HAL
open science

A Fresh Approach to Learning Register Automata

Benedikt Bollig, Peter Habermehl, Martin Leucker, Benjamin Monmege

To cite this version:

Benedikt Bollig, Peter Habermehl, Martin Leucker, Benjamin Monmege. A Fresh Approach to Learning Register Automata. Developments in Language Theory, 2013, France. pp.118-130, 10.1007/978-3-642-38771-5_12 . hal-00908998

HAL Id: hal-00908998

https://hal.science/hal-00908998v1

Submitted on 25 Nov 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Fresh Approach to Learning Register Automata*

Benedikt Bollig[1], Peter Habermehl[2], Martin Leucker[3], and Benjamin Monmege[1]

[1] LSV, ENS Cachan, CNRS & Inria, France
[2] Univ Paris Diderot, Sorbonne Paris Cité, LIAFA, CNRS, France
[3] ISP, University of Lübeck, Germany

Abstract. This paper provides an Angluin-style learning algorithm for a class of register automata supporting the notion of *fresh* data values. More specifically, we introduce *session automata* which are well suited for modeling protocols in which sessions using fresh values are of major interest, like in security protocols or ad-hoc networks. We show that session automata (i) have an expressiveness partly extending, partly reducing that of register automata, (ii) admit a symbolic regular representation, and (iii) have a decidable equivalence and model-checking problem (unlike register automata). Using these results, we establish a learning algorithm to infer session automata through membership and equivalence queries. Finally, we strengthen the robustness of our automaton by its characterization in monadic second-order logic.

1 Introduction

Learning automata deals with the inference of automata based on some partial information, for example samples, which are words that either belong to their accepted language or not. A popular framework is that of active learning defined by Angluin [2] in which a learner may consult a teacher for so-called membership and equivalence queries to eventually infer the automaton in question. Learning automata has a lot of applications in computer science. Notable examples are the use in model checking [12] and testing [3]. See [18] for an overview.

While active learning of regular languages is meanwhile well understood and is supported by freely available libraries such as learnlib [19] and libalf [8], extensions beyond plain regular languages are still an area of active research. Recently, automata dealing with potentially infinite data as first class citizens have been studied. Seminal works in this area are that of [1, 15] and [14]. While the first two use abstraction and refinement techniques to cope with infinite data, the second approach learns a sub-class of register automata.

In this paper, we follow the work on learning register automata. However, we study a different model than [14], having the ability to require that input data is *fresh* in the sense that it has not been seen so far. This feature has been

* This work is partially supported by EGIDE/DAAD-Procope (LeMon).

proposed in [24] in the context of semantics of programming languages, as, for example, fresh names are needed to model object creation in object-oriented languages. Moreover, fresh data values are important ingredients in modeling security protocols which often make use of so-called fresh nonces to achieve their security assertions [17]. Finally, fresh names are also important in the field of network protocols and are one of the key ingredients of the π-calculus [20].

In general, the equivalence problem of register automata is undecidable (even without freshness). This limits their applicability in active learning, as equivalence queries cannot be implemented (correctly and completely). Therefore, we restrict the studied automaton model to either store fresh data values or read data values from registers. In the terminology of [24], we retain global freshness, while local freshness is discarded. We call our model *session automata*. They are well-suited whenever fresh values are important for a finite period, for which they will be stored in one of the registers. Session automata correspond to the model from [7] without stacks. They are incomparable with the model from [14].

Session automata accept data words, i.e., words over an alphabet $\Sigma \times D$, where Σ is a finite set of labels and D an infinite set of data values. A data word can be mapped to a so-called symbolic word where we record for each different data value the register in which it was stored (when appearing for the first time) or from which it was read later. To each symbolic word we define a symbolic word in unique normal form representing the same data words by fixing a canonical way of storing data values in registers. Then, we show how to transform a session automaton into a unique canonical automaton that accepts the same data language. This canonical automaton can be seen as a classical finite-state automaton and, therefore, we can define an active learning algorithm for session automata in a natural way. In terms of the size of the canonical automaton, the number of membership and equivalence queries needed is polynomial (both in the number of states and in the number of registers). When the reference model are arbitrary (data) deterministic automata, the complexity is polynomial in the number of states and exponential in the number of registers.

Applicability of our framework in verification (e.g., compositional verification [10] and infinite state regular model checking [13]) is underpinned by the fact that session automata form a robust language class: While inclusion is undecidable for register automata [21], we show that it is decidable for session automata. In [7], model checking session automata was shown decidable wrt. a powerful monadic second-order logic with data-equality predicate (dMSO). Here, we also provide a natural fragment of dMSO that precisely captures session automata.

To summarize, we show that session automata (i) have a unique canonical form, (ii) have a decidable inclusion problem, (iii) enjoy a logical characterization, and (iv) can be learned via an active learning algorithm. Altogether, this provides a versatile learning framework for languages over infinite alphabets.

Outline. In Section 2 we introduce session automata. Section 3 presents an active learning algorithm for them and in Section 4 we give some language-theoretic properties of our model and a logical characterization. Missing proofs can be found in the long version: `http://hal.archives-ouvertes.fr/hal-00743240`

2 Data Words and Session Automata

We let $\mathbb{N}$ (respectively, $\mathbb{N}_{>0}$) be the set of natural numbers (respectively, non-zero natural numbers). For $n \in \mathbb{N}$, we let $[n]$ denote the set $\{1, \ldots, n\}$. In the following, we fix a non-empty finite alphabet Σ of *labels* and an infinite set D of *data values*. In examples, we usually use $D = \mathbb{N}$. A *data word* is a sequence of elements of $\Sigma \times D$, i.e., an element from $(\Sigma \times D)^*$. An example data word over $\Sigma = \{a, b\}$ and $D = \mathbb{N}$ is $(a, 4)(b, 2)(b, 4)$.

Our automata will not be able to distinguish between data words that are equivalent up to permutation of data values. Intuitively, this corresponds to saying that data values can only be compared wrt. equality. When two data words w_1 and w_2 are equivalent in that sense, we write $w_1 \approx w_2$, e.g. $(a, 4)(b, 2)(b, 4) \approx (a, 2)(b, 5)(b, 2)$. The equivalence class of a data word w wrt. $\approx$ is written $[w]_{\approx}$.

We can view a data word as being composed of (not necessarily disjoint) sessions, each session determining the scope in which a given data value is used. Let $w = (a_1, d_1) \cdots (a_n, d_n) \in (\Sigma \times D)^*$ be a data word. We let $\mathit{Fresh}(w) \stackrel{\text{def}}{=} \{i \in [n] \mid d_i \neq d_j \text{ for all } j \in \{1, \ldots, i-1\}\}$ be the set of positions of w where a data value occurs for the first time. Accordingly, we let $\mathit{Last}(w) \stackrel{\text{def}}{=} \{i \in [n] \mid d_i \neq d_j$ for all $j \in \{i+1, \ldots, n\}\}$. A set $S \subseteq [n]$ is a *session* of w if there are $i \in \mathit{Fresh}(w)$ and $j \in \mathit{Last}(w)$ such that $S = \{i, \ldots, j\}$ and $d_i = d_j$. For $i \in [n]$, let $\mathit{Session}(i)$ denote the unique session S with $d_{\min(S)} = d_i$. Thus $\mathit{Session}(i)$ is the scope in which d_i is used. Note that $\mathit{Fresh}(w) = \{\min(\mathit{Session}(i)) \mid i \in [n]\}$. For $k \geq 1$, we say that w is *k-bounded* if every position of w belongs to at most k sessions. A language L is k-bounded if every word in L is so. The set of all data words is not k-bounded, for any k. Fig. 1 illustrates a data word w with four sessions. It is 2-bounded, as no position shares more than 2 sessions. We have $\mathit{Session}(7) = \{4, \ldots, 9\}$ and $\mathit{Fresh}(w) = \{1, 2, 4, 6\}$.

Intuitively, k is the number of resources that will be needed to execute a k-bounded word. Speaking in terms of automata, a resource is a register that can store a data value. Our automata will be able to write a fresh data value into some register r, denoted $\mathsf{f}(r)$, or reuse a data value that has already been stored in r, denoted $\mathsf{r}(r)$. In other words, automata will work over (a finite subset of) the alphabet $\Sigma \times \Gamma$ where $\Gamma \stackrel{\text{def}}{=} \{\mathsf{f}(r), \mathsf{r}(r) \mid r \in \mathbb{N}_{>0}\}$. A word over $\Sigma \times \Gamma$ is called a *symbolic word*. Given a symbolic word $u = (a_1, t_1) \cdots (a_n, t_n)$ and a position $i \in [n]$, we let $\mathit{reg}(i)$ denote the register r that is used at i, i.e., such that $t_i \in \{\mathsf{f}(r), \mathsf{r}(r)\}$. Similarly, we define the type $\mathit{type}(i) \in \{\mathsf{f}, \mathsf{r}\}$ of i.

Naturally, a register has to be initialized before it can be used. So, we call u *well formed* if, for all $j \in [n]$ with $\mathit{type}(j) = \mathsf{r}$, there is $i \leq j$ such that

1	2	3	4	5	6	7	8	9
a	b	a	a	c	c	b	c	c
4	2	4	3	2	1	3	1	3

Fig. 1. A data word and its sessions

1	2	3	4	5	6	7	8	9
a	b	a	a	c	c	b	c	c
$\mathsf{f}(1)$	$\mathsf{f}(2)$	$\mathsf{r}(1)$	$\mathsf{f}(1)$	$\mathsf{r}(2)$	$\mathsf{f}(2)$	$\mathsf{r}(1)$	$\mathsf{r}(2)$	$\mathsf{r}(1)$

Fig. 2. A symbolic word

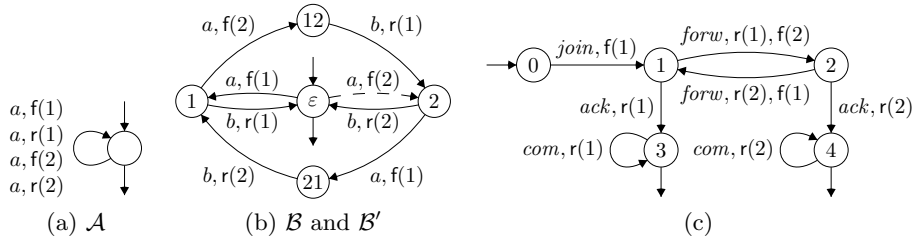

Fig. 3. (a) Session automaton, (b) Client-server system, (c) P2P protocol

$t_i = \mathsf{f}(reg(j))$. Let WF denote the set of well formed words. A well formed symbolic word is illustrated in Fig. 2. We have $type(5) = \mathsf{r}$ and $reg(5) = 2$.

A symbolic word $u = (a_1, t_1) \cdots (a_n, t_n) \in \mathrm{WF}$ generates a set of data words. Intuitively, a position i with $t_i = \mathsf{f}(r)$ opens a new session, writing a fresh data value in register r. The same data value is reused at positions $j > i$ with $t_j = \mathsf{r}(r)$, unless r is reinitialized at some position i' with $i < i' < j$. Formally, $w \in (\Sigma \times D)^*$ is a *concretization* of u if it is of the form $(a_1, d_1) \cdots (a_n, d_n)$ such that, for all $i, j \in [n]$ with $i \leq j$, (i) $i \in Fresh(w)$ iff $type(i) = \mathsf{f}$, and (ii) $d_i = d_j$ iff both $reg(i) = reg(j)$ and there is no position i' with $i < i' \leq j$ such that $t_{i'} = \mathsf{f}(reg(i))$. For example, the data word from Fig. 1 is a concretization of the symbolic word from Fig. 2. By $\gamma(u)$, we denote the set of concretizations of a well formed word u. We extend γ to sets $L \subseteq (\Sigma \times \Gamma)^*$ and let $\gamma(L) \stackrel{\text{def}}{=} \{\gamma(u) \mid u \in L \cap \mathrm{WF}\}$.

Remark 1. Let us state some simple properties of γ. It is easily seen that $w \in \gamma(u)$ implies $\gamma(u) = [w]_\approx$. Let $k \geq 1$. If $u \in \mathrm{WF} \cap (\Sigma \times \Gamma_k)^*$ where $\Gamma_k \stackrel{\text{def}}{=} \{\mathsf{f}(r), \mathsf{r}(r) \mid r \in [k]\}$, then all data words in $\gamma(u)$ are k-bounded. Moreover, $\gamma((\Sigma \times \Gamma_k)^*)$ is the set of all k-bounded data words.

Session Automata. As suggested, we consider automata over the alphabet $\Sigma \times \Gamma$ to process data words. Actually, they are equipped with a *finite* number $k \geq 1$ of registers so that we rather deal with finite automata over $\Sigma \times \Gamma_k$.

Definition 1. *Let $k \geq 1$. A k-*register session automaton *(or just* session automaton*) over Σ and D is a finite automaton over $\Sigma \times \Gamma_k$, i.e., a tuple $\mathcal{A} = (Q, q_0, F, \delta)$ where Q is the finite set of* states*, $q_0 \in Q$ the* initial state*, $F \subseteq Q$ the set of* accepting states*, and $\delta : Q \times (\Sigma \times \Gamma_k) \to 2^Q$ the transition function.*

The *symbolic language* $L_{symb}(\mathcal{A}) \subseteq (\Sigma \times \Gamma_k)^*$ of $\mathcal{A}$ is defined in the usual way, considering $\mathcal{A}$ as a finite automaton. Its *(data) language* is $L_{data}(\mathcal{A}) \stackrel{\text{def}}{=} \gamma(L_{symb}(\mathcal{A}))$. By Remark 1, $L_{data}(\mathcal{A})$ is closed under $\approx$. Moreover, it is k-bounded, which motivates the naming of our automata.

Example 1. Consider the 2-register session automaton $\mathcal{A}$ from Fig. 3(a). It recognizes the set of all 2-bounded data words over $\Sigma = \{a\}$.

Example 2. The 2-register session automaton $\mathcal{B}$ over $\Sigma = \{a, b\}$ from Fig. 3(b) represents a client-server system. A server can receive requests on two channels

of capacity 1, represented by the two registers. Requests are acknowledged in the order in which they are received. When the automaton performs $(a, \mathsf{f}(r))$, a client gets a unique transaction key, which is stored in r. Later, the request is acknowledged performing $(b, \mathsf{r}(r))$. E.g., $(a, 8)(a, 4)(b, 8)(a, 3)(b, 4)(b, 3) \in L_{data}(\mathcal{B})$.

Example 3. Next, we present a 2-register session automaton that models a P2P protocol. A user can join a host with address x, denoted by action $(join, x)$. The request is either forwarded by x to another host y, executing $(forw_1, x)(forw_2, y)$, or acknowledged by (ack, x). In the latter case, a connection between the user and x is established so that they can communicate, indicated by action (com, x). Note that the sequence of actions $(forw_1, x)(forw_2, y)$ should be considered as an encoding of a single action $(forw, x, y)$ and is a way of dealing with actions that actually take two or more data values. An example execution of our protocol is $(join, 145)(forw, 145, 978)(forw, 978, 14)(ack, 14)(com, 14)(com, 14)(com, 14)$. In Fig. 3(c), we show the 2-register session automaton for the P2P protocol.

Session automata come with two natural notions of determinism. We call $\mathcal{A} = (Q, q_0, F, \delta)$ *symbolically deterministic* if $|\delta(q, (a, t))| \leq 1$ for all $q \in Q$ and $(a, t) \in \Sigma \times \Gamma_k$. Then, δ can be seen as a partial function $Q \times (\Sigma \times \Gamma_k) \to Q$. We call $\mathcal{A}$ *data deterministic* if it is symbolically deterministic and, for all $q \in Q$, $a \in \Sigma$, and $r_1, r_2 \in [k]$ with $r_1 \neq r_2$, we have that $\delta(q, (a, \mathsf{f}(r_1))) \neq \emptyset$ implies $\delta(q, (a, \mathsf{f}(r_2))) = \emptyset$. Intuitively, given a data word as input, the automaton is data deterministic if, in each state, given a letter and a data value, there is at most one fireable transition. While "data deterministic" implies "symbolically deterministic", the converse is not true. E.g., the session automata from Fig. 3(a) and 3(b) are symbolically deterministic but not data deterministic. However, the automaton of Fig. 3(b) with the dashed transition removed is data deterministic.

Theorem 1. *Session automata are strictly more expressive than data deterministic session automata.*

The proof can be found in the long version of the paper. Intuitively, data deterministic automata cannot guess if a data value in a register will be reused later.

Session automata are expressively incomparable with the various register automata models considered in [16, 21, 23, 9, 14]. In particular, due to freshness, the languages from Ex. 1, 2, and 3 are not recognizable by the models for which a learning algorithm exists [9, 14]. On the other hand, our model cannot recognize "the set of all data words" or "every two consecutive data values are distinct". Our automata are subsumed by fresh-register automata [24], class memory automata [5], and data automata [6]. However, no algorithm for the inference of the latter is known. Note that, for ease of presentation, we consider one-dimensional data words, unlike [14] where labels have an arity and can carry several data values. Following [7], our automata can be easily extended to multi-dimensional data words (cf. Ex. 3). This also holds for the learning algorithm.

Canonical Session Automata. Our goal will be to infer the data language of a session automaton $\mathcal{A}$ in terms of a canonical session automaton $\mathcal{A}^C$.

As a first step, we associate with a data word $w = (a_1, d_1) \cdots (a_n, d_n) \in (\Sigma \times D)^*$ a *symbolic normal form* $snf(w) \in \mathrm{WF}$ such that $w \in \gamma(snf(w))$, based on the idea that data values are always stored in the first register whose data value is not needed anymore. To do so, we will determine $t_1, \ldots, t_n \in \Gamma$ and set $snf(w) = (a_1, t_1) \cdots (a_n, t_n)$. We define $\tau : Fresh(w) \to \mathbb{N}_{>0}$ inductively by $\tau(i) = \min(FreeReg(i))$ where $FreeReg(i) \stackrel{\text{def}}{=} \mathbb{N}_{>0} \setminus \{\tau(i') \mid i' \in Fresh(w)$ such that $i' < i$ and $i \in Session(i')\}$. With this, we set, for all $i \in [n]$, $t_i = \mathsf{f}(\tau(i))$ if $i \in Fresh(w)$ and $t_i = \mathsf{r}(\tau(\min(Session(i))))$ otherwise. One readily verifies that $snf(w) = (a_1, t_1) \cdots (a_n, t_n)$ is well formed and that properties (i) and (ii) in the definition of a concretization hold. This proves $w \in \gamma(snf(w))$. E.g., Fig. 2 shows the symbolic normal form of the data word from Fig. 1. The mapping *snf* carries over to languages in the expected manner.

We consider again $\mathcal{B}$ of Fig. 3(b). Let $\mathcal{B}'$ be the automaton that we obtain from $\mathcal{B}$ when we remove the dashed transition. We have $L_{data}(\mathcal{B}) = L_{data}(\mathcal{B}')$, but $snf(L_{data}(\mathcal{B})) = L_{symb}(\mathcal{B}') \subsetneqq L_{symb}(\mathcal{B})$.

Lemma 1. *Let L be a regular language over $\Sigma \times \Gamma_k$. Then, $snf(\gamma(L))$ is a regular language over $\Sigma \times \Gamma_k$.*

In other words, for every k-register session automaton $\mathcal{A}$, there is a k-register session automaton $\mathcal{A}'$ such that $L_{symb}(\mathcal{A}') = snf(L_{data}(\mathcal{A}))$ and, therefore, $L_{data}(\mathcal{A}') = L_{data}(\mathcal{A})$. We denote by $\mathcal{A}^C$ the minimal symbolically deterministic automaton $\mathcal{A}'$ satisfying $L_{symb}(\mathcal{A}') = snf(L_{data}(\mathcal{A}))$. Note that the number k' of registers effectively used in $\mathcal{A}^C$ may be smaller than k, and we actually consider $\mathcal{A}^C$ to be a k'-register session automaton.

Theorem 2. *Let $\mathcal{A} = (Q, q_0, F, \delta)$ be a k-register session automaton. Then, $\mathcal{A}^C$ has at most $2^{O(|Q| \times (k+1)! \times 2^k)}$ states. If $\mathcal{A}$ is data deterministic, then $\mathcal{A}^C$ has at most $O(|Q| \times (k+1)! \times 2^k)$ states. Finally, $\mathcal{A}^C$ uses at most k registers.*

3 Learning Session Automata

In this section, we introduce an active learning algorithm for session automata. In the usual active learning setting (as introduced by Angluin [2]), a *learner* interacts with a so-called minimally adequate *teacher* (MAT), an oracle which can answer *membership* and *equivalence queries*. In our case, the learner is given the task to infer the data language $L_{data}(\mathcal{A})$ defined by a given session automaton $\mathcal{A}$. We suppose here that the teacher knows the session automaton or any other device accepting $L_{data}(\mathcal{A})$. In practice, this might not be the case — $\mathcal{A}$ could be a black box — and equivalence queries could be (approximately) answered, for example, by extensive testing. The learner can ask if a *data* word is accepted by $\mathcal{A}$ or not. Furthermore it can ask equivalence queries which consist in giving an *hypothesis* session automaton to the teacher who either answers yes, if the hypothesis is equivalent to $\mathcal{A}$ (i.e., both data languages are the same), or gives a data word which is a counterexample, i.e., a data word that is either accepted by the hypothesis automaton but should not, or vice versa.

Given the data language $L_{data}(\mathcal{A})$ accepted by a session automaton $\mathcal{A}$ over Σ and D, our algorithm will learn the canonical k-register session automaton $\mathcal{A}^C$, i.e., the minimal symbolically deterministic automaton recognizing the data language $L_{data}(\mathcal{A})$ and the regular language $L_{symb}(\mathcal{A}^C)$ over $\Sigma \times \Gamma_k$. Therefore one can consider that the learning target is $L_{symb}(\mathcal{A}^C)$ and use any active learning algorithm for regular languages. However, as the teacher answers only questions over data words, queries have to be adapted. Since $\mathcal{A}^C$ only accepts symbolic words which are in normal form, a membership query for a given symbolic word u not in normal form will be answered negatively (without consulting the teacher); otherwise, the teacher will be given one data word included in $\gamma(u)$ (all the answers on words of $\gamma(u)$ are the same). Likewise, before submitting an equivalence query to the teacher, the learning algorithm checks if the current hypothesis automaton accepts symbolic words not in normal form[4]. If yes, one of those is taken as a counterexample, else an equivalence query is submitted to the teacher. Since the number of registers needed to accept a data language is a priori not known, the learning algorithm starts by trying to learn a 1-register session automaton and increases the number of registers as necessary.

Any active learning algorithm for regular languages may be adapted to our setting. Here we describe a variant of Rivest and Schapire's [22] algorithm which is itself a variant of Angluin's L* algorithm [2]. An overview of learning algorithms for deterministic finite state automata can be found, for example, in [4].

The algorithm is based on the notion of *observation table* which contains the information accumulated by the learner during the learning process. An observation table over a given alphabet $\Sigma \times \Gamma_k$ is a triple $\mathcal{O} = (T, U, V)$ with U, V two sets of words over $\Sigma \times \Gamma_k$ such that $\varepsilon \in U \cap V$ and T is a mapping $(U \cup U \cdot (\Sigma \times \Gamma_k)) \times V \to \{+, -\}$. A table is partitioned into an upper part U and a lower part $U \cdot (\Sigma \times \Gamma_k)$. We define for each $u \in U \cup U \cdot (\Sigma \times \Gamma_k)$ a mapping $row(u)\colon V \to \{+, -\}$ where $row(u)(v) = T(u, v)$. An observation table must satisfy the following property: for all $u, u' \in U$ such that $u \neq u'$ we have $row(u) \neq row(u')$, i.e., there exists $v \in V$ such that $T(u, v) \neq T(u', v)$. This means that the rows of the upper part of the table are pairwise distinct. A table is *closed* if, for all u' in $U \cdot (\Sigma \times \Gamma_k)$, there exists $u \in U$ such that $row(u) = row(u')$. From a closed table we can construct a symbolically deterministic session automaton whose states correspond to the rows of the upper part of the table:

Definition 2. *For a closed table $\mathcal{O} = (T, U, V)$ over a finite alphabet $\Sigma \times \Gamma_k$, we define a symbolically deterministic k-register session automaton $A_{\mathcal{O}} = (Q, q_0, F, \delta)$ over $\Sigma \times \Gamma_k$ by $Q = U$, $q_0 = \varepsilon$, $F = \{u \in Q \mid T(u, \epsilon) = +\}$, and for all $u \in Q$ and $(a, t) \in \Sigma \times \Gamma_k$, $\delta(u, (a, t)) = u'$ if $row(u(a, t)) = row(u')$. This is well defined as the table is closed.*

We now describe in detail our active learning algorithm for a given session automaton $\mathcal{A}$ given in Table 1. It is based on a loop which repeatedly constructs

[4] This can be checked in polynomial time over the trimmed hypothesis automaton with a fixed point computation labelling the states with the registers that should be used again before overwriting them.

```
initialize k := 1 and
𝒪 := (T, U, V) by U = V = {ε} and T(u, ε) for all u ∈ U ∪ U · (Σ × Γ_k) with membership queries
repeat
        while 𝒪 is not closed
            do
                find u ∈ U and (a, t) ∈ Σ × Γ_k such that for all u ∈ U :  row(u(a, t)) ≠ row(u)
                extend table to 𝒪 := (T′, U ∪ {u(a, t)}, V) by membership queries
        from 𝒪 construct the hypothesized automaton 𝒜_𝒪 (cf. Definition 2)
        if 𝒜_𝒪 accepts symbolic words not in normal form
          then let z be one of those
          else  if L_data(𝒜) = L_data(𝒜_𝒪)
                  then  equivalence test succeeds
                  else  get counterexample w ∈ (L_data(𝒜) \ L_data(𝒜_𝒪)) ∪ (L_data(𝒜_𝒪) \ L_data(𝒜))
                        set z := snf(w); find minimal k′ such that z ∈ Σ × Γ_k′
                        if k′ > k
                          then set k := k′
                                  extend table to 𝒪 := (T′, U, V) over Σ × Γ_k by membership queries
        if 𝒪 is closed /* is true if k′ ≤ k */
          then find a breakpoint for z where v is the distinguishing word
                extend table to 𝒪 := (T′, U, V ∪ {v}) by membership queries
  until equivalence test succeeds
return 𝒜_𝒪
```

Table 1. The learning algorithm for a session automaton $\mathcal{A}$

a closed table using membership queries, builds the corresponding automaton and then asks an equivalence query. This is repeated until $\mathcal{A}$ is learned. An important part of any active learning algorithm is the treatment of counterexamples provided by the teacher as an answer to an equivalence query. Suppose that for a given $\mathcal{A}_{\mathcal{O}}$ constructed from a closed table $\mathcal{O} = (T, U, V)$ the teacher answers by a counterexample data word w. Let $z = snf(w)$. If z uses more registers than available in the current alphabet, we extend the alphabet and then the table. If the obtained table is not closed, we restart from the beginning of the loop. Otherwise – and also if z does not use more registers – we use Rivest and Schapire's [22] technique to extend the table by adding a suitable v to V making it non-closed. The technique is based on the notion of breakpoint. As z is a counterexample, (1) $z \in L_{symb}(\mathcal{A}_{\mathcal{O}}) \iff z \notin L_{symb}(\mathcal{A}^C)$. Let $z = z_1 \cdots z_m$. Then, for any i with $1 \leq i \leq m+1$, let z be decomposed as $z = u_i v_i$, where $u_1 = v_{m+1} = \varepsilon$, $v_1 = u_{m+1} = z$ and the length of u_i is equal to $i-1$ (we have also $z = u_i z_i v_{i+1}$ for all i such that $1 \leq i \leq m$). Let s_i be the state visited by z just before reading the ith letter, along the computation of z on $\mathcal{A}_{\mathcal{O}}$: i is a breakpoint if $s_i z_i v_{i+1} \in L_{symb}(\mathcal{A}_{\mathcal{O}}) \iff s_{i+1} v_{i+1} \notin L_{symb}(\mathcal{A}^C)$. Because of (1) such a break-point must exist and can be obtained with $O(\log(m))$ membership queries by a dichotomous search. The word v_{i+1} is called the distinguishing word. If V is extended by v_{i+1} the table is not closed anymore ($row(s_i)$ and $row(s_i z_i)$ are different). Now, the algorithm closes the table again, then asks another equivalence query and so forth until termination. At each iteration of the loop the number of rows (each of those correspond to a state in the automaton $\mathcal{A}^C$) is increased by at least one. Notice that the same counterexample might be given several times. The treatment of the counterexample only guarantees that the table will contain one more row in its upper part. We obtain the following:

Theorem 3. *Let $\mathcal{A}$ be a k'-register session automaton over Σ and D. Let $\mathcal{A}^C$ be the corresponding canonical k-register session automaton. Let N be its number of states, K be the size of $\Sigma \times \Gamma_k$ and M the length of the longest counterexample returned by an equivalence query. Then, the learning algorithm for $\mathcal{A}$ terminates with at most $O(KN^2 + N\log(M))$ membership and $O(N)$ equivalence queries.*

Proof: This follows directly from the proof of correctness and complexity of Rivest and Schapire's algorithm [4, 22]. Notice that the equivalence query cannot return a counterexample whose normal form uses more than k registers, as such a word is rejected by both $\mathcal{A}^C$ (by definition) and by $\mathcal{A}_{\mathcal{O}}$, (by construction). ■

Let us discuss the complexity of our algorithm. In terms of the canonical session automaton, the number of required membership and equivalence queries is polynomial. When we consider data deterministic session automata, the complexity is still polynomial in the number of states, but exponential in k (with constant base). As usual, we have to add one exponent wrt. (data) non-deterministic automata. In [14], the number of equivalence queries is polynomial in the size of the underlying automaton. In contrast, the number of membership queries contains a factor n^k where n is the number of states and k the number of registers. This may be seen as a drawback, as n is typically large. Note that [14] restrict to deterministic automata, since classical register automata are not determinizable.

Table 2. The successive observation tables

$\mathcal{O}_1$	ε
ε	$+$
$(b,\mathsf{r}(1))$	$-$
$(a,\mathsf{f}(1))$	$+$
$(b,\mathsf{r}(1))_-$	$-$

$\Rightarrow$

$\mathcal{O}_2$	ε	$(b,\mathsf{r}(1))$
ε	$+$	$-$
$(b,\mathsf{r}(1))$	$-$	$-$
$(a,\mathsf{f}(1))$	$+$	$+$
$(b,\mathsf{r}(1))_-$	$-$	$-$
$(a,\mathsf{f}(1))(a,\mathsf{f}(1))$	$+$	$+$
$(a,\mathsf{f}(1))(b,\mathsf{r}(1))$	$+$	$+$

$\Rightarrow$

$\mathcal{O}_3$	ε	$(b,\mathsf{r}(1))$
ε	$+$	$-$
$(b,\mathsf{r}(1))$	$-$	$-$
$(a,\mathsf{f}(1))$	$+$	$+$
$(a,\mathsf{f}(2))$	$-$	$-$
$(b,\mathsf{r}(2))$	$-$	$-$
$(b,\mathsf{r}(1))_-$	$-$	$-$
$(a,\mathsf{f}(1))(a,\mathsf{f}(1))$	$+$	$+$
$(a,\mathsf{f}(1))(b,\mathsf{r}(1))$	$+$	$+$
$(a,\mathsf{f}(1))(a,\mathsf{f}(2))$	$-$	$+$
$(a,\mathsf{f}(1))(b,\mathsf{r}(2))$	$-$	$-$

$\Rightarrow$

$\mathcal{O}_4$	ε	$(b,\mathsf{r}(1))$
ε	$+$	$-$
$(b,\mathsf{r}(1))$	$-$	$-$
$(a,\mathsf{f}(1))$	$+$	$+$
$(a,\mathsf{f}(1))(a,\mathsf{f}(2))$	$-$	$+$
$(a,\mathsf{f}(2))$	$-$	$-$
$(b,\mathsf{r}(2))$	$-$	$-$
$(b,\mathsf{r}(1))_-$	$-$	$-$
$(a,\mathsf{f}(1))(a,\mathsf{f}(1))$	$+$	$+$
$(a,\mathsf{f}(1))(b,\mathsf{r}(1))$	$+$	$+$
$(a,\mathsf{f}(1))(b,\mathsf{r}(2))$	$-$	$-$
$(a,\mathsf{f}(1))(a,\mathsf{f}(2))(a,\mathsf{f}(1))$	$-$	$-$
$(a,\mathsf{f}(1))(a,\mathsf{f}(2))(b,\mathsf{r}(1))$	$+$	$+$
$(a,\mathsf{f}(1))(a,\mathsf{f}(2))(a,\mathsf{f}(2))$	$-$	$+$
$(a,\mathsf{f}(1))(a,\mathsf{f}(2))(b,\mathsf{r}(2))$	$-$	$+$

$\Rightarrow$

$\mathcal{O}_5$	ε	$(b,\mathsf{r}(1))$	$(b,\mathsf{r}(2))$
ε	$+$	$-$	$-$
$(b,\mathsf{r}(1))$	$-$	$-$	$-$
$(a,\mathsf{f}(1))$	$+$	$+$	$-$
$(a,\mathsf{f}(1))(a,\mathsf{f}(2))$	$-$	$+$	$-$
$(a,\mathsf{f}(1))(a,\mathsf{f}(2))(b,\mathsf{r}(1))$	$+$	$+$	$+$
$(a,\mathsf{f}(2))$	$-$	$-$	$-$
$(b,\mathsf{r}(2))$	$-$	$-$	$-$
$(b,\mathsf{r}(1))_-$	$-$	$-$	$-$
$(a,\mathsf{f}(1))(a,\mathsf{f}(1))$	$+$	$+$	$-$
$(a,\mathsf{f}(1))(b,\mathsf{r}(1))$	$+$	$+$	$-$
$(a,\mathsf{f}(1))(b,\mathsf{r}(2))$	$-$	$-$	$-$
$(a,\mathsf{f}(1))(a,\mathsf{f}(2))(a,\mathsf{f}(1))$	$-$	$-$	$-$
$(a,\mathsf{f}(1))(a,\mathsf{f}(2))(a,\mathsf{f}(2))$	$-$	$+$	$-$
$(a,\mathsf{f}(1))(a,\mathsf{f}(2))(b,\mathsf{r}(2))$	$-$	$+$	$-$
$(a,\mathsf{f}(1))(a,\mathsf{f}(2))(b,\mathsf{r}(1))(a,\mathsf{f}(1))$	$+$	$+$	$+$
$(a,\mathsf{f}(1))(a,\mathsf{f}(2))(b,\mathsf{r}(1))(b,\mathsf{r}(1))$	$+$	$+$	$+$
$(a,\mathsf{f}(1))(a,\mathsf{f}(2))(b,\mathsf{r}(1))(a,\mathsf{f}(2))$	$-$	$+$	$-$
$(a,\mathsf{f}(1))(a,\mathsf{f}(2))(b,\mathsf{r}(1))(b,\mathsf{r}(2))$	$+$	$+$	$+$

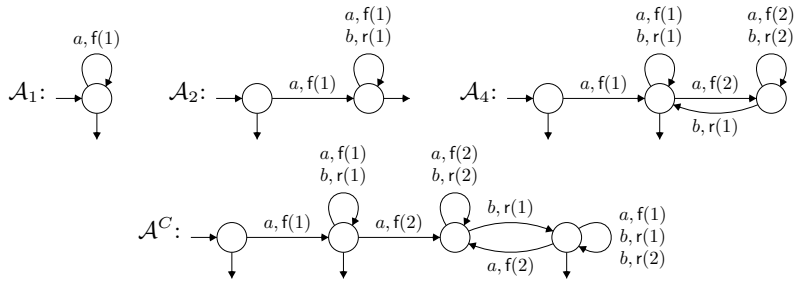

Fig. 4. The successive hypothesis automata

Example 4. We apply our learning algorithm on the data language generated by a single state automaton with loops labelled by $(a, \mathsf{f}(1))$, $(b, \mathsf{r}(1))$, $(a, \mathsf{f}(2))$ and $(b, \mathsf{r}(2))$. Table 2 shows the successive observation tables constructed by the algorithm[5], and Fig. 4 the successive automata constructed from the closed observation tables. For sake of clarity we omit the sink states. We start with the alphabet $\Sigma \times \Gamma_1 = \{(a, \mathsf{f}(1)), (a, \mathsf{r}(1)), (b, \mathsf{f}(1)), (b, \mathsf{r}(1))\}$. Table $\mathcal{O}_1$ is obtained after initialization and closing by adding $(b, \mathsf{r}(1))$ to the top: hypothesis automaton $\mathcal{A}_1$ is constructed. Suppose that the equivalence query gives back as counterexample the data word $(a, 3)(b, 3)$ whose normal form is $(a, \mathsf{f}(1))(b, \mathsf{r}(1))$. Here the breakpoint yields the distinguishing word $(b, \mathsf{r}(1))$. Adding it to V and closing the table by adding $(a, \mathsf{f}(1))$ to the top, we get table $\mathcal{O}_2$ yielding hypothesis automaton $\mathcal{A}_2$. Notice that $L_{symb}(\mathcal{A}_2) = L_{symb}(\mathcal{A}^C) \cap (\Sigma \times \Gamma_1)^*$: the equivalence query must now give back a data word whose normal form is using at least 2 registers (here $(a, 7)(a, 4)(b, 7)$ with normal form $(a, \mathsf{f}(1))(a, \mathsf{f}(2))(b, \mathsf{r}(1))$). Then we must extend the alphabet to $\Sigma \times \Gamma_2$ and obtain table $\mathcal{O}_3$. We close the table and get $\mathcal{O}_4$. After the equivalence query with the hypothesis automaton $\mathcal{A}_4$ we get $(a, \mathsf{f}(1))(a, \mathsf{f}(2))(b, \mathsf{r}(1))(b, \mathsf{r}(2))$ as normal form of the data word counterexample $(a, 9)(a, 3)(b, 9)(b, 3)$. After adding $(b, \mathsf{r}(2))$ to V and closing the table by moving $(a, \mathsf{f}(1))(a, \mathsf{f}(2))(b, \mathsf{r}(1))$ to the top, we get the table $\mathcal{O}_5$ from which the canonical automaton $\mathcal{A}^C$ is obtained and the equivalence query succeeds.

4 Language Theoretical Results

In this section, we establish some language theoretical properties of session automata, which they inherit from classical regular languages. These results demonstrate a certain robustness as required in verification tasks such as compositional verification [10] and infinite-state regular model checking [13].

Theorem 4. *Data languages recognized by session automata are closed under intersection and union. They are also closed under complementation in the following sense: given a k-register session automaton $\mathcal{A}$, the language $\gamma((\Sigma \times \Gamma_k)^*) \setminus L_{data}(\mathcal{A})$ is recognized by a k-register session automaton.*

[5] To save space some letters whose rows contain only $-$'s are omitted. Moreover, we use $_$ to indicate that all letters will lead to the same row.

Theorem 5. *The inclusion problem for session automata is decidable.*

We now provide a logical characterization of session automata. We consider *data MSO logic* (dMSO), which is an extension of classical MSO logic by the binary predicate $x \sim y$ to compare data values: a data word $w = (a_1, d_1) \cdots (a_n, d_n) \in (\Sigma \times D)^*$ with variable interpretation $x \mapsto i$ and $y \mapsto j$ satisfies $x \sim y$ if $d_i = d_j$. More background on dMSO may be found in the long version and [21, 23, 6]. Note that dMSO is a very expressive logic and goes beyond virtually all automata models defined for data words [21, 6, 11]. We identify a fragment of dMSO, called *session MSO logic*, that is expressively equivalent to session automata. While register automata also enjoy a logical characterization [11], we are not aware of logics capturing the automata model considered in [14].

Definition 3. *A* session MSO *(sMSO) formula is a dMSO sentence of the form* $\varphi = \exists X_1 \cdots \exists X_m \ (\alpha \wedge \forall x \forall y \ (x \sim y \leftrightarrow \beta))$ *such that* α *and* β *are classical MSO formulas (not containing the predicate* $\sim$*).*

Example 5. For instance, $\varphi_1 = \forall x \forall y \ (x \sim y \leftrightarrow x = y)$ is an sMSO formula. Its semantics $L_{data}(\varphi_1)$ is the set of data words in which every data value occurs at most once. Moreover, $\varphi_2 = \forall x \forall y \ (x \sim y \leftrightarrow \mathit{true})$ is an sMSO formula, and $L_{data}(\varphi_2)$ is the set of data words where all data values coincide. As a last example, let $\varphi_3 = \exists X \, \forall x \forall y \ (x \sim y \leftrightarrow (\neg \exists z \in X \ (x < z \leq y \vee y < z \leq x)))$. Then, $L_{data}(\varphi_3)$ is the set of 1-bounded data words. Intuitively, the second-order variable X represents the set of positions where a fresh data value is introduced.

Theorem 6. *A data language is recognized by a session automaton iff it is definable by an sMSO formula.*

In [7], it was already shown (for a more powerful model with pushdown stacks) that model checking for the *full* dMSO logic is decidable:

Theorem 7 ([7]). *Given a session automaton* $\mathcal{A}$ *and a dMSO sentence* φ*, one can decide whether* $L_{data}(\mathcal{A}) \subseteq L_{data}(\varphi)$*.*

5 Conclusion

In this paper, we provided a complete framework for algorithmic learning of session automata, a special class of register automata to process data words. As a key ingredient, we associated with every session automaton a canonical one, which revealed close connections with classical regular languages. This also allowed us to show that session automata form a robust language class with good closure and decidability properties as well as a characterization in MSO logic. As a next step, we plan to employ our setting for various verification tasks.

Acknowledgment. We are grateful to Thomas Schwentick for suggesting the symbolic normal form of data words.

References

1. F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F. W. Vaandrager. Automata learning through counterexample guided abstraction refinement. In *FM*, LNCS 7436, pp. 10–27. Springer, 2012.
2. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.
3. T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, and B. Steffen. On the correspondence between conformance testing and regular inference. In *FASE*, LNCS 3442, pp. 175–189. Springer, 2005.
4. T. Berg and H. Raffelt. Model checking. In *Model-based Testing of Reactive Systems*, LNCS 3472 of *LNCS*. Springer, 2005.
5. H. Björklund and Th. Schwentick. On notions of regularity for data languages. *Theoretical Computer Science*, 411(4-5):702–715, 2010.
6. M. Bojanczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Log.*, 12(4):27, 2011.
7. B. Bollig, A. Cyriac, P. Gastin, and K. Narayan Kumar. Model checking languages of data words. In *FoSSaCS*, LNCS 7213, pp. 391–405. Springer, 2012.
8. B. Bollig, J.-P. Katoen, C. Kern, M. Leucker, D. Neider, and D. Piegdon. libalf: the automata learning framework. In *CAV*, LNCS 6174, pp. 360–364. Springer, 2010.
9. S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen. A succinct canonical register automaton model. In *ATVA*, LNCS 6996, pp. 366–380. Springer, 2011.
10. J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *TACAS*, LNCS 2619, pp. 331–346. Springer, 2003.
11. T. Colcombet, C. Ley, and G. Puppis. On the use of guards for logics with data. In *Proceedings of MFCS*, LNCS 6907, pp. 243–255. Springer, 2011.
12. D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *ESEC / SIGSOFT FSE*, pp. 257–266. ACM, 2003.
13. P. Habermehl and T. Vojnar. Regular Model Checking Using Inference of Regular Languages. In *INFINITY'04*, ENTCS 138, pp. 21–36. Elsevier, 2005.
14. F. Howar, B. Steffen, B. Jonsson, and S. Cassel. Inferring canonical register automata. In *VMCAI*, LNCS 7148, pp. 251–266. Springer, 2012.
15. B. Jonsson. Learning of automata models extended with data. In *SFM*, LNCS 6659, pp. 327–349. Springer, 2011.
16. M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.
17. K. O. Kürtz, R. Küsters, and T. Wilke. Selecting theories and nonce generation for recursive protocols. In *FMSE*, pp. 61–70. ACM, 2007.
18. M. Leucker. Learning meets verification. LNCS 4709, pp. 127–151. Springer, 2007.
19. T. Margaria, H. Raffelt, B. Steffen, and M. Leucker. The LearnLib in FMICS-jETI. In *ICECCS*, pp. 340–352. IEEE Computer Society Press, 2007.
20. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, 100:1–77, Sept. 1992.
21. F. Neven, Th. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Transactions on Computational Logic*, 5(3):403–435, 2004.
22. R. Rivest and R. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103:299–347, 1993.
23. L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In *CSL*, LNCS 4207, pp. 41–57. Springer, 2006.
24. N. Tzevelekos. Fresh-register automata. In *POPL*, pp. 295–306. ACM, 2011.