A Succinct Canonical Register Automaton Model[☆]

Sofia Cassel[a], Falk Howar[b], Bengt Jonsson[a], Maik Merten[b], Bernhard Steffen[b]

[a] *Dept. of Information Technology, Uppsala University, Sweden*
[b] *Chair of Programming Systems, University of Dortmund, Germany*

Abstract

We present a novel canonical automaton model, based on register automata, that can be used to specify protocol or program behavior. Register automata have a finite control structure and a finite number of registers (variables), and process sequences of terms that carry data values from an infinite domain. We consider register automata that compare data values for equality. A major contribution is the definition of a canonical automaton representation of any language recognizable by a deterministic register automaton, by means of a Nerode congruence. This canonical form is well suited for modeling, e.g., protocols or program behavior. Our model can be exponentially more succinct than previous proposals, since it filters out 'accidental' relations between data values. This opens the way to new practical applications, e.g., in automata learning.

Keywords: register automata, data languages, canonical model, Myhill-Nerode, automata theory

[☆]Supported in part by the European Commission FP7 project CONNECT (IST 231167), and by the Swedish Research Council through the Linnaeus centre of excellence UPMARC.
This is an extended version of the paper [8] with the same name, which appeared at the *9th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, Taipei, Taiwan, October 2011.

Email addresses: `sofia.cassel@it.uu.se` (Sofia Cassel), `falk.howar@cs.tu-dortmund.de` (Falk Howar), `bengt.jonsson@it.uu.se` (Bengt Jonsson), `maik.merten@cs.tu-dortmund.de` (Maik Merten), `bernhard.steffen@cs.tu-dortmund.de` (Bernhard Steffen)

1. Introduction

When modeling a system, it is crucial to be able to capture not only control aspects but also data aspects of its behavior. Often, systems are modeled as finite automata. Finite automata can be and have been extended with data, for example as timed automata [1], counter automata, data-independent transition systems [16], and different kinds of data automata and register automata. Many of these types of automata have long been used for specification, verification, and testing (e.g., [19]).

In this paper, we focus on *register automata* [3, 14, 8]. A register automaton has a finite set of registers (a.k.a. variables) and processes sequences of terms that carry data values from an infinite domain. It can compare data values, e.g., for equality, and assign data values to registers. Register automata can thus can be regarded as a simple programming language, with variables, parallel assignments, and guards, and as such they provide a natural formalism for modeling and reasoning about systems.

Canonicity is an important property for models of systems: it is exploited in equivalence and refinement checking, e.g., through (bi)simulation based criteria [15, 18], and in automata learning (a.k.a. regular inference) [2, 11, 20]. There are standard (minimization) algorithms for obtaining canonical automata, based on the Myhill-Nerode theorem [12, 17], but it has proven difficult to carry over similar constructions to automata models over infinite alphabets, including timed automata [23].

Recently, canonical automata based on extensions of the Myhill-Nerode theorem have been proposed for languages where data values can be compared for equality [10, 3, 6], and also for inequality, when the data domain is equipped with a total order [3, 6]. These canonical models are, however, obtained at the price of rather strict restrictions on how data is stored in variables, and on which guards may be used in transitions. Examples of such restrictions are uniqueness (two variables may not store identical data values) and order (a fixed ordering is enforced between variables). Both of these restrictions may cause a blow-up in the number of locations in the canonical automaton. Another cause of unwanted blow-ups is encoding relations between data values regardless of whether they are necessary in order to recognize the language.

We propose a novel form of register automata for languages where data values are compared for equality, without the above mentioned restrictions. Our formalism avoids unnecessary blow-ups in the number of locations, while still resulting in canonical models. We do not limit access to variables to a specific order or pattern, and two variables may store identical data values. In this manner, we can obtain more intuitive models of data languages, while preserving the same expressiveness.

The idea behind our approach is to filter out 'accidental' relations between data values in a word, i.e., relations that do not affect whether the word is in the modeled language or not. We do this by organizing a set of data words in a tree, merging nodes that have isomorphic subtrees (similar to BDD minimization) and finally folding the tree into a register automaton using a form of the Nerode congruence. We then obtain a model that is both canonical and succinct. If the tree is minimal, the model is also minimal in a certain class.

By a non-technical analogy, we could compare the difference between the automata of [10, 3] and our canonical form to the difference between the region graph and zone graph constructions for timed automata. The region graph considers all possible combinations between constraints on clock values, be they relevant to acceptance of the input word or not, whereas the zone graph construction aims to consider only relevant constraints. The analogy is not perfect, however, since our automata are always more succinct than those of [10, 3].

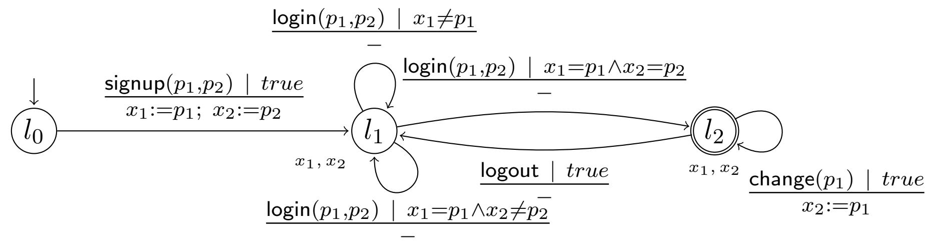
In summary, our main contribution is a Nerode congruence for a form of register automata. Our formalism can easily be used to specify protocol or program behavior since it provides a canonical representation of any (deterministic) RA-recognizable data language.

Related Work. Among the first to generalize regular languages to infinite alphabets were Kaminski and Francez [14] who introduced finite memory automata (FMA) that recognize languages with infinite input alphabets. Since then, a number of formalisms have been suggested (pebble automata, data automata, ...) that accept different flavors of data languages (see [22, 5, 4] for an overview). Most of these formalisms recognize data languages that are invariant under permutations on the data domain. In [7] a logical characterization of data languages is given plus a transformation from logical descriptions to automata.

While most of the above mentioned work focuses on non-deterministic automata and is concerned with closedness properties and expressiveness results of data languages, we are interested in a framework for (deterministic) register automata that can be used to model the behavior of protocols or (restricted) programs. This includes, in particular, the development of canonical models on the basis of a Myhill-Nerode-like theorem. Kaminski and Francez [10], Benedikt et al. [3], and Bojanczyk et al. [6] all present Myhill-Nerode theorems for data languages with equality tests, but the fact that their models tend to be large (in terms of the number of locations) and/or have restrictions on variables prompted us to develop the more succinct construction described in this paper.

Organization. In the next section, we illustrate our main ideas using a simple example. Then, we present our register automaton model as a basis for representing data languages. In Section 4, we introduce a succinct representation of data languages, allowing us to use a limited number of data words to represent an entire data language. Based on this representation, we define a Nerode congruence in Section 5 and prove that it characterizes canonical forms of deterministic register automata, called (right-invariant) DRAs. In Section 6 we relate our canonical form to other formalisms, and establish some exponential succinctness results, before we conclude in Section 7.

2. An illustrative example



login(p_1,p_2) | $x_1 \neq p_1$
–
signup(p_1,p_2) | $true$
x_1:=p_1; x_2:=p_2
login(p_1,p_2) | x_1=p_1∧x_2=p_2
–
l_0
l_1
l_2
x_1, x_2
logout | $true$
–
x_1, x_2
change(p_1) | $true$
x_2:=p_1
login(p_1,p_2) | x_1=p_1∧$x_2 \neq p_2$
–

Figure 1: Partial model for a fragment of XMPP

In this section, we provide an overview of our main ideas using a simple running example. We model the behavior of a fragment of the XMPP protocol [21], which is widely used in instant messaging. More specifically, we focus on the mechanism for signing up for an account, logging in and out, and changing the account password.

We represent our fragment of XMPP as a data language, i.e. as a set of data words; each word is an execution of the protocol. For example, the execution signup(`Bob`, `lemon`) login(`Bob`, `lemon`) represents a user first signing up, thereby setting a username (`Bob`) and password (`lemon`), and then logging in using that same username and password. In this example, we define the language $\mathcal{L}_{XMPP}$ as the set of all executions of the protocol that lead to a user being logged in. For example, the following two sequences result in the user `Bob` being logged in:

- signup(`Bob`, `lemon`) login(`Bob`, `lemon`)
- signup(`Bob`, `lemon`) login(`Bob`, `lemon`) logout login(`Bob`, `lemon`)

Let us now consider the problem of building succinct *register automata* models for languages such as $\mathcal{L}_{XMPP}$. It seems reasonable that any such automaton would need to remember the password and username supplied in the signup message in order to be able to compare them to the parameters in any subsequent login message.

A naive way of doing this is to construct an automaton that performs all possible tests on input parameters. This automaton will first test whether the username and password supplied in the signup message are equal. If they are, it will initialize one location variable; if they are not, it will initialize two location variables to store the username and password separately. Then the automaton will test whether the stored values match the new data values supplied in the login message. It will follow a transition to an accepting location if they match and to a rejecting location if they do not match. Each possible valuation of available tests (here, only equality) will correspond to a separate location, resulting in a canonical, but possibly very large, automaton.

In order to obtain a smaller number of locations, we can take a different approach to canonicity. We represent a data language as a classification λ that states for any data word whether the word is in the language or not. We claim that for any data language, there is a unique set of λ-*essential* data words whose classification can be extended to the set of all data words. Intuitively, a λ-essential data word has the property that two parameters are equal only if the data word would be classified differently without that equality. For example, the word signup(`Bob`, `lemon`) login(`Bob`, `lemon`) is essential for $\mathcal{L}_{XMPP}$: a user will only be able to login if the password (`lemon`) in the signup message matches the password in the login message.

From a set of λ-essential words, we can use a Nerode-like congruence to build a register automaton that accepts the data language represented by the classification λ. Each λ-essential word will then represent at most one location in the automaton. Since the set of λ-essential words is unique, the automaton will be canonical.

A canonical automaton that recognizes our selected fragment of XMPP is shown in Figure 1. It is a register automaton with three control locations: one initial location (l_0), one location where the user has signed up for an account but not yet logged in (l_1), and one accepting location where the user is logged in (l_2). The initial location is marked by an arrow and the accepting location is denoted by two concentric circles. The automaton has two location variables, denoted x_1, x_2, initialized in location l_1.

The automaton reads one message at a time, and then follows a transition to another location if the corresponding guard is satisfied, assigning new values to location variables. For example, if the automaton is in location l_0 and reads the message $\mathsf{signup}(\mathtt{Bob}, \mathtt{lemon})$, it will follow a transition to l_1, storing $\mathtt{Bob}$ in the variable x_1 and $\mathtt{lemon}$ in the variable x_2. Note that the automaton does not care whether the message parameters are equal or not; the message $\mathsf{signup}(\mathtt{Bob}, \mathtt{Bob})$ would store $\mathtt{Bob}$ in both x_1 and x_2.

When the next message is read, the automaton checks whether it is of the form $\mathsf{login}(p_1, p_2)$ and whether the guard $x_1 = p_1 \wedge x_2 = p_2$ is satisfied by the parameters in the message and the location variables. If this is the case, the automaton goes to the accepting location l_2. Thus, the word $\mathsf{signup}(\mathtt{Bob}, \mathtt{lemon})\ \mathsf{login}(\mathtt{Bob}, \mathtt{lemon})$ is accepted by the automaton. If the parameters and location variables do not satisfy the guard, the automaton simply loops in location l_1.

In the following sections, we will give formal definitions of the concepts in this example.

3. Data languages and register automata

Assume an unbounded domain D of data values and a finite set of *actions*. Each action has a certain *arity* that determines how many data values it carries from the domain D. A *data symbol* is a term of form $\alpha(d_1, \ldots, d_n)$, where α is an action with arity n, and $d_1, \ldots, d_n$ are data values in D. We will sometimes write $\bar{d}$ for $d_1, \ldots, d_n$. A *data word* is a sequence of data symbols. A *data language* is a set of data words that is closed under permutations on D.

Assume a set of formal parameters. A *parameterized symbol* is a term of form $\alpha(p_1, \ldots, p_n)$, where α is an action with arity n and $p_1, \ldots, p_n$ are formal parameters. We sometimes write $\bar{p}$ for $p_1, \ldots, p_n$.

Assume a finite set of *variables* (a.k.a. registers), ranged over by $x_1, x_2, \ldots$.

A *guard* is a conjunction of equalities and negated equalities over formal parameters and variables, e.g., $p_i = p_j$ or $p_i \neq x_j$, .

Definition 1 (Register automaton). A register automaton is a tuple $\mathcal{A} = (L, l_0, X, \mathit{\Gamma}, \lambda)$, where

- L is a finite set of *locations.*
- $l_0 \in L$ is the *initial location.*
- X maps each location $l \in L$ to a finite set $X(l)$ of variables, where $X(l_0)$ is the empty set.
- $\mathit{\Gamma}$ is a finite set of *transitions*, each of which of form $\langle l, (\alpha, \bar{p}), g, \rho, l' \rangle$, where
  - l is the *source location*,
  - l' is the *target location*,
  - $\alpha(\bar{p})$ is a parameterized symbol,
  - g is a guard over $\bar{p}$ and $X(l)$, and
  - ρ is the *assignment*, mapping $X(l')$ to $X(l) \cup \bar{p}$. Intuitively, the variable $x \in X(l')$ is assigned the value of $\rho(x)$.
- $\lambda : L \mapsto \{+, -\}$ maps each location to either $+$ (accept) or $-$ (reject).

$\square$

Semantics of a register automaton. A register automaton $\mathcal{A}$ classifies data words as either accepted or rejected. We define a state of $\mathcal{A}$ as consisting of a location and an assignment to the variables of that location. Then, one can describe how $\mathcal{A}$ processes a data word symbol by symbol: on each symbol, $\mathcal{A}$ finds a transition with a guard that is satisfied by the parameters of the symbol and the current assignment to variables; this transition determines a next location and an assignment to the variables of the new location.

We now describe the semantics of a register automaton more precisely. Consider the register automaton $\mathcal{A} = (L, l_0, X, \mathit{\Gamma}, \lambda)$. A *valuation* is a mapping from location variables to the domain D of data values. For a conjunction g over variables and data symbols and a valuation ν, we write $\nu \models g$ if g evaluates to true under ν.

A *state* of $\mathcal{A}$ is a pair $\langle l, \nu \rangle$ where l is a location in L and ν is a valuation over the variables in $X(l)$. The *initial state* of $\mathcal{A}$ is the pair $\langle l_0, \nu_0 \rangle$ where l_0 is the initial location and ν_0 is the empty valuation.

A *step* $\langle l, \nu \rangle \xrightarrow{\alpha(\bar{d})} \langle l', \nu' \rangle$ of a register automaton transfers the automaton from the state $\langle l, \nu \rangle$ to the state $\langle l', \nu' \rangle$ on the input $\alpha(\bar{d})$. Each step $\langle l, \nu \rangle \xrightarrow{\alpha(\bar{d})} \langle l', \nu' \rangle$ of $\mathcal{A}$ is derived from a transition $\langle l, (\alpha, \bar{p}), g, \rho, l' \rangle \in \mathit{\Gamma}$ where

1. the valuation ν is such that $\nu \models g[\bar{d}/\bar{p}]$, i.e., g evaluates to true under ν when $p_i \cdots p_n$ in g are replaced by $d_i \cdots d_n$, and
2. the valuation ν' is the updated valuation, where $\nu'(x_i) = \nu(x_j)$ whenever $\rho(x_i) = x_j$, and $\nu'(x_i) = d_j$ whenever $\rho(x_i) = p_j$.

A *run* of a register automaton $\mathcal{A}$ over a data word $\alpha_1(\bar{d}_1) \ldots \alpha_k(\bar{d}_k)$ starts in the initial location l_0 and is a sequence of steps

$$\langle l_0, \nu_0 \rangle \xrightarrow{\alpha_1(\bar{d}_1)} \langle l_1, \nu_1 \rangle \; \langle l_1, \nu_1 \rangle \xrightarrow{\alpha_2(\bar{d}_2)} \langle l_2, \nu_2 \rangle \; \ldots \; \langle l_{k-1}, \nu_{k-1} \rangle \xrightarrow{\alpha_k(\bar{d}_k)} \langle l_k, \nu_k \rangle .$$

A run is *accepting* if $\lambda(l_k) = +$ and *rejecting* if $\lambda(l_k) = -$. The automaton $\mathcal{A}$ accepts a data word w if there is an accepting run over w. The data language recognized by $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$ is the set of data words that it accepts.

A register automaton is *completely specified* if for any state $\langle l, \nu \rangle$ and any input $\alpha(\bar{d})$, there is a transition $\langle l, (\alpha, \bar{p}), g, \rho, l' \rangle \in \mathit{\Gamma}$ such that $\nu \models g[d/\bar{p}]$. A register automaton is *determinate* if the runs over any data word w are either all rejecting or all accepting. By strengthening the transition guards, a determinate register automaton can easily be made deterministic, but this transformation can be done in several ways, which is why we have chosen to focus on canonical determinate models instead. We refer to automata that are completely specified and determinate as DRAs. A data language is *DRA-recognizable* if it is accepted by a DRA.

In general, a DRA can have transition guards that imply $x_i = x_j$ or $x_i \neq x_j$ for two distinct location variables x_i, x_j. In a *right-invariant* DRA, transition guards may not imply any such relation between location variables. More precisely, a DRA is right-invariant if for each transition $\langle l, \alpha(\bar{p}), g, \rho, l' \rangle$,

- the guard g does not imply $x_i = x_j$ or $x_i \neq x_j$ for distinct $x_i,\ x_j \in X(l)$, and

- the combined effect of the guard g and the assignment ρ does not imply any equality between distinct variables $x_i,\ x_j \in X(l')$ (note that negated equalities may be implied).

In a right-invariant DRA, two data values can thus only be tested for equality if one of them is in the current input symbol. Furthermore, if a transition guard contains a test for equality between two data values, then at most one of them can be stored in a location variable of the target location. In this paper, we only consider right-invariant DRAs, and assume that any DRA mentioned is right-invariant unless otherwise stated.

4. A succinct representation of data languages

In this section, we describe how to represent any data language (not necessarily DRA-recognizable) in a succinct way. Given a data language, we will define a subset of the data words (these are the so-called *essential* words), with the property that the data language can be described succinctly by saying which essential words are accepted and which are rejected.

For any set S, let a *classification* of S be a mapping from S to $\{+, -\}$. A data language can thus be seen as a classification of a set of data words that is closed under permutations on the domain of data values. From now on, we will mostly regard a data language as a classification of data words, and will typically use the symbol λ for such classifications.

For a data word w, let $Vals(w)$ be the (ordered) sequence of data values in w, let $Acts(w)$ be the sequence of actions in w, and let $|w|$ be the number of data values in w. For $k \leq |w|$, let $Vals(w)|_k$ denote the sequence of the first k data values in w, and let $Acts(w)|_k$ denote the prefix of $Acts(w)$ that has the first k data values as parameters. For example, if $w = a(1,2)b(3)$, then $Vals(w)|_2 = 1, 2$ and $Acts(w)|_1 = a$. We say that w is *fresh at k* if the kth data value is different from all preceding data values (i.e., from the data values in $Vals(w)|_{k-1}$).

For two data words w, w', let $w \simeq w'$ denote that w' can be obtained from w by a permutation on the domain D. Without loss of generality, let the domain D be the set of positive integers. A data word w is *representative* if the set of data values that appear in any prefix of $Vals(w)$ is of form $\{1, 2, \ldots, k\}$ for some k. Thus the data word $a(1,1)b(2)$ is representative, but the data word $a(1,3)b(2)$ is not. Clearly, there is exactly one representative word in each equivalence class of $\simeq$. Thus, a data language can be uniquely described as a classification of the set of representative words. In the following, whenever a data word is mentioned, we will assume that it is a representative data word unless otherwise stated.

Let us introduce two partial orders, $\sqsubseteq$ and $<$, on the set of data words.

- Let $\boldsymbol{w \sqsubseteq w'}$ denote that w' can be obtained from w by a mapping on D, i.e., that whenever two data parameters are equal in w, they are also equal in w'. For example, $a(1,2)b(1) \sqsubseteq a(1,1)b(1)$. The smallest elements w.r.t. $\sqsubseteq$ are data words where all data values are pairwise different; the largest elements are data words where all data values are equal. We call a set W of data words $\sqsubseteq$-*closed* if $w' \in W$ and $w \sqsubseteq w'$ imply $w \in W$.

- Let $\boldsymbol{w < w'}$ denote that for some $k > 0$ with $k \leq |w|$ and $k \leq |w'|$,

  - $Vals(w)|_{k-1} = Vals(w')|_{k-1}$ and $Acts(w)|_k = Acts(w')|_k$, and
  - w is fresh at k, but w' is not fresh at k.

  Note that we do not require $Acts(w) = Acts(w')$, i.e., w and w' need not have the same length. Let $w \leq w'$ denote that $w = w'$ or $w < w$. Note that $w \sqsubseteq w'$ implies $w \leq w$.

Intuitively, $w < w'$ means that w and w' share the same prefix up to the first $k-1$ data values, for some k. Thereafter, the kth data value in w' is equal to one of the previous data values, whereas the kth data value in w is not equal to any of the previous data values. In some sense, $<$ can be thought of as a "lexicographic" ordering of data words, in which a data values that are equal to some preceding data value are larger than data values that are not. For example, $a(1,2)b(3) < a(1,2)b(2)c(3) < a(1,1)b(2)$, while $a(1,2)b(1)$ and $a(1,2)b(2)$ are not related by $<$.

Let w be a data word, let $Vals(w) = d_1 \cdots d_n$ and let $0 \leq k \leq n$. We say that w is *flat after* k if w is fresh at j for $k < j \leq n$ (i.e., the data values $d_{k+1}, \ldots, d_n$ are different from each other and from all data values among $d_1, \ldots, d_k$). For example, the word $a(1,1)b(2,3)$ is flat after 2, and the word $a(1,2)b(2,3)$ is flat after 3.

Let $w \simeq_k w'$ denote that $Vals(w)|_k = Vals(w)|_k$ and $Acts(w)|_k = Acts(w)|_k$. We say that a data word w is a *k-flattening* of a data word w' if $w \simeq_k w'$ and w is flat after k. As a special case, any word w is flat after $|w|$. Flattening can be seen as a generalization of the standard prefix relation on words, in that a k-flattening can be obtained by first taking a prefix with k data values, and thereafter extending the word with fresh data values. Let $\lfloor w \rfloor_k$ denote the k-flattening of w with $Acts(\lfloor w \rfloor_k) = Acts(w)$. Let the *flat-index* of a word w be the smallest k such that w is flat after k.

Let W and W' be sets of data words. Then W' is W*-flattening-closed* if $w \in W'$ implies that all flattenings of w that are in W, are also in W'. For example, let $w' = a(1,1)b(2,3)$, let $w'' = a(1,2)b(3,4)$, and let $W = \{w', w''\}$. Both w' and w'' are flattenings of the data word $w = a(1,1)b(1,1)$. Then the set $\{w,\ w',\ w''\}$ is W-flattening-closed.

One reason for having the parameter W in the definition of flattening-closed is that the set of flattenings of any data word is infinite. The parameter W makes it possible to say that a finite set W of data words is W-flattening-closed. Note that if W is $\sqsubseteq$-closed, then W is W-flattening-closed.

Definition 2 (Best-matching). Let W be a $\sqsubseteq$-closed set of (representative) data words and let $\Phi \subseteq W$ be a W-flattening-closed set of data words. Define the relation $\preceq_\Phi \subseteq \Phi \times W$ between Φ and W, by letting $w \preceq_\Phi w'$ iff w is a maximal (w.r.t. $<$) data word in Φ such that $w \sqsubseteq w'$. □

Intuitively, if $w \preceq_\Phi w'$, then w is a data word in Φ which "best matches" w', where "best" is taken w.r.t. the partial order $<$. For example, let $w = a(1,2)b(3,2)$, let $w' = a(1,2)b(1,3)$, let $w'' = a(1,1)b(1,1)$ and let $\Phi = \{w,\ w'\}$. Then $w \sqsubseteq w''$ and $w' \sqsubseteq w''$, but since $w < w'$, we have that $w' \preceq_\Phi w''$. Note that in general, for any w' there can be several different w with $w \preceq_\Phi w'$.

We will use the relation $\preceq_\Phi$ to define how a classification of a set W can be succinctly represented by its restriction to a subset of W.

Definition 3 (λ-sufficient). Let W be a $\sqsubseteq$-closed set of data words, let $\Phi \subseteq W$ be a non-empty W-flattening-closed set of (representative) data words, and let λ be a classification of W. Then Φ is λ*-sufficient* if $\lambda(w) = \lambda(w')$ whenever $w \preceq_\Phi w'$ for $w \in \Phi$ and $w' \in W$. □

We now define a particular λ-sufficient subset of W. For a set Φ of words, and a data word w, let $\Phi_{<w}$ denote the set of words $w' \in \Phi$ such that $w' < w$.

Definition 4 (λ-essential). Let W be a *finite* $\sqsubseteq$-closed set of data words, and let λ be a classification of W. We define the set Ψ of λ-*essential* inductively, considering the data words in W in increasing $<$-order, as follows.

- As a base case, Ψ contains all data words in W that are flat after 1.
- If the flat-index of w is $k > 1$, then $w \in \Psi$ if
  - $\lfloor w \rfloor_{k-1} \in \Psi_{<w}$, and
  - there is a data word $x \in W$ such that w is a k-flattening of x, and a λ-essential word $v \in \Psi_{<w}$, such that $v \preceq_{\Psi_{<w}} x$ and $\lambda(v) \neq \lambda(x)$.

$\square$

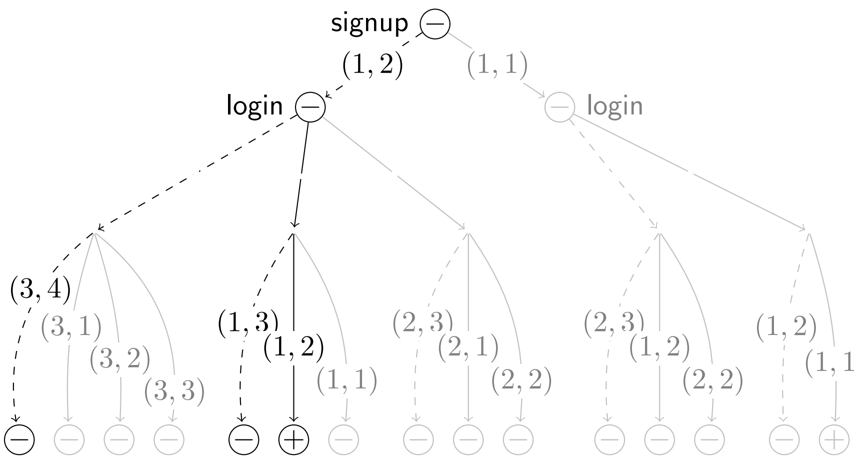


Figure 2: The set of λ-essential words for $\mathcal{L}_{XMPP}$, as a tree

Example 1 We can represent a $\sqsubseteq$-closed set W of data words, together with a classification λ (i.e., a data language), as a tree. This will let us illustrate how the set $\Psi \subseteq W$ of λ-essential data words is constructed.

Figure 2 shows $\mathcal{L}_{XMPP}$ (limited to words with at most four data values) represented as a tree. Words are read top-down, and each path from the root node represents a data word. Each node is marked $-$ and $+$, denoting the classification of the data word that reaches it.

Dashed paths denote that the last data value read is not equal to any previously seen data value. For example, there is a dashed path from the root node to $\mathsf{signup}(1, 2)$, since 2 is not equal to 1.

The partial order $<$ runs from left to right in the tree, but two nodes are incomparable if they can be reached via non-dashed paths from a common predecessor (i.e, $\mathsf{signup}(1, 2)\ \mathsf{login}(1, 3)$ and $\mathsf{signup}(1, 2)\ \mathsf{login}(2, 2)$ are incomparable).

Black paths (dashed and solid) denote λ-essential words. We now describe how to construct the set Ψ of λ-essential words for $\mathcal{L}_{XMPP}$. Words in W are added to Ψ in $<$-order. A word w is only added if it cannot be classified in the same way as an already added best-matching word, i.e., the $<$-maximal word w' in Ψ such that $w' \sqsubseteq w$.

As a base case, all data words that are flat after 1 are λ-essential. These correspond to the leftmost branch in the tree. Then, the $<$-smallest words in the tree are processed: these are the three siblings of $\mathsf{signup}(1,2)$ $\mathsf{login}(3,4)$ (incomparable among each other). None of them are λ-essential, since they all have the same classification as $\mathsf{signup}(1,2)$ $\mathsf{login}(3,4)$.

The next word by $<$ is $\mathsf{signup}(1,2)$ $\mathsf{login}(1,3)$. It is added, because (i) $\mathsf{signup}(1,2)$ $\mathsf{login}(3,4)$ is λ-essential, and because (ii) $\mathsf{signup}(1,2)$ $\mathsf{login}(1,3)$ flattens $\mathsf{signup}(1,2)$ $\mathsf{login}(1,2)$, which cannot be classified in the same way as its best-matching word $\mathsf{signup}(1,2)$ $\mathsf{login}(3,4)$. Since words are added in $<$-order, $\mathsf{signup}(1,2)$ $\mathsf{login}(1,3)$ must be added now or not at all. Finally, we add $\mathsf{signup}(1,2)$ $\mathsf{login}(1,2)$, because it is not classified in the same way as its best-matching word, which is now $\mathsf{signup}(1,2)$ $\mathsf{login}(1,3)$.

The set of λ-essential words for $\mathcal{L}_{XMPP}$ now contains exactly the words represented by the black branches. Any other data word w can use the classification of the black branch that represents its best-matching word in Ψ. For example, the data word $\mathsf{signup}(1,1)$ $\mathsf{login}(1,1)$ is not λ-essential, and it is $\sqsubseteq$-larger than any word in the tree. The best-matching word for $\mathsf{signup}(1,1)$ $\mathsf{login}(1,1)$ is $\mathsf{signup}(1,2)$ $\mathsf{login}(1,2)$ so we let $\mathsf{signup}(1,1)$ $\mathsf{login}(1,1)$ use its classification.

It is not difficult to see that Ψ as constructed in Definition 4 is λ-sufficient. In order to extend Definition 4 to the case where W is infinite, we need the following lemma.

Lemma 1 Let W be a finite $\sqsubseteq$-closed set of data words, let λ be a classification of λ, and let Ψ be the set of λ-essential words. If Φ is a λ-sufficient set of words, then $\Psi \subseteq \Phi$.

The proof of Lemma 1 is in Appendix A. Using Lemma 1, we can give a robust definition of λ-essential words.

Definition 5 (λ-essential). Let W be a $\sqsubseteq$-closed set of data words, and let λ be a classification of W. For any n, let λ_n be the restriction of λ to words with length at most n, and let Ψ_n be the set of λ_n-essential words. Define the set of λ-essential words as $\bigcup_{n=1}^{\infty} \Psi_n$. $\square$

We now get to the main theorem which states that the set of λ- essential words is the minimal λ-sufficient subset of W. The theorem slightly generalizes Theorem 1 in [8].

Theorem 1 Let W be a $\sqsubseteq$-closed set of data words. Let λ be a classification of W. Then the set of λ-essential words is the minimal λ-sufficient set.

By minimal, we mean that if Φ is any other λ-sufficient set, then $\Psi \subseteq \Phi$.

Theorem 1 implies that any classification λ of a $\sqsubseteq$-closed set W can be succinctly represented by its restriction to the set of λ-essential words. The value of λ for a λ-essential word w will then generalize to all data words $w' \in W$ with $w \preceq_\Psi w'$. Note that for each $w' \in W$, any non-empty W-flattening-closed subset $\Psi \subseteq W$ of data words must contain at least one data word w such that $w \preceq_\Psi w'$.

Proof. It is not difficult to see that Ψ is λ-sufficient by construction. To prove that Ψ is the minimal such set, it follows by Lemma 1, using the notation of Definition 5, that the words in Ψ_n must be in Ψ in order to correctly classify all words of length $\leq n$. The theorem follows. $\square$

5. Nerode congruence and canonical form

In this section, we explain how to construct a canonical DRA for a data language λ. Intuitively, this can be seen as 'folding' the set of λ-essential words so that runs of the canonical DRA correspond exactly to λ-essential words. Two λ-essential words w, w' will lead to the same location, if their possible 'continuations' into longer λ-essential words are equivalent.

We assume an infinite ordered set $D_V = \{\mathtt{1}, \mathtt{2}, \mathtt{3}, \ldots\}$, which is disjoint from D (note that we use a different font to emphasize this difference). Let a *suffix* be a data word whose data values are in $D \cup D_V$, such that the set of data values in D_V that appear in any prefix of its parameters is of form $\{\mathtt{1}, \mathtt{2}, \ldots, k\}$ for some k. (This need not hold for the data values in D; e.g., $c(\mathtt{1}, 3)d(2, \mathtt{2})$ is a suffix.) For a data word w, let a *w-suffix* be a suffix v such that $(Vals(v) \cap D) \subseteq Vals(w)$. For a data word w and a w-suffix v, let $w; v$ denote the unique representative data word of form wv' such that v' is obtained from v by an injective mapping $\sigma : D_V \mapsto (D \setminus Vals(w))$ mapping each element in D_V to a data value that does not occur in the word w. This will ensure that the data values in the suffix that are not equal to any data value in the prefix are mapped to 'fresh' data values when concatenating a prefix and a suffix.

Example 2 Let $w = a(1, 2)$ be a data word and let $v = b(\mathtt{1}, 2)c(\mathtt{2}, \mathtt{1})$ be a suffix. Then the unique representative data word $w; v = a(1, 2)b(3, 2)c(4, 3)$ is obtained by letting $\sigma(\mathtt{1}) = 3$, and $\sigma(\mathtt{2}) = 4$.

Let W be a $\sqsubseteq$-closed set of data words, let λ be a classification of W, and let w be a λ-essential word. Let a λ-*essential* w-*suffix* be a w-suffix v such that $w; v$ is a λ-essential word. The λ-*essential residual language* after w, denoted $\lfloor w^{-1}\lambda \rfloor$, is the classification of the set of λ-essential w-suffixes defined by $\lfloor w^{-1}\lambda \rfloor(v) = \lambda(w; v)$.

Definition 6 (Memorable). Let W be a $\sqsubseteq$-closed set of data words, let λ be a classification of W, and let w be a λ-essential word. Define the set of λ-*memorable* data values of w, denoted $\mathrm{mem}_\lambda(w)$, as the data values in $Vals(w)$ that also occur in some λ-essential w-suffix. $\square$

Intuitively, $\mathrm{mem}_\lambda(w)$ is the set of data values that must be remembered after processing w in order to be able to correctly classify all continuations of w.

We will define a Nerode-like congruence on data words, where two words are equivalent if their residual languages are equivalent after mapping memorable data values. The following example illustrates why such a mapping is necessary.

Example 3 Consider $\mathcal{L}_{XMPP}$ in Figure 1. Let $w = \mathsf{signup}(1, 2)\ \mathsf{login}(1, 2)$, with $\mathrm{mem}_\lambda(w) = \{1, 2\}$, and let $w' = \mathsf{signup}(1, 2)\ \mathsf{login}(1, 2)\ \mathsf{change}(3)$, with $\mathrm{mem}_\lambda(w') = \{1, 3\}$. Let $v = \mathsf{logout}\ \mathsf{login}(1, 2)$ be a suffix. We want to find out if w and w' are equivalent by checking if their residual languages classify suffixes in the same way. In the automaton, w and w' both lead to the same location. However, adding the suffix v to w and w', respectively, results in $\lambda(w; v) = +$ but $\lambda(w'; v) = -$. We can only make the residual languages equivalent by mapping the memorable data values: 1 in $\mathrm{mem}_\lambda(w)$ maps to 1 in $\mathrm{mem}_\lambda(w')$, and 2 in $\mathrm{mem}_\lambda(w)$ maps to 3 in $\mathrm{mem}_\lambda(w')$.

We now formally define this mapping. For a classification λ of a set of suffixes, and a permutation γ on D, define $\gamma\langle\lambda\rangle$ as the classification of the set of suffixes $\{\gamma(v) \mid v \in Dom(\lambda)\}$, defined by $\gamma\langle\lambda\rangle(\gamma(v)) = \lambda(v)$. For example, let $b(1,2)$ be an $a(1,2)$-suffix. If $\lambda(b(1,2)) = +$ and γ is such that $\gamma(1) = 7$ and $\gamma(2) = 5$, then $\gamma\langle\lambda\rangle$ is such that $\gamma\langle\lambda\rangle(b(7,5)) = +$.

We are now ready to define a Nerode-like congruence on a $\sqsubseteq$-closed set of λ-essential words. This congruence can then be used to construct a succinct DRA that recognizes the data language represented by the classification λ.

Definition 7 (Nerode congruence). Let $\lambda : W \mapsto \{+,-\}$ be a classification of a $\sqsubseteq$-closed set W of representative data words. We define the equivalence $\equiv_\lambda$ on the set of λ-essential words by $w \equiv_\lambda w'$ iff there is a bijection γ on D such that $\lfloor {w'}^{-1}\lambda \rfloor = \gamma\langle\lfloor w^{-1}\lambda\rfloor\rangle$. $\square$

Intuitively, two representative data words are equivalent if they induce the same residual languages modulo a remapping of their memorable parameters. The equivalence $\equiv_\lambda$ is also a congruence in the following sense: the bijection γ used in Definition 7 need only relate memorable data values, i.e., it is enough to define it as $\gamma : \mathrm{mem}_\lambda(w) \mapsto \mathrm{mem}_\lambda(w')$. Then, for any $\mathrm{mem}_\lambda(w)$-suffix v, if $w \equiv_\lambda w'$ and γ is such that $\lfloor {w'}^{-1}\lambda \rfloor = \gamma\langle\lfloor w^{-1}\lambda\rfloor\rangle$, we have that $w;v \equiv_\lambda w';\gamma(v)$.

We now state the main result of our paper, which relates our Nerode congruence to DRAs.

Theorem 2 A data language, represented by a classification λ, is recognizable by a DRA iff the equivalence $\equiv_\lambda$ on λ-essential words has finite index.

PROOF. *If:* We construct a DRA from a given equivalence $\equiv_\lambda$, as the DRA $\mathcal{A} = (L, l_0, X, \Gamma, \lambda)$, where

- L consists of the finitely many equivalence classes of the equivalence relation $\equiv_\lambda$ on λ-essential words. We let each equivalence class be represented by a particular member, called its *access string*. In the following, when we write $[w]$, this denotes that w is the access string of the equivalence class $[w]$.
- l_0 is $[\epsilon]_{\equiv_\lambda}$.
- X maps each location $[w]_{\equiv_\lambda}$ to the set $\{x_d | d \in \mathrm{mem}_\lambda(w)\}$.

- Γ is constructed as follows: For each location $l = [w]_{\equiv_\lambda}$ in L, with variables $X(l)$ and for each λ-essential one-symbol w-suffix of the form $\alpha(\bar{d})$, there is a transition in Γ of form $\langle l, (\alpha, \bar{p}), g, \rho, l' \rangle$ (where $\bar{d} = d_1 \cdots d_n$, and $\bar{p} = p_1 \cdots p_n$) where
  - $l' = [w;\alpha(\bar{d})]_{\equiv_\lambda}$. Let γ be a permutation such that $\lfloor w;\alpha(\bar{d})^{-1}\lambda \rfloor = \gamma\langle \lfloor w'^{-1}\lambda \rfloor \rangle$.
  - g is obtained from the set of λ-essential one-symbol w-suffixes in the following way. For each data value $d_j \in d_1 \cdots d_n$, each data value $d_i \in d_1 \cdots d_{j-1}$, and each variable $x_d \in X(l)$,
    - $p_i = p_j$ is a conjunct in g if $d_i = d_j$, and
    - $p_i \neq p_j$ is a conjunct in g if $d_i \neq d_j$, and there is some other λ-essential one-symbol w-suffix $\alpha(d'_1 \cdots d'_n)$ with $d'_k = d_k$ for $1 \leq k \leq j-1$, and $d'_i = d'_j$
    - $x_d = p_j$ is a conjunct in g if $x_d = d_j$, and
    - $x_d \neq p_j$ is a conjunct in g if $x_d \neq d_j$, and there is some other λ-essential one-symbol w-suffix $\alpha(d'_1 \cdots d'_n)$ with $d'_k = d_k$ for $1 \leq k \leq j-1$, and $x_d = d'_j$.
  - ρ maps location variables in l' to location variables in l and formal parameters in $\alpha(\bar{p})$, such that for $d' \in \mathrm{mem}_\lambda(w')$:
    - $\rho(x_{d'}) = p_j$ if $\gamma(d_j) = d'$ for $d_j \in \bar{d}$, and
    - $\rho(x_{d'}) = x_d$ if $\gamma(d) = d'$ for $d \in \mathrm{mem}_\lambda(w)$.
- $\lambda([w]_{\equiv_\lambda}) = \lambda(w')$.

The constructed DRA is well defined: it has finitely many locations since the index of $\equiv_\lambda$ is finite, the initial location is defined as the class of the empty word, and λ is defined from λ for the representative elements of the locations. By construction, the transition relation is total and the automaton is determinate.

To complete this direction of the proof, we need to show that the constructed automaton $\mathcal{A}$ indeed recognizes λ. This means that if a data word w generates a sequence of transitions in the automaton, the final state of that sequence is accepting iff $\lambda(w) = +$ and rejecting iff $\lambda(w) = -$.

Consider an arbitrary sequence s of transitions, of the form

$$s = \langle l_0, \alpha_1(\bar{p}_1), g_1, \rho_1, l_1 \rangle \quad \cdots \quad \langle l_{k-1}, \alpha_k(\bar{p}_k), g_k, \rho_k, l_k \rangle.$$

The guards $g_1 \cdots g_k$ can be conjoined to form a constraint $\varphi_s = g_1 \wedge \rho_1(\cdots \wedge \rho_{k-1}(g_k))$. There are (in general) several data words that can generate the sequence s. For each transition in s, the data values in any such data word satisfy the corresponding guard in φ_s. Let w be a data word with $Vals(w) = d_1, \cdots, d_n$, such that w generates s. Then, $d_i = d_j$ if $p_i = p_j \in \varphi_s$, and $d_i \neq d_j$ if $p_i \neq p_j \in \varphi_s$ for $i, j \leq n$. We then have two cases.

1. If w is $\sqsubseteq$-minimal, then by construction, w is also λ-essential since each guard in φ_s is generated from a λ-essential equality between data values. Then $\lambda(l_k) = \lambda(w)$.

2. If w is not $\sqsubseteq$-minimal, then there is another data word w' with $Vals(w') = d'_1, \cdots, d'_n$ that is $\sqsubseteq$-minimal (and thereby λ-essential) , such that w' generates the sequence s of transitions, and such that $w' \preceq_\Psi w$. We show this by contradiction. Assume that s generates a run over another data word w'' with $Vals(w'') = d''_1, \cdots, d''_n$ such that $w'' \preceq_\Psi w$, which implies that $w' < w''$ and $w'' \sqsubseteq w$. Then there are $i \leq j < k \leq n$ for which $Vals(w')|_j = Vals(w'')|_j$ and $d'_i \neq d'_k$ but $d''_i = d''_k$. Then φ_s can have no constraint over d_i, d_k, so $d''_i = d''_k$ is not a λ-essential equality. Thus w'' cannot be λ-essential which contradicts our initial assumption that $w'' \preceq_\Psi w$. Since $w' \preceq_\Psi w$, we have that $\lambda(l_k) = \lambda(w)$.

Only-if: Assume any right-invariant DRA that accepts λ. We show that two λ-essential words that cause sequences of transitions leading to the same location are also equivalent w.r.t. $\equiv_\lambda$, i.e., that one location of a DRA cannot represent more than one class of $\equiv_\lambda$. This can be shown using the definition of right-invariance, as follows.

We define a location variable $x \in X(l)$ as *non-live* if the value of x has no influence on whether a run that passes l will be accepting or rejecting (we omit the precise details). Define a location variable to be *live* if it is not non-live. We can then establish that if w is a λ-essential word that causes a sequence of transitions leading to location l, then any live location variable in $X(l)$ must be assigned to a data value in $\mathrm{mem}_\lambda(w)$. By right-invariance, there is thus a bijection between the live location variables in $X(l)$ and the data values in $\mathrm{mem}_\lambda(w)$. It follows that whenever w and w' are two λ-essential words that reach the same location, then their residual languages must be equivalent w.r.t. $\equiv_\lambda$, and thus w and w' are equivalent according to Definition 7. □

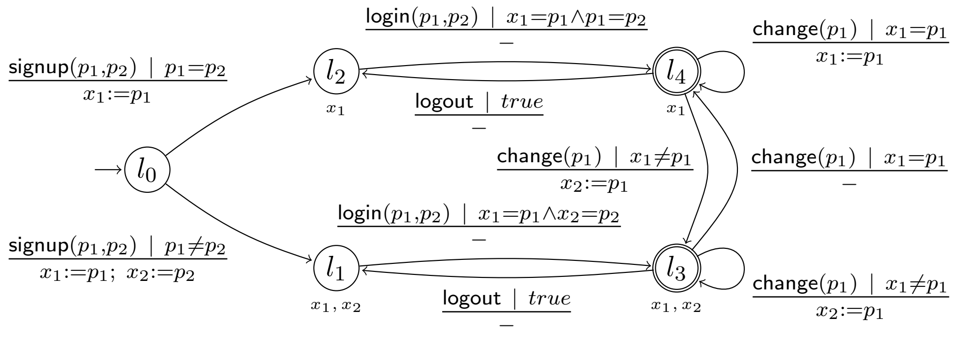

signup(p_1,p_2) | p_1=p_2
x_1:=p_1
login(p_1,p_2) | x_1=p_1∧p_1=p_2
–
change(p_1) | x_1=p_1
x_1:=p_1
l_2
x_1
logout | *true*
–
l_4
x_1
l_0
change(p_1) | x_1≠p_1
x_2:=p_1
change(p_1) | x_1=p_1
–
login(p_1,p_2) | x_1=p_1∧x_2=p_2
–
signup(p_1,p_2) | p_1≠p_2
x_1:=p_1; x_2:=p_2
l_1
x_1, x_2
logout | *true*
–
l_3
x_1, x_2
change(p_1) | x_1≠p_1
x_2:=p_1

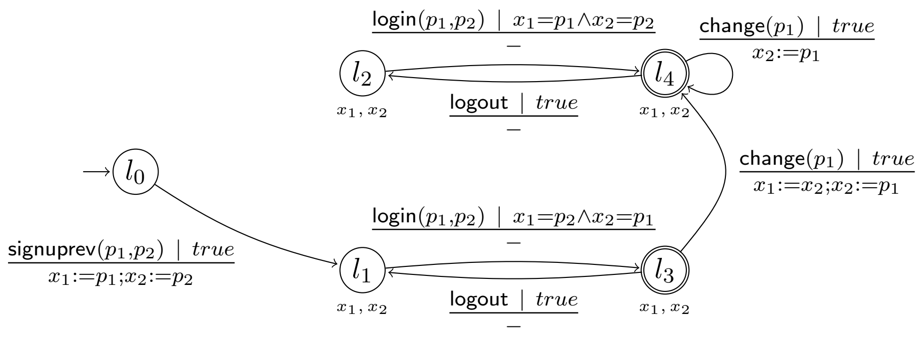
Figure 3: Deterministic unique right invariant RA

6. Comparison between different automata models

In this section, we compare our register automata to other proposed formalisms, showing that our models can be exponentially more succinct. We will discuss some of the restrictions imposed on these other kinds of register automata, and how they affect the size of the automata models. In particular, we will discuss register automata that require variables to store unique data values, to store variables in order of their last appearance, or both.

Unique-valued register automata. An RA is *unique-valued* if the valuation ν in any reachable state $\langle l, \nu \rangle$ is injective, i.e., two variables can never store identical data values. An example of a deterministic unique-valued RA (DURA) can be seen in Figure 3. This DURA accepts the same language ($\mathcal{L}_{XMPP}$) as the DRA in Figure 1. Since variables are required to store unique values, there must be two transitions to different locations after reading $\mathsf{signup}(d_1, d_2)$, depending on whether $d_1 = d_2$ or $d_1 \neq d_2$. After reading $\mathsf{change}(d_1)$, the automaton will be in either l_3 or l_4, depending on whether $d_1 = x_1$ (where x_1 stores the first data value from $\mathsf{signup}(d_1, d_2)$). Recall that the DRA for $\mathcal{L}_{XMPP}$ in Figure 1 has three location. In order to accommodate the uniqueness constraint, a DURA for the same language needs five locations.

Ordered register automata. An RA is *ordered* if there is an ordering (here, we use $\ll$) of variables. In a deterministic ordered RA (DORA), if x_i and x_j are two location variables with $x_i \ll x_j$, then the transition at which x_i was last assigned a data value must coincide with or precede the transition at which x_j was last assigned a data value. Depending on the order in which data values



$\mathsf{login}(p_1,p_2) \mid x_1{=}p_1 \wedge x_2{=}p_2$ / $-$
$\mathsf{change}(p_1) \mid true$ / $x_2{:=}p_1$
l_2
l_4
x_1, x_2
$\mathsf{logout} \mid true$ / $-$
x_1, x_2
l_0
$\mathsf{change}(p_1) \mid true$ / $x_1{:=}x_2; x_2{:=}p_1$
$\mathsf{login}(p_1,p_2) \mid x_1{=}p_2 \wedge x_2{=}p_1$ / $-$
$\mathsf{signuprev}(p_1,p_2) \mid true$ / $x_1{:=}p_1; x_2{:=}p_2$
l_1
l_3
x_1, x_2
$\mathsf{logout} \mid true$ / $-$
x_1, x_2

Figure 4: Deterministic ordered right invariant RA

in the first symbol appear, a DORA accepting $\mathcal{L}_{XMPP}$ will be different. If the first data value in the signup data symbol represents a username and the second, a password, then the DORA will be identical to the DRA for $\mathcal{L}_{XMPP}$ in Figure 1. If, on the other hand, the order is switched, the DORA will be as in Figure 4. Here, the signuprev symbol represents the signup symbol, but with the order of the data values reversed. This automaton has five locations, because after the first occurrence of the change data symbol, it will need to reorder the stored data values. This is necessary to allow changing the password later, since x_2 must be re-assigned a data value after x_1 is.

We can also define an OURA, which is both ordered and unique-valued. The automata of [3] correspond to deterministic OURAs (DOURAs). Figure 5 shows a deterministic OURA accepting $\mathcal{L}_{XMPP}$. The DOURA must at any time store either one or two data values in variables (depending on whether they are equal or not). After the initial location, the automaton will be in locations l_2 or l_4 if only one variable is needed and transition to one of the other states whenever a second variable is needed. To accommodate both the uniqueness and the ordered constraints, the DOURA thus needs seven locations.

As we have seen, automata that require variables to store unique data values, or to store data values in a certain order, may need more locations than the minimal DRA in order to accept the same data language. In the worst case, the blow-up can be exponential in the number of locations. In terms of the models described here, we can identify two such exponential blow-ups: one between DRAs and DURAs, and another between DURAs and DOURAs. The first exponential blow-up, between DRAs and DURAs,

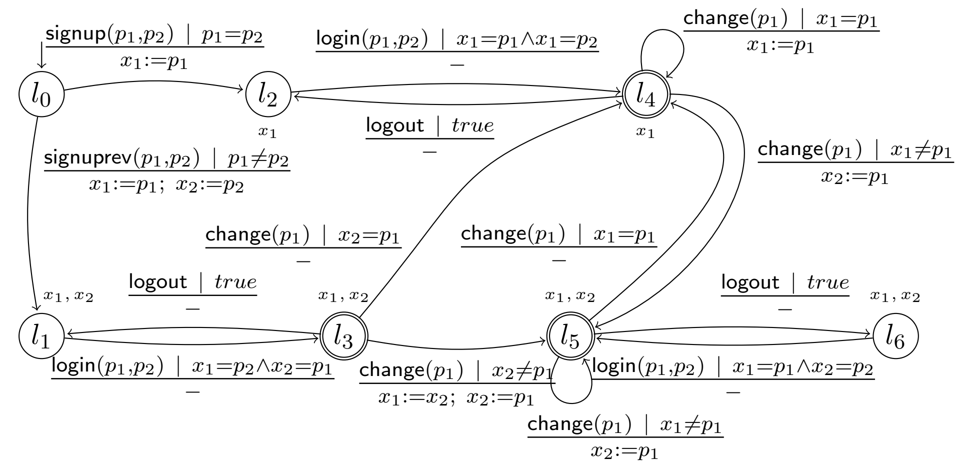

signup(p_1,p_2) | p_1=p_2
x_1:=p_1
login(p_1,p_2) | x_1=p_1∧x_1=p_2
–
change(p_1) | x_1=p_1
x_1:=p_1
l_0
l_2
l_4
x_1
x_1
logout | *true*
–
signuprev(p_1,p_2) | p_1≠p_2
x_1:=p_1; x_2:=p_2
change(p_1) | x_1≠p_1
x_2:=p_1
change(p_1) | x_2=p_1
–
change(p_1) | x_1=p_1
–
logout | *true*
–
logout | *true*
–
x_1, x_2
x_1, x_2
x_1, x_2
x_1, x_2
l_1
l_3
l_5
l_6
login(p_1,p_2) | x_1=p_2∧x_2=p_1
–
change(p_1) | x_2≠p_1
x_1:=x_2; x_2:=p_1
login(p_1,p_2) | x_1=p_1∧x_2=p_2
–
change(p_1) | x_1≠p_1
x_2:=p_1

Figure 5: Deterministic ordered unique-valued right invariant RA

can be shown by constructing a DRA that can store n independent variables. The corresponding DURA then has to maintain in the set of locations which of the n variables have the same value.

Proposition 1 There is a sequence of data languages $\mathcal{L}_1, \mathcal{L}_2, \ldots$, such that the number of locations in the minimal DRA for $\mathcal{L}_n$ is always 4, but the number of locations in the minimal DURA for $\mathcal{L}_n$ is exponential in n. $\square$

The idea is to construct a language for which the minimal DURA has to model many special cases for equal data values, while the DRA can store equal data values in multiple registers.

Proof. Consider the language

$$\mathcal{L}_n = \{a(d_1, \ldots, d_n)b(d'_1, \ldots, d'_n) \mid d_j = d'_j \text{ for } 1 \leq j \leq n.\}$$

where a and b are actions with arity n.

The minimal DRA for $\mathcal{L}_n$ has four locations: From the initial location an a-transition leads to a location with n registers storing the values of p_1 to p_n. From this location, a b-transition with a guard requiring the equality of parameters of b to registers leads to an accepting location, and one b-transition for all the other cases leads to a rejecting location. The minimal DURA, on the other hand, has an exponential number of locations after the initial a-transition encoding potential equalities between parameters of a.

Since each unique value can only be stored once in a DURA, one location is needed for every possible partition of $d_1, \ldots, d_n$ into blocks of equal data values. $\square$

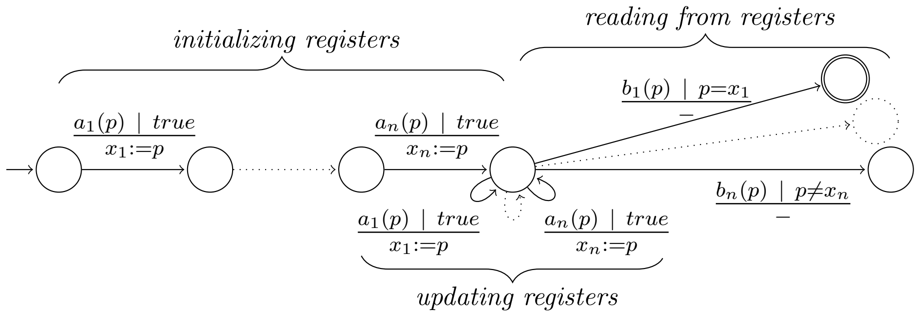
The second exponential blow-up, between DURAs and DOURAs, can be shown by constructing a DURA that allows random (write) access to n variables. The corresponding DOURA then has to encode in the set of locations the order in which the variables are written.

Proposition 2 There is a sequence of languages $\mathcal{L}_1, \mathcal{L}_2, \ldots$, such that the number of locations in the minimal DURA for $\mathcal{L}_n$ is $\mathcal{O}(n)$, while the number of locations in the minimal DOURA for $\mathcal{L}_n$ is exponential in n. $\square$

Proof. Let $A \cup B$ be a set of actions, all with arity one, where $A = \{a_1, \cdots, a_n\}$ and $B = \{b_1, \cdots, b_n\}$. Let D be a domain of data values. Each data symbol is then of the form $a_i(d)$ or $b_i(d)$ with $1 \leq i \leq n$. We can think of i, d as a key-value pair, where the data value d has the key i. We use a_i to store a value under the key i and b_i to read the value from the key i. In a DURA, labels will coincide with registers. A DOURA, on the other hand, has to store values for labels in the order they are written, which in general will not coincide with the order of the registers.

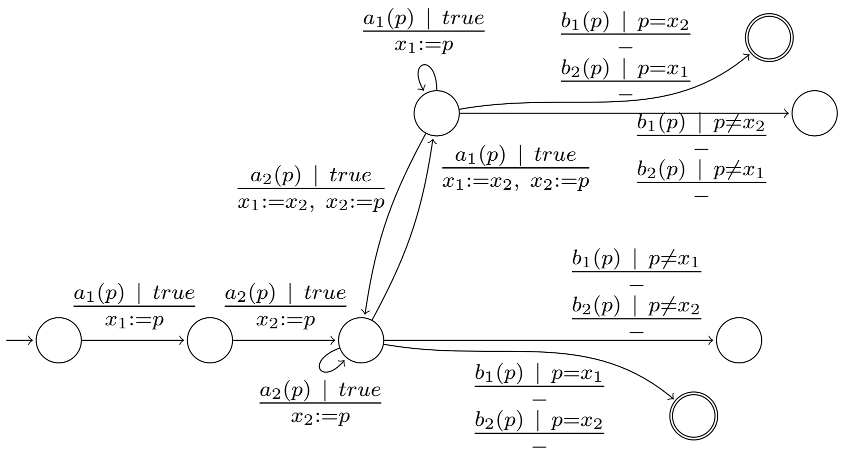
Let the data language $\mathcal{L}_n$ be the set of sequences uvw with prefix $u = a_{11}(d_1) \cdots a_{n1}(d_n)$, middle part $v \in A^*$, and last symbol $w \in B$, where for $w = b_k(d)$ the data value d equals the data value of the last a_k in uv. In order to model $\mathcal{L}_n$ as a DURA, we also require that the data values of the most recent $a_i \cdots a_n$ be unique for every prefix of uv in each data word.

A DURA for $\mathcal{L}_n$ has $n + 3$ locations. A sequence of n a-transitions and $n + 1$ locations is needed for the prefix u. After the prefix, all registers are initialized to unique data values, such that register x_i stores the data value of a_i in u. The middle part v is modeled by the $(n + 1)$th location, which has self loops on a-transitions, updating registers with unique values. For the suffix w, we need two transitions from the $(n + 1)$th location, and two more locations: a set of b-transitions guarded by $d = x_i$ for $b_i(d)$ leads to an accepting location, and a second set of b-transitions with complementary guards leads to a rejecting location. Figure 6 shows a schematic for a DURA accepting $\mathcal{L}_n$. The minimal DOURA for $\mathcal{L}_n$, on the other hand, has more than $n!$ locations. Since a DOURA must store all data values in order of occurrence, after the sequence of locations for the prefix u, a control location is needed for every possible re-ordering of the variables caused by register



initializing registers
reading from registers
$\frac{a_1(p) \mid true}{x_1{:=}p}$
$\frac{a_n(p) \mid true}{x_n{:=}p}$
$\frac{b_1(p) \mid p{=}x_1}{-}$
$\frac{b_n(p) \mid p{\neq}x_n}{-}$
$\frac{a_1(p) \mid true}{x_1{:=}p}$
$\frac{a_n(p) \mid true}{x_n{:=}p}$
updating registers

Figure 6: DURA for $\mathcal{L}_n$.



$\frac{a_1(p) \mid true}{x_1{:=}p}$
$\frac{b_1(p) \mid p{=}x_2}{-}$
$\frac{b_2(p) \mid p{=}x_1}{-}$
$\frac{b_1(p) \mid p{\neq}x_2}{-}$
$\frac{b_2(p) \mid p{\neq}x_1}{-}$
$\frac{a_2(p) \mid true}{x_1{:=}x_2,\ x_2{:=}p}$
$\frac{a_1(p) \mid true}{x_1{:=}x_2,\ x_2{:=}p}$
$\frac{b_1(p) \mid p{\neq}x_1}{-}$
$\frac{b_2(p) \mid p{\neq}x_2}{-}$
$\frac{a_1(p) \mid true}{x_1{:=}p}$
$\frac{a_2(p) \mid true}{x_2{:=}p}$
$\frac{a_2(p) \mid true}{x_2{:=}p}$
$\frac{b_1(p) \mid p{=}x_1}{-}$
$\frac{b_2(p) \mid p{=}x_2}{-}$

Figure 7: DOURA for $\mathcal{L}_n$ with $n = 2$.

updates in v. Each such location has a different set of guards on the b-transitions, reflecting the order in which a_i have updated registers. Figure 7 shows a DOURA for $\mathcal{L}_n$ with $n = 2$. □

7. Conclusions and future work

We have presented a novel form of register automata, which also has an intuitive and succinct minimal canonical form that can be derived from a Nerode-like congruence. Key to this canonical form is the representation of a data language as a classification of a set of data words. We have shown that any set of data words (with some restrictions) can be correctly classified by a minimal subset of data words. We have also formulated a Nerode-like congruence that enables the construction of a succinct and canonical register automaton from such a minimal subset. Finally, we have compared our form of register automata to other proposed formalisms, showing that our register automata can be exponentially more succinct.

As a practical application, we have used the results in this paper to generalize Angluin-style active learning to data languages over infinite alphabets. The learning algorithm, which we describe in [13], can be used to characterize, e.g., protocols, services, and interfaces.

We have also further built on the theoretical concepts, in [9], generalizing our canonical model to more expressive signatures by allowing data values to be compared using binary relations. An interesting direction for future work would be to learn such models.

References

[1] R. Alur, D.L. Dill, A theory of timed automata, Theor. Comput. Sci. 126 (1994) 183–235.

[2] D. Angluin, Learning regular sets from queries and counterexamples, Inf. Comput. 75 (1987) 87–106.

[3] M. Benedikt, C. Ley, G. Puppis, What you must remember when processing data words, in: A.H.F. Laender, L.V.S. Lakshmanan (Eds.), AMW, volume 619 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2010.

[4] H. Björklund, T. Schwentick, On notions of regularity for data languages, Theor. Comput. Sci. 411 (2010) 702–715.

[5] M. Bojanczyk, C. David, A. Muscholl, T. Schwentick, L. Segoufin, Two-variable logic on data words, ACM Trans. Comput. Log. 12 (2011) 27.

[6] M. Bojanczyk, B. Klin, S. Lasota, Automata with group actions, in: LICS, IEEE Computer Society, 2011, pp. 355–364.

[7] P. Bouyer, A logical characterization of data languages, Inf. Process. Lett. 84 (2002) 75–85.

[8] S. Cassel, F. Howar, B. Jonsson, M. Merten, B. Steffen, A succinct canonical register automaton model, in: T. Bultan, P.A. Hsiung (Eds.), ATVA, volume 6996 of *LNCS*, Springer, 2011, pp. 366–380.

[9] S. Cassel, B. Jonsson, F. Howar, B. Steffen, A succinct canonical register automaton model for data domains with binary relations, in: S. Chakraborty, M. Mukund (Eds.), ATVA, volume 7561 of *LNCS*, Springer, 2012, pp. 57–71.

[10] N. Francez, M. Kaminski, An algebraic characterization of deterministic regular languages over infinite alphabets, Theor. Comput. Sci. 306 (2003) 155–175.

[11] E.M. Gold, Language identification in the limit, Information and Control 10 (1967) 447–474.

[12] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages and Computation, Addison-Wesley, 1979.

[13] F. Howar, B. Steffen, B. Jonsson, S. Cassel, Inferring canonical register automata, in: V. Kuncak, A. Rybalchenko (Eds.), VMCAI, volume 7148 of *LNCS*, Springer, 2012, pp. 251–266.

[14] M. Kaminski, N. Francez, Finite-memory automata, Theor. Comput. Sci. 134 (1994) 329–363.

[15] P.C. Kanellakis, S.A. Smolka, CCS expressions, finite state processes, and three problems of equivalence, Inf. Comput. 86 (1990) 43–68.

[16] R. Lazic, D. Nowak, A unifying approach to data-independence, in: C. Palamidessi (Ed.), CONCUR, volume 1877 of *LNCS*, Springer, 2000, pp. 581–595.

[17] A. Nerode, Linear automaton transformations, Proc. AMS 9 (1958) 541–544.

[18] R. Paige, R.E. Tarjan, Three partition refinement algorithms, SIAM J. Comput. 16 (1987) 973–989.

[19] A. Petrenko, S. Boroday, R. Groz, Confirming configurations in EFSM testing, IEEE Trans. Software Eng. 30 (2004) 29–42.

[20] R.L. Rivest, R.E. Schapire, Inference of finite automata using homing sequences, Inf. Comput. 103 (1993) 299–347.

[21] P. Saint-Andre, Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence, RFC 6121 (Proposed Standard), 2011.

[22] L. Segoufin, Automata and logics for words and trees over an infinite alphabet, in: Z. Ésik (Ed.), CSL, volume 4207 of *LNCS*, Springer, 2006, pp. 41–57.

[23] T. Wilke, Specifying timed state sequences in powerful decidable logics and timed automata, in: H. Langmaack, W.P. de Roever, J. Vytopil (Eds.), FTRTFT, volume 863 of *LNCS*, Springer, 1994, pp. 694–715.

A. Proof of Lemma 1

To prepare the proof of Lemma 1, let us introduce some notation. For a word w, let $w[i]$ denote the ith data value of w, and let $w \models (i = j)$ denote that $w[i] = w[j]$. For words w, w' with $\mathit{Acts}(w) = \mathit{Acts}(w')$ and $|w| = |w'| = n$, let $w[j \leftarrow w']$ denote the word obtained from w by replacing the data values $w[j], \ldots, w[n]$ at positions $j, \ldots, n$ by the corresponding data values $w'[j], \ldots, w'[n]$ of w'. We now restate the lemma:

Lemma 1 Let W be a finite $\sqsubseteq$-closed set of data words, let λ be a classification of λ, and let Ψ be the set of λ-essential words. If Φ is a λ-sufficient set of words, then $\Psi \subseteq \Phi$.

Proof. We prove by contradiction that any λ-essential word w must also be in Φ. So, assume that there is some λ-essential word which is not in Φ. Let $w \in W$ be a $<$-smallest data word that is λ-essential but not in Φ (this exists trivially, since W is finite). Let k be the flat-index of w. If $k = 1$, then w must be in Φ, since Φ is flattening-closed. If $k > 1$, then by the definition of λ-essential, $\lfloor w \rfloor_{k-1} \in \Psi_{<w}$, and there is a data word $x \in W$ such that w is a k-flattening of x, and a λ-essential word $v_0 \in \Psi_{<w}$, such that $v_0 \preceq_{\Psi_{<w}} x$ and $\lambda(v_0) \neq \lambda(x)$.

In the proof, the goal is to show the existence of a word $u \in \Phi_{<w}$ such that $u \preceq_{\Phi_{<w}} x$ and $\lambda(u) \neq \lambda(x)$. This would force $w \in \Phi$. To prove such a misclassification by $\Phi_{<w}$, we must exploit the fact that words that are $<$-smaller than w are classified in the same way by $\Phi_{<w}$ and $\Psi_{<w}$. Since x is not $<$-smaller than w, we will instead use the words that are $<$-smaller than w and 'similar to' x.

So, let Y be the set of $\sqsubseteq$-maximal words y with $y < w$ and $y \sqsubseteq x$. Intuitively, Y consists of the words y that are as similar to x as possible, but still satisfying $y < w$. Let us give a more precise construction of the words in Y. Since v is fresh at k, but x is not, we have $v[k] \neq x[k]$. A word in Y can be constructed from x by letting $y \simeq_k v$ and replacing some of the occurrences of $x[k]$ that occur after the kth position by $v[k]$. More precisely, y can be constructed from x by (i) incrementing all data values that are larger than whose first occurrence in x is after the kth position by 1, (ii) letting $y[k]$ be $v[k]$ (thus achieving $y \simeq_k v$), and (iii) replacing some occurrences of $x[k]$ that occur after the kth position by $v[k]$.

For $v \in \Psi$ and $u \in \Phi$, let $v \approx_Y u$ denote that $\lambda(v) = \lambda(u)$, and furthermore that there are $y_v, y_u \in Y$ with $v \preceq_{\Psi_{<w}} y_v$ and $u \preceq_{\Phi_{<w}} y_u$ such that $\lambda(y_v[j \leftarrow z]) = \lambda(y_u[j \leftarrow z])$ for all z with $\mathit{Acts}(z) = \mathit{Acts}(x)$.

Intuitively, the slight complication in the $v \approx_Y u$ stems from the fact that there may not be an $y \in Y$ with both $v \preceq_{\Psi_{<w}} y$ and $u \preceq_{\Phi_{<w}} y$ (i.e., the case where $y_v = y_u$). Instead, $v \approx_Y u$ gives a slightly weaker definition, under which $\Psi_{<w}$ and $\Phi_{<w}$ classify each continuation of prefixes of y_v and y_u in the same way.

Let now U be the set of words $u \in \Phi_{<w}$ with $u \sqsubseteq x$, such that $v \approx_Y u$ for some $v \in \Psi_{<w}$ with $v \preceq_{\Psi_{<w}} x$ and $\lambda(v) \neq \lambda(x)$. To see that U is not empty, recall that there is a $v_0 \in \Psi_{<w}$ with $v_0 \preceq_{\Psi_{<w}} x$ and $\lambda(v_0) \neq \lambda(x)$. If we choose $y_0 \in Y$ such that $v_0 \sqsubseteq y_0$ and let u_0 be defined by $u_0 \preceq_{\Phi_{<w}} y_0$, then $u_0 \in U$.

We claim that if u is a $<$-maximal element in U, then $u \preceq_{\Phi_{<w}} x$. If the claim is true, then the lemma follows from $v \approx_Y u$ (which implies $\lambda(v) = \lambda(u)$).

We prove the claim by contradiction. Let u be a $<$-maximal word in U. If the claim is wrong, there is a $z \in \Phi_{<w}$ with $u < z \sqsubseteq x$ and $z \notin U$. By the definition of $u < z$, there is a j such that $u \simeq_{j-1} z$, such that u is fresh at j and z is not. Let y_v and y_u be as in the definition of $v \approx_Y u$. Let ρ_v be the (unique) mapping from the data values of v to the data values of y_v which establishes that $v \sqsubseteq y_v$, and let ρ_u be the mapping from the data values of u to the data values of y_u which establishes that $u \sqsubseteq y_u$. We divide into cases.

1. If $z[j]$ is neither $x[k]$ nor $v[k]$, then $z[j]$ is equal to some previous data value(s) in z. Since $z \sqsubseteq x$, then the corresponding equalities between data values hold in x, and by construction also in all $y \in Y$. This contradicts $u \preceq_{\Phi_{<w}} y_u$, since u is then not a $<$-maximal word in $\Phi_{<w}$

with $u \sqsubseteq y_u$ (since $u < \lfloor z \rfloor_j \sqsubseteq y_u$).

2. If $z[j] = x[k]$ and $y_v[j] = x[k]$, then by $v \approx_Y u$ we have $\lambda(y_u[j \leftarrow y_v]) = \lambda(y_v)\lambda(v)$. Thus, if $u' \preceq_{\Phi_{<w}} y_u[j \leftarrow y_v]$, then $u' \in U$ and $u < u'$ (since $\lfloor z \rfloor_j \leq u'$), which contradicts the maximality of u.

3. If $z[j] = x[k]$ and $y_v[j] = v[k]$, let y'_v be obtained from y_v by replacing $v[k]$ by $x[k]$, and let y'_u is obtained from y_u by replacing $v[k]$ by $x[k]$. Let y''_u be obtained from y_u by replacing $u[k]$ by a fresh data value which is different from all other data values in y_u. If, on the one hand, Ψ does not contain any v' such that $v \simeq_{j-1} v'$ and $\rho_v(v'[j])$ is either $v[k]$ or $x[k]$, then we have $v \preceq_{\Psi_{<w}} y'_v$, and we can use y'_v as y_v in the preceding case 2. If, on the other hand, Ψ contains some v' such that $v \simeq_{j-1} v'$ and $\rho_v(v'[j])$ is either $v[k]$ or $x[k]$, then if $\rho_v(v[j]) = x[k]$, we get $y_v[j] = x[k]$ (from $v \preceq_{\Psi_{<w}} y_v$), and we are back to case 2. If $\rho_v(v[j]) = v[k]$, then we have the following cases.

   (a) Assume that Ψ does not contain any v' with $v \simeq_{j-1} v'$ and $\rho_v(v'[j]) = x[k]$. Since u is fresh at j, there is no $u' \in \Phi$ such that $u \simeq_{j-1} u'$ and $\rho_u(u'[j]) = v[k]$. Hence $u \preceq_{\Phi_{<w}} y''_u$, which gives $\lambda(y''_u) = \lambda(y_u)$. Since Ψ does not contain any v' such that $v \simeq_{j-1} v'$ and $\rho_v(v'[j]) = x[k]$, then $\lambda(y''_v[(j{+}1) \leftarrow y_u]) = \lambda(y'_v[(j{+}1) \leftarrow y_u])$. By $v \approx_Y u$, we get $\lambda(y''_u) = \lambda(y''_v[(j{+}1) \leftarrow y_u])$ and $\lambda(y'_v[(j{+}1) \leftarrow y_u]) = \lambda(y'_u)$. Putting all this together yields $\lambda(y_u) = \lambda(y''_u) = \lambda(y''_v[(j{+}1) \leftarrow y_u]) = \lambda(y'_v[(j{+}1) \leftarrow y_u]) = \lambda(y'_u)$, i.e., $\lambda(y_u) = \lambda(v)$. Thus, there must be some $u' \in U$ with $u' \preceq_{\Phi_{<w}} y'_u$ and $u < u'$ (since $\lfloor z \rfloor_j \leq u'$), which contradicts the maximality of u.

   (b) Assume that Ψ contains a v' such that $v \simeq_{j-1} v'$ and $\rho_v(v'[j]) = x[k]$. Let $\tilde{v}$ be the data word obtained from v by replacing all occurrences of $v[j]$ by $v'[j]$ (i.e., "making $v[j]$ and $v'[j]$ equal"). We have two cases.

      i. If $v[j]$ and $v'[j]$ are not the values $x[k]$ and $v[k]$, then $v \preceq_{\Psi_{<w}} \tilde{v}$, and also $v'' \preceq_{\Psi_{<w}} \tilde{v}$ for some v'' with $v'' \simeq_j v'$ and $\rho_v(v''[j]) = x[k]$. Thus, we get that $v'' \simeq_{j-1} v$.

      ii. If $v[j]$ and $v'[j]$ are the values $x[k]$ and $v[k]$, then $v \preceq_{\Phi_{<w}} \tilde{v}$, and also $v'' \preceq_{\Phi_{<w}} \tilde{v}$ for some v'' with $\rho_v(v''[j]) = x[k]$. By the assumption that $w \notin \Phi$, we have $\lambda(v'') = \lambda(v)$ and $v'' \simeq_{j-1} v$.

Since in both cases $\rho_v(v''[j]) = x[k]$, we can take v'' as v in the explanation of why $u \in U$, and we are back to case 2.

4. The cases where $z[j] = v[k]$ are symmetric to the cases where $z[j] = x[k]$ (swapping the roles of $x[k]$ and $v[k]$).

We have thus, by contradiction, established that any λ-interesting word must be in Φ, hence that $\Psi \subseteq \Phi$. $\square$