A Robust Class of Data Languages and an Application to Learning

Benedikt Bollig, Peter Habermehl, Martin Leucker, Benjamin Monmege

A Robust Class of Data Languages and an Application to Learning*

Benedikt Bollig[1], Peter Habermehl[2], Martin Leucker[3], and Benjamin Monmege[4]

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

Abstract. We introduce *session automata*, an automata model to process data words, i.e., words over an infinite alphabet. Session automata support the notion of *fresh* data values, 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. Session automata have an expressiveness partly extending, partly reducing that of classical register automata. We show that, unlike register automata and their various extensions, session automata are robust: They (i) are closed under intersection, union, and (resource-sensitive) complementation, (ii) admit a symbolic regular representation, (iii) have a decidable inclusion problem (unlike register automata), and (iv) enjoy logical characterizations. Using these results, we establish a learning algorithm to infer session automata through membership and equivalence queries.

1 Introduction

The study of automata over data words, i.e., words over an infinite alphabet, has its origins in the seminal work by Kaminski and Francez [18]. Their finite-memory automata (more commonly called *register automata*) equip finite-state machines with registers in which data values (from the infinite alphabet) can be stored and be reused later. Register automata preserve some of the nice properties of finite automata: they have a decidable emptiness problem and are closed under union and intersection. On the other hand, register automata are neither determinizable nor closed under complementation, and they have an undecidable equivalence/inclusion problem. There are actually several variants of register automata, which all have the same expressive power but differ in the complexity of decision problems [13, 5]. In the sequel, many more automata models have been introduced (not necessarily with registers), aiming at a good balance between expressivity, decidability, and closure properties [25, 13, 19, 7]. As those models extend register automata, they inherit some drawbacks such as undecidability of the equivalence problem.

* This work is partially supported by EGIDE/DAAD-Procope (LeMon) and the European project Cassting.

We will follow the work on register automata and study a model that supports the notion of *freshness*. When reading a data value, it may enforce that the data value is *fresh*, i.e., it has not occurred in the whole history of the run. This feature has been proposed in [28] in the context of programming language semantics. Actually, fresh names are needed to model object creation in object-oriented languages, and they are important ingredients in modeling security protocols which often make use of so-called fresh nonces to achieve their security assertions [20]. Fresh names are also crucial in the field of network protocols, and they are one of the key features of the π-calculus [24]. Like ordinary register automata, fresh-register automata preserve some of the good properties of finite automata. However, they are not closed under complement and also come with an undecidable equivalence problem.

In this paper, we propose *session automata*, a robust automata model over data words. Like register automata, session automata are a syntactical restriction of fresh-register automata, but in an orthogonal way. Register automata drop the feature of *global freshness* (referring to the whole history) while keeping a local variant (referring to the registers). Session automata, on the other hand, discard local freshness, while keeping the global one. Session automata are well-suited whenever fresh values are important for a finite period, for which they will be stored in one of the registers. They correspond to the model from [8] without stacks. We show that session automata (i) are closed under intersection, union, and resource-sensitive complementation[5], (ii) have a unique canonical form (analogous to minimal deterministic finite automata), (iii) have a decidable equivalence/inclusion problem, and (iv) enjoy logical characterizations. Altogether, this provides a versatile framework for languages over infinite alphabets.

In a second part of the paper, we present an application of our automata model in the area of learning, where decidability of the equivalence problem is crucial. 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 [14] and testing [3]. See [22] for an overview.

While active learning of regular languages is meanwhile well understood and is supported by freely available libraries such as learnlib [23] and libalf [10], 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, 17] and [16]. While the first two use abstraction and refinement techniques to cope with infinite data, the second approach learns a sub-class of register automata. Note that session automata are incomparable with the model from [16]. Thanks to their closure and

[5] A notion similar to [21], but for a different model.

decidability properties, a conservative extension of Angluin's classical algorithm will do for their automatic inference.

Outline. The paper is structured as follows. In Section 2 we introduce session automata. Section 3 presents the main tool allowing us to establish the results of this paper, namely the use of data words in symbolic normal form and the construction of a canonical session automaton. The section also presents some closure properties of session automata and the decidability of the equivalence problem. Section 4 gives logical characterizations of our model. In Section 5, we present an active learning algorithm for session automata. This paper is an extended version of [9].

2 Data Words and Session Automata

We let $\mathbb{N}$ be the set of natural numbers and $\mathbb{N}_{>0}$ be the set of non-zero natural numbers. 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* over Σ and D is a sequence $w = (a_1, d_1) \cdots (a_n, d_n)$ of pairs $(a_i, d_i) \in \Sigma \times D$. In other words, w is an element from $(\Sigma \times D)^*$. For $d \in \{d_1, \ldots, d_n\}$, we let $\mathit{first}_w(d)$ denote the position $j \in \{1, \ldots, n\}$ where d occurs for the first time, i.e., such that $d_j = d$ and there is no $k < j$ such that $d_k = d$. Accordingly, we define $\mathit{last}_w(d)$ to be the last position where d occurs.

An example data word over $\Sigma = \{a, b\}$ and $D = \mathbb{N}$ is given by $w = (a, 8)(b, 4)(a, 8)(c, 3)(a, 4)(b, 4)(a, 9)$. We have $\mathit{first}_w(4) = 2$ and $\mathit{last}_w(4) = 6$.

This section recalls two existing automata models over data words – namely register automata, previously introduced by [18], and fresh-register automata, studied in [29] as a generalization of register automata. Moreover, we introduce the new model of session automata, our main object of interest.

Register automata (initially called finite-memory automata) equip finite-state machines with registers in which data values can be stored and be read out later. Fresh-register automata additionally come with an oracle that can determine if a data value is *fresh*, i.e., has not occurred in the history of a run. Both register and fresh-register automata are closed under union and intersection, and they have a decidable emptiness problem. However, they are not closed under complementation, and their equivalence problem is undecidable, which limits their application in areas such as model checking and automata learning. Session automata, on the other hand, are closed under (resource-sensitive) complementation, and they have a decidable inclusion/equivalence problem.

Given a set $\mathcal{R}$, we let $\mathcal{R}^{\uparrow} \stackrel{\text{def}}{=} \{r^{\uparrow} \mid r \in \mathcal{R}\}$, $\mathcal{R}^{\odot} \stackrel{\text{def}}{=} \{r^{\odot} \mid r \in \mathcal{R}\}$, and $\mathcal{R}^{\circledast} \stackrel{\text{def}}{=} \{r^{\circledast} \mid r \in \mathcal{R}\}$. In the automata models that we are going to introduce, $\mathcal{R}$ will be the set of registers. Transitions will be labeled with an element from $\mathcal{R}^{\circledast} \cup \mathcal{R}^{\odot} \cup \mathcal{R}^{\uparrow}$, which determines a register and the operation that is performed on it. More precisely, $r^{\circledast}$ writes a globally fresh value into r, $r^{\odot}$ writes a locally fresh value into r, and $r^{\uparrow}$ uses the value that is currently stored in r. For $\pi \in$

$\mathcal{R}^{\circledast} \cup \mathcal{R}^{\odot} \cup \mathcal{R}^{\uparrow}$, we let $\mathsf{reg}(\pi) = r$ if $\pi \in \{r^{\circledast}, r^{\odot}, r^{\uparrow}\}$. Similarly,

$$\mathsf{op}(\pi) = \begin{cases} \circledast & \text{if } \pi \text{ is of the form } r^{\circledast} \\ \odot & \text{if } \pi \text{ is of the form } r^{\odot} \\ \uparrow & \text{if } \pi \text{ is of the form } r^{\uparrow}\,. \end{cases}$$

Definition 1 (Fresh-Register Automaton, cf. [29]). *A* fresh-register automaton *is a tuple* $\mathcal{A} = (S, \mathcal{R}, \iota, F, \Delta)$ *where*

- S *is the non-empty finite set of* states*,*
- $\mathcal{R}$ *is the non-empty finite set of* registers*,*
- $\iota \in S$ *is the* initial state*,*
- $F \subseteq S$ *is the set of* final states*, and*
- Δ *is a finite set of* transitions*: each transition is a tuple of the form* $(s, (a, \pi), s')$ *where* $s, s' \in S$ *are the source and target state, respectively,* $a \in \Sigma$*, and* $\pi \in \mathcal{R}^{\circledast} \cup \mathcal{R}^{\odot} \cup \mathcal{R}^{\uparrow}$*. We call* (a, π) *the* transition label*.*

For a transition $(s, (a, \pi), s') \in \Delta$, we also write $s \xrightarrow{(a,\pi)} s'$. When taking this transition, the automaton moves from state s to state s' and reads a symbol $(a, d) \in \Sigma \times D$. If $\pi = r^{\uparrow} \in \mathcal{R}^{\uparrow}$, then d is the data value that is currently stored in register r. If $\pi = r^{\circledast} \in \mathcal{R}^{\circledast}$, then d is some *globally fresh* data value, which has not been read in the *whole* history of the run; d is then written into register r. Finally, if $\pi = r^{\odot} \in \mathcal{R}^{\odot}$, then d is some *locally fresh* data value, which is *currently* not stored in the registers; it will henceforth be stored in register r.

Let us formally define the semantics of $\mathcal{A}$. A *configuration* is a triple $\gamma = (s, \tau, U)$ where $s \in S$ is the current state, $\tau : \mathcal{R} \rightharpoonup D$ is a partial mapping encoding the current register assignment, and $U \subseteq D$ is the set of data values that have been used so far. By $dom(\tau)$, we denote the set of registers r such that $\tau(r)$ is defined. We say that γ is *final* if $s \in F$. As usual, we define a transition relation over configurations and let $(s, \tau, U) \xRightarrow{(a,d)} (s', \tau', U')$, where $(a, d) \in \Sigma \times D$, if there is a transition $s \xrightarrow{(a,\pi)} s'$ such that the following conditions hold:

1. $\begin{cases} d = \tau(\mathsf{reg}(\pi)) & \text{if } \mathsf{op}(\pi) = \uparrow \\ d \notin \tau(\mathcal{R}) & \text{if } \mathsf{op}(\pi) = \odot \\ d \notin U & \text{if } \mathsf{op}(\pi) = \circledast\,, \end{cases}$
2. $dom(\tau') = dom(\tau) \cup \{\mathsf{reg}(\pi)\}$ and $U' = U \cup \{d\}$,
3. $\tau'(\mathsf{reg}(\pi)) = d$ and $\tau'(r) = \tau(r)$ for all $r \in dom(\tau) \setminus \{\mathsf{reg}(\pi)\}$.

A run of $\mathcal{A}$ on a data word $(a_1, d_1) \cdots (a_n, d_n) \in (\Sigma \times D)^*$ is a sequence

$$\gamma_0 \xRightarrow{(a_1,d_1)} \gamma_1 \xRightarrow{(a_2,d_2)} \cdots \xRightarrow{(a_n,d_n)} \gamma_n$$

for suitable configurations $\gamma_0, \ldots, \gamma_n$ with $\gamma_0 = (\iota, \emptyset, \emptyset)$ (here the partial mapping $\emptyset$ represents the mapping with empty domain). The run is *accepting* if γ_n is a final configuration. The *language* $L(\mathcal{A}) \subseteq (\Sigma \times D)^*$ of $\mathcal{A}$ is then defined

as the set of data words for which there is an accepting run. Note that fresh-register automata cannot distinguish between data words that are equivalent up to permutation of data values. More precisely, given $w, w' \in (\Sigma \times D)^*$, we write $w \approx w'$ if $w = (a_1, d_1) \cdots (a_n, d_n)$ and $w' = (a_1, d'_1) \cdots (a_n, d'_n)$ such that, for all $i, j \in \{1, \ldots, n\}$, we have $d_i = d_j$ iff $d'_i = d'_j$. For instance, $(a, 4)(b, 2)(b, 4) \approx (a, 2)(b, 5)(b, 2)$. Now, if $w \approx w'$, then we actually have $w \in L(\mathcal{A})$ iff $w' \in L(\mathcal{A})$. In the following, the equivalence class of a data word w wrt. $\approx$ is written $[w]_{\approx}$.

We obtain natural subclasses of fresh-register automata when we restrict the transition labels $(a, \pi) \in \Sigma \times (\mathcal{R}^{\circledast} \cup \mathcal{R}^{\odot} \cup \mathcal{R}^{\uparrow})$ in the transitions.

Definition 2 (Register Automaton, [18]). *A* register automaton *is a fresh-register automaton where every transition label is from* $\Sigma \times (\mathcal{R}^{\odot} \cup \mathcal{R}^{\uparrow})$.

Like register automata, session automata are a syntactical restriction of fresh-register automata, but in an orthogonal way. Instead of local freshness, they include the feature of global freshness.

Definition 3 (Session Automaton). *A* session automaton *is a fresh-register automaton where every transition label is from* $\Sigma \times (\mathcal{R}^{\circledast} \cup \mathcal{R}^{\uparrow})$.

We first compare the three models of automata introduced above in terms of expressive power.

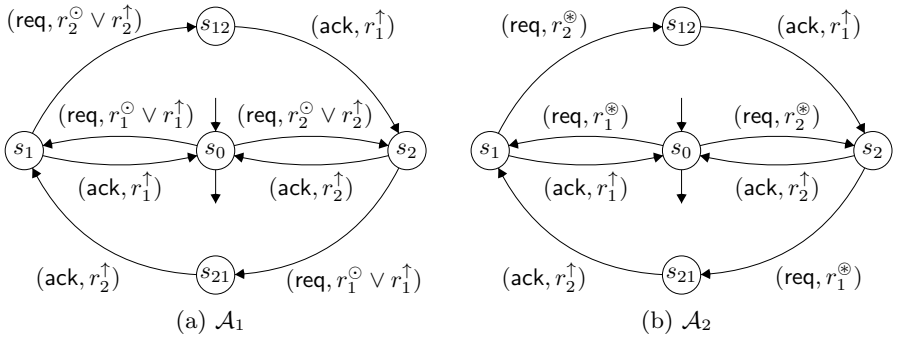
Example 4. Consider the set of labels $\Sigma = \{\mathsf{req}, \mathsf{ack}\}$ and the set of data values $D = \mathbb{N}$, representing an infinite supply of process identifiers (pids). We model a simple (sequential) system where processes can approach a server and make a request, indicated by req, and where the server can acknowledge these requests, indicated by ack. More precisely, $(\mathsf{req}, p) \in \Sigma \times D$ means that the process with pid p performs a request, which is acknowledged when the system executes (ack, p).

Figure 1(a) depicts a register automaton that recognizes the language L_1 of data words verifying the following conditions:

- there are at most two open requests at a time;
- a process waits for an acknowledgment before making another request;
- every acknowledgment is preceded by a request;
- requests are acknowledged in the order they are received.

In the figure, an edge label of the form $(\mathsf{req}, r_i^{\odot} \vee r_i^{\uparrow})$ shall denote that there are two transitions, one labeled with $(\mathsf{req}, r_i^{\odot})$, and one labeled with $(\mathsf{req}, r_i^{\uparrow})$. The automaton models a server that can store two requests at a time and will acknowledge them in the order they are received. For example, it accepts $(\mathsf{req}, 8)(\mathsf{req}, 4)(\mathsf{ack}, 8)(\mathsf{req}, 3)(\mathsf{ack}, 4)(\mathsf{req}, 8)(\mathsf{ack}, 3)(\mathsf{ack}, 8)$.

To guarantee that every process makes at most one request, we actually need the global freshness operator. Figure 1(b) hence depicts a session automaton recognizing the language L_2 of all the data words of L_1 in which every process makes at most one request. We obtain $\mathcal{A}_2$ from $\mathcal{A}_1$ by replacing any occurrence of $r_i^{\odot}$ with $r_i^{\circledast}$. While $(\mathsf{req}, 8)(\mathsf{req}, 4)(\mathsf{ack}, 8)(\mathsf{req}, 3)(\mathsf{ack}, 4)(\mathsf{req}, 8)(\mathsf{ack}, 3)(\mathsf{ack}, 8)$



(a) $\mathcal{A}_1$ (b) $\mathcal{A}_2$

Figure 1: (a) Register automaton $\mathcal{A}_1$ for L_1, (b) Session automaton $\mathcal{A}_2$ for L_2

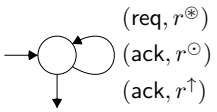


Figure 2: Fresh-register automaton $\mathcal{A}_3$ for L_3

is no longer contained in L_2, $(\mathsf{req}, 8)(\mathsf{req}, 4)(\mathsf{ack}, 8)(\mathsf{req}, 3)(\mathsf{ack}, 4)(\mathsf{ack}, 3)$ is still accepted.

As a last example, consider the language L_3 of data words in which every process makes at most one request (without any other condition). A fresh-register automaton recognizing it is given in Figure 2.

Proposition 5. *Register automata and session automata are incomparable in terms of expressive power. Moreover, fresh-register automata are strictly more expressive than both register automata and session automata.*

Proof. We use the languages L_1, L_2, and L_3 defined in Example 4 to separate the different automata models.

First, the language L_1, recognizable by a register automaton, is not recognized by any session automaton. Indeed, denoting w_d the data word $(\mathsf{req}, d)(\mathsf{ack}, d)$, no session automaton using k registers can accept

$$w_1 w_2 \cdots w_k w_{k+1} w_k \cdots w_2 w_1 \in L_1 \,.$$

Intuitively, the session automaton must store all $k+1$ data values of the requests in order to check the acknowledgement, and cannot discard any of the k first data values to store the $(k+1)$th since all of them have to be reused afterwards (and at that time they are not globally fresh anymore).

Then, the language L_2, recognizable by a session automaton, is indeed not recognizable by a register automaton, for the same reasons as already developed in Proposition 5 of [18]. Intuitively, the store automaton needs to register every data value encountered since it has to ensure the freshness of every pid.

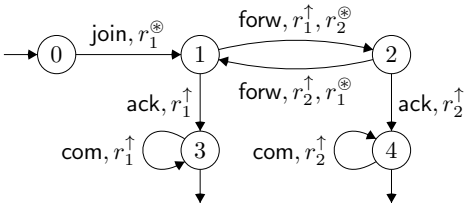


Figure 3: Session automaton for the P2P protocol

Finally, language L_3, recognized by a fresh-register automaton, is not recognized by any register automaton (see again Proposition 5 of [18]) nor by any session automaton. In particular, no session automaton with k registers can accept the data word

$$(\mathsf{req}, 1)(\mathsf{req}, 2) \cdots (\mathsf{req}, k+1)(\mathsf{ack}, 1)(\mathsf{ack}, 2) \cdots (\mathsf{ack}, k+1) \in L_3$$

since when reading the letter $(\mathsf{req}, k+1)$, all the $k+1$ data values seen so far should be registered to accept the suffix afterwards.

Example 6. To conclude the section, we present a session automaton with 2 registers 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 $(\mathsf{forw}_1, x)(\mathsf{forw}_2, y)$, 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 $(\mathsf{forw}_1, x)(\mathsf{forw}_2, y)$ 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, as considered, e.g., in [16]. An example execution of our protocol is $(\mathsf{join}, 145)(\mathsf{forw}, 145, 978)(\mathsf{forw}, 978, 14)(\mathsf{ack}, 14)(\mathsf{com}, 14)(\mathsf{com}, 14)(\mathsf{com}, 14)$. In Figure 3, we show the session automaton for the P2P protocol: it uses 2 registers. Following [8], our automata can be easily extended to multi-dimensional data words. This also holds for the learning algorithm that will be presented in Section 5.

3 Symbolic Normal Form and Canonical Session Automata

Closure properties of session automata, decidability of inclusion/equivalence and the learning algorithm will be established by means of a symbolic normal form of a data word, as well as a canonical session automaton recognizing those normal forms. The crucial observation is that data equality in a data word recognized by a session automaton only depends on the transition labels that generate it. In this section, we suppose that the set of registers of a session automaton is of the form $\mathcal{R} = \{1, \ldots, k\}$. In the following, we let $\Gamma = \mathbb{N}_{>0}^{\circledast} \cup \mathbb{N}_{>0}^{\uparrow}$ and, for $k \geq 1$, $\Gamma_k = \{1, \ldots, k\}^{\circledast} \cup \{1, \ldots, k\}^{\uparrow}$.

3.1 Data Words in Symbolic Normal Forms

Suppose a session automaton reads a sequence $u = (a_1, \pi_1) \cdots (a_n, \pi_n) \in (\Sigma \times \Gamma)^*$ of transition labels. We call u a *symbolic word*. It "produces" a data word if, and only if, a register is initialized before it is used. Formally, we say that u is *well-formed* if, for all positions $j \in \{1, \ldots, n\}$ with $\mathsf{op}(\pi_j) = \uparrow$, there is $i < j$ such that $\pi_i = \mathsf{reg}(\pi_j)^{\circledast}$. Let $\mathsf{WF} \subseteq (\Sigma \times \Gamma)^*$ be the set of all well-formed words.

With $u = (a_1, \pi_1) \cdots (a_n, \pi_n) \in (\Sigma \times \Gamma)^*$, we can associate an equivalence relation $\sim_u$ over $\{1, \ldots, n\}$, letting $i \sim_u j$ if, and only if,

- $\mathsf{reg}(\pi_i) = \mathsf{reg}(\pi_j)$, and one of the following holds:
- $i \leq j$ and there is no position $k \in \{i+1, \ldots, j\}$ such that $\pi_k = \mathsf{reg}(\pi_i)^{\circledast}$, or $j \leq i$ and there is no position $k \in \{j+1, \ldots, i\}$ such that $\pi_k = \mathsf{reg}(\pi_j)^{\circledast}$.

If u is well-formed, then the data values of any data word $w = (a_1, d_1) \cdots (a_n, d_n)$ that a session automaton "accepts via" u conform with the equivalence relation $\sim_u$, that is, we have $d_i = d_j$ iff $i \sim_u j$. This motivates the following definition. Given a well-formed word $u = (a_1, \pi_1) \cdots (a_n, \pi_n) \in (\Sigma \times \Gamma)^*$, we call $w \in (\Sigma \times D)^*$ a *concretization* of u if it is of the form $w = (a_1, d_1) \cdots (a_n, d_n)$ such that, for all $i, j \in \{1, \ldots, n\}$, we have $d_i = d_j$ iff $i \sim_u j$. For example, $w = (a, 8)(a, 5)(b, 8)(a, 3)(b, 3)$ is a concretization of $u = (a, 1^{\circledast})(a, 2^{\circledast})(b, 1^{\uparrow})(a, 2^{\circledast})(b, 2^{\uparrow})$.

Let $\gamma(u)$ denote the set of all concretizations of u. Observe that, if w is any data word from $\gamma(u)$, then $\gamma(u) = [w]_{\approx}$. Concretization is extended to sets $L \subseteq (\Sigma \times \Gamma)^*$ of well-formed words, and we let $\gamma(L) \stackrel{\text{def}}{=} \bigcup_{u \in L \cap \mathsf{WF}} \gamma(u)$. Note that, here, we first filter the well-formed words before applying the operator. Now, let $\mathcal{A} = (S, \mathcal{R}, \iota, F, \Delta)$ be a session automaton. In the obvious way, we may consider $\mathcal{A}$ as a finite automaton over the finite alphabet $\Sigma \times (\mathcal{R}^{\circledast} \cup \mathcal{R}^{\uparrow})$. We then obtain a regular language $L_{symb}(\mathcal{A}) \subseteq (\Sigma \times \Gamma)^*$ (indeed, $L_{symb}(\mathcal{A}) \subseteq (\Sigma \times \Gamma_k)^*$ if $\mathcal{R} = \{1, \ldots, k\}$). It is not difficult to verify that $L(\mathcal{A}) = \gamma(L_{symb}(\mathcal{A}))$.

Though we have a symbolic representation of data languages recognized by session automata, it is in general difficult to compare their languages, since different symbolic words may give rise to the same concretizations. For example, we have $\gamma((a, 1^{\circledast})(a, 1^{\circledast})(a, 1^{\uparrow})) = \gamma((a, 1^{\circledast})(a, 2^{\circledast})(a, 2^{\uparrow}))$. However, we can associate, with every data word, a symbolic normal form, producing the same set of concretizations. Intuitively, the normal form uses the first (according to the natural total order) register whose current data value *will not be used anymore*. In the above example, $(a, 1^{\circledast})(a, 1^{\circledast})(a, 1^{\uparrow})$ would be in symbolic normal form: the data value stored at the first position in register 1 is not reused so that, at the second position, register 1 *must* be overwritten. For the same reason, $(a, 1^{\circledast})(a, 2^{\circledast})(a, 2^{\uparrow})$ is not in symbolic normal form, in contrast to $(a, 1^{\circledast})(a, 2^{\circledast})(a, 2^{\uparrow})(a, 1^{\uparrow})$ where register 1 is read at the end of the word.

Formally, given a data word $w = (a_1, d_1) \cdots (a_n, d_n)$, we define its symbolic normal form $\mathit{snf}(w) \stackrel{\text{def}}{=} (a_1, \pi_1) \cdots (a_n, \pi_n) \in (\Sigma \times \Gamma)^*$ inductively, along with sets $\mathit{Free}(i) \subseteq \mathbb{N}_{>0}$ indicating the registers that are reusable after executing position $i \in \{1, \ldots, n\}$. Setting $\mathit{Free}(0) = \mathbb{N}_{>0}$, we define

$$\pi_i = \begin{cases} \min(\mathit{Free}(i-1))^{\circledast} & \text{if } i = \mathit{first}_w(d_i) \\ \mathsf{reg}(\pi_{\mathit{first}_w(d_i)})^{\uparrow} & \text{otherwise}\,, \end{cases}$$

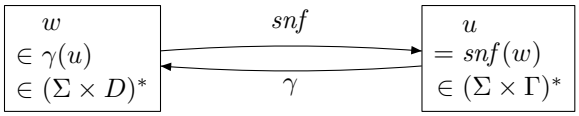
and

$$\mathit{Free}(i) = \begin{cases} \mathit{Free}(i-1) \setminus \min(\mathit{Free}(i-1)) & \text{if } i = \mathit{first}_w(d_i) \neq \mathit{last}_w(d_i) \\ \mathit{Free}(i-1) \cup \{\mathsf{reg}(\pi_i)\} & \text{if } i = \mathit{last}_w(d_i) \\ \mathit{Free}(i-1) & \text{otherwise}\,. \end{cases}$$

We canonically extend *snf* to data languages L, setting $\mathit{snf}(L) = \bigcup_{w \in L} \mathit{snf}(w)$.

Example 7. Let $w = (a,8)(b,4)(a,8)(c,3)(a,4)(b,3)(a,9)$. Then, we have $\mathit{snf}(w) = (a,1^{\circledast})(b,2^{\circledast})(a,1^{\uparrow})(c,1^{\circledast})(a,2^{\uparrow})(b,1^{\uparrow})(a,1^{\circledast})$.

The relation between the mappings γ and *snf* is illustrated below



One easily verifies that $L = \gamma(\mathit{snf}(L))$, for all data languages L. Therefore, equality of data languages reduces to equality of their symbolic normal forms:

Lemma 8. *Let L and L' be data languages. Then, $L = L'$ if, and only if, $\mathit{snf}(L) = \mathit{snf}(L')$.*

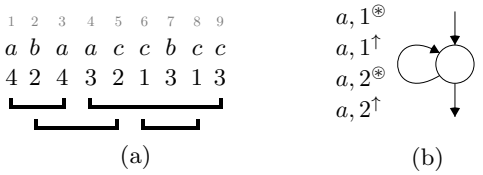
Of course, symbolic normal forms may use any number of registers so that the set of symbolic normal forms is a language over an infinite alphabet as well. However, given a session automaton $\mathcal{A}$, the symbolic normal forms that represent the language $L(\mathcal{A})$ uses only a bounded (i.e., finite) number of registers. Indeed, an important notion in the context of session automata is the *bound* of a data word. Intuitively, the bound of $w = (a_1,d_1)\ldots(a_n,d_n) \in (\Sigma\times D)^*$ is the minimal number of registers that a session automaton needs in order to execute w. Or, in other words, the bound is the maximal number of overlapping *sessions*. A session is an interval delimiting the occurrence of one particular data value. Formally, a session of w is any set $I \subset \mathbb{N}_{>0}$ of the form $\{\mathit{first}_w(d), \mathit{first}_w(d)+1, \ldots, \mathit{last}_w(d)\}$ with $d \in D$ a data value appearing in the word w. Given $k \in \mathbb{N}_{>0}$, we say that w is *k-bounded* if every position $i \in \{1,\ldots,n\}$ is contained in at most k sessions. Let DW_k denote the set of k-bounded data words, and let $\mathsf{SNF}_k \stackrel{\text{def}}{=} \mathit{snf}(\mathsf{DW}_k)$ denote the set of symbolic normal forms of all k-bounded data words.

One can verify that a data word w is k-bounded if, and only if, $\mathit{snf}(w)$ is a word over the alphabet $\Sigma \times \Gamma_k$. Notice that $\mathsf{DW}_k = \gamma((\Sigma \times \Gamma_k)^*)$. Indeed, inclusion $\mathsf{DW}_k \supseteq \gamma((\Sigma \times \Gamma_k)^*)$ is trivial. If, on the other hand, $w \in \mathsf{DW}_k$, we must have $\mathit{snf}(w) \in (\Sigma\times\Gamma_k)^*$, which implies that $w \in \gamma(\mathit{snf}(w)) \subseteq \gamma((\Sigma\times\Gamma_k)^*)$.

A data language L is said to be *k-bounded* if $L \subseteq \mathsf{DW}_k$. It is *bounded* if it is k-bounded for some k. Note that the set of all data words is not bounded.

Figure 4(a) illustrates a data word w with four different sessions. It is 2-bounded, as no position shares more than 2 sessions.

Example 9. Consider the session automaton from Figure 4(b). It recognizes the set of all 2-bounded data words over $\Sigma = \{a\}$.



(a) (b)

Figure 4: (a) A data word and its sessions, (b) Session automaton recognizing all 2-bounded data words

3.2 Deterministic Session Automata

Session automata come with two natural notions of determinism. We call $\mathcal{A} = (S, \mathcal{R}, \iota, F, \Delta)$ *symbolically deterministic* if $|\{s' \in S \mid (s, (a, \pi), s') \in \Delta\}| \leq 1$ for all $s \in S$, $a \in \Sigma$, and $\pi \in \mathcal{R}^{\circledast} \cup \mathcal{R}^{\uparrow}$. Then, Δ can be seen as a partial function $S \times (\Sigma \times (\mathcal{R}^{\circledast} \cup \mathcal{R}^{\uparrow})) \rightharpoonup S$.

We call $\mathcal{A}$ *data deterministic* if it is symbolically deterministic and, for all $s \in S$, $a \in \Sigma$, and $r_1, r_2 \in \mathcal{R}$ with $r_1 \neq r_2$, we have that $(s, (a, r_1^{\circledast})) \in dom(\Delta)$ implies $(s, (a, r_2^{\circledast})) \notin dom(\Delta)$. Intuitively, given a data word as input, the automaton is data deterministic if, in each state, given a pair letter/data value, there is at most one fireable transition.

While "data deterministic" implies "symbolically deterministic" by definition, the converse is not true. E.g., the session automaton $\mathcal{A}_2$ from Figure 1(b) and the one of Figure 4(b) are symbolically deterministic but not data deterministic. However, the session automaton obtained from $\mathcal{A}_2$ by removing, e.g., the transition from s_0 to s_2 (coupled with the transition from s_0 to s_1, it causes non-determinism when reading a fresh data value at a request), is data deterministic (and is indeed equivalent to $\mathcal{A}_2$, in the sense that it recognizes the same language $L(\mathcal{A}_2)$).

Example 10. We explain how to construct a symbolically deterministic session automaton $\mathcal{A}$, with $k \geq 1$ registers, such that $L_{symb}(\mathcal{A}) = \mathsf{SNF}_k$. Its state space is $S = \{0, \ldots, k\} \times 2^{\{1,\ldots,k\}}$, consisting of (i) the greatest register already initialized (indeed we will only use a register r if every register $r' < r$ has already been used), (ii) a subset P of registers that we promise to reuse again before resetting their value. The initial state of $\mathcal{A}$ is $(0, \emptyset)$, whereas the set of accepting states is $(\{0, \ldots, k\}) \times \{\emptyset\}$. We now describe the set of transitions. For every $a \in \Sigma$, $i \in \{0, \ldots, k\}$, $P \subseteq \{1, \ldots, k\}$, and $r \in \{1, \ldots, k\}$:

$$\Delta\big((i, P), (a, r^{\uparrow})\big) = \begin{cases} (i, P \setminus \{r\}) & \text{if } r \leq i \\ \text{not defined} & \text{otherwise} \end{cases}$$

$$\Delta\big((i, P), (a, r^{\circledast})\big) = \begin{cases} (\max(i, r), P \cup \{1, \ldots, r-1\}) & \text{if } r - 1 \leq i \wedge r \notin P \\ \text{not defined} & \text{otherwise} \end{cases}$$

Figure 5 depicts the session automaton for SNF_2 (omitting Σ).

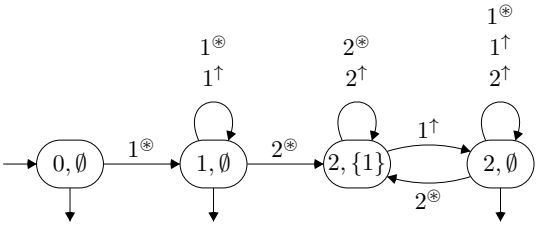


Figure 5: A session automaton recognizing SNF_2

As we shall see in the next session, every language recognized by a session automaton is also recognized by a symbolically deterministic session automata. The next theorem shows that this is not true for data deterministic session automata.

Theorem 11. *Session automata are strictly more expressive than data deterministic session automata.*

Proof. We show that the data language $L = \gamma(\mathsf{SNF}_2)$ cannot be recognized by a data deterministic session automaton. Indeed, suppose that such an automaton exists, with k registers. Then, consider the word $w = (a,1)(a,2)(a,3)\cdots(a,k+1) \in L$, where every data value is fresh. By data determinism, there is a unique run accepting w. Along this run, let $i < j$ be two positions such that their two fresh data values have been stored in the same register r (such a pair must exist since the automaton has only k registers). Without loss of generality, we can consider the greatest position j verifying this condition, and then the greatest position i associated with j. This means that register r is used for the last time when reading j, and has not been used in-between positions i and j. Now, the word $(a,1)(a,2)(a,3)\cdots(a,k+1)(a,i) \in L$ must be recognized by the automaton, but cannot since data value i appearing on the last position is not fresh anymore, and yet not stored in one of the registers (since register r was reused at j).

3.3 Canonical Session Automata

We now present the main result of this section showing that every session automaton $\mathcal{A}$ is equivalent to a *canonical* session automaton $\mathcal{A}^C$, whose symbolic language $L_{symb}(\mathcal{A}^C)$ contains only symbolic normal forms.

Theorem 12. *Let $\mathcal{A} = (S, \mathcal{R}, \iota, F, \Delta)$ be a session automaton with $\mathcal{R} = \{1, \ldots, k\}$. Then, $L(\mathcal{A})$ is k-bounded. Moreover, $snf(L(\mathcal{A}))$ is a regular language over the finite alphabet $\Sigma \times \Gamma_k$. A corresponding automaton $\tilde{\mathcal{A}}$ can be effectively computed. Its number of states is at most exponential in k and linear in $|Q|$.*

Proof. First, if $\mathcal{A}$ is a session automaton using k registers, the language $L(\mathcal{A})$ is k-bounded since $L_{symb}(\mathcal{A}) \subseteq (\Sigma \times \Gamma_k)^*$, which implies that $L(\mathcal{A}) = \gamma(L_{symb}(\mathcal{A})) \subseteq \gamma((\Sigma \times \Gamma_k)^*) = \mathsf{DW}_k$.

Example 10, constructing a symbolically deterministic session automaton for $\mathsf{SNF}_k = \mathit{snf}(\gamma((\Sigma \times \Gamma_k)^*))$, shows that regularity of the symbolic language $(\Sigma \times \Gamma_k)^*$ is preserved under the application of $\mathit{snf}(\gamma(\cdot))$. We now prove that this is the case for every regular language over $\Sigma \times \Gamma_k$. In particular, for the symbolic *regular* language $L_{\mathit{symb}}(\mathcal{A})$, this will show that $\mathit{snf}(L(\mathcal{A}))$, which is equal to $\mathit{snf}(\gamma(L_{\mathit{symb}}(\mathcal{A})))$, is regular.

Let $L \subseteq (\Sigma \times \Gamma_k)^*$ be regular. Consider first the language

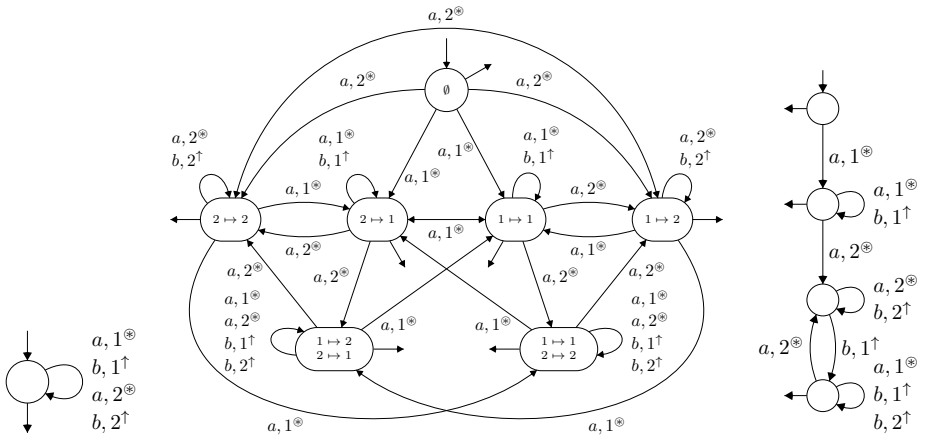
$$\tilde{L} = \{u \in \mathsf{WF} \cap (\Sigma \times \Gamma_k)^* \mid \text{there is } u' \in L \text{ such that } \gamma(u) = \gamma(u')\}$$

i.e., the set of well-formed symbolic words having the same concretizations as some word from L. We show that $\mathit{snf}(\gamma(L)) = \mathsf{SNF}_k \cap \tilde{L}$. Indeed, if $u \in \mathit{snf}(\gamma(L))$, then there are $u' \in L$ and $w \in \gamma(u')$ such that $u = \mathit{snf}(w)$. Since $u' \in (\Sigma \times \Gamma_k)^*$, we have $u \in \mathit{snf}(\gamma((\Sigma \times \Gamma_k)^*)) = \mathsf{SNF}_k$. Moreover, we have $[w]_{\approx} = \gamma(u')$ and $w \in \gamma(\mathit{snf}(w)) = \gamma(u)$ implying also $[w]_{\approx} = \gamma(u)$. Finally, we obtain $\gamma(u) = \gamma(u')$, so that $u \in \tilde{L}$. Reciprocally, if $u \in \mathsf{SNF}_k \cap \tilde{L}$, then there is $u' \in L$ such that $\gamma(u) = \gamma(u')$. Hence, starting from any word w in $\gamma(u)$ (which is non empty since u is well-formed), we have $u = \mathit{snf}(w)$ (by uniqueness of the symbolic normal form) and $w \in \gamma(u') \subseteq \gamma(L)$, so that $u \in \mathit{snf}(\gamma(L))$.

We know from Example 10 that SNF_k is regular. We now show that $\tilde{L}$ is regular: knowing that $\mathit{snf}(\gamma(L)) = \mathsf{SNF}_k \cap \tilde{L}$, this will permit to conclude that $\mathit{snf}(\gamma(L))$ is regular. To do so, let $\mathcal{A} = (S, \mathcal{R}, \iota, F, \Delta)$ be a session automaton with $\mathcal{R} = \{1, \ldots, k\}$ such that $L_{\mathit{symb}}(\mathcal{A}) = L$. We construct a session automaton $\tilde{\mathcal{A}} = (S \times \mathrm{Inj}(k), \mathcal{R}, (s_0, \emptyset), F \times \mathrm{Inj}(k), \tilde{\Delta})$ recognizing the symbolic language $\tilde{L}$. Hereby, $\mathrm{Inj}(k)$ is the set of partial injective mappings from $\{1, \ldots, k\}$ to $\{1, \ldots, k\}$, and $\emptyset \in \mathrm{Inj}(k)$ denotes the mapping with empty domain. These partial mappings are used to remember the correspondence between old registers and new ones, so they may be understood as a set of constraints. For example, the mapping $(2 \mapsto 1, 1 \mapsto 3)$ stands for "old register 2 henceforth refers to 1, and old register 1 henceforth refers to 3". Any subset of these constraints forms always a *valid* partial injective mapping. In the following, such a subset is called a sub-mapping. For example, $\sigma = (1 \mapsto 3)$ is a sub-mapping of the previous one; it can then be extended with the new constraint $2 \mapsto 2$, which we denote $\sigma[2 \mapsto 2]$. We describe now the transition relation of $\tilde{\mathcal{A}}$:

$$\begin{aligned}\tilde{\Delta} = \quad & \big\{\big((s_1, \sigma), (a, \sigma(r)^{\uparrow}), (s_2, \sigma)\big) \mid \big(s_1, (a, r^{\uparrow}), s_2\big) \in \Delta\big\} \\ \cup \; & \big\{\big((s_1, \sigma_1), (a, r_2^{\circledast}), (s_2, \sigma_2)\big) \mid \big(s_1, (a, r_1^{\circledast}), s_2\big) \in \Delta \wedge \sigma_2 = \sigma[r_1 \mapsto r_2] \\ & \qquad \text{with } \sigma \text{ maximal sub-mapping of } \sigma_1 \text{ s.t. } \sigma[r_1 \mapsto r_2] \text{ injective}\big\}\end{aligned}$$

We simulate $r^{\uparrow}$-transitions simply using the current mapping σ. For $r^{\circledast}$-transitions, we update σ, recording the new permutation of the registers: the maximal sub-mapping σ of σ_1 is either σ_1 itself or σ_1 where exactly one constraint $r_1 \mapsto r_3$ is removed to free r_1. One can indeed show that $L_{\mathit{symb}}(\tilde{\mathcal{A}}) = \tilde{L}$. Inclusion $L_{\mathit{symb}}(\tilde{\mathcal{A}}) \subseteq \tilde{L}$ is easy to show since an accepting run in $\tilde{\mathcal{A}}$ can be mapped to an accepting run in $\mathcal{A}$ using the partial injective mappings maintained in



(a) $\mathcal{A}$ (b) $\tilde{\mathcal{A}}$ (c) $\mathcal{A}^C$

Figure 6: (a) A session automaton $\mathcal{A}$, (b) its automaton $\tilde{\mathcal{A}}$, (c) its canonical automaton $\mathcal{A}^C$

the states of $\tilde{\mathcal{A}}$. For the other inclusion, it suffices to prove that for every symbolic word $u \in L$ and well-formed word u' such that $\gamma(u') = \gamma(u)$, we have $u' \in L_{symb}(\tilde{\mathcal{A}})$. By definition of γ, we know that projections of u and u' over the finite alphabet Σ are the same, and that $\sim_u = \sim_{u'}$: the latter permits to reconstruct by induction a unique sequence of partial injective mappings linking the registers used in u and in u'. An accepting run of $\mathcal{A}$ over u can therefore be mapped to an accepting run of $\tilde{\mathcal{A}}$ over u'.

Building the product of the automaton recognizing SNF_k and the automaton $\tilde{\mathcal{A}}$, we obtain a session automaton using k registers recognizing $snf(\gamma(L))$. Its number of states is bounded above by $O(|Q| \times k! \times (k+1) \times 2^k)$ (as the number of partial injective mappings in $\mathrm{Inj}(k)$ is bounded above by $O(k!)$).

From the automaton $\tilde{\mathcal{A}}$ built in the proof of the previous theorem, we can consider the (unique up to isomorphism) minimal deterministic finite-state automaton $\mathcal{A}^C$ (i.e., symbolically deterministic session automaton) equivalent to it: this automaton will be called the *canonical session automaton*. In case $\mathcal{A}$ is data deterministic, we can verify that $\tilde{\mathcal{A}}$ is symbolically deterministic, and hence the minimal automaton $\mathcal{A}^C$ has at most $O(|Q| \times k! \times (k+1) \times 2^k)$ states. Otherwise, a determinization phase has to be done resulting in a canonical session automaton with at most $2^{O(|Q| \times k! \times (k+1) \times 2^k)}$ states.

Example 13. Examples of $\mathcal{A}$ and $\tilde{\mathcal{A}}$, as defined in the previous proof, are given in Figure 6. The figure also depicts the canonical automaton $\mathcal{A}^C$ associated with $\mathcal{A}$, obtained by determinizing and minimizing the product of both $\tilde{\mathcal{A}}$ and the symbolically deterministic automaton recognizing SNF_2 (as given in Figure 5). Note that $\mathcal{A}^C$ is symbolically deterministic and minimal.

3.4 Closure Properties

Using Theorem 12, we obtain some language theoretical closure 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 [11] and infinite-state regular model checking [15].

Theorem 14. *We have the following closure properties:*

- *Session automata are closed under union and intersection.*
- *Session automata are closed under resource-sensitive complementation: Given a session automaton $\mathcal{A}$ with k registers, there is a session automaton $\mathcal{B}$ with k registers such that $L(\mathcal{B}) = \mathsf{DW}_k \setminus L(\mathcal{A})$.*

Proof. Let $\mathcal{A}$ be a session automaton using k registers, and $\mathcal{B}$ a session automaton using k' registers. Using a classical product construction for $\mathcal{A}^C$ and $\mathcal{B}^C$, we obtain a session automaton using $\min(k, k')$ registers recognizing the data language $L(\mathcal{A}) \cap L(\mathcal{B})$. The language $L(\mathcal{A}) \cup L(\mathcal{B})$ is recognized by the session automaton, using $\max(k, k')$ registers, that we obtain as the "disjoint union" of $\mathcal{A}$ and $\mathcal{B}$, branching on the first transition in one of these two automata.

Finally, let us consider a symbolically deterministic session automaton $\mathcal{A}$ using k registers. Without loss of generality, by adding a sink state, we can suppose that $\mathcal{A}$ is complete. Then, every well-formed symbolic word over $\Sigma \times \Gamma_k$ has exactly one run in $\mathcal{A}$. The automaton $\mathcal{A}'$ constructed from $\mathcal{A}$ by taking as accepting states the non-accepting states of $\mathcal{A}$ verifies that $L_{symb}(\mathcal{A}') = (\Sigma \times \Gamma_k)^* \setminus L_{symb}(\mathcal{A})$ so that $L(\mathcal{A}') = \gamma((\Sigma \times \Gamma_k)^*) \setminus L(\mathcal{A})$.

Theorem 15. *The inclusion problem for session automata is decidable.*

Proof. Considering two session automata $\mathcal{A}$ and $\mathcal{B}$, we can decide inclusion $L(\mathcal{A}) \subseteq L(\mathcal{B})$ by considering the canonical automata $\mathcal{A}^C$ and $\mathcal{B}^C$. Indeed, $L(\mathcal{A}) \subseteq L(\mathcal{B}) \iff \mathit{snf}(L(\mathcal{A})) \subseteq \mathit{snf}(L(\mathcal{B})) \iff L_{symb}(\mathcal{A}^C) \subseteq L_{symb}(\mathcal{B}^C)$. Thus, it is sufficient to check inclusion for $\mathcal{A}^C$ and $\mathcal{B}^C$.

While fresh-register automata are not complementable for the set of all data words, they are complementable for k-bounded data words, using the previous theorem. The reason is that, given a fresh-register automaton $\mathcal{A}$, one can construct a session automaton $\mathcal{B}$ such that $L(\mathcal{B}) = L(\mathcal{A}) \cap \mathsf{DW}_k$.

4 Logical Characterizations

In this section, we provide logical characterizations of session automata.

4.1 MSO Logic over Data Words

We consider *data monadic second-order logic* (dMSO), which is an extension of classical MSO logic by the binary predicate $x \sim y$ to compare data values.

We fix infinite supplies of first-order variables $x, y, \ldots$, which are interpreted as word positions, and second-order variables $X, Y, \ldots$, which are taken as sets of positions. Atomic dMSO formulae are $x = y$, $\mathrm{label}(x) = a$ (with $a \in \Sigma$), $y = x + 1$, $x \in X$, and $x \sim y$. They allow us to reason about word positions in the expected manner: given a data word $w = (a_1, d_1) \cdots (a_n, d_n) \in (\Sigma \times D)^*$, positions $i, j \in \{1, \ldots, n\}$, and a set $I \subseteq \{1, \ldots, n\}$, we have $w, i, j \models x = y$ (to be read as "w satisfies $x = y$ when x is interpreted as i and y as j") if $i = j$; $w, i \models \mathrm{label}(x) = a$ if $a_i = a$; $w, i, j \models y = x + 1$ if $j = i + 1$; and $w, i, I \models x \in X$ if $i \in I$. The atomic formula $x \sim y$ allows us to reason about data values at x and y: $w, i, j \models x \sim y$ if $d_i = d_j$. Finally, dMSO provides negation ($\neg$) and disjunction ($\vee$), as well as existential quantifiers $\exists x\, \varphi$ and $\exists X\, \varphi$ (with φ a dMSO formula), interpreted as usual. In addition, we use abbreviations such as *true*, $x \leq y$, $\forall x\, \varphi$, $\varphi \wedge \psi$, $\varphi \rightarrow \psi$, etc. A sentence is a formula without free variables. For a dMSO sentence φ, we set $L(\varphi) \stackrel{\text{def}}{=} \{w \in (\Sigma \times D)^* \mid w \models \varphi\}$.

Note that dMSO is a very expressive logic and goes beyond virtually all automata models defined for data words [25, 27, 6, 12]. However, if we restrict to bounded languages, we can show that dMSO is no more expressive than session automata.

Theorem 16. *Let L be a bounded set of data words. Then, the following statements are equivalent:*

- *There is a session automaton $\mathcal{A}$ such that $L(\mathcal{A}) = L$.*
- *There is a dMSO sentence φ such that $L(\varphi) = L$.*

Proof. The translation from automata to logic follows the standard construction and is omitted (see also proof of Theorem 20).

For the converse direction, we perform, as usual, an induction on the structure of the formula. To deal with negation, we can indeed exploit the fact that session automata are closed under complementation when considering only k-bounded data words (Theorem 14).

Let φ be a dMSO formula such that $L(\varphi)$ is k-bounded, for some $k \geq 1$. We suppose that φ is given in prenex normal form. To deal with free variables, we extend the alphabet Σ as follows. If V is the set of variables that occur in φ, we have to consider data words over $\hat{\Sigma} = \Sigma \times \{0, 1\}^V$ and D. Intuitively, these data words include the interpretation of the free variables. If a data word carries, at position i, the letter $(a, \bar{b}, d) \in \hat{\Sigma} \times D$ with $\bar{b}[x] = 1$ (where $\bar{b}[x]$ refers to the x-component of $\bar{b}$), then x is interpreted as position i. If $\bar{b}[X] = 1$, then X is interpreted as a set *containing* i. Whenever we refer to a word over the extended alphabet $\hat{\Sigma}$, we will silently assume that the interpretation of a first-order variable x is uniquely determined, i.e., there is exactly one position i where $\bar{b}[x] = 1$. This is justified, since the set of those "well-shaped" words is (symbolically) regular.

For the base case, consider the formula $\mathrm{label}(x) = a$. We construct a session automaton $\mathcal{A}$ with k registers such that $L_{symb}(\mathcal{A})$ consists of all "well-shaped" words $u \in (\hat{\Sigma} \times \Gamma_k)^*$ containing a letter $(a, \bar{b}, \pi)$ with $\bar{b}[x] = 1$. For $x = y$, the automaton has to verify that there is a letter $(a, \bar{b}, \pi)$ such that $\bar{b}[x] =$

$\bar{b}[y] = 1$. Formulae $y = x + 1$ and $x \in X$ are similar. The most interesting base case is $x \sim y$. We can construct a corresponding session automaton $\mathcal{A}$ with k registers such that $L_{symb}(\mathcal{A})$ contains exactly the symbolic words $u = (a_1, \bar{b}_1, \pi_1) \cdots (a_n, \bar{b}_n, \pi_n) \in (\hat{\Sigma} \times \Gamma_k)^*$ satisfying the following: there are two positions $i, j \in \{1, \ldots, n\}$ such that $i \sim_u j$, $\bar{b}_i[x] = 1$, and $\bar{b}_j[y] = 1$. Note that $L_{symb}(\mathcal{A})$ is indeed a regular language.

In all of the above base cases, if ψ is the atomic formula and $\mathcal{A}$ the corresponding session automaton, the following holds: given a data word $w \in \mathsf{DW}_k$, we have $w \in L(\psi)$ iff $w \in L(\mathcal{A})$.

Let us come to the induction step. For negation, suppose we have constructed a session automaton $\mathcal{A}$ with k registers such that $L(\mathcal{A}) = L(\psi) \cap \mathsf{DW}_k$. According to Theorem 14, there is a session automaton $\mathcal{B}$ with k registers such that $L(\mathcal{B}) = \mathsf{DW}_k \setminus L(\mathcal{A})$. We deduce $L(\mathcal{B}) = L(\neg\psi) \cap \mathsf{DW}_k$. To deal with disjunction, we exploit closure of session automata under union (again, Theorem 14). Finally, existential quantification corresponds, as usual, to projection.

By Theorems 15 and 16, we obtain, as a corollary, that model checking session automata against dMSO specifications is decidable (while it is undecidable for register automata). Note that this was already shown in [8] for a more powerful model with pushdown stacks.

Theorem 17. *Given a session automaton $\mathcal{A}$ and a dMSO sentence φ, one can decide whether $L(\mathcal{A}) \subseteq L(\varphi)$.*

4.2 Session MSO Logic

Next, we give an alternative logical characterization of session automata. We identify a fragment of dMSO, called *session MSO logic*, such that every formula from that fragment can be translated into a session automaton, without having to restrict the set of data words in advance. Note that register automata also enjoy a logical characterization [12]. There, *guards* are employed to tame the power of the predicate $\sim$. Similarly, our logic also uses a guard, though in a quite different way.

Definition 18. *A* session MSO *(sMSO) formula is a dMSO sentence of the form $\exists X_1 \cdots \exists X_m \, (\alpha \wedge \forall x \forall y \, (x \sim y \leftrightarrow \beta))$ such that α and β are classical MSO formulae (not containing the predicate $\sim$).*

Example 19. The formula $\varphi_1 = \forall x \forall y \, (x \sim y \leftrightarrow x = y)$ is an sMSO formula. Its semantics $L(\varphi_1)$ 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 $L(\varphi_2)$ 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, $L(\varphi_3)$ 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 20. *For any set L of data words, the following statements are equivalent:*

– *There is a session automaton $\mathcal{A}$ such that $L(\mathcal{A}) = L$.*
– *There is an sMSO sentence φ such that $L(\varphi) = L$.*

Proof. The construction of an sMSO formula $\exists X_1 \cdots \exists X_m\ (\alpha \wedge \forall x \forall y\ (x \sim y \leftrightarrow \beta))$ from a session automaton $\mathcal{A}$ was implicitly shown in [8] (with a different goal, though). The idea is that the existential second-order variables $X_1, \ldots, X_m$ are used to guess an assignment of transitions to positions. In α, it is verified that the assignment corresponds to a run of $\mathcal{A}$. Moreover, β checks if data equality corresponds to the data flow as enforced by the transition labels from Γ_k.

We turn to the converse direction and let $\varphi = \exists X_1 \cdots \exists X_m\ (\alpha \wedge \forall x \forall y\ (x \sim y \leftrightarrow \beta))$ be an sMSO sentence. By Theorem 16, it is sufficient to show that $L(\varphi)$ is bounded.

The free variables of β are among $x, y, X_1, \ldots, X_m$. As, moreover, β is a "classical" MSO formula, which does not contain $\sim$, it defines, in the expected manner, a set $L_{symb}(\beta)$ of words over the finite alphabet $\Sigma \times \{0,1\}^{m+2}$. Similarly to the proof of Theorem 16, the idea is to interpret a position carrying letter $(a, 1, b, b_1, \ldots, b_m)$ as x, and a position labeled $(a, b, 1, b_1, \ldots, b_m)$ as y, while membership in X_i is indicated by b_i. Words where x and y are not uniquely determined, are discarded. We can represent such models as tuples $(w, i, j, I_1, \ldots, I_m)$ where $w \in \Sigma^*$, i denotes the position of the 1-entry in the unique first component, and j denotes the position of the 1-entry in the second component. As $L_{symb}(\beta) \subseteq (\Sigma \times \{0,1\}^{m+2})^*$ is MSO definable (in the classical sense, without data), it is, by Büchi's theorem, recognized by some minimal deterministic finite automaton $\mathcal{A}_\beta$. Suppose that $\mathcal{A}_\beta$ has $k_\beta \geq 1$ states.

We claim that the data language $L(\varphi)$ is k_β-bounded. To show this, let $w = (a_1, d_1) \cdots (a_n, d_n) \in L(\varphi)$. There exists a tuple $\overline{I} = (I_1, \ldots, I_m)$ of subsets of $\{1, \ldots, n\}$ such that, for all $i, j \in \{1, \ldots, n\}$,

$$d_i = d_j \iff (a_1 \cdots a_n, i, j, \overline{I}) \in L_{symb}(\beta)\,. \qquad (*)$$

Suppose, towards a contradiction, that w is not k_β-bounded. Then, there are $k > k_\beta$ and a position $i \in \{1, \ldots, n\}$ such that i is contained in k distinct sessions $J_1, \ldots, J_k$. For $l \in \{1, \ldots, k\}$, let $i_l = \min(J_l)$ and $j_l = \max(J_l)$, so that $J_l = \{i_l, i_l + 1, \ldots, j_l\}$. Note that the i_l are pairwise distinct, and so are the j_l. By $(*)$, for every $l \in \{1, \ldots, k\}$, we have $w_l = (a_1 \cdots a_n, i_l, j_l, \overline{I}) \in L_{symb}(\beta)$. Thus, for every such word w_l, there is a unique accepting run of $\mathcal{A}_\beta$, say, being in state q_l after executing position i. As $\mathcal{A}_\beta$ has only k_β states, there are $l \neq l'$ such that $q_l = q_{l'}$. Thus, there is an accepting run of $\mathcal{A}_\beta$ either on a word where one of the first-order components is not unique, which is a contradiction, or on $(a_1 \cdots a_n, i_l, j_{l'}, \overline{I})$. The latter contradicts $(*)$, since J_l and $J_{l'}$ are distinct sessions.

5 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 $L(\mathcal{A})$ defined by a given session automaton $\mathcal{A}$. We suppose here that the teacher knows the session automaton or any other device accepting $L(\mathcal{A})$. 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 $L(\mathcal{A})$ accepted by a session automaton $\mathcal{A}$ over Σ and D, our algorithm will learn the canonical session automaton $\mathcal{A}^C$, that uses k registers, i.e., the minimal symbolically deterministic automaton recognizing the language $L(\mathcal{A})$ 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[6]. 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 session automaton with 1 register 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 algorithm [26] 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

[6] 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.

Algorithm 1: The learning algorithm for a session automaton $\mathcal{A}$

initialize $k := 1$ and ;
$\mathcal{O} := (T, U, V)$ by $U = V = \{\varepsilon\}$ and $T(u, \varepsilon)$ for all $u \in U \cup U \cdot (\Sigma \times \Gamma_k)$ with membership queries;
repeat
  while $\mathcal{O}$ *is not closed* **do**
    find $u \in U$ and $(a, \pi) \in \Sigma \times \Gamma_k$ such that for all $u' \in U : \ row(u(a, \pi)) \neq row(u')$;
    extend table to $\mathcal{O} := (T', U \cup \{u(a, \pi)\}, V)$ by membership queries;
  end
  from $\mathcal{O}$ construct the hypothesized automaton $\mathcal{A}_\mathcal{O}$; `// cf. Definition 21`
  if $\mathcal{A}_\mathcal{O}$ *accepts symbolic words not in normal form* **then**
    let z be one of those;
  else
    if $L(\mathcal{A}) = L(\mathcal{A}_\mathcal{O})$ **then**
      equivalence test succeeds;
    else
      get counterexample $w \in (L(\mathcal{A}) \setminus L(\mathcal{A}_\mathcal{O})) \cup (L(\mathcal{A}_\mathcal{O}) \setminus L(\mathcal{A}))$;
      set $z := snf(w)$;
      find minimal k' such that $z \in (\Sigma \times \Gamma_{k'})^*$;
      if $k' > k$ **then**
        set $k := k'$;
        extend table to $\mathcal{O} := (T', U, V)$ over $\Sigma \times \Gamma_k$ by membership queries;
      end
    end
  end
  if $\mathcal{O}$ *is closed* **then** `// is true if` $k' \leq k$
    find a breakpoint for z where v is the distinguishing word;
    extend table to $\mathcal{O} := (T', U, V \cup \{v\})$ by membership queries;
  end
until *equivalence test succeeds*;
return $\mathcal{A}_\mathcal{O}$

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 21. *For a closed table $\mathcal{O} = (T, U, V)$ over a finite alphabet $\Sigma \times \Gamma_k$, we define a symbolically deterministic session automaton $A_\mathcal{O} = (S, \mathcal{R}, \iota, F, \Delta)$ over $\Sigma \times \Gamma_k$ by $S = U$, $\mathcal{R} = \{1, \ldots, k\}$, $\iota = \varepsilon$, $F = \{u \in S \mid T(u, \varepsilon) = +\}$, and for all $u \in S$ and $(a, \pi) \in \Sigma \times \Gamma_k$, $\Delta(u, (a, \pi)) = u'$ if $row(u(a, \pi)) = row(u')$. 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 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 [26] 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 $s_i \in U$ 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 $row(s_i z_i)$ become 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 22. *Let $\mathcal{A}$ be a session automaton over Σ and D, using k' registers. Let $\mathcal{A}^C$ be the corresponding canonical session automaton. Let N be its number of states, k its number of registers and M the length of the longest counterexample returned by an equivalence query. Then, the learning algorithm for $\mathcal{A}$ terminates with at most $O(k|\Sigma|N^2 + N\log(M))$ membership and $O(N)$ equivalence queries.*

Proof. This follows directly from the proof of correctness and complexity of Rivest and Schapire's algorithm [4, 26]. 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 the session automaton $\mathcal{A}$ is data deterministic, using the discussion after the proof of Theorem 12 over the size of $\mathcal{A}^C$, the overall complexity of the learning algorithm is polynomial in the number of states of $\mathcal{A}$, but exponential in the number of registers it uses (with constant base). As usual, we have to add one exponent when we consider session automata which are not data deterministic. In [16], 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 [16] restrict to deterministic automata, since classical register automata are not determinizable.

$\mathcal{O}_1$	ε
ε	+
$(b,1^\uparrow)$	−
$(a,1^\circledast)$	+
$(b,1^\circledast)\_$	−

⇒

$\mathcal{O}_2$	ε	$(b,1^\uparrow)$
ε	+	−
$(b,1^\uparrow)$	−	−
$(a,1^\circledast)$	+	+
$(b,1^\uparrow)\_$	−	−
$(a,1^\circledast)(a,1^\circledast)$	+	+
$(a,1^\circledast)(b,1^\uparrow)$	+	+

⇒

$\mathcal{O}_3$	ε	$(b,1^\uparrow)$
ε	+	−
$(b,1^\uparrow)$	−	−
$(a,1^\circledast)$	+	+
$(a,2^\circledast)$	−	−
$(b,2^\uparrow)$	−	−
$(b,1^\uparrow)\_$	−	−
$(a,1^\circledast)(a,1^\circledast)$	+	+
$(a,1^\circledast)(b,1^\uparrow)$	+	+
$(a,1^\circledast)(a,2^\circledast)$	−	+
$(a,1^\circledast)(b,2^\uparrow)$	−	−

⇒

$\mathcal{O}_4$	ε	$(b,1^\uparrow)$
ε	+	−
$(b,1^\uparrow)$	−	−
$(a,1^\circledast)$	+	+
$(a,1^\circledast)(a,2^\circledast)$	−	+
$(a,2^\circledast)$	−	−
$(b,2^\uparrow)$	−	−
$(b,1^\uparrow)\_$	−	−
$(a,1^\circledast)(a,1^\circledast)$	+	+
$(a,1^\circledast)(b,1^\uparrow)$	+	+
$(a,1^\circledast)(b,2^\uparrow)$	−	−
$(a,1^\circledast)(a,2^\circledast)(a,1^\circledast)$	−	−
$(a,1^\circledast)(a,2^\circledast)(b,1^\uparrow)$	+	+
$(a,1^\circledast)(a,2^\circledast)(a,2^\circledast)$	−	+
$(a,1^\circledast)(a,2^\circledast)(b,2^\uparrow)$	−	+

⇒

$\mathcal{O}_5$	ε	$(b,1^\uparrow)$	$(b,2^\uparrow)$
ε	+	−	−
$(b,1^\uparrow)$	−	−	−
$(a,1^\circledast)$	+	+	−
$(a,1^\circledast)(a,2^\circledast)$	−	+	−
$(a,1^\circledast)(a,2^\circledast)(b,1^\uparrow)$	+	+	+
$(a,2^\circledast)$	−	−	−
$(b,2^\uparrow)$	−	−	−
$(b,1^\uparrow)\_$	−	−	−
$(a,1^\circledast)(a,1^\circledast)$	+	+	−
$(a,1^\circledast)(b,1^\uparrow)$	+	+	−
$(a,1^\circledast)(b,2^\uparrow)$	−	−	−
$(a,1^\circledast)(a,2^\circledast)(a,1^\circledast)$	−	−	−
$(a,1^\circledast)(a,2^\circledast)(a,2^\circledast)$	−	+	−
$(a,1^\circledast)(a,2^\circledast)(b,2^\uparrow)$	−	+	−
$(a,1^\circledast)(a,2^\circledast)(b,1^\uparrow)(a,1^\circledast)$	+	+	+
$(a,1^\circledast)(a,2^\circledast)(b,1^\uparrow)(b,1^\uparrow)$	+	+	+
$(a,1^\circledast)(a,2^\circledast)(b,1^\uparrow)(a,2^\circledast)$	−	+	−
$(a,1^\circledast)(a,2^\circledast)(b,1^\uparrow)(b,2^\uparrow)$	+	+	+

Figure 7: The successive observation tables

Example 23. We apply our learning algorithm on the data language given by the automaton $\mathcal{A}$ of Figure 6(a). In Figure 7 the successive observation tables constructed by the algorithm are given. To save space some letters whose rows contain only −'s are omitted. In Figure 8 the successive automata constructed from the closed observation tables are given. For sake of clarity we omit the sink states. We start with the alphabet $\Sigma \times \Gamma_1 = \{(a,1^\circledast),(a,1^\uparrow),(b,1^\circledast),(b,1^\uparrow)\}$. We omit

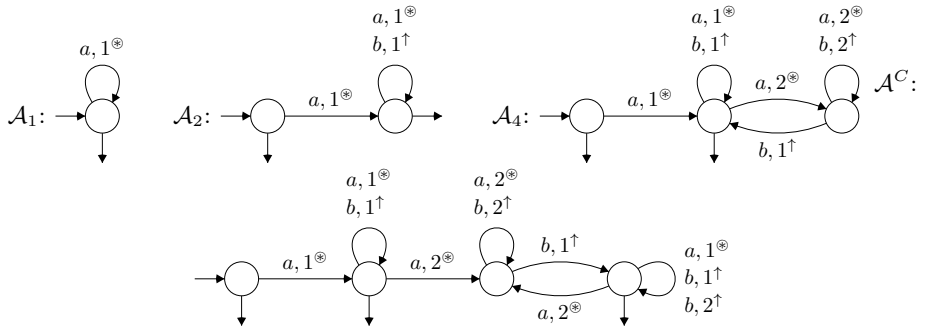


Figure 8: The successive hypothesis automata

letters $(a, 1^{\uparrow})$ and $(b, 1^{\circledast})$. Table $\mathcal{O}_1$ is obtained after initialization and closing by adding $(b, 1^{\uparrow})$ to the top. We use $\_$ to indicate that all letters will lead to the same row. From $\mathcal{O}_1$ the first hypothesis automaton $\mathcal{A}_1$ is constructed. We suppose that the equivalence query gives back as counterexample the data word $(a, 3)(b, 3)$ whose normal form is $(a, 1^{\circledast})(b, 1^{\uparrow})$. Here the breakpoint yields the distinguishing word $(b, 1^{\uparrow})$. We add it to V. The obtained table is not closed anymore. We close it by adding $(a, 1^{\circledast})$ to the top and get table $\mathcal{O}_2$ yielding hypothesis automaton $\mathcal{A}_2$. Notice that $L_{symb}(\mathcal{A}_2) = L_{symb}(\mathcal{A}^C) \cap (\Sigma \times \Gamma_1)^*$. This means that the equivalence query must give back a data word whose normal form is using at least 2 registers (here $(a, 7)(a, 4)(b, 7)$ with normal form $(a, 1^{\circledast})(a, 2^{\circledast})(b, 1^{\uparrow})$). As the word uses 2 registers, we extend the alphabet to $\Sigma \times \Gamma_2$ and obtain table $\mathcal{O}_3$. We close the table and get $\mathcal{O}_4$. From there we obtain the hypothesis automaton $\mathcal{A}_4$. After the equivalence query we get $(a, 1^{\circledast})(a, 2^{\circledast})(b, 1^{\uparrow})(b, 2^{\uparrow})$ as normal form of the data word counterexample $(a, 9)(a, 3)(b, 9)(b, 3)$. After adding $(b, 2^{\uparrow})$ to V and closing the table by moving $(a, 1^{\circledast})(a, 2^{\circledast})(b, 1^{\uparrow})$ to the top we get finally the table $\mathcal{O}_5$ from which the canonical automaton $\mathcal{A}^C$ is obtained and the equivalence query succeeds.

6 Conclusion

In this paper, we developed a theory of session automata, which form a robust class of data languages. In particular, they are closed under union, intersection, and resource-sensitive complementation. Moreover, they enjoy logical characterizations in terms of (a fragment of) MSO logic with a predicate to compare data values for equality. Finally, unlike most other automata models for data words, session automata have a decidable inclusion problem. This makes them attractive for verification and learning. In fact, we provided a complete framework for algorithmic learning of session automata, making use of their canonical normal form. 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*, volume 7436 of *Lecture Notes in Computer Science*, pages 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*, volume 3442 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2005.
4. T. Berg and H. Raffelt. Model checking. In *Model-based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. 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. M. Bojańczyk and S. Lasota. An extension of data automata that captures XPath. In *LICS 2010*, pages 243–252. IEEE Computer Society, 2010.
8. B. Bollig, A. Cyriac, P. Gastin, and K. Narayan Kumar. Model checking languages of data words. In L. Birkedal, editor, *Proceedings of FoSSaCS'12*, volume 7213 of *Lecture Notes in Computer Science*, pages 391–405. Springer, 2012.
9. B. Bollig, P. Habermehl, M. Leucker, and B. Monmege. A fresh approach to learning register automata. In *Proceedings of the 17th International Conference on Developments in Language Theory (DLT'13)*, volume 7907 of *Lecture Notes in Computer Science*, pages 118–130. Springer, 2013.
10. B. Bollig, J.-P. Katoen, C. Kern, M. Leucker, D. Neider, and D. Piegdon. libalf: the automata learning framework. In *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 360–364. Springer, 2010.
11. J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 2003.
12. T. Colcombet, C. Ley, and G. Puppis. On the use of guards for logics with data. In *Proceedings of MFCS'11*, volume 6907 of *Lecture Notes in Computer Science*, pages 243–255. Springer Berlin / Heidelberg, 2011.
13. S. Demri and R. Lazić. LTL with the freeze quantifier and register automata. *ACM Transactions on Computational Logic*, 10(3), 2009.
14. D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *ESEC / SIGSOFT FSE*, pages 257–266. ACM, 2003.
15. P. Habermehl and T. Vojnar. Regular model checking using inference of regular languages. *Electronic Notes in Theoretical Computer Science*, 138(3):21–36, 2005.
16. F. Howar, B. Steffen, B. Jonsson, and S. Cassel. Inferring canonical register automata. In *VMCAI*, volume 7148 of *Lecture Notes in Computer Science*, pages 251–266. Springer, 2012.
17. B. Jonsson. Learning of automata models extended with data. In *SFM*, volume 6659 of *Lecture Notes in Computer Science*, pages 327–349. Springer, 2011.
18. M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.
19. M. Kaminski and D. Zeitlin. Finite-memory automata with non-deterministic reassignment. *International Journal of Foundations of Computer Science*, 21(5):741–760, 2010.

20. K. O. Kürtz, R. Küsters, and T. Wilke. Selecting theories and nonce generation for recursive protocols. In P. Ning, V. Atluri, V. D. Gligor, and H. Mantel, editors, *FMSE*, pages 61–70. ACM, 2007.
21. A. Kurz, T. Suzuki, and E. Tuosto. On nominal regular languages with binders. In L. Birkedal, editor, *Proceedings of FoSSaCS'12*, volume 7213 of *Lecture Notes in Computer Science*, pages 255–269. Springer, 2012.
22. M. Leucker. Learning meets verification. In *FMCO*, volume 4709 of *Lecture Notes in Computer Science*, pages 127–151. Springer, 2007.
23. T. Margaria, H. Raffelt, B. Steffen, and M. Leucker. The LearnLib in FMICS-jETI. In *ICECCS*, pages 340–352. IEEE Computer Society Press, 2007.
24. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, 100:1–77, Sept. 1992.
25. 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.
26. R. Rivest and R. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103:299–347, 1993.
27. L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In Z. Ésik, editor, *CSL 2006*, volume 4207 of *LNCS*, pages 41–57. Springer, 2006.
28. N. Tzevelekos. Fresh-register automata. In T. Ball and M. Sagiv, editors, *POPL*, pages 295–306. ACM, 2011.
29. N. Tzevelekos. Fresh-register automata. In Th. Ball and M. Sagiv, editors, *POPL 2011*, pages 295–306. ACM, 2011.