2012) 524–538
6. Kaminski, M., Francez, N.: Finite-memory automata. Theor. Comput. Sci. **134**(2) (1994) 329–363
7. Jonsson, B.: Learning of automata models extended with data. In: SFM. Volume 6659 of Lecture Notes in Computer Science., Springer (2011) 327–349
8. Howar, F., Steffen, B., Jonsson, B., Cassel, S.: Inferring canonical register automata. In Kuncak, V., Rybalchenko, A., eds.: VMCAI. Volume 7148 of Lecture Notes in Computer Science., Springer (2012) 251–266
9. Bollig, B., Habermehl, P., Leucker, M., Monmege, B.: A fresh approach to learning register automata. In: Developments in Language Theory - 17th International Conference. Volume 7907 of Lecture Notes in Computer Science., Springer (2013) 118–130
10. Esparza, J., Leucker, M., Schlund, M.: Learning workflow Petri nets. Fundam. Inform. **113**(3-4) (2011) 205–228
11. Biermann, A.W., Feldman, J.A.: On the synthesis of finite-state machines from samples of their behaviour. IEEE Transactions on Computers **21** (1972) 592–597

		ϵ	t	st	rst
	ϵ	✓	✗	✗	✓
	r	✗	✗	✓	✗
	rst	✓	✗	✗	✗
	rs	✗	✓	✗	✗
	rrst	✗	✗	✗	✗
	rrs	✗	✗	✗	✗
	rr	✗	✗	✗	✗
ϵ	r	✗	✗	✓	✗
	s	✗	✗	✗	✗
	t	✗	✗	✗	✗
r	r	✗	✗	✗	✗
	s	✗	✓	✗	✗
	t	✗	✗	✗	✗
rst	r	✗	✗	✗	✗
	s	✗	✗	✗	✗
	t	✗	✗	✗	✗
rs	r	✗	✗	✗	✗
	s	✗	✗	✗	✗
	t	✓	✗	✗	✗
rrst	r	✗	✗	✗	✗
	s	✗	✗	✗	✗
	t	✗	✗	✗	✗
rrs	r	✗	✗	✗	✗
	s	✗	✗	✗	✗
	t	✗	✗	✗	✗
rr	r	✗	✗	✗	✗
	s	✗	✗	✗	✗
	t	✗	✗	✗	✗

Table 1: The final table produced while learning the class automaton $\mathcal{B}$ of the printer example.

	ϵ	st	stst	sstt	tst	stt	tt	t	ttst	tstt	sttt	ststt	sttst	tstst	ssttt	ststt	tsstt	ststst	ssttst	stsstt	sssttt
ϵ	✓	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
rrsstt	✗	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
rrsst	?	?	?	?	?	?	?	✗	?	?	?	?	?	?	?	?	?	?	?	?	?
rrss	?	?	?	?	?	?	✗	?	?	?	?	?	?	?	?	?	?	?	?	?	?
rrs	?	?	?	?	✓	✗	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
rr	?	?	✓	✗	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
r	?	✓	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
ϵ s	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
ϵ t	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
rrsstt r	?	✓	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
rrsstt s	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
rrsstt t	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
rrsst r	?	?	?	?	✗	✗	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
rrsst s	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
rrss r	?	?	?	?	?	?	?	?	✗	✗	✗	?	?	?	?	?	?	?	?	?	?
rrss s	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
rrs r	?	?	?	?	?	?	?	?	?	?	?	?	✗	✓	✗	✗	✗	?	?	?	?
rrs t	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
rr r	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	✓	✗	✗	✗
rr t	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
r s	?	?	?	?	?	?	?	✓	?	?	?	?	?	?	?	?	?	?	?	?	?
r t	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Table 2: The table produced while learning the base automaton $\mathcal{A}$ in the example presented in Section 3.3.