In the Maze of Data Languages

Loris D'Antoni
University of Pennsylvania

Abstract

In data languages the positions of strings and trees carry a label from a finite alphabet and a data value from an infinite alphabet. Extensions of automata and logics over finite alphabets have been defined to recognize data languages, both in the string and tree cases. In this paper we describe and compare the complexity and expressiveness of such models to understand which ones are better candidates as *regular* models.

1 Introduction

The presence of values ranging over an infinite domain in everyday programming has led to the study of languages over infinite alphabets. In areas like program verification [HMP92, AC11, BHM02, DL09] and XML processing and validation [BDM$^+$06], the finite alphabet abstraction is too restrictive to perform useful analysis.

In program verification, for example, unbound data and properties about infinite domains (like natural numbers) are commonly used. Unfortunately, when dealing with infinite domains most problems are undecidable. However, not everything is lost. In fact, some problems become decidable when restricting the set of allowed operations and/or the number of variables ranging over an infinite domain [BHM02].

In terms of XML processing, the problem of representing infinite domains occurs in several settings. Although the tree structures in XML range over a finite alphabet, the values that can be stored in the XML attributes do not [AMN$^+$01]. Most languages model XML documents with labelled, ordered, unranked trees, where the labels are from a finite alphabet, while the attribute values are usually ignored. Clearly, the current approach is too limiting, especially if we want to ask questions about the values contained in the attributes, such as, *'Does an XML specification accept trees where two nodes contain the same attribute values?'*. Again, crossing the line of undecidability is quite easy. However, several restrictions with interesting decidable properties and good expressiveness have been proposed.

One of the most studied and successful restriction is the one where *data values* from an infinite domain can be attached to the labels over a finite domain that appears in our program, string, tree, etc [BMS$^+$06, BDM$^+$06]. In particular, in a *data string* each position in the string carries a label from a finite alphabet and a data value from an infinite domain. A *string data language* is a language (set) of data strings. With this framework set, the following question arises: *"What is a regular data language?"* Several people have tried answering this question with new automata models and new logics over data strings.

When we hear the term *regular language*, we immediately think of finite automata and strings over a finite alphabet. Regular string languages gave rise to a very important concept in computer science: *regularity of languages*. But how do we extend this concept to string data languages? Before attempting to answer this, let's ask ourselves a couple of other questions to better understand the notion of regularity. Why are regular string languages so popular? What made them an appealing model?

Regular string languages have a good trade-off between expressiveness and decidability. They are also closed under a broad set of operations, and most of the interesting decision problems can be solved in polynomial time. Every regular language can also be expressed as a formula in monadic second-order logic (MSO) over strings [Tho96].

Then we might ask a more general question: *"What makes a language regular?"* All the previously mentioned properties make up a good answer, but the notion of language regularity can be generalized further. A *regular language* should have the following properties.

Expressiveness vs Decidability. They should capture the largest class of languages where properties such as emptiness, equivalence, inclusion, etc are decidable.

Efficiency. Testing whether a given element belongs to a regular language should be solvable in linear time.

Closure properties. They should be closed under operations such as intersection, complement, reverse, etc.

Robustness. The class of languages they define should have many characterizations, in particular an equivalent representation in logic.

All these properties hold for *regular string languages* and the notion of regularity has been successfully extended to other kinds of structures (languages), including infinite strings and finite or infinite, ranked or unranked trees. Many equivalent models and logics for representing regular languages over these structures have been proposed and the properties listed above are satisfied by these models. The notion of regularity has also been extended to transductions (i.e. transformations from input strings/trees to output strings/trees). In all these cases the generalizations have been quite straightforward, in the sense that they simply used natural extensions of finite automata and monadic second-order logic. Unfortunately, for data languages, the story is not as pleasant. When we try to extend the standard models to data languages, even the simplest properties become undecidable.

Since simple extensions did not work, new models have been proposed to deal with data languages. To define a language, there are mainly two approaches: the automata approach and the logic approach. Both the approaches have been explored in the case of data languages. Even though the automata approach is closer to the notion of computation, the logic approach is easier to reason about. Usually regularity occurs when a decidable logic meets an automata model. As we mentioned earlier, the equivalence of MSO over strings and finite automata is a notable example.

We present the major models proposed for data languages and compare their properties. We will not present proofs, which can be found in the references. For *string data languages* several models have been proposed in the last decade. In [NSV04] Neven et al. provide a complete analysis on expressiveness and decidability for two automata models: *register automata* and *pebble automata*. Register automata are finite automata equipped with a finite set of registers that can hold data values which can be tested for equality. Pebble automata are finite automata equipped with a finite set of pebbles that can hold pointers to positions in the string. Pebbles must be accessed with a stack discipline. Although most of the results in [NSV04] are negative, their paper renewed interest in the study of languages over infinite alphabets. In the paper they also compare several variants of these models and describe how they behave with respect to logics over data. Although this kind of comparison is standard when talking about languages, it does not work as well in the case of data languages. In fact, even in a simple logic like first-order logic (FO) with equality test and ordering over string positions, the satisfiability of a formula is already an undecidable problem.

A better idea is proposed in [BMS+06], where decidability for first-order logic over data is studied in detail. If formulae are allowed to have at most two variables, checking the satisfiability of a first-order formula over data strings is proven to be decidable. Moreover, even with three variables the problem already is undecidable. The decidability results are then extended to more powerful logics with the same limitation on the number of variables. Finally, they propose a new model called *data automata* which they prove to be equivalent to one of the decidable logics. Data automata inspect the input in two passes using a transducer over the labels of the string and an NFA over chunks of the output of the transducer.

Data automata are limited by their difficult usage. In [BS07], *class memory automata* are introduced as an equivalent variant of data automata. Class memory automata perform operations similar to data automata but in a single pass. The new model is also compared to previous models, such as register automata, and several decision problems are studied. To mention a different approach, an algebraic characterization of some classes of data languages is presented in [BPT03]. In section 2, we present several of these models over string data languages and summarize their expressiveness and complexity results.

Just as there are models for data strings, there are proposed models for *tree data languages* [CK98, KT08]. While for the string case the area is quite advanced, tree data languages are still not well understood. For languages over finite alphabets, natural extensions of string models have led to positive results with the addition of a hierarchical structure. As an example, all the decidability results for FO and MSO over string languages still hold for tree languages and, with small modifications, even for transducers. However, in the case of data languages, this approach does not work. When we extend two-variable first-order logic to work on data trees, checking whether a given formula is satisfiable becomes a difficult problem. Moreover models like register and pebble automata do not extend naturally to trees.

The only positive result, so far, is that satisfiability for a small fragment of two-variable FO over trees is decidable [BDM+06] (section 3). A variant of class memory automata that works over trees has been proposed in [BL12]. However most of the results are negative because the model is too expressive. Otherwise, the area of data tree languages is still at an embryonic stage.

There has also been some progress in the area of transducers over infinite alphabets [VHL+12]. However, no good model (even in logic) has been proposed yet for data transducers. In [VHL+12], a new approach for representing languages and transductions over infinite alphabets is presented. The approach separates the theory of the data from the theory of the automata. The new models are called *symbolic automata* and *symbolic transducers*. In this setting, expressiveness is limited, but the set of operations that can be performed on the input data is not restricted to equality. Instead, the input (and output in the case of transducers) alphabet is equipped with a decidable theory over it. If the theory is decidable in polynomial time, several problems such as checking equivalence, computing pre-images, etc are also decidable in polynomial time.

At the end of the paper (section 4) we give a summary of the results and explore future research directions and open problems in the field of data languages.

2 String Data Languages

In this section we present several models of automata and logics over data strings. Their expressiveness, closure properties, and their decision properties are then compared. Based on this analysis, we discuss which ones are better candidates to be regular models.

2.1 Basic Definitions

A *datum* $\mathbf{d}$ is a pair (l, d) where l is a label from a finite alphabet $\mathbb{L}$ and d is an element (data) of an infinite alphabet $\mathbb{D}$ (the data alphabet). The infinite alphabet $\mathbb{L} \times \mathbb{D}$ is denoted with Σ. A *data string* is a string over an infinite alphabet Σ. Formally a data string w is a sequence $(l_1, d_1) \ldots (l_n, d_n)$, where each $(l_i, d_i) \in \Sigma$ and $n \geqslant 0$. The length $|w|$ of w is n and we write $dom(w)$ for the set $\{1, \ldots, n\}$ of positions in w. For $i \in dom(w)$, we also write $val_w(i)$ for (l_i, d_i). A *string data language* is a set of data strings[1].

Without loss of generality, since we will be dealing with two-way automata, we delimit input strings with two special symbols, $(\rhd, \rhd), (\lhd, \lhd) \notin \Sigma$ for the left and the right ends of the the string. Notice that $\rhd$ and $\lhd$ do not appear in $\mathbb{L} \cup \mathbb{D}$. Throughout the paper we often just write $\rhd$ for$(\rhd, \rhd)$ (same for $\lhd$). Given a string $w = \rhd v \lhd$, where $v \in \Sigma^*$, the positions of $\rhd$ and $\lhd$ will be 0 and $|v| + 1$ respectively. We write $dom^+(w)$ for the extended set $\{0, \ldots, |v| + 1\}$ of positions. We extend the $val_w(\cdot)$ function in the natural way: $val_w(0) = \rhd$ and $val_w(|v| + 1) = \lhd$. For consistency with the pair notation, $\rhd$ and $\lhd$ will actually be considered of the form $(\rhd, \rhd)$ and $(\lhd, \lhd)$.

2.2 Models

In the following we give semi-formal definitions of several models of automata and logics for data languages. The reader can find the formal definitions in the references. This list is not complete, but it tries to capture the historically interesting models in the attempt to define regular data languages.

2.2.1 Register Automata

A *register automaton* (RA) [NSV04, DL09] is a finite state machine equipped with a finite set of registers. Registers hold values from an infinite domain $\mathbb{D}$. When processing an input symbol $(l, d) \in \Sigma$, a RA compares d (checks for equality) with the values in the registers. Based on the comparison it can store the current symbol in one of the registers. The only possible operations that can be performed on a register are: 1) checking equality with the data in the current input, and 2) updating the register to the current data. When reading the current symbol the head of the automaton can move left, right or stay in the same position and the current state can be updated.

In a RA transitions are of the following forms:

- $(i, q, l) \to (q', m)$, or
- $(q, l) \to (q', i, m)$.

[1] In [NSV04] data strings are over $\mathbb{D}$. Even though this model might seem simpler, it does not naturally extends string languages and it does not have a clear counter part in practical applications like XML processing. Moreover, no properties about the underlying label language can be expressed.

q and q' are states, i refers to the *ith* register and $m \in \{\text{stay}, \text{left}, \text{right}\}$ is the direction in which the head of the automaton moves when performing the transition.

Informally, at every point, a transition of the form $(i, q, l) \to (q', m)$ apply if the current data value is stored in the *ith* register and a transition of the form $(q, l) \to (q', i, m)$ apply if the current data value is not stored in any register. In this case the current data value is stored in register i.

An automaton starts processing a data string with an initial register assignment that assigns some starting data values to the registers. To allow the automata to recognize the symbols $\triangleright, \triangleleft$, they are always contained in the initial register assignment. Configurations of the RA will be triplets containing the current position of the string, the current register assignment and the current state. A data string w is accepted if the automaton reaches a configuration in which the current state is a final state.

A register automaton is *deterministic* if in any configuration at most one transition applies. An automaton is *one-way* if there are no left transitions.

Example 1 *Let L_0 be the following language of data strings. A data string w is in L_0 iff there exists two positions j and $j+1$ in w, such that j and $j+1$ contain the same data value. We define a one-way, deterministic, one-register* RA *$\mathcal{R}$ that recognizes L_0. $\mathcal{R}$ is a one-register automaton and is defined as follows:*

- *the set of states is $\{q_0, q_1, q_f\}$, and q_f is the only final state,*
- *the register* 1 *contains the left-mark $\tau_0(1) = \triangleright$, and*
- *$\mathcal{R}$ has the following transitions: for every $l \in \mathbb{L}$*
  1. $(1, q_0, \triangleright) \to (q_1, \text{right})$;
  2. $(1, q_1, l) \to (q_f, \text{stay})$;
  3. $(q_1, l) \to (q_1, 1, \text{right})$.

Notice that when the head reaches $\triangleleft$, no transition applies since $\triangleleft \notin \mathbb{L}$. At every point the register contains the data value d of the previous positions. If the current position also contains d (2), $\mathcal{R}$ goes to the final state. If the current position holds a value d' different from d (3), the head moves right and the register is updated to d'.

An *alternating register automaton* $\mathcal{R}$ keeps a set of so called *universal states.* Usually, when an automaton is in a configuration with state q, and is reading a data string w, we say that w is accepted from the current configuration there exists a sequence of transitions that leads the automaton to a final states. In the case of alternating automata this definition is slightly different. When the alternating RA, while reading a data string w, is in a configuration with a universal state q, and a set of transitions applies, we say that w is accepted from the current configuration if it is also accepted from *all* the configurations reached applying one of the possible transitions. If the current configuration is in a state q that is not universal, and a set of transitions applies, we say that w is accepted from the current configuration if it is also accepted from *one of the* the configurations reached applying one of the possible transitions. Whenever q is a final state, w is accepted from the current configuration.

We introduce some notation to indicate the different kinds of automata and the class of languages they define. We use the notation wC-RA to refer to the different classes of RA, where $w \in \{1, 2\}$

stands for 1 or 2-way, and $C \in \{\mathrm{D}, \mathrm{N}, \mathrm{A}\}$ stands for deterministic, non-deterministic and alternating respectively. We use the same names to express a formalism and its corresponding class of definable languages. For example, we write 1D-RA $\subseteq$ 2D-RA to say that the class of languages accepted by a 1D-RA is included in the class of languages accepted by a 2D-RA.

2.2.2 Pebble Automata

Pebble automata (PA) [NSV04] are finite state automata with a finite ordered set $\{1, \ldots, k\}$ of pebbles where every pebble points to a position in the string. Besides ordinary actions, when reading an input, a pebble automaton can add a new pebble to the string or lift the current pebble from the string. These two actions must respect a stack discipline; we can lift the *ith* pebble only if the $(i+1)th$ pebble is not present on the string, and we can add the *ith* pebble only if all the pebbles $\{1, \ldots, i-1\}$ are present on the string. The highest pebble present on the string acts as the head of the automaton. A transition depends on the current state, the current set of pebbles placed on the string, and the highest pebble placed on the string. With a transition the automaton can change the current state and update the pebbles by moving the head left or right, lifting the current pebble, or adding a new pebble. When the *ith* pebble is removed, the $(i-1)th$ pebble, becomes the new head.

Transitions are of the form $(i, q, l, P, V) \to (q', m)$ where i represents the *ith* pebble, q is the current state, l the label being read, $P, V \subseteq \{1, \ldots, i-1\}$ are subsets of pebbles, and

$$m \in \{\text{stay}, \text{left}, \text{right}, \text{place-new-pebble}, \text{lift-current-pebble}\}.$$

Informally, a transition $(i, q, l, P, V) \to (q', m)$ applies if pebble i is the current head, i is placed on the position j in the string containing the value $(l, d) \in \Sigma$, q is the current state, V is the set of pebbles whose position contains the data value d, and P is the set of pebbles whose position is j. When the transition applies the current state is updated to q' and the head of the automaton (the highest pebble) moves based on the value of m. If $m =$ place-new-pebble the $(i+1)th$ pebble is placed in the same positions of the *ith* pebble. If $m =$ lift-current-pebble the *ith* pebble is lifted and the $(i-1)th$ pebble becomes the head of the automaton. In the cases of stay, left, right, the position of the pebble remains the same, gets decremented, gets incremented respectively.

The notions of *accepting language*, *deterministic*, *two-way* and *alternating* are defined in the same way as before. A possible alternative to the previous model places new pebbles at position 0 in the string instead of at the position of the most recent pebble. This alternative is not relevant in the case of two-way pebble automata, but is crucial in the one-way case. The model we presented before is referred to as *weak* pebble automata, while the alternative one is called *strong* pebble automata.

We use the notation wC-PA to refer to the different variants of PA, where $w \in \{1, 2\}$, and $C \in \{\mathrm{D}, \mathrm{N}, \mathrm{A}\}$ have the same meanings as for RA. To indicate *weak* and *strong*, when it makes a difference, we prefix the notation with W and S, respectively. Again, we use the same names to express a formalism and the corresponding class of definable languages.

Example 2 *We consider the language L_0 of Example 1. We define a one-way, deterministic, two-pebble* PA $\mathcal{P}$ *that recognizes L_0. $\mathcal{P}$ is defined as follows:*

- *the set of states is $\{q_{\triangleright}, q_1, q_2, q_r, q_f\}$, with q_f the only final state,*
- *the first pebble is on the first position in the data string,*

- *$\mathcal{R}$ has the following transitions; for every $l \in \mathbb{L}$*

  1. $(1, q_1, l, \emptyset, \emptyset) \to (q_r, \text{place-new-pebble})$;
  2. $(1, q_r, l, \emptyset, \emptyset) \to (q_1, \text{right})$;
  3. $(2, q_r, l, \{1\}, \{1\}) \to (q_2, \text{right})$;
  4. $(2, q_2, l, \emptyset, \{1\}) \to (q_f, \text{stay})$;
  5. $(2, q_2, l, \emptyset, \emptyset) \to (q_r, \text{lift-current-pebble})$.

The automaton uses pebbles 1 and 2 to perform the comparison on adjacent positions. $\mathcal{P}$ is in state q_1 if only pebble 1 is present and $\mathcal{P}$ is going to start a sequence of transitions that will check if the data in i (the position currently in pebble 1) is the same as that in position $(i + 1)$. Starting from state q_1 the automaton places a new pebble (1) and moves the head of the current pebble (i.e. pebble 2) to position $(i + 1)$ (3). Now that the positions i and $(i + 1)$ are stored in pebbles 1 and 2, $\mathcal{P}$ checks if the value in position $(i + 1)$ is the same as the value in position i (4-5); if the value is the same (4), $\mathcal{P}$ goes to an accepting state; otherwise (5), $\mathcal{P}$ lifts the current pebble and moves the head to position $(i + 1)$ (2) to restart the sequence of transitions.

2.2.3 Data Automata

Data automata [BMS+06, BS07] were introduced as an extension of 1N-RA with better connections to logic. RAs and PAs are only able to express properties about adjacent positions, but not about positions in the same class (i.e. positions containing the same data value). A *class* of w is a maximal subset C of $dom(w)$, such that all the positions in C hold the same data value. We consider $dom(w)$ instead of $dom^+(w)$ because the automaton is one way. Given a data string $w = (l_1, d_1) \dots (l_n, d_n)$ and a class $\mathsf{C} = \{x_1, \dots, x_k\}$ of w where $x_1 < \dots < x_k$, we define the *class string* of C (we write $\mathsf{cs}(\mathsf{C})$) to be the string $l_{x_1} \dots l_{x_k}$.

A data automaton runs on data strings in two passes. During the first pass a letter-to-letter string transducer simply changes the label of each position without affecting the data. The second pass is over the result of this translation: an NFA runs on each sequence of letters having the same data value (class strings).

A letter-to-letter string transducer, from Γ_1 to Γ_2, is a finite state machine that, when reading a input symbol from Γ_1, outputs a symbol over a finite output alphabet Γ_2. Formally a non-deterministic letter-to-letter string transducer $\mathcal{T}$ from Γ_1 to Γ_2 is a tuple (Q, q_0, F, T) where Q is a set of states, $q_0 \in Q$ is an initial state, $F \subseteq Q$ is a set of final states, and $T \subseteq Q \times \Gamma_1 \times Q \times \Gamma_2$ is a set of transitions. The semantics is the one expected: a transition (q, a, q', b) applies if $\mathcal{T}$ is in state q, and the next input symbol is a; if the transition is taken, $\mathcal{T}$ goes to state q' and outputs b. A run of $\mathcal{T}$ on a string w is accepting if when reading w starting in q_0 the transducer ends in a state $q \in F$. Given a string $s = a_1 \dots a_n \in \Gamma_1^*$, we use $\mathcal{T}(s)$ to denote the set of possible outputs of $\mathcal{T}$ on s.

A *data automaton* $\mathcal{D}$ over the alphabet $\Sigma = \mathbb{L} \times \mathbb{D}$ is then a triple $(\mathcal{T}, \mathcal{A}, \Gamma)$ where $\mathcal{T}$ is a non-deterministic letter-to-letter string transducer (called the *base automaton*) from $\mathbb{L}$ to Γ, and $\mathcal{A}$ is an NFA (called the *class automaton*) over Γ. In the following we usually omit Γ when it is clear from the context.

A data automaton $\mathcal{D}$ accepts a data string $w = (l_1, d_1) \dots (l_n, d_n)$, iff there exists a string $b_1 \dots b_n$ in $\mathcal{T}(l_1 \dots l_n)$, such that, for every class $\mathsf{C} = x_1, \dots, x_k$ of w, the class automaton $\mathcal{A}$ accepts the string $b_{x_1} \dots b_{x_k}$.

Example 3 *Consider the language of data strings L_{dif}. A data string w belongs to L_{dif} if all the positions of w contain different data values. We define a data automaton $\mathcal{D} = (\mathcal{T}, \mathcal{A})$ that recognizes L_{dif}. $\mathcal{T}$ simply translates every symbol in $\mathbb{L}$ to the symbol 0 ($\Gamma = \{0\}$). The class automaton $\mathcal{A}$ checks that every class string is equivalent to 0. We omit the formal definition. Notice however that this language cannot be defined by a register automaton [Seg06].*

We can also define a data automaton $\mathcal{D}'$ that accepts the complement of L_{dif}. A data string w belongs to $\overline{L_{dif}}$ if there exists two distinct positions with the same data value. $\mathcal{D}' = (\mathcal{T}, \mathcal{A})$ works in the following way. $\mathcal{T}$ non-deterministically selects two positions and outputs 1 when reading them and 0 when reading all the other positions ($\Gamma = \{0, 1\}$). The class automaton $\mathcal{A}$ checks that every class string contains either no 1s or exactly two 1s. We omit the formal definition.

2.2.4 Class-Memory Automata

The main limitation of data automata is that they are hard to use in practice. Moreover, no notion of determinism is defined for them. In [BS07] *class-memory automata* (CMA) were introduced as a simplification of data automata that overcomes these problems.

A CMA $\mathcal{C}$ runs once, left to right, over the input data string. As usual it has a set of states and an initial state. A CMA has two kinds of accepting states F_L and F_G. They are respectively called local and global accepting states. Transitions are of the form $(q, l, \bot) \to q''$ or $(q, l, q') \to q''$. Informally the first kind of transition applies when $\mathcal{C}$ reads a data value d for the first time and it is are in state q. The second kind of transition applies when $\mathcal{C}$ is in state q, the current symbol is of the form (l, d), and the previous time $\mathcal{C}$ read a symbol with data value d the transition moved to state q'. This mechanisms allows the CMA to process class strings. A data string is accepted if the final state reached by $\mathcal{C}$ is a globally accepting state and for every data value d in the string, the last time a symbol with data value d was processed by $\mathcal{C}$ the transition moved to a locally accepting state.

A CMA is deterministic if it doesn't have two transitions with the same left-hand side. We denote them with DCMAs.

Example 4 *We construct a* CMA *$\mathcal{C}$ that recognizes L_{dif}, the language presented in Example 3. The set of states Q is $\{q_0, q_1, q_2\}$, with q_0 initial state. The set of globally accepting states is $F_G = Q$. This is due to the fact that we do not care about labels since the language does not describe any global property. The set of locally accepting states F_L will be the one representing whether a particular datum has been repeated or not. Informally, at every point in a run, for every data value d, $\mathsf{f}(d)$ will contain: 1) $\bot$, if d has not appeared in the string yet, 2) q_1, if d has appeared exactly once, 3) q_2, if d has appeared more than once. F_L will then be the set $\{q_1\}$. The transition relation is now easy to define. For every $l \in \mathbb{L}$, for every $q \in Q$, we will have:*

- $\delta(q, l, \bot) \to q_1$;
- $\delta(q, l, q_1) \to q_2$; *and*
- $\delta(q, l, q_2) \to q_2$.

It is easy to see that $\mathcal{C}$ recognizes L_{dif}.

An example of language considering both a local and a global property would be the following. Define L^a_{dif} as the language where all the positions with label a have different data values. The transducer of a DA, *and the labels in the transition function of the* CMA *are necessary to identify the positions labelled with a. We leave the encoding of this language as an exercise to the reader.*

2.2.5 FO, FO^2 and MSO over Data

In this section we leave the realm of automata and we concentrate instead, on declarative models like logics. We define *First-Order Logic* (FO) and *Monadic Second-Order Logic* (MSO) over data strings. For a brief introduction to first and second-order logic (over data structures) see [Tho96, Dav04].

First-Order Logic. As before, we are given an alphabet $\Sigma = \mathbb{L} \times \mathbb{D}$. Given a data string w, in first-order logic over strings, variables range over positions of w. For every $l \in \mathbb{L}$ there exists a predicate $l(x)$ denoting that a position x contains a label l. We are also allowed to check if two variables contain the same position with the predicate $x = y$. Given these elementary predicates we can build first-order formulae by means of logical connectives. Formally formulae are defined by the following grammar:

$$\phi ::= a(x) \mid x = y \mid \exists x.\phi \mid \phi \vee \phi \mid \neg\phi$$

A data string can then be seen as a model for a formula ϕ in this logic. As an example, the formula $\psi := \forall x(a(x))$ describes the strings where all the positions are labelled with the symbol a. We write FO to denote first-order logic over strings.

Let FO$(\sim, <, +1)$ be FO enriched with the following atomic binary predicates: $x \sim y$, $x < y$, and $x = y + 1$. The predicate $x \sim y$ holds when the positions x and y contain the same data value, $x < y$ holds when position x occurs strictly before position y, and $x = y + 1$ holds when x is the position right after y. Given a formula ϕ we say that a string w *satisfies* ϕ, if w is a model of ϕ. The language of ϕ (denoted by $L(\phi)$) is the set of data strings that satisfy a formula ϕ. A formula satisfied by at least one data string is called *satisfiable* (alternatively its language is non-empty).

FO does not impose any restriction on the number of variables appearing in formulae. We write FO^v to denote FO with at most v variables. The formula ψ above is in FO^1.

Example 5 *Consider the formula $\phi_1 := \exists x \exists y(y = x + 1 \wedge y \sim x)$ in* FO^2*. The language ϕ_1 is L_0, the same language accepted by the register automaton in example 1.*

When we want to consider a variant of the logic in which we drop some of the predicates we just omit them from the signature. For example FO$(\sim, +1)$ is the same as FO$(\sim, <, +1)$ without the $<$ predicate. The number of variables in a formula is critical for both decidability and expressiveness. As an example consider the following case.

Proposition 6 FO$^2(\sim, <) \subset$ FO$^2(\sim, <, +1)$*, while, for all $v > 2$,* FO$^v(\sim, <) =$ FO$^v(\sim, <, +1)$.

This result is due to the fact that $y = x + 1$ is equivalent to the following predicate that requires three variables:

$$x < y \wedge \forall z(x < z \implies (z = y \vee y < z))$$

Before moving on, we define another variant of FO. We write FO$(\sim, <, +\omega)$ to denote FO with the predicates $x \sim y$, $x < y$, and $x = y + k$ where $k \in \mathbb{N}$. The first two predicates are the same as before, while $x = y + k$ holds when y is k position apart from x (formally, if x represents the position i, and y the position $i + k$). Of course any given formula in this logic uses only a finite number of predicates of the form $+k$.

Second-Order Logic. In monadic second-order logic over strings (MSO) quantification over variables and also unary predicates (sets of variables) is allowed. However, quantification over $\mathbb{D}$ is not allowed. Formally first-order logic is extended with second-order variables $X, Y, \ldots$ representing sets of positions in a string, and the atomic predicate $x \in X$, meaning that the position in x belongs to the set of positions X. The adjective "monadic" refers to the fact that we can only quantify over unary predicates and not over relations. This restriction still allows to quantify over sets, which can be represented as unary predicates. Formulae in MSO are generated by the following grammar:

$$\phi ::= a(x) \mid x = y \mid x \in X \mid \exists x.\phi \mid \exists X.\phi \mid \phi \vee \phi \mid \neg\phi$$

We define MSO$(\sim, <, +1)$ to be MSO enriched with the predicates over positions and data. The following is a classical result [Tho96].

Proposition 7 MSO$(\sim, <, +1)$ *is strictly more expressive than* FO$(\sim, <, +1)$.

Example 8 *Providing an interesting example in* MSO *that is easy to understand is difficult. We present a simple language that does not actually need the expressiveness of* MSO*, however it makes* MSO *easier to understand. Let's consider the formula*

$$\phi_M := \exists X(\forall x((x \in X \implies a(x)) \wedge (x \notin X \implies \neg a(x)) \wedge (x \in X \implies \forall y(y \in X \implies x \sim y))))$$

The formula describes a language L_M in which all the positions labelled with the symbol a contain the same data value. Notice that the same language can be expressed in first-order logic with the formula: $\phi_F := \forall x(a(x) \implies (\forall y(a(y) \implies x \sim y)))$.

An interesting restriction of MSO is EMSO. In EMSO we only allow quantifiers over unary predicates to appear at the beginning of the formula and to be existential. Formally a formula ϕ is in EMSO if it is of the form $\exists X_1 \ldots \exists X_n(\psi)$ where ψ contains no quantifiers over second-order variables. We denote with EMSO$(\sim, <, +1)$ the corresponding logic enriched with data predicates, and with EMSO$^2(\sim, <, +1)$ the two-variable variant. The restriction on the number of variables only applies to first-order variables (not to the set variables).

We finally consider the logic EMSO$^2(\sim, <, +1, \oplus 1)$. $\oplus 1$ is the successor predicate for positions belonging to the same class. As we said in section 2.1, a *class* is a maximal set of positions in a data string with the same data value (i.e. an equivalence class of the relation $\sim$). We have that $x = y \oplus 1$ holds iff there exists a class with positions $i_1 < \ldots < i_k$, such that $y = i_j$ and $x = i_{j+1}$ for some $j < k$. $\oplus 1$ is called the *class successor*. Informally, the predicate holds for two positions containing the same data value d such that all the positions in between contain elements different from d.

2.3 Expressiveness

As mentioned earlier, we are looking for a model that fits the notion of regularity. We presented several models, but we cannot say anything about their properties just from their definitions. In this section we compare their expressiveness in terms of the set of languages they can define. The most important expressiveness relations are summarized in figure 1.

2.3.1 Logics

We start by summarizing the expressiveness results of the different logics we have introduced. The first notable result is that the $<$ operator cannot be simulated in $\mathrm{FO}(\sim, +1)$. However, it can be simulated by MSO.

Proposition 9 ([Dav04]) *The following relations hold:*

- $\mathrm{FO}(\sim, +1) \subset \mathrm{FO}(\sim, <, +1)$,
- $\mathrm{FO}^2(\sim, +1) \subset \mathrm{FO}^2(\sim, <, +1)$, *and*
- $\mathrm{MSO}(\sim, <, +1) = \mathrm{MSO}(\sim, +1)$

Increasing the number of variables in both FO and MSO adds expressiveness. This result is interesting not only in terms of expressiveness. Indeed, as we will see later, both FO and MSO become undecidable when more than two variables are allowed.

Proposition 10 ([Dav04]) *For all* $v \in \mathbb{N}$,

- $\mathrm{FO}^v(\sim, <, +1) \subset \mathrm{FO}^{v+1}(\sim, <, +1)$,
- $\mathrm{EMSO}^v(\sim, <, +1) \subset \mathrm{EMSO}^{v+1}(\sim, <, +1)$,
- $\mathrm{EMSO}^v(\sim, <, +1, \oplus 1) \subset \mathrm{EMSO}^{v+1}(\sim, <, +1, \oplus 1)$, *and*
- $\mathrm{MSO}^v(\sim, <, +1) \subset \mathrm{MSO}^{v+1}(\sim, <, +1)$.

We then list some results for the logics with the predicates $+\omega$ and $\oplus 1$. While the predicate $+\omega$ does not add expressiveness in general, it cannot be expressed with the two-variable restriction. Moreover, as expected, the predicates $\oplus 1$ and $\omega 1$ are orthogonal in expressiveness in the two-variable case.

Proposition 11 ([Dav04]) *The following relations hold:*

- $\mathrm{FO}^2(\sim, <, +1) \subset \mathrm{FO}^2(\sim, <, +\omega)$,
- $\mathrm{EMSO}^2(\sim, <, +1) \subset \mathrm{EMSO}^2(\sim, <, +\omega)$, *and*
- $\mathrm{EMSO}^2(\sim, <, +\omega) \nsubseteq \mathrm{EMSO}^2(\sim, <, +1, \oplus 1)$ *and* $\mathrm{EMSO}^2(\sim, <, +1, \oplus 1) \nsubseteq \mathrm{EMSO}^2(\sim, <, +\omega)$.

In general EMSO is more expressive than FO. This result holds also in the two-variable case.

Proposition 12 ([Dav04]) *The following relations hold:*

- $\mathrm{FO}^2(\sim, <, +1) \subset \mathrm{EMSO}^2(\sim, <, +1)$, *and*
- $\mathrm{FO}^2(\sim, <, +\omega) \subset \mathrm{EMSO}^2(\sim, <, +\omega)$.

We now compare the previous logics and the automata models we proposed.

2.3.2 Register Automata and Logic

The first surprising result is that deterministic register automata are in some sense *orthogonal* to FO and MSO in that they cannot express properties that are definable in first-order logic, but they can express properties that are not definable in monadic second-order logic.

Proposition 13 ([NSV04]) *There exists a language expressible by a* 2D-RA *that cannot be expressed in* MSO$(\sim, <, +1)$.

Since we are looking for a regular model, we would like our automata representation of data languages to have a counter part in logic. The previous result shows that two-way register automata are probably not a "good" model. One can wonder what happens with one-way RAs. We have that 1N-RAs are strictly less expressive than FO$^2(\sim, <, +1, \oplus 1)$. Things seem to get better, but unfortunately, the inclusion is strict (i.e. if the two models are equivalent). Our last hope is that RAs are at least equivalent to a weaker logic, for example FO$^2(\sim, <, +1)$. However this is not the case.

Proposition 14 ([BMS$^+$06]) *The language L_{dif} can be expressed in* FO$^2(\sim, <, +1)$ *but not by a* 2A-RA.

This result is really strong since it takes into consideration two-way alternating register automata, the most expressive model of RA that we have considered. Register automata do not seem to cope well with logic. Moreover, the different variants, deterministic, non-deterministic and alternating are all increasingly more expressive: 2D-RA $\subset$ 2N-RA $\subset$ 2A-RA. This result shows that the model is not determinizable. It is worthy pointing out that the strict inclusions only hold if we assume that the complexity classes LogSpace, NLogSpace and PTime are all different in expressiveness.

2.3.3 Pebble Automata and Logic

After the discouraging results on register automata we now move to pebble automata. This class seems more comparable to logic than the previous one. PAs nicely fit between first and second-order logic.

When comparing PAs to first-order logic, we have that even the weakest form of PA is more expressive than first-order logic: FO$(\sim, <, +1) \subset$ 1D-PA. Moreover, even the strongest form of PA is included in monadic-second order logic: 2A-PA $\subset$ MSO$(\sim, <, +1)$. The strictness of the containment, again, holds only under standard complexity assumptions. Even though we have not found a logic that is exactly equivalent to some form of PA, we showed that they fit between first and second-order logic. When comparing them to logic, PAs seem to behave in a *regular* way.

The following proposition shows that PAs are a robust model, in the sense that most of their variants are equivalent.

Proposition 15 ([NSV04]) 2D-PA*,* 2N-PA*,* strong 1D-PA *and* strong 1N-PA *are all equivalent models.*

PAs have appealing expressiveness properties. However, as mentioned before, a regular model should also satisfy other properties. In this subsection we concentrated more on expressiveness. In the next one we will look at decision problems and see that PAs do not behave as well in terms of decidability.

2.3.4 Class-Memory Automata, Data Automata and Logic

Data automata were introduced to overcome the incapability of register automata to describe *local properties*. Therefore they increase 1N-RA expressiveness while preserving decidability. DAs also have a logical counter part, $\text{EMSO}^2(\sim, <, +1, \oplus 1)$, and are equivalent to CMAs.

As we said earlier, CMAs are easier to use then DAs. This result makes CMAs ideal candidates for the role of regular data languages. CMA and DA are the first automata models over data strings with a logic characterization. Moreover, as we will see later, CMAs define the 'biggest' class of data languages for which emptiness is decidable. For completeness, we also mention that CMAs are not determinizable: $\text{DCMA} \subset \text{CMA}$.

We summarize the expressiveness results in Figure 1. Some other expressiveness results are added for sake of completeness. In the figure every arrow represents a strict inclusion.

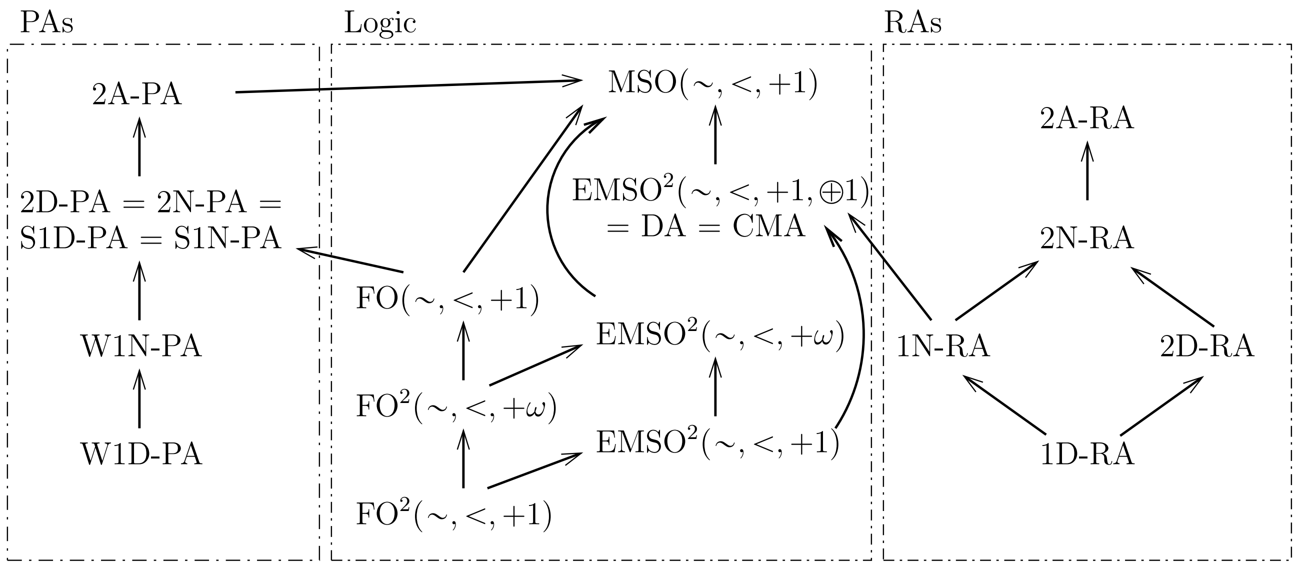

Figure 1: Expressiveness relations. All the arrows indicate strict inclusions.

2.4 Decision Problems

In the this section we discuss decision problems for the models we presented.

We start from the results on RAs and PAs [KF94, NSV04]. Given a 1N-RA $\mathcal{R}$, checking whether a data word w is accepted by $\mathcal{R}$ (the membership problem) can be decided in polynomial time. Checking whether a 1N-RA does not accept any string (emptiness) is decidable. The same problems is undecidable for 2D-RA. Checking whether a 1N-RA accepts every string (universality) is undecidable. The problems of inclusion and equivalence are undecidable in general. However if both the 1N-RAs we are comparing have at most 2 registers checking equivalence becomes decidable.

As we mentioned in section 2.3, the decidability results for PAs are discouraging. Indeed, even for 1D-PA the emptiness problem and the universality problem are already undecidable. This results immediately imply that equivalence and inclusion for 1D-PA are also undecidable.

Decidability results for logics are more interesting and well-understood.

	Emptiness	Universality	Inclusion	Equivalence
1N-RA	Dec.	Dec. w/ 2 reg. Undec.	Dec. w/ 2 reg. Undec.	Dec. w/ 2 reg. Undec.
2D-RA	Undec.	Undec	Undec.	Undec.
weak 1D-PA	Undec.	Undec	Undec.	Undec.
$\mathrm{FO}^2(\sim, <, +1)$	Dec.	Dec.	?	?
$\mathrm{FO}^3(\sim, <, +1)$	Undec.	Undec	Undec.	Undec
$\mathrm{FO}^2(\sim, <, +\omega)$	Dec.	Dec.	?	?
$\mathrm{EMSO}^2(\sim, <, +1)$	Dec.	?	?	?
$\mathrm{EMSO}^2(\sim, <, +1, \oplus 1)$	Dec.	?	?	?

Table 1: Decidability results for models over data strings.

Proposition 16 ([BMS+06]) *The satisfiability (emptiness) problem for the following logics is decidable:*

- $\mathrm{FO}^2(\sim, <, +1)$,
- $\mathrm{FO}^2(\sim, <, +\omega)$, *and*
- $\mathrm{EMSO}^2(\sim, <, +1, \oplus 1)$.

In general satisfiability of $\mathrm{FO}(\sim, <, +1)$ (i.e. emptiness of the underlying data language) is undecidable. However, a clearer boundary for decidability has been established. In fact, simply allowing FO to use three variables leads to undecidability. This property strengthens the result of proposition 16.

We finally give the precise complexity of the membership problem for CMAs. The problem of checking whether a word w is accepted by a CMA is decidable and NP-Complete [BS07].

The decidability results are summarized in table 1.

2.5 Closure Properties

In table 2.5 we present the closure properties of the models we defined. The results are from [NSV04, BMS+06, BS07].

Looking at the table we can see that very few models have been carefully studied. However, whenever a model is clearly characterized with a logic, most of the closure properties come naturally. Notice that EMSO is in general not closed under complement since when negating a formula the existential quantifiers become universal.

2.6 Regularity

In this section we discuss which models better suit the definition of regularity we gave in section 1. None of the models we presented has all the nice properties we can find in regular string languages. Unfortunately, we don't know if a model with all such properties even exists for data languages.

	$\cap$	$\cup$	Concat.	Compl.	Kleene-Star
1N-RA	?	?	?	No	?
$\mathrm{FO}^2(\sim,<,+1)$	Yes	Yes	?	Yes	?
$\mathrm{FO}^2(\sim,<,+\omega)$	Yes	Yes	?	Yes	?
$\mathrm{EMSO}^2(\sim,<,+1)$	Yes	Yes	?	?	?
$\mathrm{EMSO}^2(\sim,<,+1,\oplus 1)$,DA,CMA	Yes	Yes	Yes	No	No

Table 2: Closure properties for models over data strings. Concat. stands for concatenation and Compl. for complement.

Looking at the diagrams we presented, it is clear that the best candidates to define regular data languages are data automata and class memory automata. We first list their positive features:

1. DAs and CMAs are two equivalent formalisms and they have an equivalent counter part in logic, $\mathrm{EMSO}^2(\sim,<,+1,\oplus 1)$, and

2. their languages are closed under intersection, union and concatenation.

Now we list their negative features:

1. the membership problem for both DAs and CMAs is NP-Complete,

2. CMAs are not determinizable, and

3. their languages are not closed under complement and Kleene-star.

Even though this class of languages is "relatively small", most of its properties are discouraging. However there is still hope in finding a regular model. We have not yet found an automata model that captures $\mathrm{FO}^2(\sim,<,+1)$ or $\mathrm{FO}^2(\sim,<,+1,\oplus 1)$. These classes are both closed under complementation. Several extensions and restrictions of register automata have been studied [Tal04, CK98]. In particular, in [Tal04] they identified a subclass of RA with decidable equivalence. A context-free version of register automata is proposed in [CK98]. This model adds expressiveness at the cost of slower decision procedures. However, it does not compare well with logic.

It is clear from the results in this section (in particular, the NP completeness of CMA's membership) that either a class of languages with all the regular properties does not exist, or such a class is less expressive than CMA.

3 Tree Data Languages

String data languages allow strings to be enriched with data. However in tasks like XML reasoning, this is not enough. Languages like XPath [DSV99] are able to query XML documents performing comparison on data values. In XML documents, data appear attached to tree nodes; therefore we need a model where we can talk about tree structures and data at the same time. When allowing data comparisons, XPath is known to have undecidable emptiness checking. However, there are examples of fragments where tasks involving attribute values are decidable [AFL05, BDF+03, BMS+06].

For the automata approach most of the results are discouraging. [JL08] introduces *alternating register automata* over data trees. Emptiness is proven to be decidable for the automata that can move only down and right, and have at most one register. Every extension in the number of registers or in the control leads to undecidability. In [Fig09, FS11], a *bottom-up* version of such automata is introduced, and the decidability result is pushed further. In [BL12] the whole XPath language is captured by an extension of data automata, which, however clearly leads to an undecidable model.

In this section we present the more interesting result for regularity of data tree languages.

3.1 $\mathrm{FO}^2(\sim, E_\rightarrow, E_\downarrow)$ over Data Trees

In section 2.2.5 we have introduced first and second-order logic over data strings. In this section we consider $\mathrm{FO}^2(\sim, +1)$ applied to data trees.

3.1.1 Data Trees and Logic

We consider unranked, ordered, labelled trees with data values. A *data tree* t over $\Sigma = \mathbb{L} \times \mathbb{D}$ is a tree where every node is a datum $(l, d) \in \Sigma$. In the same way as for data strings, a data tree can be seen as a model for a logic formula. We consider FO. Now variables range over nodes instead of position. Similar to the case of data strings we will have the following predicates. For every $l \in \mathbb{L}$ there is a predicate $l(x)$ denoting that a node x contains a label a. Let $\mathrm{FO}^2(\sim, E_\rightarrow, E_\downarrow, E^*_\rightarrow, E^*_\downarrow)$ be two-variable first-order logic over trees, enriched with the following atomic predicates: $x \sim y$, $E_\rightarrow(x, y)$, $E_\downarrow(x, y)$, $E^*_\rightarrow(x, y)$, and $E^*_\downarrow(x, y)$. The binary predicate $x \sim y$ holds when the nodes x and y contain the same data value. The binary predicate $E_\rightarrow(x, y)$ holds if x and y have the same parent and y is the immediate sibling of x (the next child in order). $E_\downarrow(x, y)$ holds if y is a child of x. $E^*_\rightarrow(x, y)$, and $E^*_\downarrow(x, y)$ are the reflexive transitive closures of $E_\rightarrow(x, y)$, and $E_\downarrow(x, y)$ respectively. We write $\mathrm{FO}^2(\sim, E_\rightarrow, E_\downarrow)$ to indicate the same logic without the last two predicates.

3.1.2 Expressiveness and Decidability

The main decidability result on data trees is the following.

Proposition 17 ([BDM+06]) *The logic* $\mathrm{FO}^2(\sim, E_\rightarrow, E_\downarrow)$ *on unranked trees is decidable in* 3NEXPTIME.

This is the biggest class of tree data languages supported by a logic for which emptiness is decidable. However this bound is not proven. In the same paper, the more expressive logic $\mathrm{FO}^2(\sim, E_\rightarrow, E_\downarrow, E^*_\rightarrow, E^*_\downarrow)$ is also analysed. In the case of data strings, the extended logic $\mathrm{FO}^2(\sim, <, +1)$ is still decidable. When we have data trees, emptiness for this logic is proven to be a 'hard' open problem. In fact, the problem of emptiness for vector addition tree automata [GGS04] is reduced to satisfiability for $\mathrm{FO}^2(\sim, E_\rightarrow, E_\downarrow, E^*_\rightarrow, E^*_\downarrow)$. Emptiness for vector addition tree automata is equivalent to several notorious open problems. Therefore proving decidability for $\mathrm{FO}^2(\sim, E_\rightarrow, E_\downarrow, E^*_\rightarrow, E^*_\downarrow)$ is challenging. To understand better why this is a difficult problem we give some intuition of how hard the proof of proposition 17 is.

Proof Sketch. The proof [BDM+06] proceeds as follows: 1) first, the satisfiability problem for $\mathrm{FO}^2(\sim, E_\rightarrow, E_\downarrow)$ on unranked data trees is reduced to a *puzzle* problem that asks whether a data tree with some properties exists, then 2) a pumping argument on the size of solutions of such puzzle is given, finally, 3) a procedure for looking for such solutions is given using a class of tree automata with decidable emptiness checking.

We start discussing the first step. We do not give the formal definition of the puzzles. However, a puzzle can be seen as a variation of a data automaton over data trees. Solving the puzzle means finding a data tree that has a run over the automaton and has some restrictions on how many times each label in $\mathbb{L}$ can appear in every data class. A formula in $\mathrm{FO}^2(\sim, E_\rightarrow, E_\downarrow)$ can be converted to an instance of the puzzle as follows: a) the original formula in $\mathrm{FO}^2(\sim, E_\rightarrow, E_\downarrow)$ is converted into an equivalent formula in $\mathrm{EMSO}^2(\sim, E_\rightarrow, E_\downarrow)$ in "data normal form" (with some restrictions), b) from a formula in "data normal form" we then compute a puzzle that has solutions iff the formula is satisfiable. The formula is satisfiable iff the puzzle has a solution.

The second step of the proof, the pumping argument, is the hardest. In this step it is shown that if a puzzle admits a solution, it also admits a solution of bounded size. Moreover this size is effectively computable. This steps uses the notion of zone. In a data tree, a *zone* is a maximal connected set of nodes with the same data value. Given a solution of the puzzle, the solution can be transformed into one where only M zones contain more than N nodes. M and N are effectively computable and they only depend on the size of the puzzle. We call the transformed solution an (M, N)-reduced solution. The reduction proceeds by steps. Each step is rather technical but follows the same pattern: zones with 'problematic' properties are identified and they are transferred into other zones by changing their data values. This modification preservers the property of being a solution but decreases the number of 'problematic' zones.

The third and final step is that of finding whether a given puzzle has a (M, N)-reduced solution. The algorithm is by reduction to the emptiness of linear constraint tree automata. *Linear constraint tree automata* (LCTA) is a tree automata T together with a linear constraint on the set of states of T. A run of an LCTA is accepting if, when we instantiate the constraint with the number of times each state appears in the run, the constraint is satisfied. Given a data tree we call the tree with labels and no data its data erasure. Given a puzzle P and numbers M, N, one can compute an LCTA that recognizes the data erasure of the (M, N)-reduced solutions of P. The nice idea of this step is that of reducing a problem over data trees to a problem where no data is considered and therefore using known models.

The complexity obtained this way is 3NExpTime which is possibly not optimal.

3.2 Regularity

The only logic for which satisfiability was proved to be decidable is $\mathrm{FO}^2(\sim, E_\rightarrow, E_\downarrow)$. However, it is too early to talk about regularity, since we do not know any automata model equivalent to $\mathrm{FO}^2(\sim, E_\rightarrow, E_\downarrow)$. Indeed regularity of data tree languages is a real open problem.

Several extensions of RAs, PAs, and DAs have been proposed to deal with data trees [CK98, KT08, BL12], even though none of these extensions addresses the problem of regularity. In [KT08], a complex extension of RAs for trees was introduced, called *unification-based automata* (UBA). Emptiness is decidable for UBA, and the class is robust in the sense that the expressiveness of the bottom-up and top-down versions are equivalent. However the same expressiveness restriction of the RAs over the string case applies.

In [CK98] a pushdown version of register automata has been defined (PDRA). This extension can be used for both string and tree data languages. Emptiness for PDRA is decidable, but the model does not seem to fit in any logic characterization. In [BL12], a complex extension of class-memory automata over data trees is defined. This class has a clear logic counterpart and good closure properties. Unfortunately, the emptiness problem for CMAs over data trees is undecidable, which makes them unlikely to be a good regular model. This class is, however, useful for practical purposes, since it captures a specific fragment of XPath.

4 Conclusions and Open Problems

We presented a variety of models for representing data languages, both for data strings and data trees. Several others have been proposed in literature [BPT03], but we focused our attention on the historically interesting models in the area of regularity of data languages. We started with register and pebble automata, two classic models that were adapted to the data setting. We then explored new ideas such as data automata, class-memory automata and their corresponding logics. Even though the situation is much less favourable than for regular string languages, CMAs have some desirable properties. When comparing them to the other proposed models, CMAs appear to have a better trade-off between decidability and expressiveness.

Then we described how logic and data automata do not extend smoothly to data trees. In this area, the state of the art of regularity is discouraging. The few decidable models are not expressive enough, and automata and logic seem to travel on different tracks. Even for simple classes of tree data languages most of the decision problems are still open, and very few usable models have been proposed and analysed.

The field of regular data languages is still open, and the landscape is not clearly understood. We still need to identify the "right" classes of data languages. Here, we list some interesting open problems related to regularity:

- Is there a variant of RA (section 2.2.1) closed under complementation and with no restriction on the number of registers?
- Is there a version of FO (section 2.2.5) equivalent to one of the variants of RA?
- Is there a logic that is closed under complementation and has decidable equivalence?
- Is satisfiability of $\mathrm{EMSO}^2(\sim, <, +\omega)$ and $\mathrm{EMSO}^2(\sim, <, +\omega, \oplus 1)$ (section 2.2.5) decidable?
- Is there an automata model corresponding to $\mathrm{EMSO}^2(\sim, <, +\omega)$ (section 2.2.5)?
- Is satisfiability of $\mathrm{FO}^2(\sim, E_{\rightarrow}, E_{\downarrow}, E^*_{\rightarrow}, E^*_{\downarrow})$ (section 3) decidable?
- Is there a variant of CMA (section 3) over trees with decidable emptiness?

In this paper, we focused our attention on data strings and data trees. However, several results have been extended to more complex models such as *infinite data strings* [BMS+06]. These extensions are very useful in verification and model checking. One family of models we have not considered, that easily adapts to this setting, is that of temporal logics over data strings [DL09]. These models have good decision procedures but are not relevant for regularity.

Finally, the field of transducers lies completely open for development. The idea of *symbolic transducers* proposed in [VHL+12] is, to our knowledge, the only variant of transducers over infinite alphabets. The framework proposed in [VHL+12] is, however, different from the one we considered. In fact, the set of operations over the data is not restricted to equality. Instead, the set of operations depends on the domain considered. No model has been proposed for *data string transducers* (and *data tree transducers*) and there is still work to be done in the area.

References

[AC11] Rajeev Alur and Pavol Cerný. Streaming transducers for algorithmic verification of single-pass list-processing programs. *SIGPLAN Not.*, 46(1):599–610, January 2011.

[AFL05] Marcelo Arenas, Wenfei Fan, and Leonid Libkin. On verifying consistency of xml specifications, 2005.

[AMN$^+$01] Noga Alon, Tova Milo, Frank Neven, Dan Suciu, and Victor Vianu. Xml with data values: typechecking revisited. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '01, pages 138–149, New York, NY, USA, 2001. ACM.

[BDF$^+$03] Peter Buneman, Susan Davidson, Wenfei Fan, Carmem Hara, and Wang-Chiew Tan. Reasoning about keys for xml, 2003.

[BDM$^+$06] Mikolaj Bojańczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data trees and xml reasoning. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '06, pages 10–19, New York, NY, USA, 2006. ACM.

[BHM02] Ahmed Bouajjani, Peter Habermehl, and Richard Mayr. Automatic verification of recursive procedures with one integer parameter, 2002.

[BL12] Mikolaj Bojanczyk and Slawomir Lasota. An extension of data automata that captures xpath. *CoRR*, abs/1201.0597, 2012.

[BMS$^+$06] Mikolaj Bojanczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin, and Claire David. Two-variable logic on words with data. In *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science*, LICS '06, pages 7–16, Washington, DC, USA, 2006. IEEE Computer Society.

[BPT03] Patricia Bouyer, Antoine Petit, and Denis Thrien. An algebraic approach to data languages and timed languages, 2003.

[BS07] Henrik Bjrklund and Thomas Schwentick. On notions of regularity for data languages. In *In FCT*, pages 88–99, 2007.

[CK98] Edward Y.C. Cheng and Michael Kaminski. Context-free languages over infinite alphabets. *Acta Informatica*, 35:245–267, 1998. 10.1007/s002360050120.

[Dav04] Claire David. Mots et donneés infinis. master's thesis, 2004.

[DL09] Stéphane Demri and Ranko Lazić. Ltl with the freeze quantifier and register automata. *ACM Trans. Comput. Logic*, 10(3):16:1–16:30, April 2009.

[DSV99] Alin Deutsch, Liying Sui, and Victor Vianu. Xml path language (xpath) version 1.0. w3c recommendation, the world wide web consortium. In *Journal of Computer and System Sciences (JCSS) 2007; 73(3):442474*, 1999.

[Fig09] Diego Figueira. Satisfiability of downward xpath with data equality tests. In *Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '09, pages 197–206, New York, NY, USA, 2009. ACM.

[FS11] Diego Figueira and Luc Segoufin. Bottom-up automata on data trees and vertical xpath. In *STACS*, pages 93–104, 2011.

[GGS04] Philippe de Groote, Bruno Guillaume, and Sylvain Salvati. Vector addition tree automata. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science*, LICS '04, pages 64–73, Washington, DC, USA, 2004. IEEE Computer Society.

[HMP92] Thomas Henzinger, Zohar Manna, and Amir Pnueli. Timed transition systems, 1992.

[JL08] Marcin Jurdzinski and Ranko Lazic. Alternating automata on data trees and xpath satisfiability. *CoRR*, abs/0805.0330, 2008.

[KF94] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, November 1994.

[KT08] Michael Kaminski and Tony Tan. Pillars of computer science. chapter Tree automata over infinite alphabets, pages 386–423. Springer-Verlag, Berlin, Heidelberg, 2008.

[NSV04] Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Logic*, 5(3):403–435, July 2004.

[Seg06] Luc Segoufin. Automata and logics for words and trees over an infinite alphabet. In *Proceedings of the 20th international conference on Computer Science Logic*, CSL'06, pages 41–57, Berlin, Heidelberg, 2006. Springer-Verlag.

[Tal04] A. Tal. Decidability for inclusion for unification based automata, 2004.

[Tho96] Wolfgang Thomas. Languages, automata, and logic. In *Handbook of Formal Languages*, pages 389–455. Springer, 1996.

[VHL$^+$12] Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj Bjorner. Symbolic finite state transducers: algorithms and applications. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 137–150, New York, NY, USA, 2012. ACM.