Auto-completion learning for XML

Serge Abiteboul, Yael Amsterdamer, Tova Milo, Pierre Senellart

Auto-Completion Learning for XML

Serge Abiteboul
Collège de France
INRIA Saclay & ENS Cachan
serge.abiteboul@inria.fr

Yael Amsterdamer
INRIA Saclay
Tel Aviv University
yaelamst@post.tau.ac.il

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

Pierre Senellart
Institut Télécom; Télécom ParisTech
CNRS LTCI
pierre.senellart@telecom-paristech.fr

ABSTRACT

Editing an XML document manually is a complicated task. While many XML editors exist in the market, we argue that some important functionalities are missing in all of them. Our goal is to makes the editing task simpler and faster. We present `ALEX` (Auto-completion Learning Editor for XML), an editor that assists the users by providing *intelligent auto-completion suggestions*. These suggestions are adapted to the user needs, simply by feeding `ALEX` with a set of example XML documents to learn from. The suggestions are also guaranteed to be compliant with a given XML schema, possibly including integrity constraints. To fulfill this challenging goal, we rely on novel, theoretical foundations by us and others, which are combined here in a system for the first time.

Categories and Subject Descriptors

I.7.2 [**Document and Text Processing**]: Document Preparation—*XML*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces

General Terms

Algorithms, Design

Keywords

XML, editor, auto-completion, learning, schema

1. INTRODUCTION

XML is an extremely common and useful format for representing semistructured data. While XML is sometimes automatically generated, e.g., in Web publishing, it is often the case that XML documents must be *manually edited*. Popular examples include technical documentation in the DocBook format, XHTML, and even XML Schema documents. Editing an XML document manually is a complicated task, and an XML editor may provide functionalities that simplify the lives of the users. We argue that while existing XML editors provide many such functionalities, some important ones are missing. (We will describe them further.)

We present `ALEX` (Auto-completion Learning Editor for XML). The uniqueness of `ALEX` lies in its ability to use example documents to give the user *intelligent auto-completion suggestions*. For that, we use a *novel model* [1] that captures, in a compact manner, the structural distribution of the examples. This model is unique since it also allows expressing intricate dependencies that arise from the schema structure and from integrity constraints. Using this model, we are able to generate document parts that are compatible with the schema and resemble the examples. These parts are then suggested to the user as auto-completions, which makes the editing process significantly simpler and quicker.

To illustrate the needs for editing XML documents as well as the solution, consider a situation that every researcher regularly faces: updating the personal data on their homepage. Our example researcher is a very well-organized one, and keeps her data in an XML document conforming to a standard schema for personal information. This schema allows entering information about activities, positions, projects, interests, various publication types, contact details, and more, in a structured manner. Later on, this information can be automatically rendered and displayed on the researcher Web site. She would like to edit this document (e.g., add new activities), as quickly as possible and without much hassle.

First, the researcher does not know the schema by heart. Typically, such a schema is very rich and complex, and contains parts that are more relevant to her, as well as totally irrelevant ones. For instance, a full-time researcher may not be concerned with parts of the schema related to teaching. Thus, the researcher would like to get *suggestions on what to write*, that are both compliant with the schema and adapted to her current preferences (she may change these later, e.g., when changing positions). Many XML editors, such as Rinzo [9], suggest all possible next nodes (elements or attributes) according to the schema, but are not adaptive.

Second, instead of getting auto-completion suggestions for one node at a time, our researcher would like to be presented with suggestions containing *large bulks* of the desired XML structure. For instance, if she wants to add a publication

SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.

in conference proceedings, she would like a suggestion for the entire structure of such publication, with the most useful fields (such as title, authors, etc.). Here, user preferences play an even more significant role: as the auto-completion suggestions get larger, the number of possible completions may sharply increase. Thus, we should not show all possible suggestions, but rather only the most relevant ones.

Last, the schema our researcher uses may contain *integrity constraints*, e.g.: all publication titles must be distinct. This greatly complicates matters, since many documents that adhere to the schema structure may not be valid w.r.t. the constraints. In this case, the editor must only suggest completions to valid documents. It must also alert on constraint violations, in case the user manually types in data.

ALEX has been designed to fulfill all the aforementioned user needs. Suppose our user has a set of examples for relevant documents. This set may include homepages of colleagues from her field, the previous version of her homepage, etc. ALEX can use this set to learn a probabilistic distribution over the different schema parts, based on their frequencies in the examples. Then, this distribution can be used to randomly generate and rank the possible completions that are valid w.r.t. the schema, also supporting integrity constraints. The solution we provide is generic, and works for any schema structure, as well as important types of integrity constraints encountered in practice. It is also highly adaptive, since the user can change the editor behavior simply by adding or removing document examples. There are several technical challenges in designing and implementing such a solution. In particular, those challenges include: (i) learning a model that is compact, and describes the structural distribution in the examples (this is different from a standard data mining task, because of the need to support the intricate schema structure and integrity constraints); and (ii) using the model that has been learned to generate auto-completions relevant to the particular user needs. For that, we rely on solid theoretical foundations, developed by us and others [1, 2, 6]. The algorithms we use from these works are computationally non-trivial, and in particular the algorithm of [6] solves an NP-hard problem. We show that, in practice, using optimizations makes these algorithms feasible. In addition, the combined use of these algorithms, which we do for the first time, requires additional non-trivial adaptations.

We used the homepage example to explain the requirements from an XML editor and mention challenges. Auto-completion is more generally useful in a number of editing tasks. It is often the case, e.g., for DocBook, that the schema includes a large number of options, and that only a fragment of them are used in a given document. Also, schema languages such as XML Schema (hereafter referred to as XSD) support the use of integrity constraints, and thus a generic editor needs to take them into account as ALEX does.

Demonstration Scenario. Our demonstration will show the usefulness of ALEX for the task of creating a personal homepage. At the end of the demonstration, the participants will be able to "take home" a personalized homepage they created, in both XML and HTML formats. We use a general schema for the data, inspired by the FOAF format for personal details [4]. We collected from the Web several sets of example documents complying with this schema, containing real data about computer scientists.

We will demonstrate the use of ALEX for users with different profiles, choosing different sets of examples to feed into the system. To demonstrate the adaptivity of ALEX, we will play the roles of these users first, and then invite participants to experiment with the system. They will be able to freely choose examples for homepages. Then, they will provide those examples as input for ALEX, and use it to create XML documents with the data for their homepage. We will see what suggestions ALEX makes for each choice of examples, and demonstrate its effectiveness.

2. TECHNICAL BACKGROUND

In this section, we overview the theoretical model we use as a basis for the auto-completion. Given a set of example documents, we can *learn a model* that expresses the structural distribution in these examples, i.e., how frequently different schema parts are used. The model can then be used to generate random XML documents, according to the learned distribution, that are expected to resemble the examples. We adapt it to *generate XML parts* that can be plugged at a certain point of a given document – i.e., auto-completion suggestions. Finally, instead of generating randomly, we can deterministically generate *the most likely completions*, using a top-k algorithm. For further details, see [1, 2, 6].

The model. Let us start with briefly explaining the model for XML schemas (without constraints), which forms the basis for our model. In XSD, the possibilities for the children of a node are defined by a regular expression. For instance, we can state that for each a-labeled node, the sequence of labels of its children (from left to right) is in the regular language $\mathsf{a}^*\mathsf{bc}^*$. Instead of a regular expression, in our model we associate each node label with a deterministic finite automaton (DFA). An XML document is then modeled as a finite, labeled and ordered tree. We say that a document adheres to the schema if for every node, the sequence of its children's labels is accepted by the relevant DFA.

This schema model suggests an intuitive (nondeterministic) XML generator: start with a single node bearing the root label. Then choose an accepting run of its DFA. According to the word accepted by this run, $\mathsf{a}_1 \dots \mathsf{a}_n$, generate n children for the node, bearing the labels $\mathsf{a}_1 \dots \mathsf{a}_n$, and so on. To turn this into a probabilistic generator, we assign to each transition a probability, *t-prob*. These are the probabilities of the transitions to be selected in the course of generation. The choice of *t-prob*s is important, since we want the model to reflect the distribution of the examples. We use the algorithm of [1] for learning probabilities that maximize the likelihood of generating the example set. In a nutshell, this algorithm uses the schema to verify the example documents, and counts how many times each automaton transition was chosen. Then, the *t-prob* of each transition is the likelihood of choosing it according to the examples.

The generator defined here only generates the node structure but not data values. We use a method inspired by [1] for data value generation.

Auto-completion under integrity constraints. Real applications often involve some semantic constraints. We want to generate only auto-completions that lead to documents that are valid with respect to these constraints. Let a and b be some leaf labels. We consider here three kinds of global constraints on data values, which are also expressible in schema

languages, e.g., DTD or XSD: *key constraints*, imposing that all the values of a-labeled leafs are unique; *inclusion constraints*, imposing that for every value of a a-labeled leaf, some b-labeled leaf has the same value; and *domain constraints*, imposing that the values of a-labeled leaves are from some domain dom(a).

Allowing the use of such constraints increases the expressiveness of the schema, but poses a new difficulty: some document skeletons that adhere to the schema, may have no assignment of values that is valid w.r.t. the constraints. To avoid generating such invalid skeletons, we show in [1], based on the results of [6], that a *binary continuation-test* can be used on each transition chosen during the generation process. The continuation-test returns true iff a transition does not lead to a "dead end". This continuation-test is proved to be NP-complete, but when the schema is small relatively to the documents, the test is still feasible.

An adaptation is also required for the tuning of *t-prob*s in the presence of constraints, as explained in [1].

Finding completions in a specific point of the document. The model previously described can be used to generate an initial structure, when the user creates a new XML document. However, to allow full editing capabilities, we want to support generating document parts that can be plugged in a specific point of the document, according to user requests. Given as input the document, and a specific point in it, we first need to perform a reachability test, to see if there exists a non-empty sub-tree that can fit at this point. Then we start generating a sub-tree from the given point, but alter the continuation-test to make sure that the suffix of the original tree is eventually generated. This alteration is a novel adaptation of the original algorithm, which we created especially for the task of XML editing.

Finding the best completions. There may be unboundedly many completions for a certain document at a certain point. We can generate random completions, but we may also be interested in finding the *most likely completions*. For that, we use the top-k search algorithm from [2] that is based on the A* search algorithm.

Related work. As mentioned above, we use results from three papers [1, 2, 6], that provide the main theoretical background for the present demonstration. XML generation, which we implement here, was studied in different contexts (see, e.g., [3, 5]). We found our model from [1] convenient in the context of an XML editor, since it is automatically adapted to a set of examples. Among the various models that have been proposed for probabilistic XML and schemas, we believe our model is best suited for the task at hand because it allows generating all documents satisfying a given schema, and it supports the most standard integrity constraints.

As mentioned in the intro, ALEX provides unique features that, to the best of our knowledge, cannot be found in other editors. In contrast, other editors may provide different features: graphical representation of XML documents, debugging for certain types of XML languages, etc. Introducing such features in ALEX suggest interesting research directions.

Finally, auto-completion is widely used in different contexts, e.g., prediction of natural-language user input [8], and query auto-completion [7]. However, auto-completion in our context of XML editing faces new challenges, from learning from the *complex structure* of example documents, to resolving violations in the case of integrity constraints.

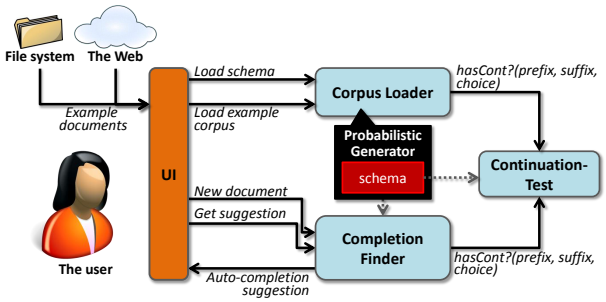

Figure 1: System Diagram

3. SYSTEM OVERVIEW

ALEX is written in Java, and is based on *Rinzo* [9], an open-source XML editor plugin for the Eclipse IDE. We next provide an overview of the main system components. These components are also shown in Figure 1.

Corpus Loader. This component gets as input an XSD file and a set of example XML files satisfying the schema. It then uses an algorithm based on [1,6] to learn a probabilistic generator that maximizes the likelihood of the examples.

Continuation-Test. This component is essentially the implementation of the continuation-test of [1], described in the previous section. It gets as input the schema tree automaton and constraints, a prefix and possibly a suffix of an XML document, and the choice of next transition. The result is Boolean; it is *yes* if there is a continuation for this particular next transition. The algorithm has been adapted to provide results as fast as possible for our demo scenario.

Completion Finder. This component gets requests for auto-completion in the form of XML document prefix and suffix. It has two modes, according to whether the prefix and suffix are empty (new documents) or not. In the first mode, k *random* document skeletons are generated, and the user may choose from them. In its second mode, the completion finder seeks the *top-k* sub-trees to plug-in between the prefix and suffix. In both modes the calls are made to the continuation-test component, to avoid reaching a dead end.

Editor User Interface. The UI of ALEX is based on an existing XML editor (the Rinzo Eclipse plugin) and thus is very intuitive and user-friendly. Via this interface the user may choose inputs to the corpus loader, to customize the auto-completion suggestions. He may then create a new XML file or choose to edit an existing one. During the editing process the user can ask for auto-completion suggestions from the completion finder in certain points of the document (according to the cursor location). A warning is displayed whenever the document violates the schema.

4. DEMONSTRATION

For the demonstration, we chose the editing task of the introduction example: updating a researcher's homepage. We will show the usefulness of ALEX for editing an XML document containing such data and, as already mentioned, at the end of the demonstration participants will be invited to create personalized homepages. Figure 2 shows a screenshot

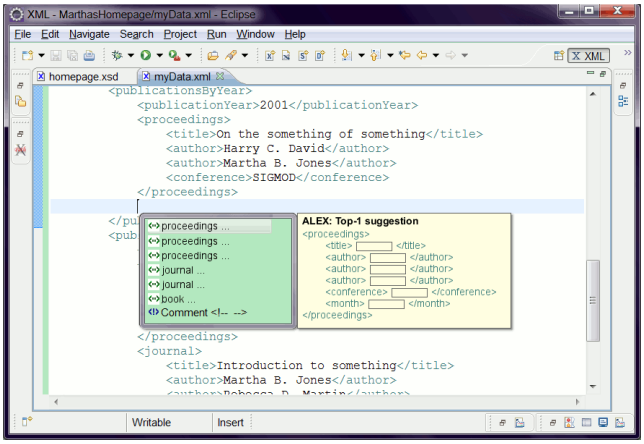

Figure 2: A screenshot of ALEX

of **ALEX**, with a list of auto-completion suggestions for the current cursor position. The panel on the right displays the content of the currently selected suggestion. Note that there are only place holders for data values (depicted as rectangles); the user can get suggestions for data values separately.

Settings. To represent personal data to display in a homepage, we use a general schema inspired by the FOAF format for personal details [4]. We have generated a corpus of example documents adhering to this schema, based on real data about computer scientists. This data was collected from the scientists information in *ArnetMiner* Web site [10] and from *DBLP*. To this data, we also add the XML documents created by the participants in the course of the demonstration. The documents include contact details, general information such as affiliation and area of expertise, publications, co-authors, etc. In addition, the documents express certain choices of style. For example, the presence of an image, or the order in which the data is displayed in the homepage. We used synthetic data for these choices, inspired by real homepages.

Let us now overview the course of demonstration.

Introducing **ALEX.** The goal of the first stage of the demonstration is to get the audience familiar with the schema, and to demonstrate the use of **ALEX**. We will start by playing the role of two researchers who want to create their homepages. To illustrate the adaptivity of **ALEX**, we will assume that those researchers come from different backgrounds, and have different tastes in style. Each of the researchers will accordingly choose some homepages they like, from the examples from the internet and those created by previous participants. For instance, one simulated researcher may belong to the SIGMOD community, and the other to PODS; and thus we will choose for each of them homepages of colleagues from their community. Two instances of **ALEX** running on two laptops in parallel will each load the examples chosen by one of the researchers. Then, we will create a new XML document using **ALEX** on each of the computers. We will show how, according to the difference in the chosen examples, **ALEX** may provide different auto-completion suggestions. For instance, an entry for a SIGMOD publication will probably be ranked lower among the suggestions for theoretical researchers; and the choice of style may affect, e.g., the presence of a picture. We will also see how **ALEX** prevents the violation of the schema structure or of the constraints.

Interactive participation. Now that the spectators are a bit familiar with **ALEX**, they can use the system. Volunteers will be invited to create homepages for themselves, each on a different laptop. Similarly to what we did in the first stage, they will choose examples for homepages, this time according to their personal preferences. They will provide those examples as input for **ALEX**, and use it to create the XML documents with their personal data. They will be able to evaluate the convenience and usefulness of **ALEX**. At the end, the created XML documents will be transformed into HTML documents, and displayed as actual Web-pages, that the participants will be able to take home. We will also offer the volunteers to perform the same editing task without **ALEX**'s intelligent auto-completion support. We will compare the different experiences, validating that **ALEX** indeed simplifies greatly the editing task and allows completing it much faster.

Under the hood. Until this point, the spectators are only exposed to **ALEX** from the user side. Now, we will invite them to take a peek under the hood into our probabilistic schema model. We will inspect the tree automaton used as a model for the schema, and see the differences in the learned probabilities for each of the chosen example sets. This will allow the users to understand better the different auto-completion suggestions each of the users got. It will also provide them with a better understanding of the technical background used in this work.

5. ACKNOWLEDGMENTS

This work has been partially funded by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC grants Webdam, agreement 226513, and MoDaS, agreement 291071, and by the Israel Ministry of Science.

6. REFERENCES

[1] S. Abiteboul, Y. Amsterdamer, D. Deutch, T. Milo, and P. Senellart. Finding optimal probabilistic generators for XML collections. In *ICDT*, 2012.

[2] Y. Amsterdamer, D. Deutch, and T. Milo. On the optimality of top-k algorithms for interactive Web applications. In *WebDB*, 2011.

[3] T. Antonopoulos, F. Geerts, W. Martens, and F. Neven. Generating, sampling and counting subclasses of regular tree languages. In *ICDT*, 2011.

[4] D. Brickley and L. Miller. FOAF vocabulary specification. `http://xmlns.com/foaf/spec/`, 2010.

[5] S. Cohen. Generating XML structure using examples and constraints. *PVLDB*, 1(1), 2008.

[6] C. David, L. Libkin, and T. Tan. Efficient reasoning about data trees via integer linear programming. In *ICDT*, 2011.

[7] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu. SnipSuggest: Context-aware autocompletion for SQL. *PVLDB*, 4(1), 2010.

[8] A. Nandi and H. V. Jagadish. Effective phrase prediction. In *VLDB*, 2007.

[9] Rinzo XML editor. `http://editorxml.sourceforge.net/`.

[10] J. Tang, J. Zhang, L. Yao, J. Li, L. Zhang, and Z. Su. ArnetMiner: extraction and mining of academic social networks. In *KDD*, 2008.