A Myhill-Nerode Theorem for Register Automata and Symbolic Trace Languages[⋆]

Frits Vaandrager and Abhisek Midya

Institute for Computing and Information Sciences
Radboud University
Nijmegen, the Netherlands
{F.Vaandrager, A.Midya}@cs.ru.nl

Abstract. We propose a new symbolic trace semantics for register automata (extended finite state machines) which records both the sequence of input symbols that occur during a run as well as the constraints on input parameters that are imposed by this run. Our main result is a generalization of the classical Myhill-Nerode theorem to this symbolic setting. Our generalization requires the use of three relations to capture the additional structure of register automata. Location equivalence $\equiv_l$ captures that symbolic traces end in the same location, transition equivalence $\equiv_t$ captures that they share the same final transition, and a partial equivalence relation $\equiv_r$ captures that symbolic values v and v' are stored in the same register after symbolic traces w and w', respectively. A symbolic language is defined to be regular if relations $\equiv_l$, $\equiv_t$ and $\equiv_r$ exist that satisfy certain conditions, in particular, they all have finite index. We show that the symbolic language associated to a register automaton is regular, and we construct, for each regular symbolic language, a register automaton that accepts this language. Our result provides a foundation for grey-box learning algorithms in settings where the constraints on data parameters can be extracted from code using e.g. tools for symbolic/concolic execution or tainting. We believe that moving to a grey-box setting is essential to overcome the scalability problems of state-of-the-art black-box learning algorithms.

1 Introduction

Model learning (a.k.a. active automata learning) is a black-box technique which constructs state machine models of software and hardware components from information obtained by providing inputs and observing the resulting outputs. Model learning has been successfully used in numerous applications, for instance for generating conformance test suites of software components [20], finding mistakes in implementations of security-critical protocols [13,15,14], learning interfaces of classes in software libraries [23], and checking that a legacy component

[⋆] Supported by NWO TOP project 612.001.852 Grey-box learning of Interfaces for Refactoring Legacy Software (GIRLS).

and a refactored implementation have the same behavior [35]. We refer to [38,26] for surveys and further references.

Myhill-Nerode theorems [32,21] are of pivotal importance for model learning algorithms. Angluin's classical L^* algorithm [3] for active learning of regular languages, as well as improvements such as [34,27,36], use an observation table to approximate the Nerode congruence. Maler and Steiger [30] established a Myhill-Nerode theorem for ω-languages that serves as a basis for a learning algorithm described in [4]. The SL^* algorithm for active learning of register automata of Cassel et al [11] is directly based on a generalization of the classical Myhill-Nerode theorem to a setting of data languages and register automata (extended finite state machines). Francez and Kaminski [16], Benedikt et al [5] and Bojańczyk et al [6] all present Myhill-Nerode theorems for data languages.

Despite the convincing applications of black-box model learning, it is fair to say that existing algorithms do not scale very well. In order to learn models of realistic applications in which inputs and outputs carry data parameters, state-of-the-art techniques either rely on manually constructed mappers that abstract the data parameters of inputs and outputs into a finite alphabet [2], or otherwise infer guards and assignments from black-box observations of test outputs [11,1]. The latter can be costly, especially for models where the control flow depends on data parameters in the input. Thus, for instance, the RALib tool [9], an implementation of the SL^* algorithm, needed more than two hundred thousand input/reset events to learn register automata with just 6 to 8 locations for TCP client implementations of Linux, FreeBSD and Windows [14]. Existing black-box model learning algorithms also face severe restrictions on the operations and predicates on data that are supported (typically, only equality/inequality predicates and constants).

A natural way to address these limitations is to augment learning algorithms with white-box information extraction methods, which are able to obtain information about the system under learning at lower cost than black-box techniques [25]. Constraints on data parameters can be extracted from the code using e.g. tools for symbolic execution [8], concolic execution [19], or tainting [22]. Several researchers have successfully explored this idea, see for instance [18,12,7,24]. Recently, we showed how constraints on data parameters can be extracted from Python programs using tainting, and used to boost the performance of RALib with almost two orders of magnitude. We were also able to learn models of systems that are completely out of reach of black-box techniques, such as "combination locks", systems that only exhibit certain behaviors after a very specific sequence of inputs [17]. Nevertheless, all these approaches are rather ad hoc, and what is missing is Myhill-Nerode theorem for this enriched settings that may serve as a foundation for grey-box model learning algorithms for a general class of register automata. In this article, we present such a theorem.

More specifically, we propose a new symbolic trace semantics for register automata which records both the sequence of input symbols that occur during a run as well as the constraints on input parameters that are imposed by this run. Our main result is a Myhill-Nerode theorem for symbolic trace languages.

Whereas the original Myhill-Nerode theorem refers to a single equivalence relation $\equiv$ on words, and constructs a DFA in which states are equivalence classes of $\equiv$, our generalization requires the use of three relations to capture the additional structure of register automata. Location equivalence $\equiv_l$ captures that symbolic traces end in the same location, transition equivalence $\equiv_t$ captures that they share the same final transition, and a partial equivalence relation $\equiv_r$ captures that symbolic values v and v' are stored in the same register after symbolic traces w and w', respectively. A symbolic language is defined to be regular if relations $\equiv_l$, $\equiv_t$ and $\equiv_r$ exist that satisfy certain conditions, in particular, they all have finite index. Whereas in the classical case of regular languages the Nerode equivalence $\equiv$ is uniquely determined, different relations relations $\equiv_l$, $\equiv_t$ and $\equiv_r$ may exist that satisfy the conditions for regularity for symbolic languages. We show that the symbolic language associated to a register automaton is regular, and we construct, for each regular symbolic language, a register automaton that accepts this language. In this automaton, the locations are equivalence classes of $\equiv_l$, the transitions are equivalence classes of $\equiv_t$, and the registers are equivalence classes of $\equiv_r$. In this way, we obtain a natural generalization of the classical Myhill-Nerode theorem for symbolic languages and register automata. Unlike Cassel et al [11], we need no restrictions on the allowed data predicates to prove our result, which drastically increases the range of potential applications. Our result paves the way for efficient grey-box learning algorithms in settings where the constraints on data parameters can be extracted from the code.

Due to the page limit, proofs have been omitted from this article, except for outlines of the proofs of main Theorems 2 and 3. All proofs can be found in the report version on arXiv [39].

2 Preliminaries

In this section, we fix some basic vocabulary for (partial) functions, languages, and logical formulas.

2.1 Functions

We write $f : X \rightharpoonup Y$ to denote that f is a partial function from set X to set Y. For $x \in X$, we write $f(x) \downarrow$ if there exists a $y \in Y$ such that $f(x) = y$, i.e., the result is defined, and $f(x) \uparrow$ if the result is undefined. We write $\mathit{domain}(f) = \{x \in X \mid f(x) \downarrow\}$ and $\mathit{range}(f) = \{f(x) \in Y \mid x \in \mathit{domain}(f)\}$. We often identify a partial function f with the set of pairs $\{(x, y) \in X \times Y \mid f(x) = y\}$. As usual, we write $f : X \to Y$ to denote that f is a total function from X to Y, that is, $f : X \rightharpoonup Y$ and $\mathit{domain}(f) = X$.

2.2 Languages

Let Σ be a set of *symbols*. A *word* $u = a_1 \ldots a_n$ over Σ is a finite sequence of symbols from Σ. The *length* of a word u, denoted $|u|$ is the number of symbols occurring in it. The empty word is denoted ϵ. We denote by Σ^* the set of all words over Σ, and by Σ^+ the set of all nonempty words over Σ (i.e. $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$). Given two words u and w, we denote by $u \cdot w$ the concatenation of u and w. When the context allows it, $u \cdot w$ shall be simply written uw. We say that u is a *prefix* of w iff there exists a word u' such that $u \cdot u' = w$. Similarly, u is a *suffix* of w iff there exists a word u' such that $u' \cdot u = w$. A *language* L over Σ is any set of words over Σ, so therefore a subset of Σ^*. We say that L is prefix closed if, for each $w \in L$ and each prefix u of w, $u \in L$ as well.

2.3 Guards

We postulate a countably infinite set $\mathcal{V} = \{v_1, v_2, \ldots\}$ of *variables*. In addition, there is also a variable $p \notin \mathcal{V}$ that will play a special role as formal parameter of input symbols; we write $\mathcal{V}^+ = \mathcal{V} \cup \{p\}$. Our framework is parametrized by a set R of relation symbols. Elements of R are assigned finite *arities*. A *guard* is a Boolean combination of relation symbols from R over variables. Formally, the set of *guards* is inductively defined as follows:

- If $r \in R$ is an n-ary relation symbol and $x_1, \ldots, x_n$ are variables from $\mathcal{V}^+$, then $r(x_1, \ldots, x_n)$ is a guard.
- If g is a guard then $\neg g$ is a guard.
- If g_1 and g_2 are guards then $g_1 \wedge g_2$ is a guard.

We use standard abbreviations from propositional logic such as $\top$ and $g_1 \vee g_2$. We write $Var(g)$ for the set of variables that occur in a guard g. We say that g is a guard *over* set of variables X if $Var(g) \subseteq X$. We write $\mathcal{G}(X)$ for the set of guards over X, and use symbol $\equiv$ to denote syntactic equality of guards.

We postulate a structure $\mathcal{R}$ consisting of a set $\mathcal{D}$ of *data values* and a distinguished n-ary relation $r^{\mathcal{R}} \subseteq \mathcal{D}^n$ for each n-ary relation symbol $r \in R$. In a trivial example of a structure $\mathcal{R}$, R consists of the binary symbol '$=$', $\mathcal{D}$ the set of natural numbers, and $=^{\mathcal{R}}$ is the equality predicate on numbers. An n-ary operation $f : \mathcal{D}^n \to \mathcal{D}$ can be modelled in our framework as an $n+1$-ary predicate. We may for instance extend structure $\mathcal{R}$ with a ternary predicate symbol $+$, where $(d_1, d_2, d_3) \in +^{\mathcal{R}}$ iff the sum of d_1 and d_2 equals d_3. Constants like 0 and 1 can be added to $\mathcal{R}$ as unary predicates.

A *valuation* is a partial function $\xi : \mathcal{V}^+ \rightharpoonup \mathcal{D}$ that assigns data values to variables. If $Var(g) \subseteq domain(\xi)$, then $\xi \models g$ is defined inductively by:

- $\xi \models r(x_1, \ldots, x_n)$ iff $(\xi(x_1), \ldots, \xi(x_n)) \in r^{\mathcal{R}}$
- $\xi \models \neg g$ iff not $\xi \models g$
- $\xi \models g_1 \wedge g_2$ iff $\xi \models g_1$ and $\xi \models g_2$

If $\xi \models g$ then we say valuation ξ *satisfies* guard g. We call g is *satisfiable*, and write $Sat(g)$, if there exists a valuation ξ such that $\xi \models g$. Guard g is a *tautology* if $\xi \models g$ for all valuations ξ with $Var(g) \subseteq domain(\xi)$.

A *variable renaming* is a partial function $\sigma : \mathcal{V}^+ \rightharpoonup \mathcal{V}^+$. If g is a guard with $Var(g) \subseteq domain(\sigma)$ then $g[\sigma]$ is the guard obtained by replacing each occurrence of a variable x in g by variable $\sigma(x)$. The following lemma is easily proved by induction.

Lemma 1. *$\xi \circ \sigma \models g$ iff $\xi \models g[\sigma]$*

3 Register Automata

In this section, we introduce register automata and show how they may be used as recognizers for both data languages and symbolic languages.

3.1 Definition and trace semantics

A register automaton comprises a set of locations with transitions between them, and a set of registers which can store data values that are received as inputs. Transitions contain guards over the registers and the current input, and may assign new values to registers.

Definition 1. *A* register automaton *is a tuple $\mathcal{A} = (\Sigma, Q, q_0, F, V, \Gamma)$, where*

- *Σ is a finite set of* input symbols*,*
- *Q is a finite set of* locations*, with $q_0 \in Q$ the* initial location*, and $F \subseteq Q$ a set of* accepting locations*,*
- *$V \subset \mathcal{V}$ is a finite set of* registers*, and*
- *Γ is a finite set of* transitions*, each of form $\langle q, \alpha, g, \varrho, q' \rangle$ where*
  - *$q, q' \in Q$ are the* source *and* target *locations, respectively; we require that $q' \in F \Rightarrow q \in F$,*
  - *$\alpha \in \Sigma$ is an input symbol,*
  - *$g \in \mathcal{G}(V \cup \{p\})$ is a guard, and*
  - *$\varrho : V \rightharpoonup V \cup \{p\}$ is an* assignment*; we require that ϱ is injective.*

Register automata are required to be completely specified *in the sense that for each location $q \in Q$ and each input symbol $\alpha \in \Sigma$, the disjunction of the guards on the α-transitions with source q is a tautology. Register automata are also required to be* deterministic *in the sense that for each location $q \in Q$ and input symbol $\alpha \in \Sigma$, the conjunction of the guards of any pair of distinct α-transitions with source q is not satisfiable. We write $q \xrightarrow{\alpha, g, \varrho} q'$ if $\langle q, \alpha, g, \varrho, q' \rangle \in \Gamma$.*

Example 1. Figure 1 shows a register automaton $\mathcal{A} = (\Sigma, Q, q_0, F, V, \Gamma)$ with a single input symbol a and four locations q_0, q_1, q_2 and q_3, with q_0, q_1, q_2 accepting and q_3 non-accepting. The initial location q_0 is marked by an arrow "start" and accepting locations are indicated by a double circle. There is just a single register x. Set Γ contains six transitions, which are indicated in the diagram. All transitions are labeled with input symbol a, a guard over formal parameter p and the registers, and an assignment. Guards represent conditions on data values. For example, the guard on the transition from q_1 to q_2, expresses that the data value of action a must be smaller than the data value currently stored in register x. We write $x := p$ to denote the assignment that stores the data parameter p in register x, that is, the function ϱ satisfying $\varrho(x) = p$. Trivial guards ($\top$) and assignments (empty domain) are omitted. Note that location q_3

is actually a sink location, i.e., there is no way to get into an accepting state from q_3. Thus the register automaton satisfies the condition that for each transition either the source location is accepting or the target location is not accepting. When drawing register automata, we often only depict the accepting locations, and leave a non-accepting sink location and the transitions leading to it implicit. Note that in locations q_1 and q_2, which have more than one outgoing transition, the disjunction of the guards of these transitions is equivalent to true, whereas the conjunction is equivalent to false.

The semantics of a register automaton is defined in terms of the set of *data words* that it accepts.

Definition 2. *Let Σ be a finite alphabet. A* data symbol *over Σ is a pair $\alpha(d)$ with $\alpha \in \Sigma$ and $d \in \mathcal{D}$. A* data word *over Σ is a finite sequence of data symbols, i.e., a word over $\Sigma \times \mathcal{D}$. A* data language *over Σ is a set of data words over Σ.*

We associate a data language to each register automata as follows.

Definition 3. *Let $\mathcal{A} = (\Sigma, Q, q_0, F, V, \Gamma)$ be a register automaton. A* configuration *of $\mathcal{A}$ is a pair (q, ξ), where $q \in Q$ and $\xi : V \rightharpoonup \mathcal{D}$. A* run of $\mathcal{A}$ over *a data word $w = \alpha_1(d_1) \cdots \alpha_n(d_n)$ is a sequence*

$$\gamma = (q_0, \xi_0) \xrightarrow{\alpha_1(d_1)} (q_1, \xi_1) \quad \ldots \quad (q_{n-1}, \xi_{n-1}) \xrightarrow{\alpha_n(d_n)} (q_n, \xi_n),$$

where, for $0 \leq i \leq n$, (q_i, ξ_i) is a configuration of $\mathcal{A}$, domain$(\xi_0) = \emptyset$, and for $0 < i \leq n$, Γ contains a transition $q_{i-1} \xrightarrow{\alpha_i, g_i, \varrho_i} q_i$ such that

- *$\iota_i \models g_i$, where $\iota_i = \xi_{i-1} \cup \{(p, d_i)\}$, and*
- *$\xi_i = \iota_i \circ \varrho_i$.*

We say that run γ is accepting *if $q_n \in F$ and* rejecting *if $q_n \notin F$. We call w the* trace *of γ, notation trace$(\gamma) = w$. Data word w is* accepted (rejected) *if $\mathcal{A}$ has an accepting (rejecting) run over w. The* data language *of $\mathcal{A}$, notation $L(\mathcal{A})$, is the set of all data words that are accepted by $\mathcal{A}$. Two register automata over the same alphabet Σ are trace equivalent if they accept the same data language.*

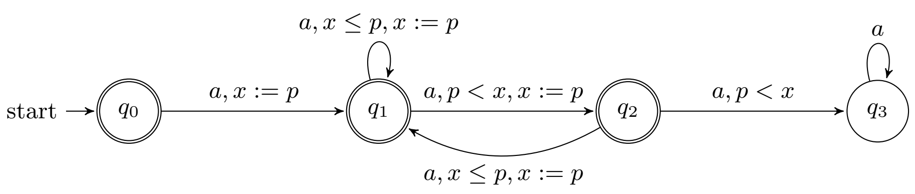

Fig. 1: Register automaton.

Example 2. Consider the register automaton of Figure 1. This automaton accepts the data word $a(1)\ a(4)\ a(0)\ a(7)$ since the following sequence of steps is a run (here ξ_0 is the trivial function with empty domain):

$$(q_0, \xi_0) \xrightarrow{a(1)} (q_1, x \mapsto 1) \xrightarrow{a(4)} (q_1, x \mapsto 4) \xrightarrow{a(0)} (q_2, x \mapsto 0) \xrightarrow{a(7)} (q_1, x \mapsto 7).$$

Note that the final location q_1 of this run is accepting. Upon receiving the first input $a(1)$, the automaton jumps to q_1 and stores data value 1 in the register x. Since 4 is bigger than 1, the automaton takes the self loop upon receiving the second input $a(4)$ and stores 4. Since 0 is less than 4, it moves to q_2 upon receipt of the third input $a(0)$ and updates x to 0. Finally, the automaton gets back to q_1 as 7 is bigger than 0.

Suppose that in the register automaton of Figure 1 we replace the guard on the transition from q_0 to q_1 by $x \leq p$. Since initial valuation ξ_0 does not assign a value to x, this means that it is not defined whether ξ_0 satisfies guard $x \leq p$. Automata in which such "runtime errors" do not occur are called *well-formed.*

Definition 4. *Let $\mathcal{A}$ be a register automaton. We say that a configuration (q, ξ) of $\mathcal{A}$ is* reachable *if there is a run of $\mathcal{A}$ that ends with (q, ξ). We call $\mathcal{A}$* well-formed *if, for each reachable configuration (q, ξ), ξ assigns a value to all variables from V that occur in guards of outgoing transitions of q, that is,*

$$(q, \xi) \ \textit{reachable} \ \wedge q \xrightarrow{\alpha, g \varrho} q' \Rightarrow \mathit{Var}(g) \subseteq \mathit{domain}(\xi) \cup \{p\}.$$

As soon as the set of data values and the collection of predicates becomes nontrivial, well-formedness of register automata becomes undecidable. However, it is easy to come up with a sufficient condition for well-formedness, based on a syntactic analysis of $\mathcal{A}$, which covers the cases that occur in practice. In the remainder of article, we will restrict our attention to well-formed register automata. In particular, the register automata that are constructed from regular symbolic trace languages in our Myhill-Nerode theorem will be well-formed.

Relation with automata of Cassel at al. Our definition of a register automaton is different from the one used in the SL^* algorithm of Cassel et al [11] and its implementation in RALib [9]. It is instructive to compare the two definitions.

1. In order to establish a Myhill-Nerode theorem, [11] requires that structure $\mathcal{R}$, which is a parameter of the SL^* algorithm, is *weakly extendible.* This technical restriction excludes many data types that are commonly used in practice. For instance, the set of integers with constants 0 and 1, an addition operator $+$, and a less-than predicate $<$ is not weakly extendable. For readers familiar with [11]: a structure (called theory in [11]) is weakly extendable if for all natural numbers k and data words u, there exists a u' with $u' \approx_{\mathcal{R}} u$ which is k-extendable. Intuitively, $u' \approx_{\mathcal{R}} u$ if data words u' and u have the same sequences of actions and cannot be distinguished by the relations in $\mathcal{R}$. Let $u = \alpha(0)\alpha(1)\alpha(2)\alpha(4)\alpha(8)\alpha(16)\alpha(11)$. Then there exists just one u' different

from u with $u' \approx_{\mathcal{R}} u$, namely $u' = \alpha(0)\alpha(1)\alpha(2)\alpha(4)\alpha(8)\alpha(16)\alpha(13)$. Now both u and u' are not even 1-extendable: if we extend u with $\alpha(3)$, we cannot find a matching extension $\alpha(d')$ of u' such that $u\alpha(3) \approx_{\mathcal{R}} u'\alpha(d')$, and if we extend u' with $\alpha(5)$ we cannot find a matching extension $\alpha(d)$ of u such that $u\alpha(d) \approx_{\mathcal{R}} u'\alpha(5)$. In the terminology of model theory [33], a structure is k-extendable if the Duplicator can win certain k-move Ehrenfeucht-Fraïssé games. For structures $\mathcal{R}$ that are homogeneous, one can always win these games, for all k. Thus, homogeneous structures are weakly extendible. An even stronger requirement, which is imposed in work of [31] on nominal automata, is that $\mathcal{R}$ is ω-categorical. In our approach, no restrictions on $\mathcal{R}$ are needed.

2. Unlike [11], we do not associate a fixed set of variables to each location. Our definition is slightly more general, which simplifies some technicalities.
3. However, we require assignments to be injective, a restriction that is not imposed by [11]. But note that the register automata that are actually constructed by SL^* are *right-invariant* [10]. In a right-invariant register automaton, two values can only be tested for equality if one of them is the current input symbol. Right-invariance, as defined in [10], implies that assignments are injective. As illustrated by the example of Figure 2, our register automata are exponentially more succinct than the right-invariant register automata constructed by SL^*. As pointed out in [10], right-invariant register automata in turn are more succinct than the automata of [16,5].
4. Our definition assumes that, for any transition from q to q', $q' \in F \Rightarrow q \in F$. Due to this assumption, which is not required in [11], the data language accepted by a register automaton is prefix closed. We need this property for technical reasons, but for models of reactive systems it is actually quite natural. RALib [9] also assumes that data languages are prefix closed.

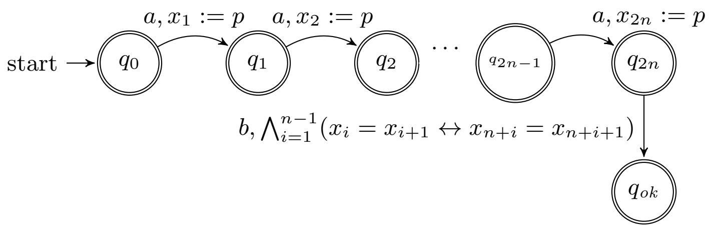

Fig. 2: For each $n > 0$, $\mathcal{A}_n$ is a register automaton that first accepts $2n$ input symbols a, storing all the data values that it receives, and then accepts input symbol b when two consecutive values in the first half of the input are equal iff the corresponding consecutive values in the second half of the input are equal. The number of locations and transitions of $\mathcal{A}_n$ grows linearly with n. There exist right-invariant register automata $\mathcal{B}_n$ that accept the same data languages, but their size grows exponentially with n.

Since register automata are deterministic, there exists a one-to-one correspondence between the accepted data words and the accepting runs. From every accepting run γ of a register automaton $\mathcal{A}$ we can trivially extract a data word $\mathit{trace}(\gamma)$ by forgetting all information except the data symbols. Conversely, for each data word w that is accepted by $\mathcal{A}$, there exists a corresponding accepting run γ, which is uniquely determined by the data word since from each configuration (q, ξ) and data symbol $\alpha(d)$, exactly one transition will be enabled.

Lemma 2. *Suppose γ and γ' are runs of a register automaton $\mathcal{A}$ such that $\mathit{trace}(\gamma) = \mathit{trace}(\gamma')$. Then $\gamma = \gamma'$.*

3.2 Symbolic semantics

We will now introduce an alternative trace semantics for register automata, which records both the sequence of input symbols that occur during a run as well as the constraints on input parameters that are imposed by this run. We will explore some basic properties of this semantics, and show that the equivalence induced by symbolic traces is finer than data equivalence.

A symbolic language consists of words in which input symbols and guards alternate.

Definition 5. *Let Σ be a finite alphabet. A* symbolic word *over Σ is a finite alternating sequence $w = \alpha_1 G_1 \cdots \alpha_n G_n$ of input symbols from Σ and guards. A* symbolic language *over Σ is a set of symbolic words over Σ.*

A symbolic run is just a run, except that the valuations do not return concrete data values, but markers (variables) that record the exact place in the run where the input occurred. We use variable v_i as a marker for the i-th input value. Using these symbolic valuations (variable renamings, actually) it is straightforward to compute the constraints on the input parameters from the guards occurring in the run.

Definition 6. *Let $\mathcal{A} = (\Sigma, Q, q_0, F, V, \Gamma)$ be a register automaton. A* symbolic run *of $\mathcal{A}$ is a sequence*

$$\delta \;=\; (q_0, \zeta_0) \xrightarrow{\alpha_1, g_1, \varrho_1} (q_1, \zeta_1) \;\ldots\; \xrightarrow{\alpha_n, g_n, \varrho_n} (q_n, \zeta_n),$$

where ζ_0 is the trivial variable renaming with empty domain and, for $0 < i \leq n$,

- *$q_{i-1} \xrightarrow{\alpha_i, g_i, \varrho_i} q_i$ is a transition in Γ,*
- *ζ_i is a variable renaming with $\mathit{domain}(\zeta_i) \subseteq V$, and*
- *$\zeta_i = \iota_i \circ \varrho_i$, where $\iota_i = \zeta_{i-1} \cup \{(p, v_i)\}$.*

We also require that $G_1 \wedge \cdots \wedge G_n$ is satisfiable, where $G_i \equiv g_i[\iota_i]$, for $0 < i \leq n$.

We say that symbolic run δ is accepting *if $q_n \in F$ and* rejecting *if $q_n \notin F$. The* symbolic trace *of δ is the symbolic word $\mathit{strace}(\delta) = \alpha_1\ G_1 \cdots \alpha_n\ G_n$. Symbolic word w is* accepted (rejected) *if $\mathcal{A}$ has an accepting (rejecting) symbolic run δ with $\mathit{strace}(\delta) = w$. The symbolic language of $\mathcal{A}$, notation $L_s(\mathcal{A})$, is the set of all symbolic words accepted by $\mathcal{A}$. Two register automata over the same alphabet Σ are symbolic trace equivalent if they accept the same symbolic language.*

Example 3. Consider the register automaton of Figure 1. The following sequence constitutes a symbolic run:

$$(q_0, \zeta_0) \xrightarrow{a, \top, x:=p} (q_1, x \mapsto v_1) \xrightarrow{a, x \leq p, x:=p} (q_1, x \mapsto v_2)$$

$$\xrightarrow{a, p<x, x:=p} (q_2, x \mapsto v_3) \xrightarrow{a, x \leq p, x:=p} (q_1, x \mapsto v_4).$$

Since

$$\begin{aligned}
\iota_1 &= \{(p, v_1)\} \\
\iota_2 &= \{(x, v_1), (p, v_2)\} \\
\iota_3 &= \{(x, v_2), (p, v_3)\} \\
\iota_4 &= \{(x, v_3), (p, v_4)\}
\end{aligned}$$

and the final location q_1 of this symbolic run is accepting, the register automaton accepts the symbolic word $w = a \top a\ v_1 \leq v_2\ a\ v_3 < v_2\ a\ v_3 \leq v_4$. Note that the guard of w is satisfiable, for instance by valuation ξ with $\xi(v_1) = 1$, $\xi(v_2) = 4$, $\xi(v_3) = 0$ and $\xi(v_4) = 7$.

The two technical lemmas below state some basic properties about variable renamings in a symbolic run. The proofs are straightforward, by induction.

Lemma 3. *Let δ be a symbolic run of $\mathcal{A}$, as in Definition 6. Then range$(\zeta_i) \subseteq \{v_1, \ldots, v_i\}$, for $i \in \{0, \ldots, n\}$, and range$(\iota_i) \subseteq \{v_1, \ldots, v_i\}$, for $i \in \{1, \ldots, n\}$.*

As a consequence of our assumption that assignments in a register automaton are injective, all the variable renamings in a symbolic run are injective as well.

Lemma 4. *Let δ be a symbolic run of $\mathcal{A}$, as in Definition 6. Then, for each $i \in \{0, \ldots, n\}$, ζ_i is injective, and for each $i \in \{1, \ldots, n\}$, ι_i is injective.*

All symbolic words accepted by a register automaton satisfy some basic sanity properties: guards may only refer to the markers for values received thus far, and the conjunction of all the guards is satisfiable. We call symbolic words that satisfy these properties *feasible*. Note that if a symbolic word is feasible, any prefix is feasible as well.

Definition 7 (Feasible). *Let $w = \alpha_1 G_1 \cdots \alpha_n G_n$ be a symbolic word. We write length$(w) = n$ and guard$(w) = G_1 \wedge \cdots \wedge G_n$. Word w is* feasible *if guard(w) is satisfiable and Var$(G_i) \subseteq \{v_1, \ldots, v_i\}$, for each $i \in \{1, \ldots, n\}$. A symbolic language is* feasible *if it is prefix closed and consists of feasible symbolic words.*

Lemma 5. *$L_s(\mathcal{A})$ is feasible.*

Since register automata are deterministic, each symbolic trace of $\mathcal{A}$ corresponds to a unique symbolic run of $\mathcal{A}$.

Lemma 6. *Suppose δ and δ' are symbolic runs of a register automaton $\mathcal{A}$ such that strace(δ) = strace(δ'). Then $\delta = \delta'$.*

Definition 8. *Let $\mathcal{A}$ be a register automaton and $w \in L_s(\mathcal{A})$. Then we write $symb(w)$ for the unique symbolic run δ of $\mathcal{A}$ with $strace(\delta) = w$.*

There exists a one-to-one correspondence between runs of $\mathcal{A}$ and pairs consisting of a symbolic run of $\mathcal{A}$ and a satisfying assignments for the guards from its symbolic trace.

Lemma 7. *Let δ be a symbolic run of $\mathcal{A}$, as in Definition 6, and $\xi : \{v_1, \ldots, v_n\} \to \mathcal{D}$ a valuation such that $\xi \models G_1 \wedge \cdots \wedge G_n$. Let $run_{\mathcal{A}}(\delta, \xi)$ be the sequence obtained from δ by (a) replacing each input α_i by data symbol $\alpha_i(\xi(v_i))$ (for $0 < i \leq n$), (b) removing guards g_i and assignments ϱ_i, and (c) replacing valuations ζ_i by $\xi_i = \xi \circ \zeta_i$ (for $0 \leq i \leq n$). Then $run_{\mathcal{A}}(\delta, \xi)$ is a run of $\mathcal{A}$.*

Lemma 8. *Let γ be a run of register automaton $\mathcal{A}$. Then there exist a valuation ξ and symbolic run δ such that $run_{\mathcal{A}}(\delta, \xi) = \gamma$.*

Using the above lemmas, we can prove that whenever two register automata accept the same symbolic language, they also accept the same data language.

Theorem 1. *Suppose $\mathcal{A}$ and $\mathcal{B}$ are register automata with $L_s(\mathcal{A}) = L_s(\mathcal{B})$. Then $L(\mathcal{A}) = L(\mathcal{B})$.*

Example 4. The converse of Theorem 1 does not hold. Figure 3 gives a trivial example of two register automata with the same data language but a different symbolic language.

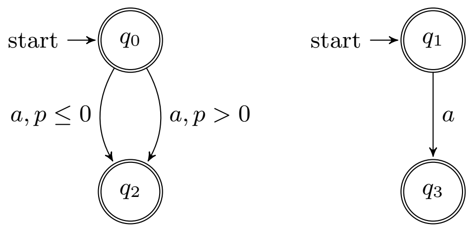

Fig. 3: Trace equivalent but not symbolic trace equivalent.

Lemma 7 allows us to rephrase the well-formedness condition of register automata in terms of symbolic runs.

Corollary 1. *Register automaton $\mathcal{A}$ is well-formed iff, for each symbolic run δ that ends with (q, ζ), $q \xrightarrow{\alpha, g\varrho} q' \Rightarrow Var(g) \subseteq domain(\zeta) \cup \{p\}$.*

4 Nerode Equivalence

The Myhill-Nerode equivalence [32,21] deems two words w and w' of a language L equivalent if there does not exist a suffix u that distinguishes them, that is, only one of the words $w \cdot u$ and $w' \cdot u$ is in L. The Myhill-Nerode theorem states that L is regular if and only if this equivalence relation has a finite index, and moreover that the number of states in the smallest deterministic finite automaton (DFA) recognizing L is equal to the number of equivalence classes. In this section, we present a Myhill-Nerode theorem for symbolic languages and register automata. We need three relations $\equiv_l$, $\equiv_t$ and $\equiv_r$ on symbolic words to capture the structure of register automata. Intuitively, two symbolic words w and w' are *location equivalent*, notation $w \equiv_l w'$, if they lead to the same location, *transition equivalent*, notation $w \equiv_t w'$, if they share the same final transition, and marker v of w, and marker v' of w' are *register equivalent*, notation $(w, v) \equiv_r (w', v')$, when they are stored in the same register after occurrence of words w and w'. Whereas $\equiv_l$ and $\equiv_t$ are equivalence relations, $\equiv_t$ is a partial equivalence relation (PER), that is, a relation that is symmetric and transitive. Relation $\equiv_r$ is not necessarily reflexive, as $(w, v) \equiv_r (w, v)$ only holds when marker v is stored after symbolic trace w. Since a register automaton has finitely many locations, finitely many transitions, and finitely many registers, the equivalences $\equiv_l$ and $\equiv_t$, and the equivalence induced by $\equiv_r$, are all required to have finite index.

Definition 9. *A feasible symbolic language L over Σ is* regular *iff there exist three relations:*

- *an equivalence relation $\equiv_l$ on L, called* location equivalence*,*
- *an equivalence relation $\equiv_t$ on $L \setminus \{\epsilon\}$, called* transition equivalence*, and*
- *a partial equivalence relation $\equiv_r$ on $\{(w, v_i) \in L \times \mathcal{V} \mid i \leq \mathit{length}(w)\}$, called* register equivalence*. We say that w* stores *v if $(w, v) \equiv_r (w, v)$.*

We require that equivalences $\equiv_l$ and $\equiv_t$, as well as the equivalence relation obtained by restricting $\equiv_t$ to $\{(w, v) \in L \times \mathcal{V} \mid w \text{ stores } v\}$ have finite index. We also require that relations $\equiv_l$, $\equiv_t$ and $\equiv_r$ satisfy the conditions of Table 1, for $w, w', u, u' \in L$, $\mathit{length}(w) = m$, $\mathit{length}(w') = n$, $\alpha, \alpha' \in \Sigma$, G, G' guards, $v, v' \in \mathcal{V}$, and $\sigma : \mathcal{V} \rightharpoonup \mathcal{V}$. Condition 1 implies that, given w, w' and v, there is at most one v' s.t. $(w, v) \equiv_r (w', v')$. Therefore, we may define $\mathit{matching}(w, w')$ as the variable renaming σ satisfying:

$$\sigma(v) = \begin{cases} v' & \text{if } (w, v) \equiv_r (w', v') \\ v_{n+1} & \text{if } v = v_{m+1} \\ \text{undefined otherwise} & \end{cases}$$

Intuitively, the first condition captures that a register can store at most a single value at a time. When $w\alpha G$ and $w'\alpha' G'$ share the same final transition, then in particular w and w' share the same final location (Condition 2), input symbols α and α' are equal (Condition 3), G' is just a renaming of G (Condition 4),

$$
\begin{aligned}
&(w,v) \equiv_r (w,v') \Rightarrow v = v' && (1)\\
&w\alpha G \equiv_t w'\alpha' G' \;\;\Rightarrow\;\; w \equiv_l w' && (2)\\
&w\alpha G \equiv_t w'\alpha' G' \;\;\Rightarrow\;\; \alpha = \alpha' && (3)\\
&w\alpha G \equiv_t w'\alpha G' \wedge \sigma = \mathit{matching}(w,w') \;\;\Rightarrow\;\; G[\sigma] \equiv G' && (4)\\
&w \equiv_t w' \Rightarrow w \equiv_l w' && (5)\\
&w \equiv_t w' \wedge w \text{ stores } v_m \Rightarrow (w,v_m) \equiv_r (w',v_n) && (6)\\
&u \equiv_t u' \wedge u = w\alpha G \wedge u' = w'\alpha G' \wedge (w,v) \equiv_r (w',v') \wedge u \text{ stores } v\\
&\qquad \Rightarrow (u,v) \equiv_r (u',v') && (7)\\
&u \equiv_t u' \wedge u = w\alpha G \wedge u' = w'\alpha G' \wedge (u,v) \equiv_r (u',v') \wedge v \neq v_{m+1}\\
&\qquad \Rightarrow (w,v) \equiv_r (w',v') && (8)\\
&w \equiv_l w' \wedge w\alpha G \in L \wedge v \in \mathit{Var}(G) \setminus \{v_{m+1}\} \Rightarrow \exists v' : (w,v) \equiv_r (w',v') && (9)\\
&w \equiv_l w' \wedge w\alpha G \in L \wedge \sigma = \mathit{matching}(w,w')\\
&\qquad \wedge\; \mathit{Sat}(\mathit{guard}(w') \wedge G[\sigma]) \Rightarrow w'\alpha G[\sigma] \in L && (10)\\
&w \equiv_l w' \wedge w\alpha G \in L \wedge w'\alpha G' \in L \wedge \sigma = \mathit{matching}(w,w')\\
&\qquad \wedge\; \mathit{Sat}(G[\sigma] \wedge G') \Rightarrow w\alpha G \equiv_t w'\alpha G' && (11)
\end{aligned}
$$

Table 1: Conditions for regularity of symbolic languages.

and $w\alpha G$ and $w'\alpha' G'$ share the same final location (Condition 5) and final assignment (Conditions 6, 7 and 8). Condition 6 says that the parameters of the final input end up in the same register when they are stored. Condition 7 says that when two values are stored in the same register, they will stay in the same register for the rest of their life (this condition can be viewed as a right invariance condition for registers). Conversely, if two values are stored in the same register after a transition, and they do not correspond to the final input, they were already stored in the same register before the transition (Condition 8). Condition 9 captures the well-formedness assumption for register automata. As a consequence of Condition 9, $G[\sigma]$ is defined in Conditions 4, 10 and 11, since $\mathit{Var}(G) \subseteq \mathit{domain}(\sigma)$. Condition 10 is the equivalent for symbolic languages of the well-known right invariance condition for regular languages. For symbolic languages a right invariance condition

$$w \equiv_l w' \wedge w\alpha G \in L \wedge \sigma = \mathit{matching}(w,w') \Rightarrow w'\alpha G[\sigma] \in L$$

would be too strong: even though w and w' lead to the same location, the values stored in the registers may be different, and therefore they will not necessarily enable the same transitions. However, when in addition $\mathit{guard}(w') \wedge G[\sigma]$ is satisfiable, we may conclude that $w''\alpha G[\sigma] \in L$. Condition 11, finally, asserts that L only allows deterministic behavior.

The simple lemma below asserts that, due to the determinism imposed by Condition 11, the converse of Conditions 2, 3 and 4 also holds. This means that

$\equiv_t$ can be expressed in terms of $\equiv_l$ and $\equiv_r$, that is, once we have fixed $\equiv_l$ and $\equiv_r$, relation $\equiv_t$ is fully determined.

Lemma 9. *Suppose symbolic language L over Σ is regular, and equivalences $\equiv_l$, $\equiv_t$ and $\equiv_r$ satisfy the conditions of Definition 9. Then*

$$w \equiv_l w' \wedge w\alpha G \in L \wedge w'\alpha G' \in L \wedge \sigma = \mathit{matching}(w, w') \wedge G' \equiv G[\sigma] \\ \Rightarrow w\alpha G \equiv_t w'\alpha G'.$$

We can now state our "symbolic" version of the celebrated result of Myhill & Nerode. The symbolic language of any register automaton is regular (Theorem 2), and any regular symbolic language can be obtained as the symbolic language of some register automaton (Theorem 3).

Theorem 2. *Suppose $\mathcal{A}$ is a register automaton. Then $L_s(\mathcal{A})$ is regular.*

Proof. (outline) Let $L = L_s(\mathcal{A})$. Then, by Lemma 5, L is feasible. Define equivalences $\equiv_l$, $\equiv_t$ and $\equiv_r$ as follows:

- For $w, w' \in L$, $w \equiv_l w'$ iff $\mathit{symb}(w)$ and $\mathit{symb}(w')$ share the same final location.
- For $w, w' \in L \setminus \{\epsilon\}$, $w \equiv_t w'$ iff $\mathit{symb}(w)$ and $\mathit{symb}(w')$ share the same final transition.
- For $w, w' \in L$ and $v, v' \in \mathcal{V}$, $(w, v) \equiv_r (w', v')$ iff there is a register $x \in V$ such that the final valuations ζ of $\mathit{symb}(w)$ stores v in x, and the final valuation ζ' of $\mathit{symb}(w')$ stores v' in x, that is, $\zeta(x) = v$ and $\zeta'(x) = v'$.
  (Note that, by Lemma 3, $\mathit{range}(\zeta) \subseteq \{v_1, \ldots, v_m\}$, for $m = \mathit{length}(w)$, and $\mathit{range}(\zeta') \subseteq \{v_1, \ldots, v_n\}$, for $n = \mathit{length}(w')$.)

Then $\equiv_l$ has finite index since $\mathcal{A}$ has a finite number of locations, $\equiv_t$ has finite index since $\mathcal{A}$ has a finite number of transitions, and the equivalence induced by $\equiv_r$ has finite index since $\mathcal{A}$ has a finite number of registers. We refer to the full version of this paper for a proof that, with this definition of $\equiv_l$, $\equiv_t$ and $\equiv_r$, all 11 conditions of Table 1 hold.

The following example shows that in general there is no coarsest location equivalence that satisfies all conditions of Table 1. So whereas for regular languages a unique Nerode congruence exists, this is not always true for symbolic languages.

Example 5. Consider the symbolic language L that consists of the following three symbolic words and their prefixes:

$$\begin{aligned} w &= a\ v_1 > 0\ a\ v_1 > 0\ b\ \top \\ u &= a\ v_1 = 0\ a\ v_1 = 0\ b\ \top \\ z &= a\ v_1 < 0\ c\ v_1 + v_2 = 0\ a\ v_2 > 0\ c\ \top \end{aligned}$$

Symbolic language L is accepted by both automata displayed in Figure 4. Thus, by Theorem 2, L is regular. Let w_i, u_i and z_i denote the prefixes of w, u and

z, respectively, of length i. Then, according to the location equivalence induced by the first automaton, $w_1 \equiv_l u_1$, and according to the location equivalence induced by the second automaton, $u_1 \equiv_l z_2$. Therefore, if a coarsest location equivalence relation would exist, $w_1 \equiv_l z_2$ should hold. Then, by Condition 9, $(w_1, v_1) \equiv_r (z_2, v_2)$. Thus, by Lemma 9, $w_2 \equiv_t z_3$, and therefore, by Condition 5, $w_2 \equiv_l z_3$. But now Condition 10 implies $a\ v_1 > 0\ a\ v_1 > 0\ c\ \top \in L$, which is a contradiction.

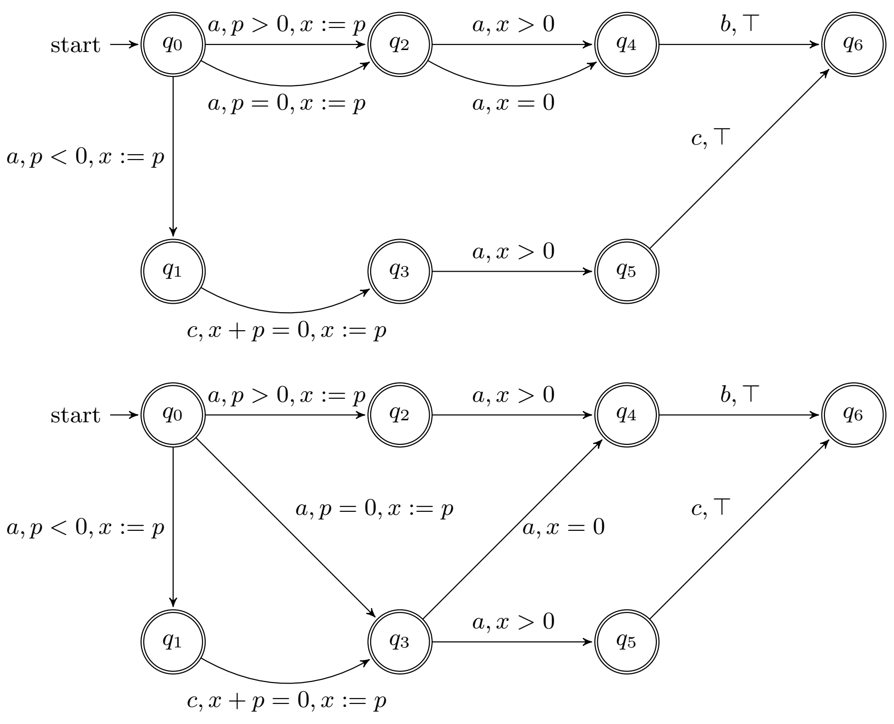

Fig. 4: There is no unique, coarsest location equivalence.

Theorem 3. *Suppose L is a regular symbolic language over Σ. Then there exists a register automaton $\mathcal{A}$ such that $L = L_s(\mathcal{A})$.*

Proof. (Outline) Since any register automaton without accepting locations accepts the empty symbolic language, we may assume without loss of generality that L is nonempty. Let $\equiv_l, \equiv_t, \equiv_r$ be relations satisfying the properties stated in Definition 9. We define register automaton $\mathcal{A} = (\Sigma, Q, q_0, F, V, \Gamma)$ as follows:

– $Q = \{[w]_l \mid w \in L\} \cup \{q_{sink}\}$, where q_{sink} is a special *sink location*.
  (Since L is regular, $\equiv_l$ has finite index, and so Q is finite, as required.)

- $q_0 = [\epsilon]_l$.
  (Since L is regular, it is feasible, and thus prefix closed. Therefore, since we assume that L is nonempty, $\epsilon \in L$.)
- $F = \{[w]_l \mid w \in L\}$.
- $V = \{[(w, v)]_r \mid w \in L \wedge v \in \mathcal{V} \wedge w \text{ stores } v\}$.
  (Since L is regular, the equivalence induced by $\equiv_r$ has finite index, and so V is finite, as required. Note that registers are supposed to be elements of $\mathcal{V}$, and equivalence classes of $\equiv_r$ are not. Thus, strictly speaking, we should associate a unique register of $\mathcal{V}$ to each equivalence class of $\equiv_r$, and define V in terms of those registers.)
- Γ contains a transition $\langle q, \alpha, g, \varrho, q'\rangle$ for each equivalence class $[w\alpha G]_t$, where
  - $q = [w]_l$
    (Condition 2 ensures that the definition of q is independent from the choice of representative $w\alpha G$.)
  - (Condition 3 ensures that input symbol α is independent from the choice of representative $w\alpha G$.)
  - $g \equiv G[\tau]$ where τ is a variable renaming that satisfies, for $v \in \mathit{Var}(G)$,

$$\tau(v) = \begin{cases} [(w, v)]_r & \text{if } w \text{ stores } v \\ p & \text{if } v = v_{m+1} \wedge m = \mathit{length}(w) \end{cases}$$

    (By Condition 9, w stores v, for any $v \in \mathit{Var}(G) \setminus \{v_{m+1}\}$, so $G[\tau]$ is well-defined. Condition 4 ensures that the definition of g is independent from the choice of representative $w\alpha G$.) Also note that, by Condition 1, τ is injective.)
  - ϱ is defined for each equivalence class $[(w'\alpha G', v')]_r$ with $w'\alpha G' \equiv_t w\alpha G$ and $w'\alpha G'$ stores v'. Let $n = \mathit{length}(w')$. Then

$$\varrho([(w'\alpha G', v')]_r) = \begin{cases} [(w', v')]_r & \text{if } w' \text{ stores } v' \\ p & \text{if } v' = v_{n+1} \end{cases}$$

    (By Condition 8, either $v' = v_{n+1}$ or w' stores v', so $\varrho([(w'\alpha G', v')]_r)$ is well-defined. Also by Condition 8, the definition of ϱ does not depend on the choice of representative $w'\alpha G'$. By Conditions 6 and 7, assignment ϱ is injective.)
  - $q' = [w\alpha G]_l$
    (Condition 5 ensures that the definition of q' is independent from the choice of representative $w\alpha G$.)

  In order to ensure that $\mathcal{A}$ is completely specified, we add transitions to the sink location q_{sink}. More specifically, if $q \in Q$ is a location with outgoing α-transitions with guards $g_1, \ldots, g_m$, then we add a transition $\langle q, \alpha, \neg(g_1 \vee \cdots \vee g_m), \varrho_0, q_{sink}\rangle$ to Γ, for ϱ_0 the trivial assignment with empty domain. Finally, we add, for each $\alpha \in \Sigma$, a self loop $\langle q_{sink}, \alpha, \top, \varrho_0, q_{sink}\rangle$ to Γ. Since L is regular, $\equiv_t$ has finite index and therefore Γ is finite, as required.

We claim that $\mathcal{A}$ is deterministic and prove this by contradiction. Suppose $\langle q, \alpha, g', \varrho', q'\rangle$ and $\langle q, \alpha, g'', \varrho'', q''\rangle$ are two distinct α-transitions in Γ with $g' \wedge g''$

satisfiable. Then there exists a valuation ξ such that $\xi \models g' \wedge g''$. Note that $q \neq q_{sink}$, $q' \neq q_{sink}$ and $q'' \neq q_{sink}$. Let the two transitions correspond to (distinct) equivalence classes $[w'\alpha G']_t$ and $[w''\alpha G'']_t$, respectively. Then $g' = G'[\tau']$ and $g'' = G''[\tau'']$, with τ' and τ'' defined as above. Now observe that $G'[\tau'] \equiv G'[\sigma][\tau'']$, for $\sigma = \mathit{matching}(w', w'')$. Using Lemma 4, we derive

$$\xi \models g' \wedge g'' \Leftrightarrow \xi \models G'[\sigma][\tau''] \wedge G''[\tau''] \Leftrightarrow \xi \models (G'[\sigma] \wedge G'')[\tau''] \Leftrightarrow \xi \circ \tau'' \models G'[\sigma] \wedge G''.$$

Thus $G'[\sigma] \wedge G''$ is satisfiable and we may apply Condition 11 to conclude $w'\alpha G' \equiv_t w''\alpha G''$. Contradiction.

So using the assumption that L is regular, we established that $\mathcal{A}$ is a register automaton. Note that for this we essentially use that equivalences $\equiv_l$, $\equiv_t$ and $\equiv_r$ have finite index, as well as all the conditions, except Condition 10. We claim $L = L_s(\mathcal{A})$.

5 Concluding Remarks

We have shown that register automata can be defined in a natural way *directly* from a regular symbolic language, with locations materializing as equivalence classes of a relation $\equiv_l$, transitions as equivalence classes of a relation $\equiv_t$, and registers as equivalences classes of a relation $\equiv_r$.

It is instructive to compare our definition of regularity for symbolic languages with Nerode's original definition for non-symbolic languages. Nerode defined his equivalence for all words $u, v \in \Sigma^*$ (not just those in L!) as follows:

$$u \equiv_l v \Leftrightarrow (\forall w \in \Sigma^* : uw \in L \Leftrightarrow vw \in L).$$

For any language $L \subseteq \Sigma^*$, the equivalence relation $\equiv_l$ is uniquely determined and can be used (assuming it has finite index) to define a unique minimal finite automaton that accepts L. As shown by Example 5, the equivalence $\equiv_l$ and its corresponding register automaton are not uniquely defined in a setting of symbolic languages. For such a setting, it makes sense to consider a symbolic variant of what Kozen [29] calls *Myhill-Nerode relations*. These are relations that satisfy the following three conditions, for $u, v \in \Sigma^*$ and $\alpha \in \Sigma$,

$$u \equiv_l v \;\Rightarrow\; (u \in L \Leftrightarrow v \in L) \tag{12}$$
$$u \equiv_l v \;\Rightarrow\; u\alpha \equiv_l v\alpha \tag{13}$$
$$\equiv_l \text{ has finite index} \tag{14}$$

Note that Conditions 12 and 13 are consequences of Nerode's definition. Condition 13 is the well-known right invariance property, which is sound for non-symbolic languages, since finite automata are completely specified and every state has an outgoing α-transition for every α. A corresponding condition

$$u \equiv_l v \Rightarrow u\alpha G \equiv_l v\alpha G$$

for symbolic languages would not be sound, however, since locations in a register automaton do not have outgoing transitions for every possible symbol α and every possible guard G. We see basically two routes to fix this problem. The first route is to turn $\equiv_l$ into a partial equivalence relation that is only defined for symbolic words that correspond to runs of the register automaton. Right invariance can then be stated as

$$\begin{aligned} & w \equiv_l w' \wedge w\alpha G \equiv_l w\alpha G \wedge \sigma = \mathit{matching}(w, w') \wedge w'\alpha G[\sigma] \equiv_l w'\alpha G[\sigma] \\ & \quad \Rightarrow w\alpha G \equiv_l w'\alpha G[\sigma]. \end{aligned} \tag{15}$$

The second route is to define $\equiv_l$ as an equivalence on L and restrict attention to prefix closed symbolic languages. This allows us to drop Condition 12 and leads to the version of right invariance that we stated as Condition 10. Since prefix closure is a natural restriction that holds for all the application scenarios we can think of, and since equivalences are conceptually simpler than PERs, we decided to explore the second route in this article. However, we conjecture that the restriction to prefix closedness is not essential, and Myhill-Nerode characterization for symbolic trace languages without this restriction can be obtained using Condition 15.

An obvious research challenge is to develop a learning algorithm for symbolic languages based on our Myhill-Nerode theorem. Since for symbolic languages there is no unique, coarsest Nerode congruence that can be approximated, as in Angluin's algorithm [3], this is a nontrivial task. We hope that for register automata with a small number of registers, an active algorithm can be obtained by encoding symbolic traces and register automata as logical formulas, and using SMT solvers to generate hypothesis models, as in [37].

As soon as a learning algorithm for symbolic traces has been implemented, it will be possible to connect the implementation with the setup of [17], which extracts symbolic traces from Python programs using an existing tainting library for Python. We can then compare its performance with the grey-box version of the RALib tool [17] on a number of benchmarks, which include data structures from Python's standard library. An area where learning algorithms for symbolic traces potentially can have major impact is the inference of behavior interfaces of legacy control software. As pointed out in [28], such interfaces allow components to be developed, analyzed, deployed and maintained in isolation. This is achieved using enabling techniques, among which are model checking (to prove interface compliance), observers (to check interface compliance), armoring (to separate error handling from component logic) and test generation (to increase test coverage). Recently, automata learning has been applied to 218 control software components of ASMLs TWINSCAN lithography machines [40]. Using black-box learning algorithms in combination with information from log files, 118 components could be learned in an hour or less. The techniques failed to successfully infer the interface protocols of the remaining 100 components. It would be interesting to explore whether grey-box learning algorithm can help to learn models for these and even more complex control software components.

References

1. F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F.W. Vaandrager. Automata learning through counterexample-guided abstraction refinement. In *Proceedings FM 2012*, LNCS 7436, pages 10–27. Springer, August 2012.
2. F. Aarts, B. Jonsson, J. Uijen, and F.W. Vaandrager. Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods in System Design*, 46(1):1–41, 2015.
3. D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
4. D. Angluin and D. Fisman. Learning regular omega languages. *Theor. Comput. Sci.*, 650:57–72, 2016.
5. M. Benedikt, C. Ley, and G. Puppis. What you must remember when processing data words. In *Proceedings International Workshop on Foundations of Data Management*, *CEUR Workshop Proceedings* 619. CEUR-WS.org, 2010.
6. M. Bojanczyk, B. Klin, and S. Lasota. Automata with group actions. In *Proceedings LICS 2011*, pages 355–364. IEEE Computer Society, 2011.
7. M. Botinčan and D. Babić. Sigma*: Symbolic learning of input-output specifications. In *Proceedings POPL'13*, pages 443–456, New York, NY, USA, 2013. ACM.
8. C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):8290, February 2013.
9. S. Cassel, F. Howar, and B. Jonsson. RALib: A LearnLib extension for inferring EFSMs. In *Proceedings DIFTS 15*, Austin, Texas, 2015.
10. S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen. A succinct canonical register automaton model. *J. Log. Algebr. Meth. Program.*, 84(1):54–66, 2015.
11. S. Cassel, F. Howar, B. Jonsson, and B. Steffen. Active learning for extended finite state machines. *Formal Asp. Comput.*, 28(2):233–263, 2016.
12. C. Cho, D. Babić, P. Poosankam, K. Chen, E. Wu, and D. Song. Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *Proceedings SEC'11*, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.
13. P. Fiterău-Broştean, R. Janssen, and F.W. Vaandrager. Combining model learning and model checking to analyze TCP implementations. In *Proceedings CAV'16*, LNCS 9780, pages 454–471. Springer, 2016.
14. P. Fiterău-Broştean and F. Howar. Learning-based testing the sliding window behavior of TCP implementations. In *Proceedings FMICS-AVoCS 2017*, LNCS 10471, pages 185–200. Springer, 2017.
15. P. Fiterău-Broştean, T. Lenaerts, E. Poll, J. de Ruiter, F. Vaandrager, and P. Verleg. Model learning and model checking of SSH implementations. In *Proceedings SPIN 2017*, pages 142–151, New York, NY, USA, 2017. ACM.
16. N. Francez and M. Kaminski. An algebraic characterization of deterministic regular languages over infinite alphabets. *Theor. Comput. Sci.*, 306(1-3):155–175, 2003.
17. B. Garhewal, F. Vaandrager, F. Howar, T. Schrijvers, T. Lenaerts, and R. Smits. Grey-box learning of register automata. To appear in *Proceedings ICTAC'20*. Full version available as CoRR arXiv:2009.09975, September 2020.
18. D. Giannakopoulou, Z. Rakamarić, and V. Raman. Symbolic learning of component interfaces. In *Proceedings SAS'12*, pages 248–264. Springer-Verlag, 2012.
19. P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.
20. A. Hagerer, T. Margaria, O. Niese, B. Steffen, G. Brune, and H. Ide. Efficient regression testing of CTI-systems: Testing a complex call-center solution. *Annual review of communication, Int.Engineering Consortium (IEC)*, 55:1033–1040, 2001.

21. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
22. M. Höschele and A. Zeller. Mining input grammars from dynamic taints. In *Proceeding ASE 2016*, pages 720–725. ACM, 2016.
23. F. Howar, M. Isberner, B. Steffen, O. Bauer, and B. Jonsson. Inferring semantic interfaces of data structures. In *Proceedings ISoLA 2012, Part I*, LNCS 7609, pages 554–571. Springer, 2012.
24. F. Howar, D. Giannakopoulou, and Z. Rakamarić. Hybrid learning: Interface generation through static, dynamic, and symbolic analysis. In *Proceedings ISSTA 2013*, pages 268–279, New York, NY, USA, 2013. ACM.
25. F. Howar, B. Jonsson, and F. Vaandrager. Combining black-box and white-box techniques for learning register automata. In *Computing and Software Science - State of the Art and Perspectives*, LNCS 10000, pages 563–588. Springer, 2019.
26. F. Howar and B. Steffen. Active automata learning in practice. In *Machine Learning for Dynamic Software Analysis: Potentials and Limits: International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers*, pages 123–148. Springer International Publishing, 2018.
27. M. Isberner, F. Howar, and B. Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In *Proceedings RV 2014*, pages 307–322, Cham, 2014. Springer International Publishing.
28. M. Jasper, M. Mues, A. Murtovi, M. Schlüter, F. Howar, B. Steffen, M. Schordan, D. Hendriks, R. Schiffelers, H. Kuppens, and F. Vaandrager. RERS 2019: Combining synthesis with real-world models. In *Proceedings TACAS: TOOLympics, Part III*, LNCS 11429, pages 101–115. Springer, 2019.
29. D. Kozen. *Automata and computability*. Undergraduate texts in computer science. Springer, 1997.
30. O. Maler and L. Staiger. On syntactic congruences for omega-languages. *Theor. Comput. Sci.*, 183(1):93–112, 1997.
31. J. Moerman, M. Sammartino, A. Silva, B. Klin, and M. Szynwelski. Learning nominal automata. In *Proceedings POPL 2017*, pages 613–625. ACM, 2017.
32. A. Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958.
33. B. Poizat. *A Course in Model Theory – An Introduction to Contempary Mathematical Logic*. Springer-Verlag New York, 2000.
34. R.L. Rivest and R.E. Schapire. Inference of finite automata using homing sequences. *Inf. Comput.*, 103(2):299–347, 1993.
35. M. Schuts, J. Hooman, and F.W. Vaandrager. Refactoring of legacy software using model learning and equivalence checking: an industrial experience report. In *Proceedings iFM*, LNCS 9681, 2016.
36. M. Shahbaz and R. Groz. Inferring mealy machines. In *Proceedings FM 2009*, LNCS 5850, pages 207–222. Springer, 2009.
37. R. Smetsers, P. Fiterau-Brostean, and F. Vaandrager. Model learning as a satisfiability modulo theories problem. In *Proceedings LATA 2018*, LNCS 10792, pages 182–194. Springer, 2018.
38. F.W. Vaandrager. Model learning. *Communications of the ACM*, 60(2):86–95, February 2017.
39. F.W. Vaandrager and A. Midya. A Myhill-Nerode theorem for register automata and symbolic trace languages. CoRR arXiv:2007.03540, July 2020.
40. N. Yang, K. Aslam, R. Schiffelers, L. Lensink, D. Hendriks, L. Cleophas, and A. Serebrenik. Improving model inference in industry by combining active and passive learning. In *Proceedings SANER 2019*, pages 253–263. IEEE, 2019.