Regular Rewriting of Active XML and Unambiguity[*]

Serge Abiteboul
INRIA-Futurs & U. Paris Sud-LRI
<fname>.<lname>@inria.fr

Tova Milo
Tel-Aviv University
milo@cs.tau.ac.il

Omar Benjelloun
Stanford University
benjello@db.stanford.edu

ABSTRACT

We consider here the exchange of Active XML (AXML) data, i.e., XML documents where some of the data is given explicitly while other parts are given only intensionally as calls to Web services. Peers exchanging AXML data agree on a *data exchange schema* that specifies in particular which parts of the data are allowed to be intensional. Before sending a document, a peer may need to *rewrite* it to match the agreed data exchange schema, by calling some of the services and *materializing* their data. Previous works showed that the rewriting problem is undecidable in the general case and of high complexity in some restricted cases. We argue here that this difficulty is somewhat artificial. Indeed, we study what we believe to be a more adequate, from a practical view point, rewriting problem that is (1) in the spirit of standard 1-unambiguity constraints imposed on XML schema and (2) can be solved by a single pass over the document with a computational device not stronger than a finite state automaton. Following previous works, we focus on the core of the problem, i.e., on the problem on *words*. The results may be extended to (A)XML trees in a straightforward manner.

1. INTRODUCTION

Active XML documents (AXML for short) are documents where some of the data is given explicitly while other parts are given only intensionally by means of calls to Web services [2, 14, 7, 8].

When exchanged between two applications, AXML documents have a crucial property: since Web services may be called from anywhere on the Web, data can either be materialized before sending, or sent in its intensional form, thus leaving the receiver the freedom to materialize it if and when needed. More generally, a hybrid approach can be adopted, where some data is materialized by the sender before the document is sent, and some is materialized by the receiver.

[*]This work was partially supported by the European Project EDOS, the ACI project MDP2P, the ARC project ASAX and the Israeli Science Foundation.

This choice may be influenced by various parameters such as performance, capabilities, and security considerations [10]. For instance, if communication is expensive for the sender, deferring the materialization to the receiver is preferable. On the other hand, a particular receiver may not be capable of invoking some service calls due, for example, to limited access rights or security considerations. Therefore, this particular portion of the data needs to be materialized by the sender. Just like for purely extensional XML data, a receiver can specify her preferences/constraints regarding the structure of data and its allowed intensional portions using an (A)XML schema [10]. Before sending a document, the sender must check if the data, in its current structure, matches the schema expected by the receiver. If some of data is intensional, while required by the receiver to be extensional, the sender needs to invoke the necessary service calls to transform the data into the desired structure, if possible.

Consider a sender trying to cast a given document d to the requested data exchange schema τ. Following [10], we call the sequence of invoked calls a *rewriting*. The problem of verifying whether there exists a sequence of calls that transforms d into a document of type τ is called the *rewriting problem*. The selection of the calls to be invoked in such a rewriting is called a *rewriting strategy*.

The goal of this paper is to study the rewriting problem and advocate the use of a particularly efficient class of rewriting strategies, which we call one-pass regular rewriting.

Trees vs. words. It was shown in [10] that to solve the rewriting problem for AXML trees, it suffices to solve it for each individual node in the tree. For each node n with label (type) l, one needs to check if n is "well typed", i.e. if the sequence of labels of n's children forms a *word* in the *regular language* associated with type l in the schema. If not, the word needs to be rewritten to match the type. So, as in [10, 13], we study here the rewriting problem for words. The results extend to (AXML) trees in a straightforward manner.

Before describing our contribution, let us first discuss the difficulty of the rewriting problem and limitations of previous approaches. The rewriting problem was shown to be undecidable in general, and of very high complexity even in the restricted decidable cases that were studied in [13, 10]. This was rather discouraging since rewriting needs to be performed each time an AXML document is sent. Furthermore, in a Peer-to-Peer context (the typical context of AXML data exchange), a peer is often a device with very limited resources such as a PDA or a cell phone. Thus, it is particularly important to have rewriting algorithms with very low complexity. This lead us to consider restrictions on the types that are used in data exchange schemas so that the rewriting would be dramatically simplified. Intuitively these restrictions are in the spirit of the standard ones imposed on XML schemas to ensure efficient document parsing. The following goals motivate the restrictions we will impose:

- We would like the rewriting to be performed in a single pass over the word. We introduce for that the notion of *one-pass rewriting.*
- We would like to be able to rewrite a word using no more computation power than a finite state automaton (FSA). We introduce the notion of *regular rewriting.*
- As in [10, 13], we would like, when possible, to perform the rewriting in a "safe" manner, i.e., be sure to succeed in the rewriting whenever possible, and refrain from invoking services as soon as failure is clear.
- As in [4], we would like, when possible, to use parallelism to accelerate the rewriting.

The first goal, namely a one-pass rewriting, bears some resemblance to the left-to-right rewritings studied in [10, 13]. However, there is a subtle difference. In both, left-to-right and one-pass rewritings, service calls are invoked in the order they appear in the document, i.e., once a service has been invoked, no services to its left may be invoked. A key difference is that in left-to-right rewriting, the selection of the services to call may depend on the entire word, whereas in the one-pass it is based only on what has been read so far. Intuitively, this corresponds to a sequential read of the word, making invocation decisions based only on what have been seen so far, in a one-pass filtering manner. We argue that this is a desirable restriction, from a practical view point. Indeed, we will advocate the use of finite state devices to perform the filtering. This is what we call *regular rewriting.*

A first contribution of the paper is a comparison between the two previously studied classes of rewritings, namely, the *safe* and *left-to-right* rewritings[10, 13], and the new *one-pass.* A second contribution is the study of the class of (one-pass) *regular* rewritings. As we shall see, the complexity of the rewriting demonstrated in [13] is somewhat artificially due to ambiguities in the schemas. Unambiguity conditions are often imposed on XML types [12, 9]. For instance, in XML schemas [18], the grammar specifying the allowed labels of the children of a node is typically restricted to be 1-unambiguous [5]. It thus seems reasonable to impose as well unambiguity conditions on the intensional information and on how it is rewritten. We will see that simple unambiguity conditions will lead to very efficient regular rewriting.

Practical corner. The present work was motivated by an analysis of the AXML rewriting algorithm [10] provided in an open-source system [3]. The code involves complex automata manipulations and is rather inefficient. Two directions were pursued. The first [4] lead to a re-engineering of the system based on an intensive use of parallelism. Although the performance are much better than in the original implementation, rewriting is still costly. More importantly it requires processing resources that may be beyond that found on a small peer such as a PDA or a cell phone. This motivated the present work where the focus is primarily on rewriting performed by finite state devices. An important advantage is that unlike that of [10, 4], some of the rewriting techniques we mention here may be performed by a standard XML validator, a piece of software that one expects to find even on a small device.

The paper is organized as follows. Section 2 discusses one-pass strategies. Section 3 is about regular strategies. Section 4 considers parallel rewriting. The last section is a conclusion.

2. ONE-PASS STRATEGIES

In this section, we introduce one-pass rewriting strategies and consider their relationship with safe and left-to-right strategies.

We assume a finite alphabet Σ and a subset $\Sigma_f \subseteq \Sigma$ of function names. The letters in Σ_f represent Web service names and those in $\Sigma - \Sigma_f$ represent XML elements. In the following, f, g, h denote function names, a, b, c other letters in Σ, and u, v, w are words over Σ.

In practice, the signatures of Web services, namely the expected types of their arguments and results, are typically given in their WSDL description [16]. This is modeled here by a signature function τ that associates to each function name $f \in \Sigma_f$ a regular expression R_f over Σ called the *output type* of f. (As standard in studying the rewriting problem [10, 13], we ignore the arguments of service calls in our model.) A *rewriting system* is thus specified by Σ, Σ_f and τ, or simply τ when Σ, Σ_f are understood.

We use $L(R_f)$ to denote the regular language of R_f, and assume in the following that for each f, R_f is ϵ-free (i.e., $\epsilon \notin L(R_f)$). For a function f and its output type R_f, we use further the notation f:-R_f to denote the fact that $\tau(f) = R_f$.

A *target schema* (that corresponds to the agreed data exchange type) is modeled by a regular expression R. In the following, Σ, Σ_f and τ will often be understood, possibly also the target language R. Given a word w and a target schema R, the goal is to "rewrite" w into a word in $L(R)$.

EXAMPLE 2.1. *For example, consider a newspaper document where some of the data is given explicitly, e.g. the title and date, and some intensionally, e.g. the temperature is obtained by calling a a weather forecast Web service GetTemp, and the listing of current art exhibits is obtained from the* TimeOut *local guide. The document can be modeled by the word $w = title.date.GetTemp.TimeOut$, with $GetTemp$:-$temp$, and $TimeOut$:- $(exhibit{+}performance)^*$. Here $GetTemp, TimeOut \in \Sigma_f$ while the other element names are in Σ. Assume we need to send the newspaper to a customer, and the agreed data exchange schema for her is*

$$R = (title.date.temp.TimeOut) + (title.date.GetTemp.(exhibit + performance)^*).$$

We can rewrite w into $L(R)$ by invoking GetTemp or TimeOut *(but not both).*

Observe that since the actual answers of functions are not known in advance (only their signature is given) the rewriting has to account for all their potential answers. So for the target schema

$$R' = (title.date.temp.TimeOut) + (title.date.GetTemp.exhibit^*),$$

the only sure *way to rewrite w into $L(R')$ would be by invoking GetTemp since we do not know in advance whether*

TimeOut will return exhibits or performances. Finally, for the target schema
$R'' = (title.date.GetTemp.exhibit^*)$,
there is no sure way of rewriting w. □

Rewritings and rewriting strategies. The invocation of service calls and their replacement by their answers is called a *rewriting*, and is modeled as follows.

DEFINITION 2.1. *For a word* w *over* Σ, *we say that* $w \rightarrow_i w'$ *if the i-th letter in* w *is some function* f *with output type* R_f, *and* w' *is obtained from* w *by replacing the i-th letter by a word in* $L(R_f)$. *If* $w = w_1 \rightarrow_{i_1} w_2 \ldots \rightarrow_{i_{n-1}} w_n$, *we say that* w rewrites *to* w_n.

Observe that given w, the set of words w' s.t. w rewrites to w' is context-free [1]. The intersection with $L(R)$ (for R the target schema) gives us a very rough test for potential success: If the intersection is empty, we are certain to fail rewriting w into a word in $L(R)$.

Now, given a word w and a target schema R, a rewriting of w into R can be viewed as a game between two players, *Juliet* and *Romeo*, which play in rounds [13]. In each round, Juliet selects a position i in the current word, where the i-th letter is some function name f_i. Romeo then chooses some word in $L(R_{f_i})$ and replaces f_i by this word. The game stops and *Juliet wins* if after some finite number of rounds, the resulting string is in $L(R)$. The game stops and *Romeo wins* if after some rounds, the resulting string does not rewrite into any word in $L(R)$ or if Juliet has not won yet and decides to abandon. It is possible for the game to continue forever.

A *strategy* σ (for a target language R) is a (deterministic[1]) function that Juliet uses to pick at each round the position i. If the function is undefined for some word $w \notin L(R)$, this means that Juliet declares failure. The strategy is *winning* for w and R, if, no matter how Romeo plays, Juliet wins.

Given a target schema R and a strategy σ, we denote by $\mathcal{L}(R, \sigma)$ the set of words w for which σ is winning. The set $\widehat{\mathcal{L}}(R, \sigma)$ on the other hand denotes the set of words w for which Romeo has some way to win when Juliet plays according to σ. The remaining words are those for which the game continues forever and neither Juliet nor Romeo wins. We say that two strategies σ, σ' and *equivalent* if $\mathcal{L}(R, \sigma) = \mathcal{L}(R, \sigma')$. Two strategies are *semi-equivalent* if $\widehat{\mathcal{L}}(R, \sigma) = \widehat{\mathcal{L}}(R, \sigma')$. Observe the difference between equivalence and semi-equivalence. For two equivalent strategies, Juliet wins for exactly the same set of words. For two semi-equivalent strategies, Juliet may win on some word w in one of the strategies but may neither loose nor win (i.e., loop forever) in the other.

Particular rewriting strategies. As mentioned in the introduction, we are interested in this paper in particular classes of rewriting strategies. We next recall the definition of the left-to-right strategies of [10, 13].

DEFINITION 2.2. *A* left-to-right (L2R-)strategy *is one that selects the functions to call from left to right only, i.e., if in a word* ufv, *this occurrence of* f *is selected, then no occurrence of functions in* u *may be later selected by the strategy*[2].

[1] One could also define non-deterministic strategies (and we will mention some) but the focus here is on deterministic ones.

We next illustrate these notions with some examples. For brevity, rather than using in the examples "real life" element and function names, we will represent them by letters (f, g, h for function names, a, b, c, d, e for element names.)

EXAMPLE 2.2. **[not winning]** *Consider* $w = abfh$ *with* f:-c *and* h:-$(d{+}e)^*$. *No winning strategy exists for the target language* $abfd^*$. *One can try to call* h, *but since it may return* e*'s the rewriting may fail.* □

EXAMPLE 2.3. **[winning but not L2R]** *Consider a word* $w = fg$ *with* f:-a, g:-$(b{+}b')$, *and a target schema* $(fb{+}ab')$. *A winning strategy for* w *is one that first invokes* g *and then, depending on its answer (i.e. Romeo's choice) decides to invoke* f *or not. (If* g *returns* b', f *is invoked; otherwise, the rewriting succeeded already.) In contrast, no left-to-right winning strategy exists for* w. *If we choose to invoke* f, *the adversary answers* a. *We thus have to invoke* g *and the adversary answers* b, *and the rewriting fails. If we choose to invoke* g *first (so we will never be able to invoke* f*), the adversary answers* b' *and the rewriting also fails.*

Observe that in this example a right-to-left strategy exists. It is not difficult, following similar lines, to construct an example where neither left-to-right nor right-to-left winning strategies exist. □

L2R-strategies were originally considered to reduce the space of possible rewritings and thereby simplify the task of rewriting a document to a given type. Indeed, the first rewriting module of the AXML system used a L2R strategy. We now focus on particular L2R strategies inspired by a streaming mode, where the rewriting strategy only sees the word "up to" the function f that it must decide to call or not, and not any further. A *one-pass strategy* (1P-strategy) is defined as follows. The store consists of a stack. The input is originally on the stack with its first symbol on top. We only see this top symbol. A move is either to read the top symbol or to call it. If we call, the top symbol f is replaced by some word in $\tau(f)$. We win if after emptying the stack, the list of letters we read is a word in the target language.

We will also consider a variant where we have access to the size of the stack. In other words, we also assume, for instance, that we know the size of the input and that each result of a service call is prefixed by its size. Note that this is very reasonable in practice and that without such restriction, the game may be somewhat biased. The adversary may just keep producing a longer and longer answer to a service call and there is simply no chance of being able to actually cast the document. We will speak of *one-pass strategy with size* (1PS-strategy).

It is easy to see that 1P-strategies (and 1PS-strategies) are particular L2R-strategy. Their practical interest should be clear. We read the word and decide (based only on what has been read so far) whether a function that is encountered should be called or not. Note that this is not directly related to the "1-pass pre-order typing" of [9]. However, the underlying motivation is similar.

The following examples illustrate the notion of one-pass strategy.

[2] A right-to-left variant may be defined in a similar manner.

EXAMPLE 2.4. **[1P]** *Consider the target language $(\Sigma - f)^*$, where f:-$(a+b)$. An obvious one-pass strategy consists in calling the f functions that are encountered and leaving all other functions unchanged.* □

EXAMPLE 2.5. **[1PS but not 1P]** *Consider the system f:-a with the target language $a + ff$. When we read an f we should call it if the input size is 1, and not if the size is 2. Without the size of the stack, we cannot.* □

When rewriting words to fit a target type, one would clearly like to use the best possible rewriting strategy. A strategy σ is *optimal* (for a target language R), if there does not exist another strategy σ' that is strictly better, i.e., such that $\mathcal{L}(R, \sigma) \subset \mathcal{L}(R, \sigma')$. Similarly, a L2R-strategy (resp. a 1P-strategy) is *optimal* if there does not exist a L2R-strategy (resp. a 1P-strategy) that is strictly better. If there exists an optimal strategy σ that is better than any other strategy, i.e., such that $\mathcal{L}(R, \sigma') \subseteq \mathcal{L}(R, \sigma)$ for all σ', we say that σ is an *optimum strategy* (similarly for *optimum* L2R- and *optimum* 1P-strategy).

EXAMPLE 2.6. **[L2R but no optimum 1P]** *Consider the system with f:-a and target $(fb + ab')$. It has an obvious optimum L2R-strategy. Consider an input of length 2, fx where x is not yet known. We cannot choose between calling f (in case $x = b'$) or not (in case $x = b$. There is no optimum 1P-strategy. The one strategy that decides, for instance, not to call f is optimal. Observe that because f:-a, the expression $(fb + ab')$ is quite related to $(ab + ab')$ which is 1-ambiguous [5]. We will show in the next section that an unambiguity for AXML schemas will guarantee the existence of optimum 1P-strategy and efficient rewriting.* □

The proof of the next result highlights properties of the various rewritings that will guide us in the next section in our quest for an efficient 1P-rewriting. The result is stated for one-pass but a similar result also holds for one-pass with size. In general, unless stated otherwise, in what follows, results stated for one-pass also holds for one-pass with size. We therefore sketch it.

THEOREM 2.3. *1. For every target language R, there exist an optimum strategy and an optimum L2R-strategy.*

2. There exists a target language R that has infinitely many incomparable optimal 1P-strategies and does not have an optimum 1P-strategy.

3. There exists some target languages R that has infinitely many distinct optimum strategies, and similarly for optimum L2R- and 1P-strategies.

PROOF. (sketch) We start with (1). Let the *depth* of a word w for R be the smallest k such that there exists a strategy that wins on input w after at most k rounds. We define the optimum strategy σ by induction on the depth of words. For words of depth 0 (i.e. words in $L(R)$), σ invokes no functions and wins immediately. Now, suppose that we have defined σ for all words of depth k or less. Let w be a word of depth $k+1$. Observe that σ is not defined for w and there exists a winning strategy σ_w for w that wins after at most $k+1$ rounds. Let the choice of σ on input w be exactly that of σ_w. Similarly for all such words w. The strategy σ is now winning for all words of depth $k+1$ or less. By induction, we obtain a strategy σ that wins for all words w such that some winning strategy exists for w. Similarly for L2R.

To prove (3), consider Example 2.4 above. For each i, let σ_i be the 1P-strategy defined as follows: σ_i calls all the f functions in the word and, additionally, if the i^{th} function that it encounters is not an f it calls it too. Note that the invocation of this additional function is not necessary but that it allows to distinguish between infinitely many optimum strategies.

Finally, for (2) we present an example of a target schema for which no optimum 1P-strategy exists. The goal of the example is also to illustrate *the link between an absence of such strategy and the presence of some ambiguities in the target language*, an issue that will be further investigated in the next section. Consider the target type $(fb + ab')^*$, where f:-a. It is easy to see that there exists an optimum L2R-strategy for the target language $(fb + ab')^*$. On input w, test if w has the form $(\ (f+a)(b+b')\ )^*$. If not, declare failure. If yes, look at the letters of w in even positions and call or not their preceding f's accordingly.

The situation is different for 1P-strategies. Consider some particular occurrence of f. One 1P-strategy may decide not to call f and succeed if the next symbol is a b. Another one may choose to call f and succeed if the next symbol is a b'. Indeed an infinite set of optimal 1P-strategies σ_i may be defined as follows. For each i, σ_i invokes the i-th f and no other function. One can verify that each σ_i is optimal. □

We denote by $safe(R)$ and $L2R(R)$ the set of words for which the optimum strategy and the optimum left-to-right strategies win, resp. The term "safe" comes from [10]: the words in $safe(R)$ are those that can be safely rewritten to a word in the target language, independently of the choices of the adversary.

When a target language R has an optimum 1P-strategy, say σ, we will denote $1P(R)$ the set $\mathcal{L}(R, \sigma)$.

We proved above the existence of optimum general and L2R strategies. Observe that this was not a constructive proof. Indeed, it is in general undecidable whether a winning strategy exists for a given word w and a target language R [13]. In contrast, for left-to-right rewritings, one can construct the optimum strategy. More precisely, it was shown in [13] that (1) for every R the set of words for which a winning left-to-right strategy exists is regular, (2) the regular language for it can be effectively defined, and (3) a optimum corresponding rewriting strategy can be computed. Note however that this construction is rather expensive. It was proved to be double-exponential even for some very restricted cases. This motivates our quest for more efficient optimal 1P-strategies.

For some rewriting system and a target language, there is no optimum 1P-strategy. One can question whether can construct an optimal 1P-strategy. The issue is still open. However, we next show that one can construct an *optimal one-pass strategy with size*. Note that this is precisely what matters in practice since size is typically available. The proof is quite involved and will thus be only sketched.

THEOREM 2.4. *For each rewriting system and each target language R, there exists a decidable, optimal 1PS-strategy.*

PROOF. (sketch) Given a system $(\Sigma, \Sigma_f,)\tau$ and a target language R, we transform the problem of 1PS-strategy for this system into a series of problems of L2R strategy for another system $(\widehat{\Sigma}, \widehat{\Sigma'}), \widehat{\tau}$ and a new target language $\widehat{R}$, as follows.

The L2R-system will use two new function symbols h (for hidden symbol) and $\tilde{h}$ (for last hidden). Each one will provide a letter. The intuition is the following. Consider a call to f in the 1PS-strategy with f:-R. It returns a string that starts by some g and says the result is of length n, say 5. It will be simulated in the L2R system by considering that f returned $g[:: R/g]hhh\tilde{h}$ where $[:: R/g]$ is an annotation that will be explained shortly. For now, we note that R/g is the quotient of R by g, i.e., the regular language consisting of all the words w such that gw is in R. In general, one function symbol will possibly be carrying left and right annotations: what it was left to do and what it leaves to its successors.

Now, nothing would prevent the first h above from choosing to generate a word that is different from R/g. This will be taken care of by the target language.

The idea is that at some point in time we can represent all the knowledge of the 1PS-strategy in the L2R-system. The 1PS-strategy has a choice between calling some f or not. It will consider the set of words for the stack for which there is a L2R-winning strategy depending of that choice. If one is larger, follow its advice. Otherwise, if they are equal or incomparable: it does not matter which one I call, so for instance, read f. □

The exact complexity of computing this optimal 1PS-strategy is still open. It is also opened whether like for L2R-strategies, the set of words for which this particular strategy is winning, is regular. We strongly conjecture that it is indeed regular.

The situation is simpler when there is a unique optimum strategy.

Proposition 2.7. *When a target language R has a* unique *optimum 1P-strategy, the set of words $1P(R)$ for which this strategy wins is regular.*

The proof follows from Theorem 3.1 and Proposition 3.1 in next section, showing how to construct a regular language for $1P(R)$.

We conclude this section comparing the respective power of one-pass, L2R- and arbitrary rewriting strategies.

Theorem 2.5. 1. *For each R and each 1P-strategy σ, $\mathcal{L}(R,\sigma) \subseteq L2R(R) \subseteq safe(R)$.*

2. *There exist R s.t. $L2R(R) \subset safe(R)$. There exists R s.t. for each 1P-strategy σ. $\mathcal{L}(R,\sigma) \subset L2R(R)$.*

3. *One cannot decide given R, whether $safe(R) = L2R(R)$, and whether $safe(R) = \mathcal{L}(R,\sigma)$ for some 1P-strategy σ.*

4. *When R has a unique optimum 1P-strategy, it is decidable whether $1P(R) = L2R(R)$.*

Proof. (sketch) The proof for (1) follows from the definition of the 1P, L2R, and arbitrary (optimum) strategies. The proof for (2) is by Examples 2.3 and 2.6.

For (4) The decidability for $1P(R) = L2R(R)$ follows from the fact that, in this case, both $1P(R)$ and $L2R$ are regular. (See Proposition 2.7 above, and its preceding discussion). Hence equivalence can be tested by comparing the corresponding regular languages.

The undecidability proof for (3) is by reduction to the acceptance problem for Turing machines. Given a Turing machine M, we constructs a target schema R_M s.t. $safe(R_M) = L2R(R_M) = 1P(R_M)$ iff the Turning machine accepts no words. The construction follows a line similar to the one used in [13] for showing that testing whether $w \in safe(R)$ for some word w and a target language R is undecidable. We sketch the main idea below.

The undecidability proof in [13] constructs, given a Turing machine TM with initial state q_0, a target schema S_{TM}. The schema is constructed such that a word w of the form $w = q_0(0+1)^* \diamond^* \odot$ has a safe rewriting iff (1) the sequence of zeros and ones between the symbols q_0 and $\diamond$ describe a word that is accepted by the Turing machine TM, and (2) the number of $\diamond$ symbols in the word correspond to the amount of space consumed by the machine when running on this word. (Each $\diamond$ represents a "blanc" cell in the machine tape, to be used by the computation, and $\odot$ denotes the end of the tape.)

In [13] the $\diamond$ was a constant symbol. In our proof we use it instead as a function $\diamond$:- $\diamond\diamond$ that can generate as many cells as needed (so there is no need to know in advance the required space). We also defined two additional functions g:-$b + b'$; $f \rightarrow a$.

Let Σ denote the set of all letters. Our target schema $R = R_1 + R_2$ is the following.

$$R_1 = S_{TM}\ (fb + ab')$$
$$R_2 = (\Sigma^* - (q_0(0+1)^* \diamond^* \odot))fg$$
$$R = R_1 + R_2$$

Observe that $safe(R) = safe(R_1) \cup safe(R_2)$. Thus, a word $w = w'fg$ has a safe rewriting to R iff it is in R_1, so w' is of the form $q_0 a_1 \ldots a_n \diamond^* \odot$ where $a_1 \ldots a_n$ is accepted by the Turing machine TM, or it is in R_2, so w' is not an encoding of an input for the machine.

One can show that $L2R(R_1) = 1P(R_1)$ is empty and $safe(R_2) = L2R(R_2) = 1P(R_2)$. So, $safe(R) = L2R(R) = 1P(R)$ iff $safe(R_1)$ is empty, i.e., if there is no word accepted by the Turing machine. □

The above result shows that, in general, one may loose some potential rewritings when resorting to restricted strategies. Interestingly, it turns out that this problem does not occur for a large class of schemas, introduced in the next section, that holds certain unambiguity properties. Indeed, we will see that for these schemas $safe(R) = L2R(R) = 1P(R)$, hence a 1P-rewriting suffices for achieving the best possible results.

3. REGULAR REWRITING

In this section, we consider a particular class of 1P-rewritings called *regular rewritings*. In regular rewritings, the rewriting strategy is defined by a simple computational device, namely a finite state automaton. We particularly focus on a class of regular rewritings that obey certain unambiguity conditions. These are similar to restrictions imposed, in standard XML, to ensure efficient document validation on XML schemas [18]. As for XML, we will see that, for AXML, unambiguity facilitates efficient document rewriting.

1-unambiguity. First, we briefly recall the notion of unambiguity. Intuitively, the types assigned to XML elements are required to be unambiguous in the sense that when arriving to a given element, (after processing its preceding siblings), the type of the element can be inferred in a deterministic way by looking only at the element name. This entails that for each element, the regular expressions R associated with the element type is required to be 1-unambiguous [5]. A typical example of a 1-ambiguous regular expression is $(ab+ac)$. When reading the a, we cannot decide whether we are parsing ab or ac. In this simple case, the equivalent regular expression $a(b+c)$ is 1-unambiguous. The formal definition of 1-ambiguity is rather complex and for space reasons we omit it here. A property of such regular expression that is of interest here, is that when R is 1-unambiguous, *the* deterministic automaton $A(R)$ corresponding to R can be constructed in PTIME [5]. This automaton is such that (i) each state is reachable and (ii) for each non-error state, there exists a word that brings from this state to an accepting state.

In the following, when we do not assume that R is 1-unambiguous, we still associate, w.l.o.g, to R a deterministic automaton $A(R)$ with the properties (i) and (ii) above; the only difference is that the size of this automaton may now be exponential in that of R.

For brevity, we overload below the notation and, whenever clear from the context, use R sometimes to denote both the regular expression and its corresponding deterministic automaton $A(R)$.

Regular strategies. A *regular strategy* is a 1P-strategy where the decision of which functions to call is performed by an FSA. More precisely, a regular strategy is specified by a deterministic automaton B. The automaton computes exactly like a standard finite state automaton except that when reading a function f in state q, if B has no transition for symbol f, rather than detecting an error and halting, the function f is invoked and replaced by some word in R_f (the value is selected by the adversary). Then the run continues with the modified word. We say that the regular strategy wins on input w if independently of the adversary, the machine reaches the end of the word in an accepting state ($\forall$ acceptance). We denote by $\mathcal{L}(B)$ the sets of words for which the regular strategy wins. Observe that this is not taking into account any target language R. We can assume w.l.o.g that the regular strategy B also does the test for R. Namely, when machine B reaches the end of a word in an accepting state, the word on the tape is in $L(R)$. In this case, Juliet, aka Regular Strategy B, won the game for R; otherwise Romeo won. Given a regular strategy B and a target language R, the set of words for which the regular strategy wins is $\mathcal{L}(R,B)$ (or $\mathcal{L}(B)$ when it is understood that B also checks for R).

A main contribution of this paper is a study of regular strategies for some target language R, and their relationship to other rewriting strategies.

Properties of regular strategies. We have seen above that the set of words $L2R(R)$ for which the optimum L2R-strategy wins, is regular, and similarly for the optimum 1P-strategy when it is unique. (Proposition 2.7). Interestingly, for regular strategies, the set of word won by *any* regular strategy is regular (even for the non optimum ones).

PROPOSITION 3.1. *For each regular strategy B, $\mathcal{L}(B)$ is regular. (For each regular strategy B and target language R, $\mathcal{L}(R,B)$ is regular.)*

PROOF. (sketch) To prove the proposition we show how to construct a particular *universal* FSA for $\mathcal{L}(B)$ in time polynomial in the size of B, and τ (the definition of function signatures). This universal automaton, denoted $Accept(B)$, accepts a word if it is accepted in *all* possible runs[3]. The moves of the universal automaton are defined using a datalog program. For each function name f and each subexpression R' of R_f (including R_f itself) with the exception of $R' = e$ for some letter e, let $a(R')$ be a fresh alphabet symbol not in Σ. We assume that for each letter e, $a(e)$ is e itself. The datalog program $P(B,\sigma)$ defines a relation $Move(q,e,q')$ denoting the fact that the device moves from state q to q' with the letter e. We have the following rules.

1. $Move(q,e,q') \leftarrow .$
   for each q, q', e such that B moves from q to q' when reading e (e is possibly a function symbol).

2. $Move(q,f,q') \leftarrow Move(q,a(R_f),q')$
   if there is no transition from q with letter f, i.e., f has to be called by B.

3. We have rules for defining the transitions with each R':

   - $Move(q,a(R_1R_2),q') \leftarrow Move(q,a(R_1),q''), Move(q'',a(R_2),q')$
   - Similarly for $a(R_1^+), a(R_1^*), a(R_1?), a(R_1{+}R_2)$.

The transition function δ of $Accept(B)$ is given by: For each q, q' and each $a \in \Sigma$, $q' \in \delta(q,a)$ iff $Move(q,a,q')$ holds. The accepting states are those of B. $\square$

A deterministic version of the universal automaton described in $Accept(B)$ can be constructed as for standard non-deterministic automaton. (The only difference from the standard powerset automaton construction is that now the accepting states are the sets where all states are accepting.) The size of the deterministic automaton may be exponential in the size of $Accept(B)$. The next proposition shows that there are cases where this exponential blowup is indeed necessary. Note that the construction for L2R-strategies requires two exponentials [13].

PROPOSITION 3.2. *There exists B such that any deterministic automaton for $\mathcal{L}(B)$ must have a number of states exponential in the size of B.*

PROOF. (sketch) We will use in the proof the following language. For some fixed n: $R_n = (0+1)^*0(0+1)^n$. There exists a non-deterministic automaton for R_n with $O(n)$ states, but every deterministic one must have a number of states exponential in n [6].

Let $\Sigma_f = \{0, n_0, l_0\}$ and $\Sigma = \{1\} \cup \Sigma_f$ with the signatures: 0:-(n_0+l_0); n_0:-1; l_0:-1. Consider a regular strategy B defined as follows. The automaton loops in a state q_0 when reading 1's. When it reads a 0, it calls the function and stays in q_0. When it reads n_0, it calls it and transforms it into 1 while going back to state q_0. In state q_0, it accepts. When B reads l_0 (think of it as "last 0"), it starts counting characters. After counting exactly n letters, it goes into a

[3]This differs from the standard nondeterministic automaton that accept a word if it is accepted by *some* possible run.

non-accepting state. All others states are accepting. Details omitted. The regular strategy B wins on all sequences of zero and one, except for those in R_n. So, there cannot be a deterministic automaton for $\mathcal{L}(B)$ with a number of states that is polynomial in n, since this would easily provide a deterministic automaton for R_n with a number of states polynomial in n. $\square$

1P- vs. regular-strategies. We next examine the relationship between arbitrary 1P strategies and regular ones. An ideal situation would be if (*) each optimum 1P-strategy had an equivalent regular one. This would mean that for such strategies, we do not loose any rewriting power when resorting to the simpler FSA computation. Unfortunately, (*) is still open. We can show the following, weaker but nevertheless encouraging results.

THEOREM 3.1. *1. If a target language R has a* unique *optimum 1P-strategy, then this strategy is regular.*

2. Every optimum 1P-strategy of a target language R has a semi-equivalent regular strategy[4]*.*

3. Furthermore, if R is 1-unambiguous, the regular strategy can be obtained from $A(R)$ by simply removing some of its states and transitions.

PROOF. (sketch) We first prove (1) and its counterpart in (3). Consider the deterministic automaton without redundant states for the target language R. For a state q_i in R, we use R_i to denote the suffix language corresponding to the paths that start at q_i and lead to an accepting state of R.

Let σ be the unique optimum 1P-strategy, i.e. for all distinct 1P-strategies σ', $\mathcal{L}(R,\sigma') \subset \mathcal{L}(R,\sigma)$. Let W denote the set of all words to which words in Σ^* may be rewritten into, using the 1P-strategy σ. Let R' be the automaton obtained from R by deleting all the states that are never reachable from the start state of R, when running words in W. Let q be some state in R' with an outgoing edge labeled by a function f having the output type R_f. Let q_f denote the state to which the edge of f leads. Let W_f denote the set of words to which words in R_f may be rewritten into, using the σ, and let $q_1 ... q_k$ be the states to which we get, starting from q, when running on R' words in W_f.
(*) It must be the case that either
$\cap_{i=1...k}\mathcal{L}(R_{q_i},\sigma) \subset \mathcal{L}(R_{q_f},\sigma)$, or
$\mathcal{L}(R_{q_f},\sigma) \subset \cap_{i=1...k}\mathcal{L}(R_{q_i},\sigma)$.
For suppose not. This contradicts the fact that σ is a unique optimum strategy. There are two cases that need to be examined:

1. $\cap_i\mathcal{L}(R_{q_i},\sigma) = \mathcal{L}(R_{q_f},\sigma)$. Consider some word wf where σ, when applied on w generates a word that, when run on R', brings to q. If σ decides to call (or resp. not to call), consider an alternative strategy σ' that works always exactly like σ except that on wf chooses the other option. Then σ' is different than σ and it is also an optimum, which contradicts the hypothesis.

2. suppose that there are two words, w', w'', s.t. $w' \in \cap_i\mathcal{L}(R_{q_i},\sigma)$ but $\notin \mathcal{L}(R_{q_f})$, and $w'' \in \mathcal{L}(R_{q_f},\sigma)$ but $\notin \cap_i\mathcal{L}(R_{q_i},\sigma)$. Consider again the input word wf from above. If σ decides to call f, then consider an alternative strategy σ' that behaves exactly like σ except that on wf it does not call f. One can see that $wfw'' \in \mathcal{L}(R,\sigma')$, but $\notin \mathcal{L}(R,\sigma)$, which contradicts the fact that σ is an optimum. Similarly, if σ does nor call f.

[4]Recall from Section 2 that a semi-equivalent strategy is one where Romeo wins on exactly the same words.

This concludes the proof of (*).

This yields an optimum regular strategy. For every state q, do not call f if $\cap_i\mathcal{L}(R_{q_i},\sigma) \subset \mathcal{L}(R_{q_f},\sigma)$, and call f if $\mathcal{L}(R_{q_f},\sigma) \subset \cap_i\mathcal{L}(R_{q_i},\sigma)$. This strategy can be described by the FSA obtained from R' by deleting the f edges from all states q where $\mathcal{L}(R_{q_f},\sigma) \subset \cap\mathcal{L}(R_{q_i},\sigma)$. To compute it, it suffices to consider all automata B obtained from R by removing some nodes and edges labeled by function names. By Proposition 3.1, one can then compute the automata corresponding to the languages for $\mathcal{L}(R,B)$ and compare the automata to find the one with a maximum language.

The proof for (2) (and its counterpart in (3)) follows similar lines. The only difference is that when there are several optimum strategies it is possible that $\cap_i\mathcal{L}(R_{q_i},\sigma) = \mathcal{L}(R_{q_f},\sigma)$ for some q. Strategy σ may decide, when in state q, sometimes to call f and sometimes not. We are not allowed to do in a regular strategy. So, we have to choose one of the two options. However, observe that the suffixes accepted by the two choices are the same. So Romeo will not win with B on more words than he did with σ. $\square$

To conclude observe that just like for general 1P-strategies, the optimal regular strategy for a given target language may not be unique. And furthermore there may be infinitely many incomparable optimal regular strategies. To see this, consider the proof of Theorem 2.3(2).

Unambiguous rewriting. Given a target language R, we are interested in regular strategies for R. We have seen that depending on R: (1) the strategy may be very complex (testing membership in $\mathcal{L}(B,R)$ may require an FSA with an exponential number of states), (2) there may be infinitely many optimal strategies and (3) resorting to regular strategies may cause us to miss some possible rewriting.

We would like to impose syntactic restrictions that are easy to test, reasonable in practice and prevent (1), (2) and (3) to happen. As we will see, such restrictions exist. They are similar in spirit to the ones typically imposed on XML schemas and are essentially based on preventing some forms of ambiguity. We next introduce a very simple and natural one. It is based on the following classical notions:

- We denote by $sem(f)$ the set of words to which f may rewrite to.

- For each language L, $start(L) = \{a \mid a \in \Sigma \text{ and } \exists u(au \in L)\}$.

It is easy to see that $sem(f)$ is context-free and not always regular. Clearly, for each context-free language L, it is straightforward to compute $start(L)$.

DEFINITION 3.2. *A 1-unambiguous regular grammar R is said to be* rewriting-1-unambiguous*, denoted r1-unambiguous, if in its corresponding automaton $A(R)$, there does not exists a state q with transitions for both a function name f and some letter $a \in start(sem(f))$.*

PROPOSITION 3.3. *If the target schema R is r1-unambiguous, then $safe(R) = L2R(R) = 1P(R) = \mathcal{L}(R, R)$, i.e., the regular strategy, defined by the automaton R itself, provides an optimum rewriting strategy for the target language R.*

PROOF. (sketch) Consider how R operates when playing the role of a regular strategy. When a function call f is met in state q, either q has an outgoing edge labeled f, in which case f is not invoked, or otherwise the function must be invoked. The crux is that when q has an outgoing edge labeled f, there is no point in considering the option of invoking f since, due to the r1-unambiguity, we know that its output cannot match any of the edges outgoing from q. $\square$

Clearly, r1-unambiguity is easy to test. Furthermore, this test may be reduced to a standard test for 1-unambiguity. This is of practical importance for it means that (transposed to AXML documents) the test can be performed using a standard XML schema validator. To see that, let $check(R)$ be the grammar obtained from R by replacing each function name f by $(a_1+...+a_n)$, where $start(sem(f)) = \{a_1, ..., a_n\}$. One can see that $check(R)$ can be computed in PTIME and that

R is r1-unambiguous iff $check(R)$ is 1-unambiguous.

Clearly r1-unambiguity is a sufficient but not necessary condition for Proposition 3.3, as illustrated by the following example.

EXAMPLE 3.4. *Consider a target schema $R = a(fc+deg)$ with f:-dh; g:-c'. It is not r1-unambiguous for the same arguments as above.*

However, consider a word $w = af...$ When reading f, one may be tempted to call it since $start(sem(f))$ contains d and d is meaningful after reading a. However, dh, the return value of f, does not match deg, the suffix allowed by the schema, so there is no point in calling f. Indeed, one can find an optimum rewriting strategy for R which is regular, although it is not r1-unambiguous. $\square$

One can define more general (and more complex) unambiguity conditions. (Omitted here for space reasons). In some sense, Theorem 3.1 also provides some condition of unambiguity. The advantage of r1-unambiguity is that it is easy to test and seems reasonable in practice. One example where this is too restrictive is, for instance, when a document may contain a phone number or a function returning a phone number. In such case, calling or not calling just does not make a difference. This is quite different from situations where the success of rewriting depends on making the right decision.

4. NON-BLOCKING REGULAR REWRITING

Web services typically take time to answer. Rather than blocking and waiting for the service answer, we would like to continue meanwhile and rewrite the remaining of the word, going back to processing the service result only when it arrives. A difficulty is that the decision whether to call or not future functions may depend on f's answer[5]. Non-blocking rewriting is the main problem studied in [4]. We consider here the issue in the context of regular strategies.

[5]In principle, one could also call some services in advance "just in case" their result is needed (in the spirit of pre-fetching). This may be undesirable if the services have side effects or induce some fees. We do not consider this here.

The following example illustrates the issue of non-blocking rewriting.

EXAMPLE 4.1. *Consider a target language $R = (ag + a'b)$ with f:-$(a + a')$; g:-b. Suppose we try to rewrite the word $w = fg$. When we read f, we know that we have to call it. But we need to wait for its return value to decide if to call g or not. On the other hand, for a target schema $R' = (a+a')b$ we can call f and, without waiting for its answer, proceed and call g immediately. However, f may take us to two distinct states of the automaton for R, which we may view as some kind of ambiguity.* $\square$

Homogeneous unambiguity.

One can impose in practice some radical restrictions both on the schema and on the function signatures to guarantee a non-blocking rewriting. For instance, suppose that we only allow functions that return nonempty homogeneous collections, e.g., one or more b elements (for some b). We will say that such functions are *homogeneous.* Note that now, the only freedom that the adversary (Romeo) has is in the number of elements that are returned in the answer. The following notion of *homog-unambiguity* guarantees that the number of returned elements makes no difference for the rewriting.

Definition: A 1-unambiguous regular language R is *homog-unambiguous* if it is r1-unambiguous and in each state q in R and each b, any nonempty sequence of b's brings to the same state q_b.

In some sense, all is now deterministic: the choice to call or not to call a function (because of r1-unambiguous) and the effect of the adversary choice (because of homog-unambiguous). Under such restrictions the rewriting needs not wait for the result of the function and can continue immediately the rewriting for the remaining of the word, from state q_b.

Analysis of potential runs.

A non-blocking service invocation is possible also for non homog-unambiguous target schemas, but requires a more delicate analysis. We next sketch the main principles.

Recall from the previous section the $Move$ predicate defined by the datalog program in the proof of Proposition 3.1. Given a regular strategy B, a state q in B and a function f with signature R_f, the set of states

$$\{q' \mid Move(q, a(R_f), q') \text{ holds}\},$$

are the states of B that may be reached from q after rewriting some possible output of f. Observe that, for homog-unambiguous schemas, this set consists of the single state q_b, but that this set typically may have more that one elements. Since we cannot know in advance which of these states will indeed be reached, we need to consider all possible runs for the remaining of the word. More precisely, the set of states which may be reached from q, when reading a sequence of letters $a_1 \ldots a_n$ (of element or function names), can be computed by enriching the datalog program with the following rules, for $j = 2 \ldots n$:

$$\begin{aligned} Move(q, a(a_1...a_j), q') \;\leftarrow\; & Move(q, a(a_1...a_{j-1}), q''), \\ & Move(q'', a_j, q') \end{aligned}$$

Now, suppose that a_j is the function f'. Consider all the states q'' such that $Move(q_0, a(a_1...a_{j-1}), q'')$ holds. The function f' (a_j) is invoked for sure if none of these states q'' has an outgoing edge labeled by f'. Alternatively, if all such states contain an outgoing edge labeled by f, we know *for sure* that the function must not be invoked. If some of the states have an outgoing edge labeled by f' and some not, no sure decision can be made for f', in other words, it depends on the actual output of the previous functions. So one can decide in PTIME which function occurrences will surely be called and which surely not.

Observe that deciding whether a function occurrence is called or not is very closely related to deciding whether the strategy is winning for a word $a_1 \dots a_j$. Indeed, the strategy is winning if for each q such that $Move(q_0, a(a_1...a_j), q)$ holds, q is an accepting state.

We omit the formal definition of the notion of *sure decisions* and the correctness proof for the above results for space reasons.

5. CONCLUSION

Motivated by the exchange of Active XML documents, we studied several document rewriting strategies, including regular, one-pass with and without size, L2R and general rewritings. More generally, our results can be interpreted as results on context-free games in the sense of [13] with the bias that we are primarily interested in one-pass strategies. Perhaps the most important open question in this setting is whether for each problem, there is an optimal one-pass strategy that is regular.

Typing issues in XML Schema have recently motivated a number of interesting works such as [15, 11, 9] that are based on tree automaton. In the present paper, the focus was on strings and automata. However when (AXML) trees are considered, tree automaton clearly becomes the essential tool.

We are currently implementing a new rewriting module for AXML. The simplicity of the implementation and its efficiency are essential issues in particular for devices with limited computational power. In particular, one of our goals is to implement the rewriting module using a standard XML validator to benefit from the genericity and efficiency of such tools, and also limit the quantity of the AXML specific code needed in a peer, a critical aspect for small devices. Intuitively, the implementation would run the XML validator and, when an error is detected, analyze whether this is a a "real" XML error or simply a function that needs to be invoked. So, we basically have to catch exceptions and proceed from there. For technical reasons that will be omitted here, it is not immediate to do so with validators such as Xerces [17].

Finally, the focus here was on deterministic and winning rewriting strategies. We already mentioned the interest of non-deterministic rewriting. As discussed in [10], there are cases when the rewriting is not guaranteed to succeed, therefore one would like to consider "possible rewriting". We believe that the study presented here on 1P and regular rewriting could be similarly extended to possible rewriting but this still remains to be explored.

Acknowledgments. We thank Luc Segoufin and Anca Muscholl for discussions on the topic.

6. REFERENCES

[1] Serge Abiteboul, Omar Benjelloun, Bogdan Cautis, Ioana Manolescu, Tova Milo, and Nicoleta Preda. Lazy query evaluation for active xml. In *Proc. of ACM SIGMOD*, pages 227–238, 2004.

[2] The Active XML homepage. `http://activexml.net/`.

[3] Open Source Active XML (activexml). `http://forge.objectweb.org/`.

[4] O. Benjelloun, T. Milo, and S. Raj Mathur. Towards efficient exchange of active xml data, 2004. Technical Report.

[5] Anne Bruggemann-Klein and Derick Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, 1998.

[6] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 1979.

[7] Jakarta Project, Jelly: Executable XML. `http://jakarta.apache.org/`.

[8] Macromedia Coldfusion MX 6.1. `http://www.macromedia.com/`.

[9] W. Martens, F. Neven, and T. Schwentick. Which xml schemas admit 1-pass preorder typing? In *Proc. of ICDT*, 2005.

[10] T. Milo, S. Abiteboul, B. Amann, O. Benjelloun, and F. Dang Ngoc. Exchanging intensional XML data. In *Proc. of ACM SIGMOD*, 2003.

[11] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 11–22. ACM, 2000.

[12] M. Murata, D. Lee, and M. Mani. "Taxonomy of XML Schema Languages using Formal Language Theory". In *Extreme Markup Languages*, Montreal, Canada, 2001.

[13] A. Muscholl, T. Schwentick, and L. Segoufin. Active Context-Free Games. In *Proc. of STACS*, 2004.

[14] J. Powell and T. Maxwell. Integrating Office XP Smart Tags with the Microsoft .NET Platform. `http://msdn.microsoft.com`.

[15] T. Schwentick. Trees, automata and xml. In *Proc. of ACM PODS*, 2004.

[16] The World Wide Web Consortium. `http://www.w3.org/`.

[17] The Xerces Java Parser. `http://xml.apache.org/xerces-j/`.

[18] The XML Schema specification. `http://www.w3.org/TR/XML/Schema`.