Automata and Logics for Words and Trees over an Infinite Alphabet

Luc Segoufin

INRIA and Université Paris 11
http://www-rocq.inria.fr/~segoufin

Abstract. In a *data word* or a *data tree* each position carries a label from a finite alphabet and a data value from some infinite domain. These models have been considered in the realm of semistructured data, timed automata and extended temporal logics.

This paper survey several know results on automata and logics manipulating data words and data trees, the focus being on their relative expressive power and decidability.

1 Introduction

In many areas there is a need for good abstract models manipulating explicitly data values. We mention two of them here.

In program verification one has to decide statically whether a program satisfies some given specification. Programs may contain several procedures calling each other recursively. Procedures may have parameters and data could be exchanged via the parameters. In program verification, variables over unbounded domains such as integers, arrays, parameters etc. are usually abstracted to finite range domains and configuration graphs of pushdown automata have been used quite successfully in order to model recursive dependencies between procedures. It is then possible to check properties expressed by temporal logics such as LTL or CTL. The modelization using a finite domain has some obvious limitations but it is hard to avoid while remaining decidable. One notable exception is [8] where procedures can have one parameter whose value can range over the integers and the procedures can perform limited arithmetic on that parameter.

In the database context also, most theoretical work on XML and its query languages models XML documents by labeled ordered unranked trees, where the labels are from a finite set. Attribute values are usually ignored. This has basically two reasons, which are not independent. First, the modeling allows to apply automata based techniques, as automata operate on trees of this kind. Second, extending the model by attribute values (data values) quickly leads to languages with undecidable static analysis (see, for instance [1,4,18,27]). Nevertheless, there are examples of decidable static reasoning tasks involving attribute values.

One of them is validation of schema design. A schema contains a structural part and integrity constraints such as *keys* and *inclusion constraints*. In

Z. Ésik (Ed.): CSL 2006, LNCS 4207, pp. 41–57, 2006.

a semistructured model such as XML the structural part is mainly a mechanism for assigning types to nodes of the document tree. It is then natural to ask whether a specification is consistent and whether a set of integrity constraints is minimal or not (*implication problem*). Decidable cases were proposed for XML schemas in [2].

Another one is query optimization and checking whether one query is included into or equivalent to another one. Each of these inclusion tests could be relativized by the presence of a schema. In the context of XML and one of its most popular language XPath several decidable fragments were proposed in [27,4,18,6].

Each of the papers cited above propose a decidable framework where data values are explicitly manipulated. In each of them the decidability was obtained using a non-trivial ad-hoc argument. It is natural to wonder whether there exists a general decidable suitable theoretical framework which could be used to infer all this kind of results. This framework is yet to be discovered and this paper is a modest attempt to gather known results in this direction. We tried to group here interesting models of automata and logical frameworks which could be used in order to code some of the problems mentioned above.

To make this survey finite size we have restricted our attention to an approach that, in our opinion, deserves more attention from the theoretical community. All over this survey we model data values using an infinite domain (like the integers). Moreover our structures have a very simple shape as they are either finite strings or finite trees. More precisely we are given two alphabets over strings and trees, one which is finite and can be used to code names and constants appearing in the programs, in the schemas, or in the queries, and another one which is infinite and which can be used to code data values. We call such structures *data words* or *data trees*.

We present several models of automata and several logics which are evaluated on data words and data trees. We focus on their relative expressive power and on decidability.

When dealing with an infinite domain, like the integers for instance, the allowed arithmetic is a central issue in order to obtain decidability. We avoid this problem here by allowing only the simplest arithmetical operation which is equality test: The only operation that can be performed in our infinite domain is checking whether to values are equal. We will see that this is already enough to make the story interesting.

We have modified slightly several known concepts so that we could present each of them within a uniform framework in order to compare them. We hope that the reader familiar with one of them will not be too much disturbed by this.

This survey contains no proof at all. Those can be found in the references provided all over the paper. We wish we had time to include more references, especially in the verification context.

2 Some Notations

In this paper we consider two kinds of model: *data words* and *data trees*. Let Σ be a finite alphabet of *labels* and D an infinite set of *data values*.

A *data word* $w = w_1 \cdots w_n$ is a finite sequence over $\Sigma \times D$, i.e., each w_i is of the form (a_i, d_i) with $a_i \in \Sigma$ and $d_i \in D$. A *data word language* is a set of data words.

A *data tree* t is a finite unranked, ordered tree where each node is of the form (a_i, d_i) with $a_i \in \Sigma$ and $d_i \in D$. As above a *data tree language* is a set of data trees.

Given a node x of a data tree or a position x of a data word, we denote respectively by $x.d$ and $x.l$ the data value of D and the label of Σ associated to x. The idea is that the alphabet Σ is accessed directly, while data values can only be tested for equality. This amounts to considering words and trees over the finite alphabet Σ endowed with an equivalence relation on the set of positions. With this in mind, given a data word or a data tree w, we will call *class* of w a set of positions/nodes of w having the same data value. Finally, given a data word w (or a data tree t), the *string projection* of w (the *tree projection* of t), denoted by $\textsc{str}(w)$ ($\textsc{tree}(t)$), is the word (tree) constructed from w (t) by keeping only the label in Σ and projecting out all the data values.

For each integer $n \in \mathbb{N}$ we note $[n]$ the set of all integers from 1 to n. Given a data word w we denote its length by $|w|$.

3 Automata

In this section we present several models of automata over data words and data trees. We present them in details in the word case and only briefly discuss the extension to trees.

3.1 Register Automata

Register automata are finite state machines equiped with a finite number of registers. These registers can be used to store temporarily values from D. When processing a string, an automaton compares the data value of the current position with values in the registers; based on this comparison, the current state and the label of the position it can decide on its action. We stress that the only allowed operation on registers (apart from assignment) is a comparison with the symbol currently being processed. The possible actions are storing the current data value in some register and specifying the new state. This model has been introduced in [20] and was later studied more extensively in [28]. We give here an equivalent formalism that fits with data words.

Definition 3.1. *A* k-register automaton $\mathcal{A}$ *is a tuple* (Q, q_0, F, τ_0, T) *where*

- Q *is a finite set of states;* $q_0 \in Q$ *is the initial state;* $F \subseteq Q$ *is the set of final states;*

- $\tau_0 : \{1, \ldots, k\} \to D$ *is the initial register assignment; and,*
- T *is a finite set of transitions of the forms* $(q, a, E) \to q'$ *or* $(q, a, E) \to (q', i)$. *Here,* $i \in \{1, \ldots, k\}$, $q, q' \in Q$, $a \in \Sigma$ *and* $E \subseteq \{1, \ldots, k\}$.

Given a data word w, a *configuration of* $\mathcal{A}$ *on* w is a tuple $[j, q, \tau]$ where $0 \leq j \leq |w|$ is the current position in the word, $q \in Q$ is the current state, and $\tau : \{1, \ldots, k\} \to D$ is the current register assignment. The *initial* configuration is $\gamma_0 := [1, q_0, \tau_0]$. A configuration $[j, q, \tau]$ with $q \in F$ is *accepting*. Given $\gamma = [j, q, \tau]$, the transition $(p, a, E) \to \beta$ *applies* to γ iff $p = q$, $w_j.l = a$ and $E = \{l \mid w_j.d = \tau(l)\}$ is the set of registers having the same data value as the current position.

A configuration $\gamma' = [j', q', \tau']$ is a successor of a configuration $\gamma = [j, q, \tau]$ iff there is a transition $(q, a, E) \to q'$ that applies to γ, $\tau' = \tau$, and $j' = j + 1$; or there is a transition $(q, a, E) \to (q', i)$ that applies to γ, $j' = j + 1$, and τ' is obtained from τ by setting $\tau'(i)$ to $w_j.d$. Based on this, reachable configuration and acceptance of a data word is defined in the standard way. We denote by $L(\mathcal{A})$ the language of data words accepted by the register automata $\mathcal{A}$.

Example 3.2. There exists a 1-register automata which checks whether the input data words contains two positions labeled with a with the same data value (a is not a key): it non-deterministically moves to the first position labeled with a with a duplicated data value, stores the data value in its register and then checks that this value appears again under another position labeled with a.

The complement of this language, all node labeled with a have different data values, which would be useful for key integrity constraints in the database context is not expressible with register automata. To prove this, one can show that for each data word accepted by a given register automata, there exists another accepted data word that have the same string projection but uses a number of data values depending only on the automaton (see [20] for more details).

This shows that register automata are not closed under complementation. They are also not closed under determinization as the first example cannot be achieved with a deterministic register automata.

The main result for register automata is that emptiness is decidable.

Theorem 3.3. *[20] Emptiness of register automata is decidable.*

In term of complexity the problem is PSPACE-complete [14]. Without labels it was shown to be NP-complete in [30]. As illustrated with the examples above, register automata are quite limited in expressive power. This limitation can also be formalized with the following proposition which should be contrasted with the similar one in the case of data automata presented later (Proposition 3.14).

Proposition 3.4. *[20,9] Given a language L of data words accepted by a register automata the language of strings* $\text{STR}(L)$ *is regular.*

There exist many obvious extensions of register automata. For instance one could add alternation or 2-wayness. Unfortunately those extensions are undecidable.

Theorem 3.5. *1. Universality of register automata is undecidable [28].*
2. Emptiness of 1-register 2-way automata is undecidable [12].

An immediate corollary of Theorem 3.5 is:

Corollary 3.6. *[28] Inclusion and equivalence of register automata is undecidable.*

There are several variants that weaken or strengthen the model of register automata as presented above without affecting much the results. We only briefly mention two of them. One possible extension is to allow the automata to store a non-deterministically chosen value from D in one of the register. With this extension the language of data words such that the data value of the last position of the word is different from all the other positions can be accepted. Without it the language is not accepted by a register automata. This model was studied in [11,22] and is a little bit more robust than the one presented here. In the model presented above the automata knows the equality type of the current data value with all the registers. One possible weakening of the model is to allow only an equality test of the current data value with a fix register. That is the transitions are of the form $(q, a, i) \longrightarrow q'$ or $(q, a, i) \longrightarrow (q', j)$. This setting was studied in [32]. The language of data words such that the first two data values are distinct can no longer be accepted [21]. On the other hand this model has equivalent regular expressions [21]. Another notable property of this weaker model is that inclusion becomes decidable [32]. To conclude this discussion one should mention that the model of the register automata as presented above have algebraic characterizations, see [9,17] for more details.

Trees. To extend this model to binary trees the technical difficulty is to specify how registers are splited (for the top-down variants) or merged (for the bottom-up variant). In the top-down case the natural solution is to propagate the content of the registers to the children of the current node. In the bottom-up case a function can be included into the specification of the transitions of the automata which specify, for each register, whether the new value should come from some register of the left child or from some register of the right one. It is possible to do this so that the family obtained is decidable and robust in the sense that the bottom-up variant correspond to the top-down variant. This model has all the properties of register automata over data words: emptiness is decidable and the tree projection is regular. We refer to [21] for more details.

3.2 Pebble Automata

Another model of automata uses pebbles instead of registers. The automata can drop and lift pebbles on any position in the string and eventually compare the current value with the ones marked by the pebbles. To ensure a "regular" behavior, the use of pebbles is restricted by a stack discipline. That is, pebbles are numbered from 1 to k and pebble $i + 1$ can only be placed when pebble i is present on the string. A transition depends on the current state, the current

label, the pebbles placed on the current position of the head, and the equality type of the current data value with the data values under the placed pebbles. The transition relation specifies change of state, and possibly whether the last pebble is removed or a new pebble is placed. In the following more formal definition we follow [28] and assume that the head of the automata is always the last pebble. This does not make much difference in term of expressive power, at least in the 2-way case which is the most natural for this model, but simplifies a lot the notations:

Definition 3.7. A *k-pebble automaton* $\mathcal{A}$ is a tuple (Q, q_0, F, T) where

- Q is a finite set of *states*; $q_0 \in Q$ is the *initial state*; $F \subseteq Q$ is the set of *final states*; and,
- T is a finite set of *transitions* of the form $\alpha \to \beta$, where
  - α is of the form (q, a, i, P, V, q), where $i \in \{1, \ldots, k\}$, $a \in \Sigma$, $P, V \subseteq \{1, \ldots, i-1\}$, and
  - β is of the form (q, d) with $q \in Q$ and d is either DROP of LIFT.

Given a data word w, a *configuration of* $\mathcal{A}$ *on* w is of the form $\gamma = [q, j, \theta]$ where i is the current position in w, $q \in Q$ the current state, $j \in \{1, \ldots, k\}$ the number of pebbles already dropped, and $\theta : \{1, \ldots, j\} \to [|w|]$ the positions of the pebbles. Note that the current position always correspond to $\theta(j)$. We call θ a pebble assignment. The *initial* configuration is $\gamma_0 := [q_0, 1, \theta_0]$, with $\theta_0(1) = 1$. A configuration $[q, j, \theta]$ with $q \in F$ is *accepting*.

A transition $(p, a, i, P, V, p) \to \beta$ *applies to a configuration* $\gamma = [q, j, \theta]$, if

1. $i = j$, $p = q$,
2. $V = \{l < j \mid w_{\theta(l)}.d = w_{\theta(j)}.d\}$ is the set of pebbles placed on a position which has the same data value than the current position.
3. $P = \{l < j \mid \theta(l) = \theta(j)\}$ is the set of pebbles placed at the current position, and
4. $w_j = a$.

Intuitively, $(p, a, i, P, V) \to \beta$ applies to a configuration if pebble i is the current head, p is the current state, V is the set of pebbles that see the same data value as the top pebble, P is the set of pebbles that sit at the same position as the top pebble, and the current label seen by the top pebble is a.

A configuration $[q', j', \theta']$ is a successor of a configuration $\gamma = [q, j, \theta]$ if there is a transition $\alpha \to (p, d)$ that applies to γ such that $q' = p$ and $\theta'(i) = \theta(i)$, for all $i < j$, and

- if d=DROP, then $j' = j + 1$, $\theta'(j) = \theta(j) = \theta'(j + 1)$,
- if d=LIFT, then $j' = i - 1$.

Note that this implies that when a pebble is lifted, the computation resumes at the position of the previous pebble.

Example 3.8. The languages of data words such that all nodes labeled with a have different data values is accepted by a 2-pebble automata as follows. At each

position labeled with a the automata drops pebble 1 and then check that the data value never occurs under a position labeled with a to the right of the pebble. When this is done it comes back to pebble 1 and move to the next position.

Register automata and pebble automata are likely to be incomparable in expressive power. The example above can be expressed by a pebble automata but not by a register automata. On the other hand, pebble automata have a stack discipline constraint on the use of pebbles while register automata can update their register in an arbitrary order. Based on this, examples of languages expressible with register automata but not with pebble automata are easy to construct but we are not aware of any formal proof of this fact.

Strictly speaking the model presented above is not really 1-way as when it lifts a pebble the automata proceed at the position of previous pebble. It is also immediate to extends this model in order to make it 2-way. The advantage of this definition is that the model is robust:

Theorem 3.9. *[28] Pebble automata are determinizable and the 2-way variant has the same expressive power than its 1-way variant.*

But unfortunately the model is too strong in expressive power. In the result below, recall that with our definition the head of the automata counts as 1 pebble. Therefore in a sense the result is optimal.

Theorem 3.10. *[28,12] Emptiness of 2-pebble automata over data words is undecidable.*

Trees. The most natural way to extend this model to trees is to consider tree walking automata with pebbles on trees. This model extend the 2-way variant of pebble automata as presented here in a natural way with moves allowing to "walk" the tree in all possible directions (up, down, left right). Of course, this model remains undecidable.

3.3 Data Automata

This model was introduced in [6,7]. It extends the model of register automata, remains decidable and has better connections with logic. Data automata runs on data words in two passes. During the first one a letter-to-letter transducers is run. This transducers does not have access to the data values and change the label of each position. During the second pass, an automata is run on each sequence of letters having the same data value, in other words, on each class.

A *data automaton* $\mathcal{D} = (\mathcal{A}, \mathcal{B})$ consists of

- a non-deterministic letter-to-letter string transducer $\mathcal{A}$ (the *base automaton*) with input alphabet Σ, for some output alphabet Γ (letter-to-letter means that each transition reads and writes exactly one symbol), and
- a non-deterministic string automaton $\mathcal{B}$ (the *class automaton*) with input alphabet Γ.

A data word $w = w_1 \cdots w_n$ is accepted by $\mathcal{D}$ if there is an *accepting* run of $\mathcal{A}$ on the string projection of w, yielding an output string $b_1 \cdots b_n$, such that, for *each class* $\{x_1, \ldots, x_k\} \subseteq \{1, \ldots, n\}$, $x_1 < \cdots < x_k$, the class automaton accepts $b_{x_1}, \ldots, b_{x_k}$.

Example 3.11. The language of data words such that all positions labeled with a have different data values can be accepted by a data automaton as follows. The base automaton does nothing and only copy its input. The class automata accepts only words containing only one a.

The complement of the above language, the set of all data words containing two positions labeled with a with the same data value, is done as follows. The base automata non-deterministically selects two positions labeled with a and output **1** on each of them and **0** everywhere else. The class automata checks that each class contains either no **1** or exactly 2.

The language of data words such that there exists two positions labeled with a and having the same data value but with no positions labeled with b (no matter what the data value is) in between is accepted by a data automaton. The base automata outputs **0** on all positions excepts two, labeled with a, that it selects non-deterministically and on which it outputs **1**. It also checks that between the two selected positions no b occurs. The class automata checks that each class contains either no **1** or exactly 2.

It is not clear how to do the complement of this language using a data automata. This suggest that data automata are not closed under complementation but we are not aware of any formal proof of this result. It will follow from the results of Section 4 that if they would be closed under complementation then they would not be effectively closed under complementation, by this we mean that there would be no algorithm computing the complement of a data automata.

By just looking at the definitions it is not immediate that data automata extends the model of register automata presented in Section 3.1. But it is indeed the case.

Proposition 3.12. *[5] For each register automata there exists a data automaton accepting the same language.*

The main result on data automata is that emptiness remains decidable. We will use this fact to show decidability of several logics in Section 4.

Theorem 3.13. *[7] Emptiness of data automata is decidable.*

In order to better understand the expressive power of data automata we show that the string projection of languages accepted by data automata correspond to languages accepted to multicounter automata. Recall that for register automata the string projection remains regular (Proposition 3.4).

We first briefly review the definition of multicounter automata. An ϵ-free *multicounter automaton* is a finite automaton extended by a finite set $C = \{1, \ldots, n\}$ of counters. It can be described as a tuple $(Q, \Sigma, C, \delta, q_I, F)$. The set of states Q, finite alphabet Σ, initial state $q_I \in Q$ and final states $F \subseteq Q$ are

as in a usual finite automaton. The transition relation δ is a finite subset of $Q \times \Sigma \times (dec^*(i)\, inc^*(i))_{i \in C} \times Q$.

The idea is that in each step, the automaton can change its state and modify the counters, by incrementing or decrementing them, according to the current state and the current letter on the input. In a step, the automaton can apply to each counter $i \in C$ a sequence of decrements, followed by a sequence of increments. Whenever it tries to decrement a counter of value zero the computation stops. Besides this, the transition of a multicounter automaton does not depend on the value of the counters. In particular, it cannot test whether a counter is exactly zero (otherwise the model would be undecidable). Nevertheless, by decrementing a counter k times and incrementing it again afterward it can test whether the value of that counter is at least k.

A *configuration* of such an automaton is a tuple $c = (q, (c_i)_{i \in C}) \in Q \times \mathbb{N}^n$, where q is the current state and c_i is the value of the counter i. A transition

$$(q, a, (dec^{k_i}(i) inc^{l_i}(i))_{i \in C}, q') \in \delta$$

can be applied if the current state is q, the current letter is a and for every counter $i \in C$, the value c_i is at least k_i. The successor configuration is $d = (q', (c(i) - k_i + l_i)_{i \in C})$. A *run* over a word w is a sequence of configurations that is consistent with the transition function δ. The acceptance condition is given by a subset R of the counters C and the final states. A run is *accepting* if it starts in the state q_I with all counters empty and ends in a configuration where all counters in R are empty and the state is final.

Emptiness of multicounter automata is known to be decidable [26,23].

Proposition 3.14. *[7]*

- *If L is a language of data words accepted by a data automata then* $\text{STR}(L)$ *a language of strings accepted by a multicounter automata.*
- *If L is a language of strings accepted by a multicounter automata then there exists a language L' of data words accepted by a data automata such that $h(\text{STR}(L')) = L$, where h is an erasing morphism.*

The constructions of Proposition 3.14 are constructive (the time complexity requires a tower of 3 exponentials in the first case and is only polynomial in the second case) and thus Proposition 3.14 implies Theorem 3.13. Therefore the emptiness problem of data automata is elementary equivalent to the emptiness problem of multicounter automata which is not yet known to be elementary (the best known lower-bound being ExpSpace [24] and the best known upper-bound being non-elementary [26,23]). The precise complexity of Theorem 3.13 is therefore still an open issue.

Trees. This model can be extended to unranked ordered trees as follows. Both the base automaton and the class automaton are bottom-up tree automaton. The base automata works as in the word case and reassigns a label to each node. Each class can now be viewed as an unranked ordered tree by contracting the initial

tree edges which contains a node not in the class. The class automata is then run on the resulting trees. Unfortunately decidability of data tree automata remains an open issue. Data tree automata still have some connection with multicounter tree automata. It is not clear yet whether the first part of Proposition 3.14 holds or not, but the second one does hold. Therefore showing decidability of data automata would implies showing decidability of multicounter tree automata which has eluded all efforts so far (see [13] and the references therein).

3.4 Other Generalizations

We have consider so far only automata for data words or data trees that are based on the classical notion of finite state automata. It is also possible to consider more powerful devices for manipulating data words such as pushdown automata.

Historically the first models of pushdown automata over infinite alphabet were presented in [3] and [19]. Those models extend the notions of Context-Free grammars and of pushdown automata in the obvious way by allowing infinitely many rules and infinitely many transition functions, one for each letter in D. Hence the models are not even finitely presented and decidability does not make much sense. In term of expressive power, with some technical constraints on the pushdown automata models (see [19]), both the context-free grammars and pushdown automata defines the same class of languages.

The most robust decidable notion so far was given in [11] using ideas similar to register automata. We describe it next. Intuitively the automata has k registers and makes its choices given, the current state, the label of the current node, and the equality type of the current data value with the values present in the registers. The possible actions are: change the current state, update some of the registers with the current data value, push the values stored into some registers into the stack together with some finite content, pop the top of the stack. Moreover, in order to have a robust model with an equivalent context-free grammar, the model allows ϵ transition which can store into one of the registers a new, non-deterministically chosen, data value.

Definition 3.15. *A* k-register pushdown automaton $\mathcal{A}$ *is a tuple* $(Q, q_0, F, \Gamma, \tau_0, T)$ *where*

- Q *is a finite set of states;* $q_0 \in Q$ *is the initial state;* $F \subseteq Q$ *is the set of final states;*
- Γ *is a finite set of labels for stack symbols;*
- $\tau_0 : \{1, \ldots, k\} \to D$ *is the initial register assignment; and,*
- T *is a finite set of transitions of the form* $(q, a, E, \gamma) \to (q', i, \overline{s})$ *or* $(q, \epsilon, E, \gamma) \to (q', i)$.
  Here, $i \in \{1, \ldots, k\}$, $q, q' \in Q$, $a \in \Sigma$, $\gamma \in \Gamma$, $\overline{s} \in (\Gamma \times \{1, \ldots, k\})^*$ *and* $E \subseteq \{\top, 1, \ldots, k\}$.

Given a stack symbol s we denote by $s.l$ and $s.d$ respectively the label in Γ and the data value in D of s. Given a data word w, a *configuration of* $\mathcal{A}$ *on* w is a tuple $[j, q, \tau, \mu]$ where $1 \leq j \leq |w|$ is the current position in the data words,

$q \in Q$ is the current state, $\tau : \{1, \ldots, k\} \to D$ is the current register assignment, and $\mu \in (\gamma \times D)^*$ is the current stack content. The *initial* configuration is $\nu_0 := [1, q_0, \tau_0, \epsilon]$. A configuration $[j, q, \tau, \mu]$ with $q \in F$ is *accepting*. Given $\nu = [j, q, \tau, \mu]$ with top stack symbol s, the transition $(p, a, E, \gamma) \to \beta$ *applies* to ν iff $p = q$, $w_j.l = a$, $s.l = \gamma$, and $E = \{l \mid w_j.d = \tau(l)\} \cup S$ with S is empty if $w_j.d \neq s.d$ and is $\{\top\}$ otherwise.

A configuration $\nu' = [j', q', \tau', \mu']$ is a successor of a configuration $\nu = [j, q, \tau, \mu]$ iff there is a transition $(q, a, E, \gamma) \to (q', i, \overline{s})$ that applies to ν, $\tau'(l) = \tau(l)$ for all $l \neq i$, $\tau'(i) = w_j.d$, $j' = j + 1$ and μ' is μ with the top of the stack removed and replaced by $(a_1, d_{i_1}) \cdots (a_m, d_{i_m})$ where $\overline{s} = (a_1, i_1) \cdots (a_m, i_m)$ and d_l is the current value of register l; or there is a transition $(q, a, E, \gamma) \to (q', i)$ that applies to ν, $j' = j$, $\mu = \mu'$, and τ' is obtained from τ by setting $\tau'(i)$ to some arbitrary value of D. Based on this, reachable configuration and acceptance of a data word is defined in the standard way.

Note that we allow ϵ transitions which can introduce non-deterministically a new symbol in some register. This make the model more robust.

Example 3.16. The language of data words such that the data values form a sequence ww^R where w^R is the reverse sequence of w is accepted by a register data automata in the obvious way by first pushing the data values into the stack and then poping them one by one.

The language of data words such that all positions labeled with a have different data values is not accepted by a register pushdown automata. This uses argument similar than for register automata. See [11] for more details and more example.

This model of register pushdown automata extends in a natural way the classical notion of pushdown automata. It can be shown that the good properties of Context-Free languages can be extended to this model. In particular we have:

Theorem 3.17. *[11] Emptiness of register pushdown automata is decidable.*

This model is also robust and has an equivalent presentation in term of Context-Free grammar that we don't present here. The interested reader will find in [11] more details and more properties of the model. We conclude with a characterization of the projection languages defined by register pushdown automata similar to Proposition 3.4.

Proposition 3.18. *If L is a language accepted by a register pushdown automata then* $\text{STR}(L)$ *is Context-Free.*

4 Logics

The previous section was concerned with finding decidable automata for manipulating words and trees containing data values. In this section we are looking for declarative decidable tools such as logics.

Data words and data trees can be seen as models for a logical formula.

In the case of data words, the universe of the model is the set of positions in the word and we have the following built-in predicates: $x \sim y$, $x < y$, $x = y + 1$, and a predicate $a(x)$ for every $a \in \Sigma$. The interpretation of $a(x)$ is that the label in position x is a. The order $<$ and successor $+1$ are interpreted in the usual way. Two positions satisfy $x \sim y$ if they have the same data value. Given a formula ϕ over this signature, we write $L(\phi)$ for the set of data words that satisfy the formula ϕ. A formula satisfied by some data word is *satisfiable*.

In the case of data trees, the universe of the structure is the set of nodes of the tree with the following predicates available:

- For each possible label $a \in \Sigma$, there is a unary predicate $a(x)$, which is true for all nodes that have the label a.
- The binary predicate $x \sim y$ holds for two nodes if they have the same data value.
- The binary predicate $E_{\rightarrow}(x, y)$ holds for two nodes if x and y have the same parent node and y is the immediate successor of x in the order of children of that node.
- The binary predicate $E_{\downarrow}(x, y)$ holds if y is a child of x.
- The binary predicates $E_{\Rightarrow}$ and $E_{\Downarrow}$ are the transitive closures of $E_{\rightarrow}$ and $E_{\downarrow}$, respectively.

For both words and trees, we write $\mathrm{FO}(\sim, <, +1)$ for first-order logic over a signature containing all the predicates mentioned above for the corresponding context. We also write $\mathrm{FO}(\sim, +1)$ when the predicates $E_{\Rightarrow}$ and $E_{\Downarrow}$ -in the case of trees- and without $<$ -in the case of words are missing.

A logic L is said to be decidable over a class $\mathcal{M}$ of models if, given a sentence $\varphi \in L$, it is decidable whether there exists a model $M \in \mathcal{M}$ such that $M \models \varphi$.

4.1 First-Order logics

Example 4.1. The language of data words such that all positions labeled with a have different data values can be expressed in $\mathrm{FO}(\sim, <, +1)$ by $\forall x, y \;\; (a(x) \wedge a(y) \wedge x \neq y) \longrightarrow x \not\sim y$.

The complement of the above language, the set of all data words containing two positions labeled with a having the same data value, is thus $\exists x, y \;\; a(x) \wedge a(y) \wedge x \neq y \wedge x \sim y$.

The language of data words where each position labeled with a has a data value which also appears under a position labeled with b (inclusion dependency) is expressed in $\mathrm{FO}(\sim, <, +1)$ by $\forall x \exists y \;\; a(x) \longrightarrow (b(y) \wedge x \sim y)$.

Let L be the language of data words such that (i) any two positions labeled with a have a distinct data value, (ii) any two positions labeled with b have a distinct data value and (iii) the sequence of data values of the positions labeled a is exactly the same as the sequence of data values labeled with b. L can be expressed in $\mathrm{FO}(\sim, <, +1)$ as follows. First we use sentences as given in the first three examples in order to express (i), (ii) and the fact that each data value appears exactly twice under two positions, one labeled a and one labeled b. Then the following sentence shows that the sequences are the same: $\forall x, y, z \;\; (a(x) \wedge a(y) \wedge x < y \wedge b(z) \wedge x \sim z) \longrightarrow (\exists x \;\; b(x) \wedge x \sim y \wedge z < x)$.

The relative expressive power of register and pebble automata with logics has been studied in [28] over data words. In term of expressive power, $\mathrm{FO}(\sim, <, +1)$ is not comparable with register automata and it is strictly included into pebble automata.

As $\mathrm{FO}(\sim, <, +1)$ is a fragment of pebble automata it is natural to wonder whether it forms a decidable fragment. We write FO^k for formulas using at most k variables (possibly reusing them at will). In the examples above note that the first three are definable in $\mathrm{FO}^2(\sim, <, +1)$ while the third one requires three variables. This last example can be pushed a little bit in order to code PCP. Therefore we have:

Theorem 4.2. *[7]* $\mathrm{FO}^3(\sim, <, +1)$ *is not decidable over data words.*

However the two-variable fragment of $\mathrm{FO}(\sim, <, +1)$, which turns out to be often sufficient for our needs, especially in the database context, is decidable.

Theorem 4.3. *[7]* $\mathrm{FO}^2(\sim, <, +1)$ *is decidable over data words.*

Theorem 4.3 follows from the fact that any data word language definable by a formula of $\mathrm{FO}^2(\sim, <, +1)$ is accepted by a data word automata described in Section 3. Actually we can show a stronger result than this. Consider the new binary predicate $+1$ which holds true at position x and y if the positions denoted by x and y have the same data value and y is the successor of x in their class. Note that this predicate is not expressible in $\mathrm{FO}^2(\sim, <, +1)$. Consider now the logic $\mathrm{EMSO}^2(\sim,<,+1,+1)$ which extends $\mathrm{FO}^2(\sim,<,+1,+1)$ by existential monadic second-order predicates quantification in front of $\mathrm{FO}^2(\sim,<,+1,+1)$ formulas.

Theorem 4.4. *[7] For any data word language L the following are equivalent.*

1. *L is definable in* $\mathrm{EMSO}^2(\sim,<,+1,+1)$,
2. *L is accepted by a data word automata.*

Moreover the translations between $\mathrm{EMSO}^2(\sim,<,+1,+1)$ *formulas and data word automata are effective.*

This immediately yields:

Corollary 4.5. $\mathrm{EMSO}^2(\sim,<,+1,+1)$ *is decidable over data words.*

Again the precise complexity is not known as it more or less correspond to deciding emptiness of multicounter automata.

Note that adding a little bit more of arithmetic, like assuming that D is linearly order and that the logic contains an extra binary predicate checking for this order among data values, yields undecidability even for the 2-variables fragment of FO [7].

Trees. The step from data words to data trees seems non trivial and we don't yet know whether Theorem 4.4 holds in the data tree case. In particular we don't know whether languages definable in $\mathrm{FO}^2(\sim, <, +1)$ are always accepted by a data tree automata. On the other hand $\mathrm{FO}^2(\sim, <, +1)$ is expressive enough to simulate multicounter tree automata therefore satisfiability of $\mathrm{FO}^2(\sim, <, +1)$ over data trees is likely to be a difficult problem.

On the other hand we can show:

Theorem 4.6. *[6]* $\mathrm{FO}^2(\sim, +1)$*, and therefore* $\mathrm{EMSO}^2(\sim, +1)$*, is decidable over data trees.*

The proof of Theorem 4.6 is quite involved and does not use automata but rather some kind of *puzzles*. It is shown, using this puzzles, that any satisfiable sentences of $\mathrm{FO}^2(\sim, +1)$ has a tame model and that deciding whether a sentence has a tame model is decidable. The complexity obtained this way is 3NExpTime which is possibly not optimal.

4.2 LTL with Freeze Quantifier

This logic was studied in the context of data words in [14,15]. Roughly speaking it extends the linear temporal logic LTL with a freeze operator which binds the data value of the current position in the data words to a register for later use.

We consider the temporal operators X for Next, X^{-1} for Previous, F for Sometime in the future, F^{-1} for sometime in the past, U for Until, U^{-1} for Previous. As usual we regard G as an abbreviation for $\neg F \neg$. We define $\mathrm{LTL}_n^{\downarrow}$ using the following grammar:

$\mathrm{S} ::= \top \mid a \mid \downarrow_r S \mid S \wedge S \mid \neg S \mid O(S...S) \mid \uparrow_r$

where $r \in [1, n]$, $a \in \Sigma$ and O is a temporal operator in $\{X, X^{-1}, F, F^{-1}, U, U^{-1}\}$.

We only consider well-formed formulas where $\uparrow_r$ is only used in the scope of a $\downarrow_r$. When we want to use only a subset $\mathcal{O}$ of the temporal operators then we write $\mathrm{LTL}_n^{\downarrow}(\mathcal{O})$.

The semantic is classical for the Boolean and temporal operators. The formula a holds at any position whose label is a. The formula $\downarrow_r S$ stores the data value of the current position in the register r and then checks for S starting at the current position. The formula $\uparrow_r$ checks that the data values stored in the register r equals the data value of the current position.

Example 4.7. The language of data words containing two positions labeled with a having the same data value, can be expressed in $\mathrm{LTL}_1^{\downarrow}$ by $F(a \wedge \downarrow_1 XF(a \wedge \uparrow_1))$.

The complement of the above language, data words such that all positions labeled with a have different data values, is thus $\neg F(a \wedge \downarrow_1 XF(a \wedge \uparrow_1))$.

The language of data words where each position labeled with a has a data value which also appears under a position labeled with b (inclusion dependency) is expressed in $\mathrm{LTL}_1^{\downarrow}$ by $G(a \longrightarrow (\downarrow_1 (F(b \wedge \uparrow_1) \vee F^{-1}(b \wedge \uparrow_1))))$.

The language of data words such that any two distinct positions labeled with a and having the same data value have a b in between can be expressed in $\mathrm{LTL}_1^{\downarrow}$ by: $G\big((a \wedge \downarrow_1 F(a \wedge \uparrow_1)) \longrightarrow (\neg(a \wedge \uparrow_1) U b)\big)$

Theorem 4.8. *[14,15]*

1. $\mathrm{LTL}_1^{\downarrow}(X, U)$ *is decidable over data words.*
2. $\mathrm{LTL}_2^{\downarrow}(X, F)$ *is undecidable over data words.*
3. $\mathrm{LTL}_1^{\downarrow}(X, F, F^{-1})$ *is undecidable over data words.*

The logics $\mathrm{LTL}_1^{\downarrow}(X, U)$ and $\mathrm{FO}^2(\sim, <, +1)$ are incomparable. Indeed the third example above (inclusion dependency) is expressible in $\mathrm{FO}^2(\sim, <, +1)$ but not in $\mathrm{LTL}^{\downarrow}$ without past temporal operators. On the other hand the last example above is expressible in $\mathrm{LTL}_1^{\downarrow}(X, U)$ but not in $\mathrm{FO}^2(\sim, <, +1)$ (and most likely this property is also not expressible using a data automaton).

There exists a fragment of $\mathrm{LTL}_1^{\downarrow}$ which correspond to $\mathrm{FO}^2(\sim, <, +1)$. A $\mathrm{LTL}_1^{\downarrow}$ formula is said to be *simple* if any temporal operator is immediately preceded by a $\downarrow_1$ and there are no occurrences of $\downarrow_1$ operators. Then simple-$\mathrm{LTL}_1^{\downarrow}(X, X^{-1}, X^2 F, X^{-2}F^{-1})$ has exactly the same expressive power than $\mathrm{FO}^2(\sim, <, +1)$. This fact was first mentioned in [14]. The translations in both directions are effective and use the same ideas as in [16].

Trees. The extension of this ideas to trees, using languages like CTL, remains to be done.

5 Conclusion

We have presented several models of decidable automata and logics over data words and data trees. The logical approach has the advantage of compositionality and has many other interesting closure properties which makes it easier for coding problems into it. The complexities obtained for logics are usually quite high which makes them quite unsuited for practical applications. However it is possible to obtain lower complexities by imposing some extra constraints, see for instance [6,7].

Several of the results presented in this paper were extended to infinite data words. This is useful in the context of verification in order to code infinite computations. For instance Theorem 4.3 have been extended over infinite data words in [7].

The tree case is usually a lot more difficult than the word case. If several decidability results were obtained over data trees, like for register automata or $\mathrm{FO}^2(\sim, +1)$, many decidability questions remains open, like the decidability of $\mathrm{FO}^2(\sim, <, +1)$.

The decidability result of $\mathrm{EMSO}^2(\sim, +1)$ over data trees presented in Theorem 4.6 was used in a database context in [6] in order to show decidability of the inclusion problem of certain fragments of XPath in the presence of DTDs. It was also used to show decidability of validation of DTDs in the presence of unary key and foreign key constraints [6]. Those two results were obtained via different coding of the problems into data trees and $\mathrm{EMSO}^2(\sim, +1)$. This builds on the fact that any regular tree language (and therefore the structural part of DTDs)

can be expressed in $\text{EMSO}^2(\sim, +1)$ (without using the predicate $\sim$) and XPath 1.0 is also intimately related with first-order logics with two variables [25].

Altogether we hope that we have convinced the reader that there is a need for more decidable automata models and more decidable logics over data words and data trees. This is a challenging topic which has a lot of applications, in particular in database and in program verification.

Acknowledgment. We thanks Mikołaj Bojańczik, Anca Muscholl, and Thomas Schwentick for all the interesting discussions we had while writing this paper.

References

1. N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with Data Values: Typechecking Revisited. Journal of Computer and System Sciences, 66(4):688 727, 2003.
2. M. Arenas, W. Fan, and L. Libkin. Consistency of XML specifications. Inconsistency Tolerance, Springer, LNCS vol. 3300, 2005.
3. J.-M. Autebert, J. Beauquier, and L. Boasson. Langages des alphabets infinis. Discrete Applied Mathematics, 2:120, 1980.
4. M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. In Proc. ACM Symp. on Principles of Database Systems, pages 2536, 2005.
5. H. Björklund and T. Schwentick. Personnal communication.
6. M. Bojańczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin. Two- Variable Logic on Data Trees and XML Reasoning. In Proc. ACM Symp. on Principles of Database Systems, 2006.
7. M. Bojańczyk, A. Muscholl, T. Schwentick, L. Segoufin, and C. David. Two- Variable Logic on Words with Data. In Proc. IEEE Conf. on Logic in Computer Science, 2006.
8. A. Bouajjani, P. Habermehl, and R. Mayr. Automatic Verification of Recursive Procedures with one Integer Parameter. Theoretical Computer Science, 295, 2003.
9. P. Bouyer, A. Petit, and D. Thérien. An algebraic approach to data languages and timed languages. Inf. Comput., 182(2):137162, 2003.
10. P. Buneman, S. B. Davidson, W. Fan, C. S. Hara, and W. C. Tan. Reasoning about keys for XML. Inf. Syst., 28(8):10371063, 2003.
11. E. Cheng and M. Kaminski. Context-Free Languages over Infinite Alphabets. Acta Inf., 35(3):245267, 1998.
12. C. David. Mots et données infinis. Master's thesis, Université Paris 7, LIAFA, 2004.
13. P. de Groote, B. Guillaume, and S. Salvati. Vector Addition Tree Automata. In Proc. IEEE Conf. on Logic in Computer Science, pages 6473, 2004.
14. S. Demri and R. Lazić. LTL with the Freeze Quantifer and Register Automata. In Proc. IEEE Conf. on Logic in Computer Science, 2006.
15. S. Demri, R. Lazić, and D. Nowak. On the freeze quantifier in Constraint LTL. In TIMES, 2005.
16. K. Etessami, M. Vardi, and T. Wilke. First-Order Logic with Two Variables and Unary Temporal Logic. Inf. Comput., 179(2):279295, 2002.
17. N. Francez and M. Kaminski. An algebraic characterization of deterministic regular languages over infinite alphabets. Theoretical Computer Science, 306(1-3):155175, 2003.

18. F. Geerts and W. Fan. Satisfiability of XPath Queries with Sibling Axes. In Proc. workshop on Database Programming Language, pages 122137, 2005.
19. J. Idt. Automates a pile sur des alphabets infinis. In STACS, 1984.
20. M. Kaminski and N. Francez. Finite memory automata. Theoretical Computer Science, 134(2):329363, 1994.
21. M. Kaminski and T. Tan. Regular Expressions for Languages over Infinite Alphabets. Fundam. Inform., 69(3):301318, 2006.
22. M. Kaminski and T. Tan. Tree Automata over Infinite Alphabets. Poster at the 11th International Conference on Implementation and Application of Automata, 2006.
23. S. Kosaraju. Decidability of reachability in vector addition systems. In Proc. ACM SIGACT Symp. on the Theory of Computing, pages 267281, 1984.
24. R. Lipton. The reachability problem requires exponential space. Technical report, Dep. of Comp.Sci., Research report 62, Yale University, 1976.
25. M. Marx. First order paths in ordered trees. In Proc. of Intl. Conf. on Database Theory, 2005.
26. E. Mayr. An algorithm for the general Petri net reachability problem. SIAM J. of Comp., 13:441459, 1984.
27. F. Neven and T. Schwentick. XPath Containment in the Presence of Disjunction, DTDs, and Variables. In Proc. of Intl. Conf. on Database Theory, 2003.
28. F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. ACM Trans. Comput. Logic, 15(3):403435, 2004.
29. F. Otto. Classes of regular and context-free languages over countably infinite alphabet. Discrete and Apllied Mathematics, 12:4156, 1985.
30. H. Sakamoto and D. Ikeda. Intractability of decision problems for finite-memory automata. Theoretical Computer Science, 231:297308, 2000.
31. Y. Shemesh and N. Francez. Finite-State Unification Automata and Relational Languages. Inf. Comput., 114(2):192213, 1994.
32. A. Tal. Decidability of inclusion for unification based automata. Master's thesis, Department of Computer Science, Technion - Israel Institute of Technology, 1999.