A Succinct Canonical Register Automaton Model for Data Domains with Binary Relations *

Sofia Cassel[1], Bengt Jonsson[1], Falk Howar[2], and Bernhard Steffen[2]

[1] Dept. of Information Technology, Uppsala University, Sweden
{sofia.cassel|bengt.jonsson}@it.uu.se
[2] Chair for Programming Systems, Technical University Dortmund, Germany
{falk.howar|steffen}@cs.tu-dortmund.de

Abstract. We present a novel canonical automaton model for languages over infinite data domains, that is suitable for specifying the behavior of services, protocol components, interfaces, etc. The model is based on register automata. A major contribution is a construction of succinct canonical register automata, which is parameterized on the set of relations by which elements in the data domain can be compared. We also present a Myhill Nerode-like theorem, from which minimal canonical automata can be constructed. This canonical form is as expressive as general deterministic register automata, but much better suited for modeling in practice since we lift many of the restrictions on the way variables can be accesed and stored: this allows our automata to be significantly more succinct than previously proposed canonical forms. Key to the canonical form is a symbolic treatment of data languages, which allows us to construct minimal representations whenever the set of relations can be equipped with a so-called branching framework.

1 Introduction

Our aim is to develop automata formalisms that can be used for systems specification, verification, testing, and modeling. It is crucial to be able to model not only control, but also data aspects of a system's behavior, and express relations between data values and how they affect control flow. For example, we may want to express that a password entered matches a previously registered one, that a sequence number is in some interval, or that a user identity can be found in some specific group.

There are many kinds of automata augmented with data, for example timed automata [1], counter automata, data-independent transition systems [15], and different kinds of register automata. Many of these types of automata have long been used for specification, verification, and testing (e.g., [18]). In our context, *register automata* is a very interesting formalism [3, 13, 7]. A register automaton has a finite set of registers (or state variables) and processes input symbols using a predefined set of operations (tests and updates) over input data and registers.

* Supported in part by the European FP7 project CONNECT (IST 231167).

Modeling and reasoning about systems becomes significantly easier if automata can be transformed into a canonical form: this is exploited in equivalence and refinement checking, e.g., through (bi)simulation based criteria [14, 17], and in automata learning (aka regular inference) [2, 9, 19]. There are standard algorithms for minimization of finite automata, based on the Myhill-Nerode theorem [11, 16], but it has proven difficult to carry over such constructions to automata models over infinite alphabets, including timed automata [21].

More recently, canonical automata based on extensions of the Myhill-Nerode theorem have been proposed for languages in which data values can be compared for equality [8, 3, 6], and also for inequality when the data domain has a total order (in [3, 6]). In these works, canonicity is obtained at the price of rather strict restrictions on how data is stored in variables and which guards may be used in transitions: two variables may not store the same data value, and in the ordered case each state enforces a fixed ordering between its variables. These restrictions often cause a blow-up in the number of states, since they require testing and encoding accidental as well as essential[3] relations between data values in a word. For instance, a cross-product of two independent automata, representing, e.g., the interleaving of two independent languages, will result in a blow-up due to the recording of accidental relations between data values of the two languages.

In [7], we presented a succinct canonical automaton model, based on a Myhill-Nerode characterization, for languages where data is compared for equality. Our model does not require different variables to store different values, and allows representing only essential relations between data values. Our approach results in register automata that are minimal in a certain class, and that can be exponentially more succinct than similar, previously proposed automata formalisms [8, 3, 6]. We have also exploited our model for active learning of data languages [12].

In this paper, we extend our canonical automaton model of [7] to data domains where data values can be compared using an arbitrary set of relations. We consider data languages that are able to distinguish words by comparing data values using only the relations in this set, and propose a form of RA that accept such languages. To achieve succinctness, our construction must be able to filter out unnecessary tests between data values, and also produce the weakest possible guards that still make the necessary distinctions between data words. It is a challenge to achieve such succinctness while maintaining canonicity. We approach it by using a symbolic representation of data languages in the form of decision-tree-like structures, called *constraint decision trees*. Constraint decision trees have superficial similarities with decision diagrams or BDDs, but since relations on the data domain typically impose asymmetries in the tree, we cannot use the minimization techniques for BDDs. Instead, we introduce a signature-specific *branching framework* that may restrict the allowable guards, and also allows us to compare branches in the tree in order to filter out unnecessary guards. Under some conditions on the branching framework, we obtain the nontrivial result that our decision trees are minimal.

[3] By *essential* relation, we mean a test which is necessary for recognizing the language.

As an illustration, if our data domain is equipped with tests for equality and (ordered) inequality, then after processing three data values (say, d_1, d_2, d_3), it may be that the only essential test between a fourth value d_4 and these three is whether $d_4 \leq d_1$ or not (i.e., all other comparisons not essential for determining whether the data word is accepted). In previous automaton proposals, this would typically result in 7 different cases, representing all possible outcomes of testing d_4 against the three previous values. In our proposal, however, we take into account whether comparisons are essential or not, resulting in only 2 cases.

Related work. An early work on generalizing regular languages to infinite alphabets is due to Kaminski and Francez [13], 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 [20, 5, 4] for an overview). Many of these formalisms recognize data languages that are invariant under permutations on the data domain, corresponding to the ability to test for equality on the data domain. Much of the work focuses on non-deterministic automata and are concerned with closedness properties and expressiveness results of data languages. A model that represents relations between data values without using registers or variables is proposed by Grumberg et al. [10].

Our interest lies in canonical deterministic RAs that can be used to model the behavior of protocols or (restricted) programs. Kaminski and Francez [8], Benedikt et al. [3], and Bojanczyk et al. [6] all present Myhill-Nerode theorems for data languages with equality tests. Canonicity is achieved by restricting how state variables are stored, which prompted us to propose a more succinct construction in [7].

There are a few extensions of Myhill-Nerode theorems to more general sets of relations between data values. Benedikt et al. [3] and Bojanczyk et al. [6] consider the case where the data domain is equipped with a total order. They present canonical automata, in which stored variables must be known to obey a total order, and in which guards must be as tight as possible (we term such automata *complete*): such restrictions may lead to unintuitive and significant blow-ups in the number of control locations.

Organization. In the next section we provide a motivating example, and introduce the register automaton model as a basis for representing data languages. In Section 3, we introduce a succinct representation of data languages, which suppresses non-essential tests, in the form of a decision tree-like structure called *constraint decision trees* (CDTs). Based on this representation, in Section 4 we define a Nerode congruence, and show that it characterizes minimal canonical forms of deterministic RAs. We also discuss the effects of restricting the RA in different ways. Conclusions are provided in Section 5.

2 Data languages and register automata

In this section, we introduce *data languages*. Data languages can be seen as languages over finite alphabets augmented with data. A data symbol is of the form $\alpha(d)$ where each α is an action and each d is a data value from some (possibly infinite) domain. A data word is a sequence of data symbols, and a data language is a set of data words.

We will consider data languages that can be recognized by comparing data values using relations from a given set $\mathcal{R}$ of binary relations. For example, if the set $\mathcal{R}$ of binary relations includes only the equality relation, this means that data languages will be closed under permutations on the data domain.

A *register automaton* (RA) is an automaton model capable of recognizing a data language. It reads data values as input and has registers (or variables) for storing them. When reading a data value, a register automaton can compare it to one or more variables in order to determine, e.g., its next transition. In the following sections, we will describe a register automaton model that recognizes data languages parameterized on a set of binary relations.

Example. Let $\mathcal{R} = \{<, =\}$ and let data values be rational numbers. Consider the data language L_2, consisting of data words where the last data value is the second-largest one in the entire data word. (Whenever the largest data value occurs several times, we call a data value second-largest if it is equal to the largest data value.) This language contains, for example, the data words $\alpha(\mathtt{3})\ \alpha(\mathtt{4})\ \alpha(\mathtt{4})$ and $\alpha(\mathtt{9})\ \alpha(\mathtt{1})\ \alpha(\mathtt{4})\ \alpha(\mathtt{2})\ \alpha(\mathtt{8})$. In the first case, the last data value is equal to the largest data value in the word. In the second case, the last data value is smaller than the largest data value in the word.

A register automaton ($\mathcal{A}_2$) that recognizes L_2 is shown in Figure 1. (For brevity, we have omitted the actions in the figure.) The $\mathcal{A}_2$ automaton has three locations, each with a set of associated variables. Accepting locations are denoted by two concentric circles, and the initial location is marked by an arrow. Arcs represent transitions, and they are labeled with guards and variable assignments. Informally, at each transition, a new data value, represented by the formal parameter p, is read by the automaton and compared to the existing location variables (using the guards). Depending on the outcome of the comparison, variables may be assigned new values, either the current data value or the value of another variable.

The second-largest data value seen so far is always kept in the variable x_2 and the largest data value seen so far in the variable x_1. Thus after having read the first two data values in any data word, the automaton will have reached a 'steady-state' where both variables x_1 and x_2 have stored data values, and any new data values are compared to these. The automaton will then alternate between locations l_2 and l_3 until the end of the data word is reached. This is because whenever a new data value is read by the automaton, it needs only distinguish between three cases: p is smaller than x_2, p is larger than x_1, or $x_2 \leq p \leq x_1$ in order to determine whether to transition to the accepting location l_2 or the rejecting location l_3. □

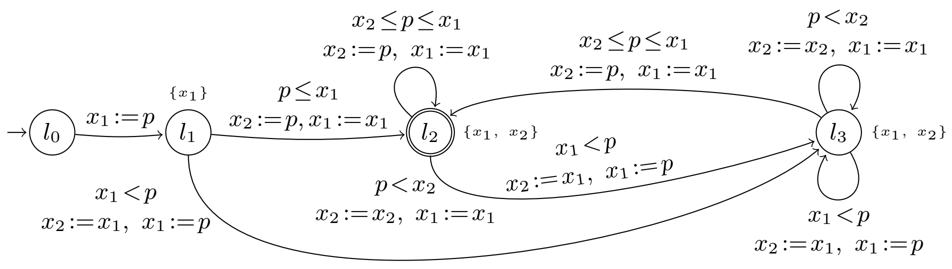

Fig. 1. Running example: the A_2 automaton

2.1 Data languages

Assume an unbounded domain $\mathcal{D}$ of data values, and a set $\mathcal{R}$ of binary relations on $\mathcal{D}$. Assume a set of *actions*, each with an *arity* that determines how many parameters it takes from the domain $\mathcal{D}$. In this paper, we assume that all actions have arity 1; it is straightforward to extend the results to the general case.

A *data symbol* is a term of form $\alpha(d)$, where α is an action and d is a data value from the domain $\mathcal{D}$. A *data word* is a sequence of data symbols. Two data words $w_d = \alpha_1(d_1)\ldots\alpha_n(d_n)$ and $w'_d = \alpha_1(c_1)\ldots\alpha_n(c_n)$, are *equivalent*, denoted $w_d \approx_{\mathcal{R}} w'_d$, if $d_i\ R\ d_{i'} \leftrightarrow c_i\ R\ c_{i'}$ whenever $R \in \mathcal{R}$, for $1 \leq i, i' \leq n$ and $1 \leq i \leq n_j, 1 \leq i' \leq n_{j'}$. Intuitively, w_d and w'_d are equivalent if they have the same sequences of actions and they cannot be distinguished by the relations in $\mathcal{R}$. A *data language* is a set $\mathcal{L}$ of data words, which respects $\mathcal{R}$ in the sense that $w_d \approx_{\mathcal{R}} w'_d$ implies $w_d \in \mathcal{L} \leftrightarrow w'_d \in \mathcal{L}$. We will often represent a data language as a mapping from the set of data words to $\{+, -\}$, where $+$ stands for accept and $-$ for reject.

2.2 Register automata

Assume a set of *formal parameters*, ranged over by $p_1, p_2, \ldots$, and a finite set of *variables* (or registers), ranged over by $x_1, x_2, \ldots$.

A *parameterized symbol* is a term of form $\alpha(p)$, where α is an action and p is a formal parameter. A *parameterized word* is a sequence of parameterized symbols in which all formal parameters are distinct, i.e., we assume a (re)naming scheme that avoids clashes. A *guard* is a conjunction of negated and unnegated relations (from $\mathcal{R}$) between formal parameters or variables.

Definition 1 (RA). *A* register automaton *(RA) is a tuple $\mathcal{A} = (L, l_0, X, T, \lambda)$, 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,*
– *T is a finite set of* transitions, *each of form $\langle l, \alpha(\overline{p}), g, \pi, l' \rangle$, where*
  - *l is a source location,*
  - *l' is a target location,*
  - *$\alpha(\overline{p})$ is a parameterized symbol,*
  - *g is a guard over $\overline{p}$ and $X(l)$, and*
  - *π (the* assignment*) is a mapping from $X(l')$ to $X(l) \cup \overline{p}$ (intuitively, the variable $x \in X(l')$ is assigned the value of $\pi(x)$), and*
– *$\lambda : L \mapsto \{+, -\}$ maps each location to either $+$ (accept) or $-$ (reject),*

such that for any location l and action α, the disjunction of all guards g in transitions $\langle l, \alpha(\overline{p}), g, \pi, l' \rangle \in T$ is equivalent to true *i.e., $\mathcal{A}$ is* completely specified. □

Semantics of a register automaton. A register automaton $\mathcal{A}$ classifies data words as either accepted or rejected. A standard way to describe how this is done is to 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 data 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. When the last symbol has been processed, the word is accepted if an accepting location has been reached, otherwise the word is rejected. We omit a more formal account.

An RA is *determinate* (called a DRA) if no data word can be processed in two different ways to reach both accepting and rejecting locations. A data word is *accepted* (*rejected*) by a DRA $\mathcal{A}$ if processing the word reaches an accepting (rejecting) location. We define $\mathcal{A}(\mathtt{w}_d)$ to be $+$ $(-)$ if the data word $\mathtt{w}_d$ is accepted (rejected) by $\mathcal{A}$. The language recognized by $\mathcal{A}$ is the set of data words that it accepts.

3 Symbolic representation of data languages

A given data language may be accepted by many different DRAs. In order to obtain a succinct, canonical form of DRAs, we will in this section define a canonical representation of data languages; in the next section we will describe how to derive canonical DRAs from this representation.

We first introduce a symbolic representation for sets of data words, called *constrained words.* These can also be regarded as representing runs of a register automaton. We can then use sets of constrained words, together with a classification of these words as “accepted” or “rejected”, as a representation of data languages. Such classified sets will be called *constraint decision trees.* We establish, as a central result (in Theorem 1), that any data language can be represented by a *minimal* set of constrained words, corresponding to a minimal constraint decision tree. This minimal set will correspond to the set of runs of our canonical automaton, and will serve several purposes during automata construction:

– it will allow us to keep only the essential relations between data values and filter out inessential (accidental) relations between data values,
– from it, we can derive the parameters an automaton must store in variables after processing a data word, and
– we can transform parts of it directly into transitions when constructing a canonical DRA.

Constrained words. A *parameterized word* $w = \alpha_1(p_1) \cdots \alpha_k(p_k)$ is a data word where concrete data values are replaced by formal parameters. We let parameters be indexed $1 \cdots |w|$, where $|w|$ is the number of formal parameters in w, i.e., the sequence of parameters is $p_1 \cdots p_{|w|}$. A *literal* is of the form $p_i \; R \; p_j$ or $\neg(p_i \; R \; p_j)$, where p_i and p_j are formal parameters and $R \in \mathcal{R}$. A *constraint* ϕ is a conjunction of literals. We say that a constraint ϕ is *weaker than* a constraint ϕ', and that ϕ' is *stronger than* ϕ if ϕ' implies ϕ. A *constrained word* is a pair $\langle w, \phi \rangle$ consisting of a parameterized word w and a constraint ϕ over the formal parameters of w. If l is a literal of form $p_i \; R \; p_j$ or $\neg(p_i \; R \; p_j)$, then the *level of* l *in* w is the maximum of i and j. A constraint ϕ is k*-level in* w if it contains only literals of level k in w, and it is $\leq k$*-level in* w if it contains only literals of level $\leq k$ in w. Let $\phi^w|_k$ denote the conjunction of all k-level literals in $\langle w, \phi \rangle$. Similarly, define $\phi^w|_{<k}$ as the conjunction of all literals of ϕ of level smaller than k in w. Define $\phi^w|_{\leq k}$, $\phi^w|_{>k}$, and $\phi^w|_{\geq k}$ analogously.

A constraint ϕ is a k*-atom* if all its literals are of level at most k, and ϕ implies either $p_i \; R \; p_j$ or $\neg(p_i \; R \; p_j)$ for any $R \in \mathcal{R}$ whenever $i, j \leq k$. Intuitively, an atom is a maximal consistent constraint, i.e., it cannot be more specified without becoming inconsistent. A constrained word $\langle w, \phi \rangle$ is an *atom* if ϕ is a $|w|$-atom.

A data word $\mathtt{w}_d$ *satisfies* a constrained word $\langle w, \phi \rangle$, denoted $\mathtt{w}_d \models \langle w, \phi \rangle$, if w and $\mathtt{w}_d$ have the same sequence of actions, and the data values in $\mathtt{w}_d$ satisfy ϕ in the obvious way.

Example. Let $w = \alpha_1(p_1) \; \alpha_2(p_2) \; \alpha_3(p_3)$ be a parameterized word, and let $\phi = p_1 \leq p_3 \leq p_2$. Then $\langle w, \phi \rangle$ is a constrained word. Let $w_d = \alpha_1(\mathtt{3}) \; \alpha_2(\mathtt{7}) \; \alpha_3(\mathtt{4})$ be a data word. Then $w_d \models \langle w, \phi \rangle$, and that $\langle w, \phi \rangle$ is an atom. □

3.1 Constraint decision trees

We will now introduce constraint decision trees, and how they recognize data languages. Let a k*-branching* be a set of k-level constraints whose disjunction is equivalent to *true*. Let ϕ be a $\leq (k-1)$-level constraint. A k-level constraint ψ is ϕ*-admissible* (or admissible after ϕ) if ϕ implies $(\phi \wedge \exists p_k \; \psi)$, i.e., if ψ does not add any additional constraint between the parameters of ϕ. A k-branching Ψ is ϕ*-admissible* if each k-level constraint in Ψ is ϕ-admissible.

A set Φ of constrained words is *prefix-closed* if $\langle wv, \phi \rangle \in \Phi$ implies $\langle w, \phi^{wv}|_{\leq |w|} \rangle \in \Phi$. A set Φ is *extension-closed* if for any $\langle w, \phi \rangle \in \Phi$ and any action α, the set of $(|w| + 1)$-level constraints ψ such that $\langle w\alpha(p_{|w|+1}), \phi \wedge \psi\rangle \in \Phi$ forms a ϕ-admissible $(|w| + 1)$-branching.

Definition 2 (CDT). *A* constraint decision tree *(CDT) $\mathcal{T}$ is a pair $\langle Dom(\mathcal{T}), \lambda_{\mathcal{T}}\rangle$ where $Dom(\mathcal{T})$ is a non-empty prefix-closed and extension-closed set of constrained words, and $\lambda_{\mathcal{T}} : Dom(\mathcal{T}) \mapsto \{+, -\}$ is a mapping from $Dom(\mathcal{T})$ to $\{+, -\}$.* □

A CDT $\mathcal{T}$ is *determinate* (called a DCDT) if $\lambda_{\mathcal{T}}(\langle w, \phi\rangle) = \lambda_{\mathcal{T}}(\langle w, \phi'\rangle)$ whenever $\mathtt{w}_d \models \langle w, \phi\rangle$ and $\mathtt{w}_d \models \langle w, \phi'\rangle$ for some data word $\mathtt{w}_d$. It is *complete* if all constrained words in $Dom(\mathcal{T})$ are atoms.

A DCDT defines a language $\lambda_{\mathcal{T}}$ defined by $\lambda_{\mathcal{T}}(\mathtt{w}_d) = \lambda_{\mathcal{T}}(\langle w, \phi\rangle)$ whenever $\mathtt{w}_d \models \langle w, \phi\rangle$. Intuitively, a CDT can be thought of as a set of runs of a register automaton. Each constrained word $\langle w, \phi\rangle$ represents a path through the automaton: the parameterized word w is the sequence of actions and formal parameters, and ϕ is the conjunction of all guards that are tested along the path. A design constraint for our canonical automaton model is that all essential tests concerning the relationship between a parameter p_i and previously received parameters (i.e., parameters p_j with $j < i$) should be performed when p_i is processed. This is reflected in the property of admissibility, which intuitively means that a guard should not retroactively constrain the relation between previously received parameters. The property of extension-closed implies that the CDT is completely specified in the sense that it can classify any data word as accepted or rejected.

In the following, we will show that for each language $\mathcal{L}$, we can construct a canonical CDT that faithfully represents $\mathcal{L}$, and which is also minimal under some restrictions. We will first try to provide some intuition for our construction.

Constructing a canonical DCDT. A first attempt at constructing a canonical DCDT $\mathcal{T}$ could be to simply include all atoms in the domain $Dom(\mathcal{T})$, thus resulting in a complete DCDT. This DCDT will surely be able to correctly classify a data language, but it will typically be prohibitively large. We need to find ways to reduce the size of the CDT while still rendering it capable to correctly classify the language it represents.

Example. Let $\mathcal{D}$ be the set of rational numbers, and let $\mathcal{R} = \{<, =\}$. Assume that $\mathcal{T}$ is a complete DCDT. The constrained words in $Dom(\mathcal{T})$ of the form $\langle a(p_1)b(p_2)c(p_3), \phi\rangle$ would then be such that ϕ specifies some total order between p_1, p_2, p_3. The question is then whether we actually need a total order between the parameters in order to correctly classify the data language represented by $\mathcal{T}$. Perhaps this language is insensitive to the ordering between p_2 and p_3, or it simply does not distinguish the case $p_2 < p_3$ from $p_2 = p_3$. We would like the CDT to reflect this by replacing atoms by weaker constrained words. A constrained word is weaker than an atom if the atom implies the constrained word. This means that a constrained word can be used to represent several atoms, i.e., we can 'merge' the atoms.

Two atoms can be represented by the same constraint if any constraint that is admissible after the one atom is also admissible after the second atom (and vice versa), and their classifications (accept/reject) match. However, sometimes we want to merge atoms that do not fulfill these conditions. Consider the atoms $p_1 < p_2 \wedge p_2 < p_3$ and $p_1 < p_2 \wedge p_2 = p_3$. We can *not* add the same set of constraints after both atoms; for instance, the 4-level constraint $p_2 < p_4 \wedge p_4 < p_3$ is admissible after the atom $p_1 < p_2 \wedge p_2 < p_3$, but not after $p_1 < p_2 \wedge p_2 = p_3$.

We can solve this problem by introducing an ordering $\sqsubseteq_\phi$ between extensions of an atom. We then try to use the $\sqsubseteq_\phi$-smaller extension to classify the larger extension. This can be done if the classifications match and the resulting constraints are admissible. In the above example, $p_2 < p_3$ and $p_2 = p_3$ extend the atom $p_1 < p_2$. If we order them as $p_2 < p_3 \sqsubseteq_\phi p_2 = p_3$, we can check if the classification of the extensions of $p_1 < p_2 \wedge p_2 = p_3$ matches the classification of the extensions of $p_1 < p_2 \wedge p_2 < p_3$. If they do, we can merge the atoms into $p_1 < p_2 \wedge p_2 \leq p_3$. $\square$

3.2 Branching frameworks

Let us now describe the structure that must be predefined in order to define a canonical DCDT. We assume that the set $\mathcal{R}$ of binary relations on $\mathcal{D}$ is fixed.

Let ϕ be a $k{-}1$-constraint. A *guard hierarchy for* ϕ is a set $\mathcal{P}$ of ϕ-admissible k-level constraints, in which the set of maximally strong (wrp. to implication) constraints ψ (called *atomic branches* of $\mathcal{P}$) are such that $\phi \wedge \psi$ is an atom, and such that the set of atomic branches of $\mathcal{P}$ forms a ϕ-admissible k-branching.

Definition 3 (Branching framework). *A* branching framework *is a mapping $\mathcal{M}$ which to each k–1-atom ϕ assigns a pair $\langle \mathcal{P}, \sqsubseteq_\phi \rangle$, where $\mathcal{P}$ is a guard hierarchy for ϕ, and $\sqsubseteq_\phi$ is a partial order on the atomic branches of $\mathcal{P}$.* $\square$

Intuitively, the elements of $\mathcal{P}$ are the possible k-level constraints that test the next parameter that follows after ϕ in a CDT. The atomic branches represent the "most constrained" k-level constraints that completely characterize how the next parameter p_k is related to previous parameters $p_1, \ldots, p_{k-1}$. Note that different guards or atomic branches need not be mutually exclusive.

For any k-level constraint g in $\mathcal{P}$, define the *support of g after ϕ*, denoted $\mathit{supp}^\phi(g)$, as the set of atomic branches $\psi \in \mathcal{P}$ such that $(\phi \wedge \psi)$ implies $(\phi \wedge g)$. Since the set of atomic branches forms a k-branching, it follows that any element g in $\mathcal{P}$ represents the set of atomic branches in $\mathit{supp}^\phi(g)$ in the sense that

$$(\phi \wedge g) \leftrightarrow (\phi \wedge \bigvee_{\psi \in \mathit{supp}^\phi(g)} \psi) \quad .$$

Definition 4. *A branching framework $\mathcal{M}$ is* adequate *if whenever $\mathcal{M}(\phi) = \langle \mathcal{P}, \sqsubseteq_\phi \rangle$, then*

- *for each constraint g in $\mathcal{P}$, the set $\mathit{supp}^\phi(g)$ contains a unique minimal (wrp. to $\sqsubseteq_\phi$) atomic branch, called the* principal atomic branch *of g, and*
- *Whenever two constraints g, g' in $\mathcal{P}$ have the same principal atomic branch, then $g \vee g'$ is in $\mathcal{P}$.* $\square$

The concept of principal atomic branch can be extended from k-constraints to arbitrary constrained words as follows. For a CDT $\mathcal{T}$, an adequate branching framework $\mathcal{M}$, and any constrained word $\langle w, \phi\rangle$, define the *principal atom* of ϕ inductively, as follows:

- The principal atom of the empty constraint *true* over the empty sequence of parameters is *true*
- If ϕ is a constraint over $p_1, \ldots, p_k$, let ϕ' be the principal atom of $\phi^w|_{\leq k-1}$, let ψ be $\phi^w|_k$, and let $\mathcal{M}(\phi')$ be $\langle \mathcal{P}, \sqsubseteq_\phi\rangle$. Then the principal atom of ϕ over $p_1, \ldots, p_k$ is the atom $\phi' \wedge \psi'$, where ψ' is the principal atomic branch of ψ.

The branching framework determines what kinds of constraints we allow in the CDT. For example, if we are dealing with data values that are rational numbers and subject to some order $<$, we might allow constraints to specify intervals (such as $p_2 < p_4 < p_1$). Whenever we can extend a constraint ϕ by some other constraint g, we must also be able to extend the principal atom of ϕ by g, i.e., the constraints ϕ and g must be compatible.

Example. Consider the case where $\mathcal{R}$ is $\{=\}$. We can obtain the model in our previous work [7], by a branching framework which assigns to a $\leq (k-1)$-atom ϕ stating that the parameters $p_1, \ldots, p_{k-1}$ are all different, the pair $\langle \mathcal{P}, \sqsubseteq_\phi\rangle$, where

- $\mathcal{P}$ consists of all conjunctions of subsets of the literals $p_1 \neq p_k, \cdots, p_{k-1} \neq p_k$ (of these, only the maximal one is an atom), as well as the $k-1$ constraints of form $p_i = p_k$ for $i = 1, \ldots, k-1$, and
- $(p_1 \neq p_k \wedge \cdots \wedge p_{k-1} \neq p_k) \sqsubseteq_\phi p_i = p_k$ for $i = 1, \ldots, k-1$, but $p_i = p_k$ and $p_j = p_k$ are not ordered for $i \neq j$.

Consequently, each k-level constraint with nontrivial support will have $p_1 \neq p_k \wedge \cdots \wedge p_{k-1} \neq p_k$ as principal atomic branch. □

Example. Let us next consider the case where $\mathcal{D}$ is the set of rational numbers, and $\mathcal{R}$ is $\{<, =\}$ (it is important not to let $\mathcal{D}$ be some set of integers, since that case is more complicated). An atom ϕ over $p_1, \ldots, p_{k-1}$ will specify some order between $p_1, \ldots, p_{k-1}$, for example $p_1 < \cdots < p_{k-1}$. A suitable branching framework will assign to a $\leq (k-1)$-atom ϕ the pair $\langle \mathcal{P}, \sqsubseteq_\phi\rangle$, where

- Ψ contains all possible non-empty intervals between to parameters p_i, p_j with $1 \leq i \leq j \leq k$, e.g., $p_2 \leq p_k < p_5$, or (some degenerate cases) $p_k = p_i$, or $p_k < p_1$, or $p_{k-1} < p_k$. Among these, only the minimal intervals are atoms.
- The relation $\sqsubseteq_\phi$ will then be the smallest transitive relation that fulfills the following conditions:
  - $p_i < p_k < p_{i+1} \sqsubseteq_\phi p_{i-1} < p_k < p_i$ (for $i = 2, \ldots, k-2$), and
    $p_1 < p_k < p_2 \sqsubseteq_\phi p_k < p_1$, and
    $p_{k-1} < p_k \sqsubseteq_\phi p_{k-2} < p_k < p_{k-1}$,
  - $p_{i-1} < p_k < p_i \sqsubseteq_\phi p_k = p_i$ (for $i = 2, \ldots, k-1$), and
    $p_k < p_1 \sqsubseteq_\phi p_k = p_1$.

□

Definition 5. Let $\mathcal{T}$ be a CDT, and let $\mathcal{M}$ be an adequate branching framework. Then $\mathcal{T}$ is an $\mathcal{M}$-*CDT* if for any $\langle w, \phi \rangle \in Dom(\mathcal{T})$ with ϕ' being the principal atom of ϕ, it is the case that whenever ψ is a $|w| + 1$-level constraint with $\langle w\alpha(p_{|w|+1}), \phi \wedge \psi \rangle \in Dom(\mathcal{T})$, then $\psi \in \mathcal{P}$, where $\langle \mathcal{P}, \sqsubseteq_\phi \rangle = \mathcal{M}(\phi')$ $\square$

3.3 Minimal constraint decision trees

We will now show how to obtain a minimal constraint decision tree for a data language.

Theorem 1 (Minimal DCDT). Let $\mathcal{M}$ be an adequate branching framework. Then for any data language $\mathcal{L}$, there is a unique minimal $\mathcal{M}$-DCDT $\mathcal{T}$ such that $\mathcal{L} = \lambda_\mathcal{T}$. $\square$

By minimal, we mean that if $\mathcal{T}'$ is any other $\mathcal{M}$-DCDT with $\mathcal{L} = \lambda_{\mathcal{T}'}$, then any constrained word in $Dom(\mathcal{T}')$ is contained in a constrained word in $Dom(\mathcal{T})$. Intuitively, this means that $Dom(\mathcal{T})$ uses the weakest possible constraints that are necessary in order to be able to correctly recognize the language $\mathcal{L}$. We will sometimes use the term $\mathcal{L}$-*essential* (constrained) words (or just $\mathcal{L}$-essential words) for members of $Dom(\mathcal{T})$ where $\mathcal{T}$ is the minimal DCDT with $\mathcal{L} = \lambda_\mathcal{T}$.

Proof. (Sketch.) We prove Theorem 1 by construction. The minimal DCDT for $\mathcal{L}$ is constructed starting from the $\sqsubseteq_\phi$-minimal atoms that serve as the leaf nodes. Atoms are merged to form guards in a bottom-up fashion. This is only possible if we assume a bounded length of words that are classified by $\mathcal{L}$. We will therefore assume a maximal length n of data words and construct a minimal "truncated" DCDT $\mathcal{T}_n$, which correctly classifies data words of length at most n. We can then show that $\mathcal{T}_n$ "grows monotonically" with increasing n, so that $\mathcal{T}$ can be taken as a limit of the trees $\mathcal{T}_n$.

Example. We will now describe how to construct the canonical CDT for our running example L_2. Recall that a branching framework assigns a set of partially ordered branches to each atom. The partial order determines in what order we will add branches as guards in the minimal CDT, and also which branches can be merged to form guards. We will consider atoms of increasing k-level. At each level, we will construct the subtree (set of constraints) of each atomϕ in increasing $\sqsubseteq_\phi$-order. We will then check if any of the atoms can be merged. At the leaf level in the tree, atoms can be merged if they have the same classification. For reasons of brevity, we will here only consider words of length 3 at most.

In the L_2 example, the 1-level atom is *true*. We generate the set of 2-level atomic branches, ordered as $p_1 < p_2 \sqsubseteq_\emptyset p_2 < p_1 \sqsubseteq_\emptyset p_1 = p_2$.

We apply the branching framework to the smallest branch $p_1 < p_2$, and obtain the set of 3-level atomic branches after $p_1 < p_2$, ordered as $\{p_2 < p_3 \sqsubseteq_{(p_1 < p_2)}$ $p_1 < p_3 < p_2 \sqsubseteq_{(p_1 < p_2)} p_3 = p_2, \quad p_1 < p_3 < p_2 \sqsubseteq_{(p_1 < p_2)} p_3 < p_1 \sqsubseteq_{(p_1 < p_2)} p_3 = p_1\}$. The smallest branch is $p_2 < p_3$, which is a rejecting leaf. We construct the first 3-level L_2-essential constrained word as $p_2 < p_3$.

The next smallest branch is $p_1 < p_3 < p_2$ which is accepted. It can be merged with the larger branches $p_1 = p_3$ and $p_2 = p_3$, since they are both also accepting. (Because we are at the leaf level in the tree, this is sufficient to determine whether branches can be merged.) We thus construct the second 3-level L_2-essential constraint as $p_1 \leq p_3 \leq p_2$. The only branch left is now $p_3 < p_1$, which will become the third L_2-essential constraint.

There are no more 3-level branches to classify, so we go back to level 2. Since we have already constructed the subtree after $p_1 < p_2$, we proceed to construct the subtree after the next smallest 2-level branch, $p_2 < p_1$ in the same manner. This results in the L_2-essential constraints $p_2 \leq p_3 \leq p_1$, $p_3 < p_2$ and $p_1 < p_3$.

The last remaining 2-level branch is $p_1 = p_2$. We construct the subtree, resulting in the constraints $p_1 = p_2 = p_3$, $p_3 < p_1 = p_2$ and $p_1 = p_2 < p_3$. It is possible to use the subtree after $p_2 < p_1$ to classify the constraints after $p_1 = p_2$, so we can merge these atoms into $p_2 \leq p_1$.

We obtain the set of L_2-essential constrained words of length 3 as
$\{\langle w, p_3 < p_2 \leq p_1\rangle, \langle w, p_2 \leq p_3 \leq p_1\rangle, \langle w, p_2 \leq p_1 < p_3\rangle, \langle w, p_3 < p_1 < p_2\rangle,$
$\langle w, p_1 \leq p_3 \leq p_2\rangle, \langle w, p_1 < p_2 < p_3\rangle\}$ where $w = \alpha(p_1)\alpha(p_2)\alpha(p_3)$.

The L_2-essential constrained words directly correspond to paths in the $\mathcal{A}_2$ automaton in Figure 1. For example, the path $l_0 \rightarrow l_1 \rightarrow l_3 \rightarrow l_3$ corresponds to the constrained word $\langle w, p_3 < p_1 < p_2\rangle$. Since $\mathcal{A}_2$ always stores the largest data value seen so far in the variable x_1 and the second-largest seen so far in x_2, this path requires variables to be reassigned. The variable x_1 will first store the first data value. Then, because the second data value is larger than the first, x_1 will be re-assigned the second data value (and the first data value will be moved to x_2).

4 Nerode congruence

We will now define a Nerode-like congruence on constrained words. As in the classical Nerode congruence for regular languages, we will define constrained words to be equivalent if their suffixes are equivalent.

In order to describe how to fold a constraint decision tree into a register automaton, we need to decide which parameters to store as variables in the automaton. We associate a set of *memorable* parameters with each constrained word. These are the parameters that occur in the word and are needed in some guard in some of its suffixes. When the CDT is folded into a register automaton, the memorable parameters of a node will become location variables at the location that corresponds to that particular node.

Let us first define how a constrained word can be split into a prefix and a suffix. Consider a constrained word $\langle w, \phi\rangle$, where the parameterized word w is a concatenation uv, and u has k parameters. We can make a corresponding split of ϕ as $\phi^w|_{\leq k} \wedge \phi^w|_{>k}$. Then $\langle u, \phi^w|_{\leq k}\rangle$ (the prefix) is a constrained word, but $\langle v, \phi^w|_{>k}\rangle$ (the suffix) is in general not, since $\phi^w|_{>k}$ refers to parameters that are not in v. We therefore define a *$\langle u, \phi\rangle$-suffix* as a tuple $\langle v, \psi\rangle$, where ψ is a constraint over parameters of u and v, in which each literal contains at least one parameter from v, and such that $\langle uv, \phi \wedge \psi \rangle$ (which we often denote $\langle u, \phi \rangle; \langle v, \psi \rangle$) is a constrained word.

Definition 6 (Memorable). *Let $\mathcal{L}$ be a data language, let $\mathcal{M}$ be an adequate branching framework, and let $\mathcal{T}$ be the minimal $\mathcal{M}$-DCDT recognizing $\mathcal{L}$. The $\mathcal{L}$-*memorable parameters *of a constrained word $\langle w, \phi \rangle \in Dom(\mathcal{T})$, denoted $mem_{\mathcal{L}}(\langle w, \phi \rangle)$, is the set of parameters in w that occur in some $\langle w, \phi \rangle$-suffix $\langle v, \psi \rangle$ such that $\langle w, \phi \rangle; \langle v, \psi \rangle \in Dom(\mathcal{T})$.* □

In order for our canonical form to capture exactly the causal relations between parameters, we will allow memorable parameters to be permuted. when comparing words. Two words will be considered equivalent if they require equivalent parameters to be stored, independent of their ordering or their names.

Definition 7 (Nerode congruence). *Let $\mathcal{L}$ be a data language, and let $\mathcal{T}$ be the minimal DCDT recognizing $\mathcal{L}$. We define the equivalence $\equiv_{\mathcal{L}}$ on constrained words by $\langle w, \phi \rangle \equiv_{\mathcal{L}} \langle w', \phi' \rangle$ if there is a bijection $\gamma : mem_{\mathcal{L}}(\langle w, \phi \rangle) \mapsto mem_{\mathcal{L}}(\langle w', \phi' \rangle)$ such that*

- *$\langle v, \psi \rangle$ is a $\langle w, \phi \rangle$-suffix such that $\langle wv, \phi \wedge \psi \rangle \in Dom(\mathcal{T})$ iff $\langle v, \gamma(\psi) \rangle$ is a $\langle w', \phi' \rangle$-suffix such that $\langle w'v, \phi' \wedge \gamma(\psi) \rangle \in Dom(\mathcal{T})$, and then*
- *$\mathcal{L}(\langle wv, \phi \wedge \psi \rangle) = \mathcal{L}(\langle w'v, \phi' \wedge \gamma(\psi) \rangle)$,*

where $\gamma(\psi)$ is obtained from ψ by replacing all parameters in $mem_{\mathcal{L}}(\langle w, \phi \rangle)$ by their image under γ.

Intuitively, two constrained words are equivalent if they induce the same residual languages modulo a remapping of their memorable parameters. The equivalence $\equiv_{\mathcal{L}}$ is also a congruence in the following sense. If $\langle w, \phi \rangle \equiv_{\mathcal{L}} \langle w', \phi' \rangle$ is established by the bijection $\gamma : \mathrm{mem}_{\mathcal{L}}(\langle w, \phi \rangle) \mapsto \mathrm{mem}_{\mathcal{L}}(\langle w', \phi' \rangle)$, then for any $\mathrm{mem}_{\mathcal{L}}(\langle w, \phi \rangle)$-suffix $\langle v, \psi \rangle$ we have $\langle w, \phi \rangle; \langle v, \psi \rangle \equiv_{\mathcal{L}} \langle w', \phi' \rangle; \langle v, \gamma(\psi) \rangle$.

By using the Nerode congruence, we can 'fold' a constraint decision tree into a register automaton, mapping constrained words that are equivalent by this congruence to the same location.

We are now able to relate our Nerode congruence to DRAs.

Theorem 2 (Myhill-Nerode). A data language $\mathcal{L}$ is recognizable by a DRA iff the equivalence $\equiv_{\mathcal{L}}$ on $\mathcal{L}$-essential words has finite index.

Proof. (Sketch.) We will not detail the extension of the Myhill-Nerode theorem here, but refer to the version stated in [7]For the if-direction, we construct a DRA from a congruence. The locations of the resulting DRA will be given by the finitely many equivalence classes of the Nerode relation on essential words. Transitions will be extracted from the representative words in each equivalence class. Location variables will be given by the memorable parameters of the representative words. For the only-if direction, we assume any DRA that accepts $\mathcal{L}$. The proof idea then is to show that two $\mathcal{L}$-essential constrained words corresponding to sequences of transitions that lead to the same location are also equivalent w.r.t. $\equiv_{\mathcal{L}}$, i.e., that one location of a DRA cannot represent more than one class of $\equiv_{\mathcal{L}}$. This can be shown using congruence properties. □

Let us now reconsider our running example. In general, many different register automata can be constructed that accept some given data language – for example, one that simply stores each new data value in a location variable, regardless of whether or not it will be referenced later. This leads to an unnecessary blowup in the number of location variables. Another automaton might only store data values that will be referenced later, but perform all possible tests on newly received data values. Such an automaton would distinguish between the two cases $x_2 \leq p < x_1$ and $x_2 = p < x_1$ in the self-loop at location l_2 in the automaton of Figure 1. In general, canonical models are easier to define if strong restrictions are imposed on their form. In previous proposals, typical such restrictions include to maintain a priori relations between stored variables (e.g., that $x_2 < x_1$), or that guards always be as strong as possible (thus duplicating the self-loop at location l_2). In this paper, we have lifted several such restrictions, while still producing canonical automata.

Example. Consider the $\mathcal{A}_2$ automaton in Figure 1. Here, the locations l_2 and l_3 both have two location variables x_1 and x_2. The variable x_1 always stores the largest data value seen so far, and the variable x_2 stores the second-largest data value seen so far. There is no uniqueness restriction on the variables, so it may well be that x_1 and x_2 store equal data values. Imposing the uniqueness restriction on the location variables of this automaton will lead to l_2 and l_3 being duplicated. The automaton will thus have two accepting states and two rejecting, depending on whether the two largest data values seen so far are equal or not. (The 'initial' state l_0 has no location variables; l_1 only has one location variable, and they will both stay that way.) $\square$

5 Conclusions and future work

In this paper, we have presented a succinct canonical register automaton model for data languages, in which data values can be compared by an arbitrary given set of relations. This construction consistently and significantly generalizes our previous work [7], which considered only the equality relation. Our construction gives rise to automata that are often considerably more succinct than those of previous proposals.

The main technical contribution of the paper is the symbolic treatment of data languages using branching frameworks to organizate relations on the data domain canonically. This allows us to extend our ideas from [7] resulting in a Myhill Nerode-like theorem for this larger class of data languages.

Our immediate plans are to use these results to derive canonical models of realistic protocols, services, and interfaces, as well as generalizing Angluin-style active learning to this class of systems.

References

1. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

2. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.
3. M. Benedikt, C. Ley, and G. Puppis. What you must remember when processing data words. In *Proc. 4th Alberto Mendelzon Int. Workshop on Foundations of Data Management, Buenos Aires, Argentina*, volume 619 of *CEUR Workshop Proceedings*, 2010.
4. H. Björklund and T. Schwentick. On notions of regularity for data languages. *Theoretical Computer Science*, 411:702–715, January 2010.
5. M. Bojanczyk, C. D. andA. Muscholl, T. Schwentick, and L. Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Log.*, 12(4):27, 2011.
6. M. Bojanczyk, B. Klin, and S. Lasota. Automata with group actions. In *LICS*, pages 355–364. IEEE Computer Society, 2011.
7. S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen. A succinct canonical register automaton model. In *ATVA*, volume 6996 of *Lecture Notes in Computer Science*, pages 366–380. Springer Verlag, 2011.
8. N. Francez and M. Kaminski. An algebraic characterization of deterministic regular languages over infinite alphabets. *Theoretical Computer Science*, 306(1-3):155–175, 2003.
9. E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.
10. O. Grumberg, O. Kupferman, and S. Sheinvald. Variable automata over infinite alphabets. In A. Dediu, H. Fernau, and C. Martín-Vide, editors, *LATA*, volume 6031 of *Lecture Notes in Computer Science*, pages 561–572. Springer, 2010.
11. J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
12. F. Howar, B. Steffen, B. Jonsson, and S. Cassel. Inferring canonical register automata. In *VMCAI*, volume 7148 of *Lecture Notes in Computer Science*, pages 251–266. Springer Verlag, 2012.
13. M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.
14. P. Kanellakis and S. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, May 1990.
15. R. Lazic and D. Nowak. A unifying approach to data-independence. In C. Palamidessi, editor, *Proc. CONCUR 2000,* 11^{th} *Int. Conf. on Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 581–595, 2000.
16. A. Nerode. Linear Automaton Transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958.
17. R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, 1987.
18. A. Petrenko, S. Boroday, and R. Groz. Confirming configurations in EFSM testing. *IEEE Trans. on Software Engineering*, 30(1):29–42, 2004.
19. R. Rivest and R. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.
20. L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In *CSL: Proc. 20th Int. Workshop on Ccomputer Science Logic, Szeged, Hungary*, volume 4207 of *Lecture Notes in Computer Science*, pages 41–57. Springer, 2006.
21. T. Wilke. Specifying timed state sequences in powerful decidable logics and timed automata. In *Proc. FTRTFT*, volume 863 of *Lecture Notes in Computer Science*, pages 694–715. Springer, 1994.