XTaGe: A flexible generation system for complex XML collections

María Pérez, Ismael Sanz, and Rafael Berlanga

Universitat Jaume I, Spain
{mcatalan,isanz,berlanga}uji.es

Abstract. We introduce XTaGe (XML Tester and Generator), a system for the synthesis of XML collections meant for testing and micro-benchmarking applications. In contrast with existing approaches, XTaGe focuses on complex collections, by providing a highly extensible framework to introduce controlled variability in XML structures. In this paper we present the theoretical foundation, internal architecture and main features of our generator; we describe its implementation, which includes a GUI to facilitate the specification of collections; we discuss how XTaGe's features compare with those in other XML generation systems; finally, we illustrate its usage by presenting a use case in the Bioinformatics domain.

1 Introduction

Testing is an essential step in the develoment of XML-oriented applications and in most practical settings, this requires the creation of synthetic data.

Existing XML generators focus on either the creation of collections of a given size (for stress testing and workload characterization purposes) or with a fixed schema and little variation. These systems do not suit the requirements of an emerging class of important applications in fields such as Bioinformatics and GIS, which have to deal with large collections that present complex structural features, and specialized content such as protein sequences or vectorial map data.

In this context, the main drawback of existing systems in our application context is the lack of extensibility, since all systems are limited by the support of a limited number of predefined generation primitives. Another limitation is the uneven support for the introduction of *controlled* variability in generated structures, useful for example for micro-benchmarking purposes. Finally, the specification of collections is generally done through the manual creation of a text-based specification file, which can be tedious and error-prone.

In this paper we introduce XTaGe (XML Tester and Generator), which focuses on the creation of collections with complex structural constraints and domain-specific characteristics. XTaGe contributes (i) a flexible component-based framework to create highly tailored generators, (ii) a ready-made set of components that model common patterns that arise in complex collections, (iii) easy adaptability to new use cases using a high-level language (XQuery itself)

The resulting system makes it possible to generate test collections whose characteristics would be very difficult, or impossible, to replicate using the existing generic XML generators.

Related work. As indicated above, current approaches for generating synthetic XML data can be classified as either *schema-unaware* or *template-based*. The former are based on the specification of a few global structural characteristics, such as maximum depth, fan-out and amount of textual data. They are commonly used in benchmarking applications. Examples include Niagdatagen [2], *xmlgen* (developed for the XMark [16]) and *genxml* (used in the X007 Benchmark [5]).

In contrast, template-based generators use as input an annotated schema that precisely describe the desired structure of the output documents. The best-known example is ToXgene [4], which defines an extension of XML Schema, the *Template Specification Language* (TSL) to describe the generated XML document content. It has some support for generating variability through the use of probability distributions to create heterogeneous structures. Many benchmarks applications, such as [17] and [14], generate their testing collections using ToXgene. Other examples of template-based generators are: VeXGene [10], MeMBeR [3] and [6], which bases the XML data generation on a DTD and examples of XML instances and support the specifications of constraints of the generated collection, such required XPath expressions.

As a special case, other approaches attempt to create new collections by transforming existing ones, such as [7], which can adapt existing documents for experiments meant to evaluate semantic query optimization methods; they provide a set of four transformations to adapt existing XML documents. Another relevant system is [15], which can modify the content of XML documents by creating duplicates or by removing content of the documents in order to create "dirty" documents suitable for testing data cleaning algorithms.

Outline of the paper. The remainder of this paper is structured as follows. First, the foundations of XTaGe are presented in Section 2. Then, the XTaGe component-based framework is described in detail in Section 3. In Section 4 we present the prototype and a use case that shows an application of the generation model. Finally, Section 5 presenta a short discussion of XTaGe's features and introduces directions of future work.

2 Foundations of XTaGe

One of the main goals of XTaGe is to provide precise control of the generated data when creating heterogeneous collections and, as a consequence, we will use template-based techniques as a basis for our approach. In this section we provide a formal basis for the definition of XML generators, which will allow us to define a flexible mechanism for the creation and adaptation of XML generators for complex domains.

We adopt the *XML Store* [9] as a suitable abstraction of XML collections, which is commonly used in the context of XML update languages. Following [8], we will use the following notations: the set $\mathcal{A}$ denotes the set of all atomic values, $\mathcal{V}$ is the set of all nodes, $\mathcal{S} \subseteq \mathcal{A}$ is the set of all strings, and $\mathcal{N} \subseteq \mathcal{S}$ is the set of strings that may be used as tag names. The set $\mathcal{V}$ is partitioned into the sets of document nodes ($\mathcal{V}^d$), element nodes ($\mathcal{V}^e$), attribute nodes ($\mathcal{V}^a$), and text nodes ($\mathcal{V}^t$).

Definition 1 (XML Store). *An* XML store *is a 6-tuple* $St = (V, E, <, \nu, \sigma, \delta)$ *where:*

- V *is a finite subset of* $\mathcal{V}$*; we write* V^d *for* $V \cap \mathcal{V}^d$ *(resp.* V^e *for* $V \cap \mathcal{V}^e$*,* V^a *for* $V \cap \mathcal{V}^a$*,*V^t *for* $V \cap \mathcal{V}^t$*);*
- (V, E) *is an acyclic directed graph (with nodes* V *and directed edges* E*) where each node has an in-degree of at most one, and hence it is composed of trees; if* $(m, n) \in E$ *then we say that* n *is a child of* m*; we denote by* E^* *the reflexive transitive closure of* E*;*
- $<$ *is a strict partial order on* V *that compares exactly the different children of a common node. Hence for two distinct nodes* n_1 *and* n_2 *it holds that* $((n_1 < n_2) \vee (n_2 < n_1)) \Leftrightarrow \exists m \in V((m, n_1) \in E \wedge (m, n_2) \in E))$
- $\nu : V^e \cup V^a \rightarrow \mathcal{N}$ *labels the element and attribute nodes with their* node name
- $\sigma : V^a \cup V^t \rightarrow \mathcal{S}$ *labels the attribute and text nodes with their* string value
- $\delta : \mathcal{S} \rightarrow \mathcal{V}^d$ *a partial function that associates a document node with an URI or a file name. It is called the* document function*. This function represents all the URIs of the Web and all the names of the files, together with the documents they contain. We suppose that all these documents are in the store.*

The following properties must hold for an XML store: each document node of V^d is the root of a tree and has only one child element; attribute nodes of V^a and text nodes of V^t do not have any children; in the $<$-order attribute children precede the element and text children, i.e. if $n_1 < n_2$ and $n \in V^a$ then $n_1 \in V^a$; there are no adjacent text children, i.e. if $n_1, n_2 \in V^t$ and $n_1 < n_2$ then there is an $n_3 \in V^e$ with $n_1 < n_3 < n_2$; for all text nodes n_t of V^t holds $\sigma(n_t) \neq$ "" ; all the attribute children of a common node have a different name, i.e. if $(m, n_1), (m, n_2) \in E$ and $n_1, n_2 \in V^a$ then $\nu(n_1) \neq \nu(n)_2$.

Given an XML Store St we will use following auxiliary notations and functions:

- V_{St},E_{St},ν_{St},σ_{St} and δ_{St} return the corresponding components of St. We also define V^d_{St}, V^e_{St} and V^a_{St}.
- $genDocNode()$, $genElement()$, $genAttribute()$ return members from $\mathcal{V}^d$, $\mathcal{V}^e$ and $\mathcal{V}^a$ which do not exist in V^d_{St}, V^e_{St} and V^a_{St} respectively. These functions are abstractions of the creation of new element document and attribute nodes. Note that text nodes will be generated by appropriately specific functions.
- $root(St)$ is the root node of the store.
- $descendants_{St}(n)$ is the set of all nodes in St which are descendants of n.

For simplicity, and without loss of generality, in the remainder of this paper we will ignore the partial order $<$, and we will not indicate the name of the XML Store when it is obvious from context.

2.1 Creating XML documents from scratch

We will model XML generators as functions that create XML Stores. Like in other schema-based systems, the generation will be based on the specification of a *base model*, whose expressivity must be the same of XML Schema languages ([13]).

Definition 2 (base model, generator, interpretation). *An* XTaGe base model*, M, is a tree that represents a* generating functional expression *(or* generator*, for short) f whose* interpretation $Gen_M(f)$ *is an XML store.*

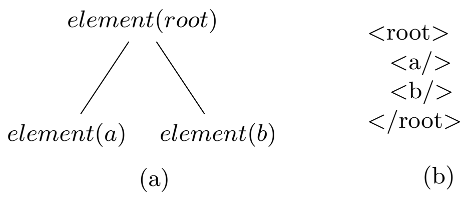
A base model is, therefore, conceptually similar to the expression trees that appear when parsing programming languages. For example, consider the following operation:

Example 1. Given a generator model tree M, The $element(name)$ component is a generating functional expression whose interpretation $Gen_M(element(name))$ generates a XML Store $(V, E, <, \nu, \sigma, \delta)$ such that:

- $newNode = genElement()$*, that is, a new node*
- $V = \{newNode\} \cup V_{Gen(c_i)}$ for each $c_i \in children_T(c)$
- $E = \{(newNode, root(Gen(c_i))\} \cup (\bigcup E_{Gen(c_i)})$ for each $c_i \in children_T(c)$
- $\nu = \{(newNode, name)\} \cup (\bigcup\{(a, b) : (a, b) \in \nu_{Gen(c_i)}\})$ for each $c_i \in children_T(c)$
- $\delta = \bigcup\{(a, b) : (a, b) \in \delta_{Gen(c_i)}\}$ for each $c_i \in children_T(c)$
- $\sigma = \bigcup\{(a, b) : (a, b) \in \sigma_{Gen(c_i)}\}$ for each $c_i \in children_T(c)$

where $children_M(c)$ represents the children of node c in the generation model tree M. We treat the *attribute* nodes similarly.[1]

Figure 1 represents a simple generation tree that uses the *element* generator.



```
<root>
  <a/>
  <b/>
</root>
```

Fig. 1. A simple tree generation using the *element* generator component and the corresponding XML tree.

[1] The main difference with the treatment of *element* nodes is that the *attribute* nodes cannot be nested.

In order to provide a functionality similar to basic XML Schema, we introduce the following generators that account for the possible content models:

- $sequence(name, attr, n, minOccurs, maxOccurs)$: A functional component that is a generalization of the previously introduced *element*, including support for attributes ($attr$) a number of repetitions (n) and cardinality constraints ($minOccurs$ and $maxOccurs$).
- $choice(name, attr, n, minOccurs, maxOccurs)$: A functional equivalent of the XML Schema *choice* content model, represented by a bar (|) in DTDs.

The features of the generation model presented so far support the creation of XML documents based on a fixed XML-like schema. We now introduce two features which are specifically designed to introduce controlled variability in collections: *distributions* and *probability-labeled arcs*.

Value distributions. In XTaGe, *every* value is extracted from a probability distribution, including constant numbers and strings (which are considered to be extracted from a suitably defined constant distribution). This includes:

- Parameters of functional components, for example the number of repetitions n in sequences or choices.
- Synthetic content in attributes or text nodes.

This allows us to easily express empirical properties of the generated data, such as that the number of children of a given node is normally distributed, or Zipf-like distribution for words in textual content.

Probability-labeled tree arcs. Another mechanism to introduce controlled variability in generated XML is introducing the notion of probabilistic labeling. Each arc (u, v) in a generator tree M is labeled with a probability value $p((u, v)) \in [0, 1]$. The meaning of a probability-labeled arc is that the child functional expression v will be ignored with probability $1 - p((u, v))$.

Example 2. Figure 9 shows a simple generator tree that includes probability-labeled arcs, and some of the XML trees that could be generated by it.

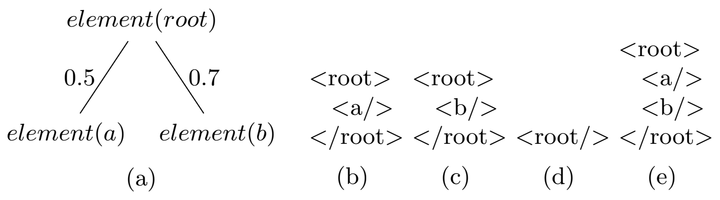


Fig. 2. A generator tree with probabilistic arcs and some possible generated trees.

2.2 Transforming XML documents

The second mechanism by which a new XML collection may be generated is by the controlled transformation of an existing one. Analogously to tree generation model trees, we introduce a *transforming functional expressions* (or *transformations*, in short). Moreover, these transformations provide the foundations required to model global contraints over XML documents. Since there are many transformation languages available for XML (XSLT, XDuce), we will focus in the introduction of controlled variability into XML collections through the definition of transforms.

Definition 3 (locator). *A* locator *is a function that takes an XML Store St and returns an XML Store St' such that:*

- $V'_{St} \subseteq V_{St}$
- $E'_{St} \subseteq E_{St}$
- St' *is well formed according to Definition 1*

Example 3. Given a locator and an XML Store St, the *delChild* transformation removes a random child of the root node of a tree with probability p, and returns a new XML Store St'. We can define it as follows:

- Choose a node $n \in children(root(l))$, where l is the subtree induced by the locator, with probability p.
- $V_{St'} = V_l \backslash (\{n\} \cup descendants_l(n))$
- $E_{St'} = E_l \backslash \{(u, v)$ such that $u, v \in \{n\} \cup descendants(n)\}$
- $\nu_{St'}$, $\delta_{St'}$ and $\sigma_{St'}$ are suitably modified.

Probability-labeled tree arcs are also used in transformation trees to determine if a transformation will be applied or not in the XML documents. It is important also to remark that transformations are applied on the original XML Store, and not on a "different" XML store induced by the locator, which must then be grafted on the original tree. This property allows us to define meta-transformation operations, which can be used to combine different transformations into complex operations. The main meta-transformation is the macro-transformation:

Definition 4 (macro-operation). *Given an XML Store St and list of (locator, transformation) pairs, the macro-operation* $macro_{St}[(l_1, t_1), \ldots, (l_n, t_n)]$ *is defined as the sequential application of all pairs to St.*

3 Component-based framework

The concepts outlined in the previous section have been realized in XTaGe by means of a lightweight component-oriented software model, outlined in Figure 3. The use of components as an abstraction of the generation and transformation functional expressions has a number of benefits directly related to the goals of XTaGe.

First of all, it makes metadata about components and their relationships explicit. This facilitates greatly the construction of support tools such as GUIs, and it also allows the simplified creation of new components without requiring the modification of the XTaGe code. This can be accomplished by a simple two-stage process: (*i*) create the function in a high-level language (XQuery in our case, as we will explain presently) and (*ii*) register it by filling in the appropriate metadata. For the most complex cases (usually involving calling external libraries), new components can be coded by implementing the appropriate interface. Libraries of related components (e.g. for testing of biomedical data sets) may be put together and maintained independently.

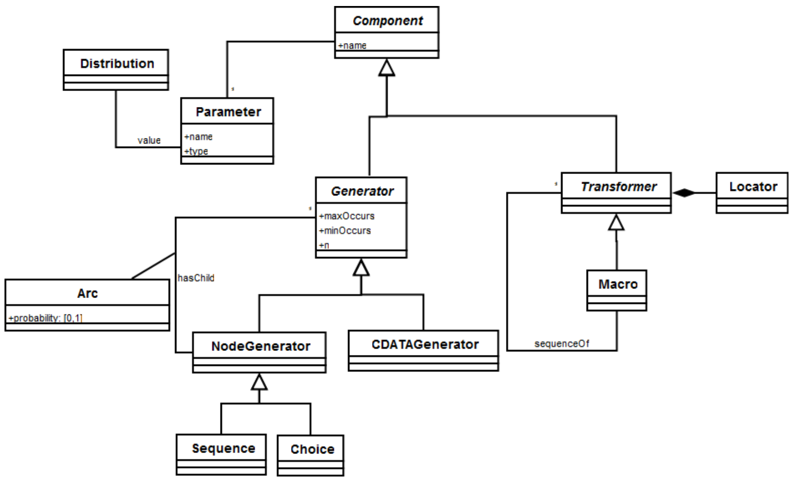


Fig. 3. Simplified UML diagram of the XTaGe component architecture.

3.1 Generators in XTaGe

In addition to the basic XML Schema-related components described in Section 2, a number of components useful for the generation of collections with controlled variability are pre-defined:

XOR This generator chooses one node between all its descendants according to their *xor probability* values. The descendants of a XOR constructor have an additional probability parameter, *xor probability*, which determines the likelihood of a node of being chosen by its parent.

Combi This generator creates a new node whose tag is a combination of the tags of its descendants.

DminDmax The functionality of this generator is creating a new node located n levels below its ancestor. The value of n depends on the values of the attributes *dmin* and *dmax* of the constructor. The value of n is a random number between *dmin* and *dmax*.

IfAncestor This generator determines if a node appears in the new document depending on the tag of its ancestor.

An XML node generated by one of these components is assigned a unique id value, which can be specified by the user, automatically by the system, or by a user-defined function. Attributes (and IDREFs, which are treated as a special case) can be defined by the user in the generator model tree.

As mentioned above, the preferred way to create new components is by means of the creation of an XQuery functions. XQuery was chosen because it is inherently well-suite to define operations on XML trees. The function must conform to the following signature:

```
declare function component−name($comp as node()) as node()∗;
```

where $comp represents the component metainformation (serialized as XML), including its parameters.

To support the creation of components, a library of XQuery functions has been defined that permits access to the defined distributions and the structure of the model tree.

Example 4. The following XQuery function implements the component *DminDmax*, using an auxiliary function to handle recursion:

```
declare function dmindmax($comp as node())) as node()∗ {
   let $dist := $comp/param/dist
   return dmindmax_aux($comp, $dist) }

declare function dmindmax_aux($comp as node()∗, $dist)) as node()∗ {
   if ($dist = 1)
   then xtg:create_node($comp)
   else(          let $actual_dist := xs:integer($dist) − 1
                  let $random_tag := xtg::randomTag()
                  return element{$random_tag} {dmindmax_aux($comp, $actual_dist)} )
}
```

where library function `xtg:create_node($comp)` creates an element based on the parameters of the component and `xtg:randomTag()` returns random strings.

Note the use of XPath to extract the value of the parameters from the component metadata object. Besides the `$comp` parameter, the component accepts `$dist`, that is a number obtained from a user-defined distribution that must lie between the values of the parameters *dmin* and *dmax* of the component.

3.2 Transformations in XTaGe

In order to apply controlled transformations, XTaGe includes a few pre-defined XML transformation components:

Add This transformation component takes as input two XML trees, t_1 and t_2. The component adds t_2 to t_1 as a descendant of the node or nodes of t_1 determined by the component *locator*.

Delete This component removes the node determined by the component *locator* of the XML document tree. The component has a parameter called *recursive*, whose value determines if the operation is executed recursively or not. If its value is 1, the node and all its descendants are removed; if its value is 0, only the node is removed and its descendants occupy its place. In case the nodes have references, XTaGe allows the user to specify whether the references must be automatically re-calculated.

Change Order This component changes the order of a node and one of its siblings by changing their positions. The user has to determine the ancestor of the node that is going to be changed and, optionally, the node that is going to change its position. If the user does not specify this node, the component chooses a descendant of the *ancestor node* randomly. The user can also determine the new position of the node; in case the user does not specify it, the component determines randomly the new position.

Change Level This component changes the position of a node and one of its descendants, chosen by the user or randomly. The descendant will be now the ancestor of its siblings and its ancestor will become one of its descendants.

XTaGe also allows the definition of new transformations using XQuery. The functions must conform to the following signature:

```
declare function component−name($context as node(), $locator as node()∗,
                                        $comp as node()) as node()∗;
```

where `$context` represents the current context node in the source XML document, that defines where the transformation will be applied; `$locator` is the set of nodes induced by the locator; and `$comp` contains metainformation about the component.

Example 5. The XQuery function that implements the functionality of the component *Change_order* may be implemented as follows:

```
declare function change_order($context, $loc, $config) as node()∗ {
   let $newpos:= xtg:newPos($config)
   let $child:= xtg:child($context, $config)
   let $sibling:= xtg:sibling($context, $config)
   return element{fn:local−name($context)}
              {for $att in $context/@∗
                 return attribute{fn:local−name($att)}{$att},
               for $c at $pos in $context/∗
                 return if ($pos = $newpos)
                        then xtg:traverse($child, $context, $config)
                        else ( if($c is $child)
                               then xtg:traverse( $sibling, $context, $config)
                               else xtg:traverse($c,$context, $config))}
}
```

where:

`newPos($config)` returns the new position of the node, if this value is not set in the parameter *NewPos* of the component Change_order, the function returns a random value in the range of [1, number of descendants of the *context* node].

`child($context, $config)` returns the element that is going to be changed. This node is retrieved by the *locator* component.

`sibling($context, $config)` returns the element that now occupies the position *newpos*.

`traverse($context, $loc, $config)` is a function that traverses the XML document, element by element.

This set of transformations does not allow the user to model global constraints but it is possible to create and add new components that support non-local contraints models.

4 Prototype and Use Cases

The next section presents a use case in the Bioinformatics domain in which XTaGe is applied. It addresses the problem of evaluating techniques based on the modification of specific characteristics of a XML collection.

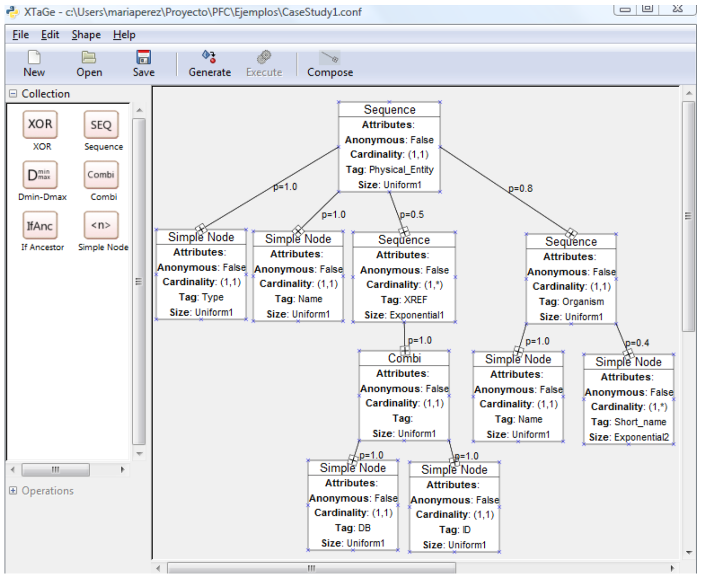
4.1 Generating controlled testing collections

We consider a problem of exploratory search of XML collections in the Bioinformatics domain. This domain is characterized by the existence of a great number of complex, large and heterogeneous XML data sources, which poses serious issues in data integration applications [11]. Usually, a first step in the design of these applications is the characterization of a sample of these collections, which requires the use of approximate querying techniques due to the lack of a schema and the presence of complex, domain-specific data such as protein sequences, which inherently require approximate matching algorithms.

Testing of these systems is difficult, since (i) they may not correspond well with the expected work load (ii) they might not exercise all possible structural variations that might appear in the production system or (iii) they may contain errors which need to be corrected [12].

In such a context, the features of XTaGe for the introduction of controlled variability in collections may facilitate the design process greatly. Consider the case of an application trying to integrate information coming from the complex BioPAX collection [1], which is derived from OWL specifications and exhibits an essentially free-form schema when translated into XML.

To account for the possible variations, XTaGe can be used to automatically generate test cases that can be used to check for unexpected structural variations. For example, Figure 4 shows an example of a user-defined schema that



Sequence
Attributes:
Anonymous: False
Cardinality: (1,1)
Tag: Physical_Entity
Size: Uniform1

Simple Node
Attributes:
Anonymous: False
Cardinality: (1,1)
Tag: Type
Size: Uniform1

Simple Node
Attributes:
Anonymous: False
Cardinality: (1,1)
Tag: Name
Size: Uniform1

Sequence
Attributes:
Anonymous: False
Cardinality: (1,*)
Tag: XREF
Size: Exponential1

Sequence
Attributes:
Anonymous: False
Cardinality: (1,1)
Tag: Organism
Size: Uniform1

Combi
Attributes:
Anonymous: False
Cardinality: (1,1)
Tag:
Size: Uniform1

Simple Node
Attributes:
Anonymous: False
Cardinality: (1,1)
Tag: Name
Size: Uniform1

Simple Node
Attributes:
Anonymous: False
Cardinality: (1,*)
Tag: Short_name
Size: Exponential2

Simple Node
Attributes:
Anonymous: False
Cardinality: (1,1)
Tag: DB
Size: Uniform1

Simple Node
Attributes:
Anonymous: False
Cardinality: (1,1)
Tag: ID
Size: Uniform1

Fig. 4. An example of a user-defined schema. Note that the GUI shows the list of available components (left pane) which can be dragged and dropped to create the schema.

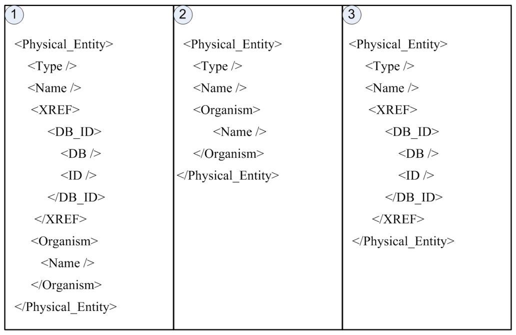
generates XML structures with BioPAX-derived information. Figure 5 shows different XML structures generated by this schema. Note how the structure of the documents varies due to the probabilities and the patterns of the components.

This is also useful to help assess the performance of the approximate techniques being used for data exploration, in particular in the presence of characteristics of interest. This calculation requires the generation of different versions of a same collection, each one exhibiting a different characteristic. To achieve this goal, XTaGe can be used to generate such a set of XML collections.

Figure 6 shows the steps to generate the new versions of the XML collection. The approach is based in a multi-step process. First, a "background collection" is determined; this can be synthetic, or a sample of existing. Next, to facilitate comparisons, a XML structure suitable for transformation is determined. A number of transformations are written, in order to exercise the different structural characteristics that should be tested (e.g. presence/absence of nodes or subtrees, changes in ordering, and so on). Finally, new versions of the background collections are created and subjected to testing.

Next we explain with further details the two main steps required to obtain the different versions of the background collection and we clarify them with an example.

Creating the new XML structure. In this first phase the user has to define the schema of the XML fragment that, in the following phase, is going to be modified



```
1
<Physical_Entity>
   <Type />
   <Name />
    <XREF>
        <DB_ID>
            <DB />
            <ID />
        </DB_ID>
     </XREF>
    <Organism>
      <Name />
    </Organism>
</Physical_Entity>
```

```
2
<Physical_Entity>
   <Type />
   <Name />
   <Organism>
        <Name />
   </Organism>
</Physical_Entity>
```

```
3
<Physical_Entity>
    <Type />
    <Name />
     <XREF>
         <DB_ID>
             <DB />
             <ID />
         </DB_ID>
      </XREF>
</Physical_Entity>
```

Fig. 5. Structures generated by the schema shown in Figure 4.

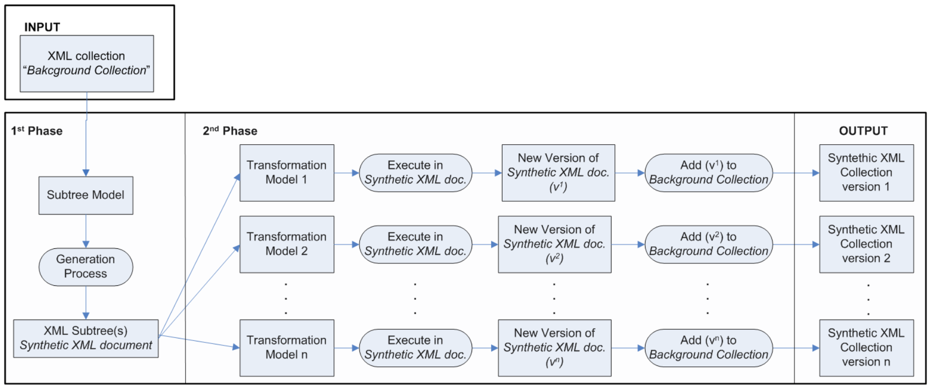

INPUT
XML collection "Bakcground Collection"
1st Phase
Subtree Model
Generation Process
XML Subtree(s) Synthetic XML document
2nd Phase
Transformation Model 1
Execute in Synthetic XML doc.
New Version of Synthetic XML doc. (v1)
Add (v1) to Background Collection
Transformation Model 2
Execute in Synthetic XML doc.
New Version of Synthetic XML doc. (v2)
Add (v2) to Background Collection
Transformation Model n
Execute in Synthetic XML doc.
New Version of Synthetic XML doc. (vn)
Add (vn) to Background Collection
OUTPUT
Synthetic XML Collection version 1
Synthetic XML Collection version 2
Synthetic XML Collection version n

Fig. 6. Steps to generate new versions of an existing XML collection.

and finally, added to the background collection. The result of executing this schema is an XML document, which we call *synthetic XML document*, whose structure and tags are well-known by the user.

In our example, we have designed a XML document that contains information about publications, its schema is shown in Figure 7. Figure 8b shows the synthetic XML document generated by it.

Creating synthetic collections. In this phase, the user has to define the schemas of the transformations that are going to be executed in the synthetic XML document in order to exercise the different structural characteristics that should be tested. The goal is to create a set of n synthetic collections where each one differs from the others in a known characteristic. The steps to do that are the following:

1. Create n transformation schemas, one per each characteristic to be analyzed. Each schema is composed by a set of transformations whose execution modifies a specific characteristic of the synthetic XML document.

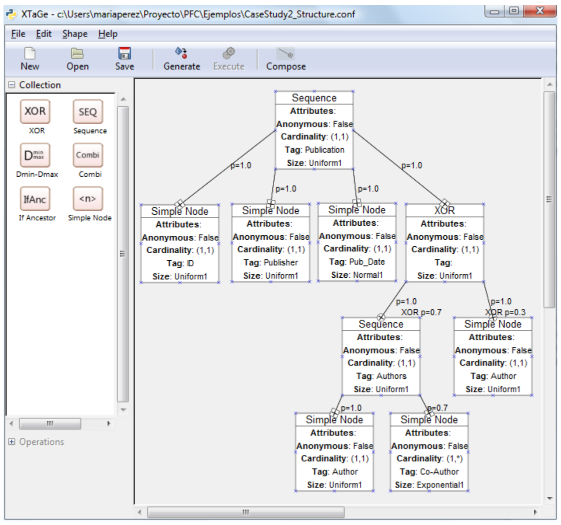

XTaGe - c:\Users\mariaperez\Proyecto\PFC\Ejemplos\CaseStudy2_Structure.conf
File Edit Shape Help
New Open Save Generate Execute Compose
Collection
XOR XOR
SEQ Sequence
Dmin-Dmax
Combi Combi
IfAnc If Ancestor
<n> Simple Node
Operations
Sequence Attributes: Anonymous: False Cardinality: (1,1) Tag: Publication Size: Uniform1
p=1.0
Simple Node Attributes: Anonymous: False Cardinality: (1,1) Tag: ID Size: Uniform1
Simple Node Attributes: Anonymous: False Cardinality: (1,1) Tag: Publisher Size: Uniform1
Simple Node Attributes: Anonymous: False Cardinality: (1,1) Tag: Pub_Date Size: Normal1
XOR Attributes: Anonymous: False Cardinality: (1,1) Tag: Size: Uniform1
p=1.0 XOR p=0.7
p=1.0 XOR p=0.3
Sequence Attributes: Anonymous: False Cardinality: (1,1) Tag: Authors Size: Uniform1
Simple Node Attributes: Anonymous: False Cardinality: (1,1) Tag: Author Size: Uniform1
p=1.0
p=0.7
Simple Node Attributes: Anonymous: False Cardinality: (1,1) Tag: Author Size: Uniform1
Simple Node Attributes: Anonymous: False Cardinality: (1,*) Tag: Co-Author Size: Exponential1

Fig. 7. A schema that generates XML documents with information about publications.

2. Execute each transformation schema on the synthetic XML document. The result is a set of n modified versions of this synthetic XML document.
3. Add each modified version of the synthetic XML document to the background collection. The result of this last step is a set of n versions of the background collection ready to testing experiments.

Figure 8a shows the transformation schema we have created to modify the synthetic XML document. Then, Figure 8b shows the result of executing this transformation schema on the synthetic XML document generated by the generator schema shown in Figure 7. The execution of this schema consists on:

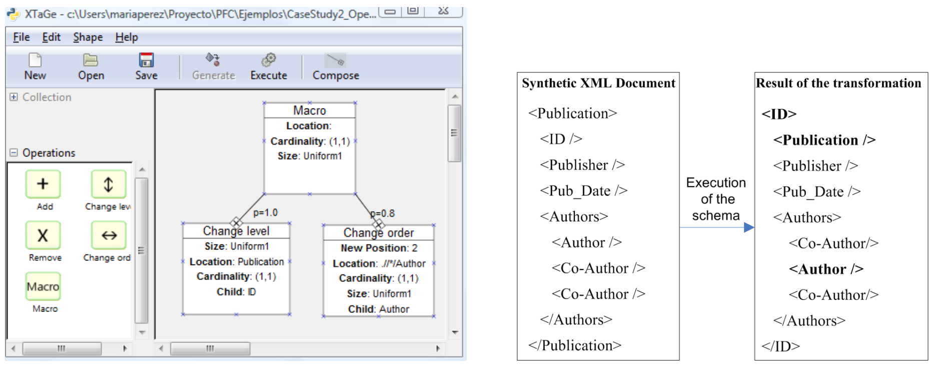
1. Change Level: The execution of this component will change the positions of the elements "Publication" and "ID", being now the element "ID" the ancestor and "Publication" the descendant.
2. Change order: The element "Author" will occupy the second position in its siblings set wherever it appears, due to the ancestor's path specified in the *location* parameter, "$.//*/[Author]$".

Later, this modified XML structure will be added to the background collection by using an *Add* transformation component.

5 Discussion and Conclusions

We have presented how the XTaGe XML generator can be used to overcome the limitations of existing system when dealing with complex collections.

The first issue that XTaGe addresses is the lack of adaptability of the current generators to new domains or new use cases. Most of them cannot be adapted



Macro
Location:
Cardinality: (1,1)
Size: Uniform1

Change level
Size: Uniform1
Location: Publication
Cardinality: (1,1)
Child: ID

Change order
New Position: 2
Location: .//*/Author
Cardinality: (1,1)
Size: Uniform1
Child: Author

Synthetic XML Document

```
<Publication>
  <ID />
  <Publisher />
  <Pub_Date />
  <Authors>
    <Author />
    <Co-Author />
    <Co-Author />
  </Authors>
</Publication>
```

Result of the transformation

```
<ID>
  <Publication />
  <Publisher />
  <Pub_Date />
  <Authors>
    <Co-Author/>
    <Author />
    <Co-Author/>
  </Authors>
</ID>
```

Fig. 8. (a) A transformation schema and (b) a resulting document.

to new domains because they have been designed for specific purposes in a specific domain. XTaGe provides a flexible component-based framework that makes possible to adapt it to new specifications. The user can add new components that implement new functionalities in order to fulfill the new requirements.

In addition, XTaGe also supports creating different versions of an existing XML collection by applying a set of user-defined transformations as [7] does. However, [7] supports only a limited set of 4 functions that have been implemented for their specific purpose, the evaluation of semantic query optimization techniques, and they don't mention any way to expand this set with new functions. XTaGe also provides a set of basic transformation components that encapsulate typical XML tree-based transformations but, in contrast to [7], XTaGe allows the user to expand this set with other components according to the new requirements.

Future directions for research include extending and generalizing the features of XTaGe. We are currently focusing on the automatic generation of preliminary generation models by examining existing samples of collections. Another important issue is the lack of support for non-local constraint specification (as in [6]), although XTaGe's architecture sets the foundations to model these constraints, specific components have to be implemented and added. Another relevant direction is the extension of the component model to be able to better organize collections of pre-defined components and managing their dependencies. Finally, we aim to design specific generators, such as OWL instances generator and GIS data generator, based on XTaGe. The implementation of these specific generators will be based on an MDA architecture, where the specific generator models will be transformed into the XTaGe's model.

In conclusion, XTaGe builds upon the main features of existing schema-aware generators, and extends them in order to provide support for complex collections. The resulting system has excellent support for the creation of *controlled variability*, which is useful in testing complex and highly specific features of XML collections in particular domains. The component-based architecture is the basis for a GUI, which facilitates the specification of new collections.

Acknowledgements

This work has been partially supported by the Ministry of Science and Innovation (TIN2008-01825).

References

1. Biopax. http://www.biopax.org/.
2. Ashraf Aboulnaga, Jeffrey F. Naughton, and Chun Zhang. Generating Synthetic Complex-structured XML Data. In *WebDB'2001*, 2001.
3. L. Afanasiev, I. Manolescu, and P. Michiels. MemBeR XML Generator. http://ilps.science.uva.nl/Resources/MemBeR/member-generator.html.
4. Denilson Barbosa and Alberto O. Mendelzon. Declarative generation of synthetic XML data. *Software: Practice and Experience*, 36:1051–1079, May 2006.
5. Stéphane Bressan, Mong Li Lee, Ying Guang Li, Zoé Lacroix, and Ullas Nambiar. The XOO7 Benchmark. In *Efficiency and Effectiveness of XML Tools, and Techniques (EEXTT2002)*, pages 146–147, London, UK, 2002. Springer-Verlag.
6. Sara Cohen. Generating XML Structure Using Examples and Constraints. In *VLDB*, 2008.
7. Ke Geng and Gillian Dobbie. An XML Document Generator for Semantic Query Optimization Experimentation. In *iiWAS 2006*, pages 367–376, 2006.
8. Jan Hidders, Stefania Marrara, Jan Paredaens, and Roel Vercammen. On the expressibility of functions in XQuery fragments. *Information Systems*, 33:435–455, 2008.
9. Jan Hidders, Philippe Michiels, Jan Paredaens, and Roel Vercammen. LiXQuery: A formal foundation for XQuery research. *SIGMOD Record*, 34(4):21–26, 2005.
10. Hoe Jing Jeong and Sang Ho Lee. A Versatile XML Data Generator. *International Journal of Software Effectiveness and Efficiency*, 1:21–24, 2006.
11. Marco Mesiti, Ernesto Jiménez-Ruiz, Ismael Sanz, Rafael Berlanga, Giorgio Valentini, Paolo Perlasca, and David Manset. Data integration issues and opportunities in biological XML data management. In *Open and Novel Issues in XML Database Applications: Future Directions and Advanced Technologies*. IGI Global, 2009.
12. I. Mlynkova, K. Toman, and J. Pokorny. Statistical Analysis of Real XML Data Collections. In *COMAD'06*, 2006.
13. Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Techn.*, 5(4):660–704, 2005.
14. Matthias Nicola, Irina Kogan, and Berni Schiefer. An XML Transaction Processing Benchmark. In *SIGMOD'07*, 2007.
15. Sven Puhlmann, Felix Naumann, and Melanie Weis. The Dirty XML Generator.
16. Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolesc, and Ralph Busse. XMark: A Benchmark for XML Data Management. In *VLDB*, pages 974–985, 2002.
17. Benjamin Bin Yao, M. Tamer zsu, and John Keenleyside. XBench - A Family of Benchmarks for XML DBMSs. 2003.