Declarative generation of synthetic XML data

Denilson Barbosa¹, Alberto O. Mendelzon²

¹Department of Computer Science, University of Calgary
²Department of Computer Science, University of Toronto

SUMMARY

Synthetic data can be extremely useful in testing and evaluating algorithms, tools and systems. Most synthetic data generators available today are the result of individual benchmarking efforts. Typically, these are complex programs in which the specifications of both the structure and the contents of the data are hard-coded. As a result, it is often difficult to customize these tools for producing synthetic data tailored for specific needs. In this article, we describe the ToXgene synthetic data generator which is a declarative tool for generating realistic XML data for benchmarking as well as testing purposes. We present our template specification language, which consists of augmenting XML Schema with probabilistic models that guide the data generation process; we discuss the architecture of our current implementation; and we argue about ToXgene’s usefulness by discussing experimental results as well as describing two projects that use our tool.

KEY WORDS: XML; synthetic data; benchmarking; probabilistic generative models.

1. Introduction

Synthetic data workloads can be extremely useful in testing and evaluating algorithms, tools and systems, and have been used for benchmarking in various communities and application domains [18, 30]. Good reasons for using synthetic data abound. First, the data generator can be designed to produce “realistic data” (i.e., data that are similar to a typical real dataset in a given application domain). Second, the data generator can be used for obtaining workloads of varying sizes, thus allowing scalability tests. Third, the synthetic data can be customized for evaluating specific aspects of the system, either by using different parameters or by modifying the data generator accordingly. Fourth, synthetic data can be publicly disclosed,
which facilitates the publication and comparison of performance results of competing systems and is a requirement for public benchmarks. Finally, and perhaps most importantly, it is difficult to obtain real data that possess all the above properties.

Most synthetic data generators available at the time of writing are byproducts of benchmarking efforts; that is, special-purpose software tools in which the characteristics of the data (i.e., a schema and rules for populating that schema) are hard-coded. These data generators accept few parameters, allowing little customization in their output. Typically, these parameters are a random seed for allowing the generation of identical datasets and a scaling factor determining how much data to generate. Thus, while these tools are adequate for benchmarking, where the workloads must adhere to strict rules, they are generally not as useful for generating testing data. In a typical testing scenario, one might want to try the system with datasets of varying degrees of “complexity” (i.e., varying numbers of attributes in the schema), or one might want to compare the performance of a given algorithm with skewed data versus uniformly distributed data (i.e., varying rules for populating the schema). Therefore, system developers are often faced with the choice of either undertaking the complex task of developing or modifying an existing data generator to obtain appropriate test data, or settling for less-than-ideal data produced by a data generator already available.

While obtaining suitable test data is a problem common to all domains, it is particularly hard for those applications dealing with XML (Extensible Markup Language) data, for two main reasons. First, XML is a meta-language flexible enough for encoding of virtually any kind of data, ranging from highly structured data (e.g., relational data) to semistructured data (e.g., structured documents) to fairly unstructured data (e.g., results of scientific experiments). Second, XML is really a means to an end, and has been used in a wide range of applications (e.g., encoding payloads in Web services [19] calls or encoding documents in standard office applications [31]). It is thus unlikely that there will ever be benchmarks covering all aspects of XML usage.

This article discusses a flexible, extensible, and declarative generator for synthetic XML data called ToXgene, which was developed in the context of the ToX (Toronto XML Server)
DECLARATIVE GENERATION OF SYNTHETIC XML DATA

The data generation in ToXgene is guided by templates that specify a schema for the desired data as well as rules for populating such schema; such templates are XML Schema specifications augmented with specific annotations for guiding the data generation. For instance, annotations are used for defining probability distributions specifying the number of elements in a document or the values of numeric types, and for defining content sharing, which allows different elements (possibly in different documents) to have some content in common. Unlike the procedural approach for data generation taken by hard-coded tools, our tool provides the users with a declarative approach to data generation; that is, the user is able to specify what data is needed, instead of how to generate it. Also, by supporting a large subset of a standard schema language for XML, ToXgene eases the development and modification of templates. ToXgene is implemented in Java and can be used as command line tool or invoked by other applications through a simple API, which also provides a means for application-specific synthetic data generators to be used together with the predefined ones. Finally, our current implementation allows the use of secondary memory for storing temporary data required during data generation; in effect, this allows the generation of datasets that do not fit in the main memory of the host machine.

ToXgene is the result of a collaboration between the University of Toronto Database Group and IBM Canada, and is available free of charge for non-commercial use. ToXgene is also part of the IBM Alphaworks project, and has been widely used by the researchers and practitioners in industry since its first release in 2002. We are aware of over 50 universities and over 200 companies and research labs around the world where ToXgene has been used; also, as of February 2005, our tool has had over 5000 individual downloads.

Running example. We will illustrate the discussion in the article describing how to model a data generator for a bookstore application; the data consists of a book catalog, several book orders, and several book reviews written by customers. The catalog consists of several XML documents, each describing a single book (e.g., its title, ISBN, price, etc.); similarly, each review is stored as a separate XML document. Figures 1(a) and 1(b) show a typical book and a typical book review in this domain, respectively.

1.1. XML data

Despite being originally designed for large-scale content publishing, XML is a natural format for representing and exchanging semistructured data on the Web. Moreover XML is already widely used as a content publishing format: a recent study shows that XML can be found in all major Internet domains and in all geographic regions of the globe. While a proper introduction to XML is unnecessary here (we refer the reader to Bosak for a discussion of XML, its use and importance for the Web), a brief discussion of XML content is in order.

The building blocks of XML content are called elements, which consist of text enclosed by matching start tags, such as <book>, and end tags, such as </book>. The content of an element is everything enclosed by its tags, and may consist of text (called parsed character data), other elements, or a mix of these. For instance, the content of the element in Figure 1(a) is the text Donald Knuth, while the content of the element in the figure consists of three elements. An XML element may be adorned with one or more...
attributes, each consisting of a name and value that describes or qualifies the content of the element. For instance, the attribute currency="CDN" in the <price>... </price> element qualifies that the price in question is given in Canadian dollars. The XML standard defines two special kinds of attributes: ID attributes, which uniquely identify the elements that bear them, and IDREF(S) attributes, which are used for defining references among elements in a document. XML allows empty elements as well (i.e., elements with no content), which can be abbreviated by having a / at the end of the start tag. For instance, the <book ... /> element in Figure 1(b) is an empty element with a single attribute.

In XML, the content of an element is considered to be well-formed if all start and end tags contained in it are balanced; that is, they are closed in inverse order to that in which they are opened. Well-formedness naturally imposes a tree structure to XML content, as it forbids an element to be part of the content of two or more other elements. Thus, if p and c are elements such that p contains c and there is no element x that contains c but not p, we say that p is the parent of c, and c is a child of p. The notions of descendants and siblings are defined as usual. For instance, in Figure 1(a), the element <price>... </price> is a sibling of <author>... </author> while both are children of <book>... </book>. Being a document encoding language, XML imposes an ordering to elements and text nodes inside a document; for instance, in Figure 1(a), the <title> element comes before the <author> element. Thus, well-formed XML content can be naturally modeled as ordered labeled trees in which: (1) leaf nodes represent attributes, textual content, or empty elements, while internal nodes represent elements; and (2) edges represent the parent-child relationship between elements as well as the containment relationship among elements and attributes or text nodes.

Each XML document contains a single element, called its root element, and may contain a reference to a schema. There are two standard schema languages for XML: DTDs (Document Type Definitions) [12], and XML Schema [32]. A document is said to be valid (resp. schema-valid) if its root element is well-formed and satisfies the constraints in a DTD (resp. XML Schema specification). While differing in many aspects, both formalisms allow the specification of constraints on the ways elements can be nested (i.e., governing the structure of the XML documents), as well as on the valid contents of elements and attributes. In particular, XML Schema allows the specification of common database constraints such as keys and foreign keys [1]; moreover, XML Schema provides a rich collection of types compared to DTDs, as well as type specialization. For these reasons, we chose XML Schema as the basis for ToXgene’s template specification language [5]. Lee and Chu [23] compare several schema languages for XML.

Outline. The remainder of the article is organized as follows. Section 2 provides a brief discussion of using probabilistic models for generating useful synthetic data. Section 3 discusses ToXgene’s template specification language through examples that illustrate the main features of our tool as well as the extensions to XML Schema that we introduced. Section 4 discusses our current implementation, focusing on generating large datasets and ToXgene’s API. Section 5 discusses an experimental validation of ToXgene as well as two successful projects based on our tool, thus arguing about its usefulness. Finally, Section 6 discusses related work and Section 7 concludes the article.
2. Synthetic data generation with probabilistic generative models

In this section we discuss ToXgene’s data generation framework, which is based on augmenting XML schemas with probabilistic generative models. The main purpose of developing ToXgene was being able to generate realistic synthetic XML datasets for a given application domain in a declarative way. By realistic datasets here we mean collections of valid XML documents (with respect to a given set of schemas) with characteristics that are similar to those of most real datasets in a given application domain. It should be noted that a rich schema for the data alone is not sufficient for generating realistic synthetic data. This is so because, in most real applications, the actual data either follow skewed probability distributions or are determined by application-specific rules that cannot be easily captured by specifying datatypes only. For instance, in our bookstore example, both individual book prices and total order prices could be described in a schema \( S \) as having of the same type (e.g., decimal numbers in the range \([1,1000]\)); however, it would be reasonable to expect that book prices follow a skewed distribution (e.g., an exponential distribution with mean 35), while the total price of an order be equal to the sum of the individual prices of the books in that order. In other words, not all valid instances of \( S \) are equally likely to occur in practice.

In order to generate a realistic XML datasets, each ToXgene template specifies both the schemas for the documents as well as rules for the generation of synthetic XML content similar to that of typical real documents. More precisely, each piece of XML content generated by ToXgene is described by a tree-structured gene, which consists of an XML Schema type and can be annotated with probabilistic models describing various aspects of typical instances of that type. Two kinds of genes are defined: content genes describe elements, attributes and textual content, while control genes are used for various purposes, among which allowing content sharing and specifying complex data dependencies among elements. We say that a piece of XML content that conforms to the type specification in a gene is an instance of that gene; conversely, we refer to instantiating a gene the process of generating a piece of synthetic XML content from the gene’s description. As any other XML Schema types, genes are composed of other genes, forming a hierarchical structure that describes valid and realistic XML content. Thus, at a high level, a ToXgene template consists of one or more genes describing typical XML documents (recall each document consists of a single root element).

In the remainder of this section we briefly discuss content and control genes though examples. We defer a detailed discussion using ToXgene’s notation until the next section.

2.1. Content genes

Content genes are used for describing elements, attributes and textual content. We discuss the different kinds of annotations are used for each kind of content gene through examples next. The simplest content genes are those for describing textual content. A content gene describing book prices (textual content) consists of a numeric domain (e.g., real numbers in \([10,1000]\)) annotated with a probability distribution function (PDF for short) over that domain; instantiating such a gene consists of sampling from the associated PDF. A content gene for natural language text, such as paragraphs or author names, consist of probabilistic context-free grammars [27] (PCFG’s for short) over a given vocabulary.
Figure 2. A gene describing book documents. Element genes are represented by hollow circles, attribute genes are represented by filled circles, and text genes are represented by round-edge square nodes.

In a sense, elements and attributes are very similar: they have a label and some content; the main difference between them is that an attribute’s “content” (i.e., its value) must be text, while an element’s content can be any well-formed XML content. An attribute gene consists of a label and a content gene that describes the values of those attributes. An element gene, on the other hand, consists of an element label and a sequence of genes, which may include other element genes as well as attribute and text genes, specifying the content of those elements. Since similar elements may appear together inside a document, we associate with each element gene a probability distribution over an interval \([x, y]\) where \(x\) and \(y\) are the minimum and maximum number of occurrences allowed for the element, as defined by the schema underlying the data generation process. The value of an attribute, while being a single string of text, may be a list of atomic values separated by blank spaces. Thus, we also associate with each attribute gene a probability distribution that determines how many individual items appear in an actual instance of that gene.

Figure 2 shows a graphical representation of a gene describing book elements such as the one in Figure 1(a). In the figure, each element (resp. attribute) gene is annotated with a pair \((l, p)\), where \(l\) is the element (resp. attribute) label, and \(p\) is the probability distribution describing the number of occurrences of the element (resp. items in the attribute’s value) in each instance.

\(p_1\) : PDF uniform distribution over \([1,1]\)
\(p_2\) : PDF uniform distribution over \([1,10]\)
\(p_3\) : PDF normal distribution over \([10,1000]\) with mean 35
\(pcfg_1\) : PCFG defined over a vocabulary of names
\(pcfg_2\) : PCFG defined over a vocabulary of English words
\(pcfg_3\) : PCFG defined over \{“CDN”\}

†While both DTDs and XML Schema allow specification of types where the maximum number of occurrences of an element is unbounded, ToXgene requires an upper bound in all element genes for obvious practical reasons; if no bound is given, ToXgene uses a default value, which is a parameter that can be set by the user.
‡ToXgene allows other characters to be used as list separators.
Input: a gene $g$
Output: an instance of $g$
1. If $g$ is a text gene then return a sample from the probabilistic model of $g$
2. If $g$ is an attribute gene annotated with $(l, p)$ then
   3. Let $n$ be a sample from $p$;
   4. For $1 \leq i \leq n$ let $v_i$ be an instance of the text gene that is the child of $g$
   5. Return an attribute with label $l$ and value $v_1 \otimes \Box \otimes v_2 \otimes \Box \cdots \otimes v_n$
6. If $g$ is an element gene annotated with $(l, p)$ then
   7. Let $n$ be a sample from $p$; let $g_1, \ldots, g_k$ be the genes that are children of $g$
   8. For $1 \leq i \leq n$ do
      9. Let $e_i$ be an empty element with label $l$
     10. For $1 \leq j \leq k$ let $v_{ij}$ be an instance of $g_j$
     11. Insert $v_{ij}$ into $e_i$
   12. Return $e_1, e_2, \ldots, e_n$

Figure 3. Algorithm for generating instances of ToXgene content genes; $\otimes$ denotes the string concatenation operation, while $\Box$ denotes the attribute list separator.

Figure 4. A gene describing review documents. Choice genes are represented by diamond-shaped nodes, and group genes are represented as oval nodes.

of the gene. Similarly, text nodes are annotated with their type and the probabilistic model describing their actual values.
The instantiation of a gene can be done by a simple recursive algorithm that visits each node in the gene tree, starting from the root and in a depth-first-search fashion, and proceeds accordingly to the current node, as depicted in Figure 3. For a gene describing textual content (line 1), we simply return a sample from the associated probabilistic model (e.g., a PCFG for textual content or a PDF for numeric content). For a gene describing an XML attribute, we determine the number of literals that will be in the attribute’s value (line 3), generate as many instances of an associated gene, and concatenate such instances using a user-specified separator character (line 5). For a gene describing XML elements, we first determine the number of elements to be generated (line 7), and, for each such element, we iterate over the genes defining the element’s content, insert their instances appropriately.

2.2. Control genes

Several kinds of control genes are defined in ToXgene, for various purposes. The goal of such genes is to allow the encoding of constraints that cannot be easily captured by schema languages. For instance, suppose we want to model that 25% of the book reviews written by the customers of our bookstore are brief (say, 1 or 2 lines at most), while the remainder of them are long and detailed (say, several paragraphs long), and that only 40% of the reviews have the email address of the customer and a date associated with them. In ToXgene, this can be specified using choice and group genes as follows. A choice gene contains one or more group genes, each of which associates a probability of occurrence to a sequence of one or more genes. When a choice gene is instantiated, exactly one group gene is chosen randomly and all (and only) those genes in the chosen group are instantiated. Figure 4 illustrates such a gene; note that we model the different kinds of comments by using different PCFG’s in the content of the gene for comment elements.

It should be noted that the simple algorithm in Figure 3 can be adapted to handle control genes in a straightforward way. We define other kinds of control genes for specifying recursive elements, content sharing, integrity constraints, and conditional instantiation of (groups of) genes (i.e., if-then-else constructs). Next section discusses the genes in our tool; our discussion is not exhaustive, and we refer the reader to ToXgene’s documentation [5] for further details.

3. ToXgene template specification language

In this section we describe the different kinds of genes in ToXgene templates using the notation in our Template Specification Language (TSL) [5]. TSL is an extension of XML Schema that includes annotations for guiding the data generation process; we assume the reader has a basic understanding of XML Schema and focus on the extensions we introduce. Figure 5 contains most of the examples we will use to illustrate the discussion in this section. Note that all ToXgene-related annotations are prefixed by the keyword tox; for clarity of exposition, we present all such annotations using a different font.

Our discussion is organized as follows. Next Section gives an overview of ToXgene templates and the use of probability distributions in them. Content genes are covered in Section 3.2.
which describes the specification of textual content as well as elements and attributes. The discussion about control genes is divided into four further sections: we cover irregular and recursive XML content in Section 3.3, content sharing in Section 3.4, integrity constraints in Section 3.5, and a miscellany of other aspects in Section 3.6.

3.1. The anatomy of a template

An XML Schema specification defines one or more types of two kinds: simple types describe textual content, while complex types describe XML elements. XML Schema defines several base simple types that correspond to most common datatypes: strings, numbers, dates, etc., and allows the specialization of these types (e.g., specifying a range for an integer type). Complex types may include both simple as well as other complex types in them, thus forming type hierarchies, which are, in general, represented as trees (XML Schema allows recursive types, which we discuss separately). In XML Schema, a type is defined by a set of values, called the domain of the type, and a set of lexical representations for the elements in that domain [15]. For example, “100” and “1.0E2” are lexical representations of the same element in the domain of integer numbers. (We note in passing that the problem of determining whether a document
satisfies the constraints of an XML Schema specification is called validation and is central to XML data management. In essence, the validation problem consists of checking whether there is an assignment of types to elements in the document such that the type of the root element in the document is consistent with the XML Schema specification [29].

As discussed in Section 2, a detailed schema is often insufficient for generating realistic synthetic data; thus, we introduced the notion of genes, which add generative models to the type definition. To avoid confusion of terminology with XML Schema, throughout the article, we refer to a (possibly annotated) XML Schema type as a gene; also, we say that a given XML content is an instance of a gene if it is a lexical representation of an element in the domain of that gene. Note that the definition of instance applies both to content genes as well as to control genes, which essentially define rules for instantiating (possibly a subset) the content genes they contain. Finally, a ToXgene template may define several genes, describing either individual documents or collections with several similar documents; each run of ToXgene results in producing instances of those genes describing XML documents in the template.

Defining probability distribution functions. XML Schema, naturally, does not provide notation for specifying probability distribution functions, which are central to the definition of genes. In ToXgene, PDF’s are declared using tox-distribution annotations and are uniquely identified and referenced to by their names. For example, Figure 5(b) shows the declaration of a PDF named \( c_1 \) that is defined by an exponential distribution over the interval \( [5,100] \) with mean 35. ToXgene supports the uniform, normal, exponential and log-normal distributions, and multinomial distributions.

As discussed in Section 2, PDF’s are used both as generative models for numeric values and for determining the number of instances of element and attribute genes. Other uses of PDF’s are: determining the length of instances of genes describing textual content, and determining the number of days to be added to a starting date in date genes (see below). For instance, in Figure 5(d), the PDF \( c_1 \) is used for determining the length of the strings in the content of the title elements.

If the template does not specify a PDF for a given aspect of a gene (e.g., the number of occurrences of an element), ToXgene uses an uniform distribution over the associated interval of values given in the XML Schema type specification. For instance, in Figure 5(d), the number of elements in an instance of the element gene defining author elements is assumed to be uniformly distributed over \([1-5]\). Also, if the user specifies a PDF that is inconsistent with the type specification, ToXgene adjusts the PDF whenever possible and notifies the user through a warning message. ToXgene also has customizable default values to use in those cases where a type specification allows “unbounded” quantities or values (e.g., a simple type may specify only that the domain of values is the set of integer numbers).

3.2. Content genes

We discuss the genes based on simple types, which we call literal genes, and genes that extend complex types, called structured genes.
3.2.1. Literal genes

ToXgene provides annotations for defining genes for most of the base types specified in XML Schema [15], namely: all numerical types, string and date. We discuss each kind of gene separately.

Dates and numbers. Numeric genes are specified by a probability distribution over a closed interval (i.e., no infinite domains are allowed), and a pattern for producing the lexicographic representations of values sampled from that distribution. Infinite domains (e.g., the domain of the positiveInteger type is the set of natural numbers greater than 0) are replaced by finite domains where $-\infty$ and $+\infty$ are replaced by the smallest and largest values of that type that can be represented by the machine, respectively. For example, Figure 5(c) shows the specification of a gene for integer numbers drawn from a skewed probability distribution specified in Figure 5(b).

Date genes are specified in a similar way, by a interval of dates and a probability distribution. Instances of a date gene are obtained by adding a random number of days, sampled from the given distribution, to the starting date in the interval.

Strings. String genes are specified by a generative model (e.g., a PCFG), and a probability distribution for determining the lengths of their instances. ToXgene comes with several generic built-in models for commonly used nouns such as names of countries, names of persons, email addresses, etc. We also provide three probabilistic models for producing gibberish or text according to the TPC-H [33] and XMark [28] standards, and a simple model based on regular expressions. The probability distribution for the lengths of the strings is optional, and the user can customize a default distribution for this purpose. Figure 5(a) shows the declaration of a literal gene for string literals that match a regular expression; Figure 5(d), on the other hand, contains two string genes that use the TPC-H text generation model. Finally, as we discuss in Section 4, our tool can be easily extended to use other text generation models for producing domain-specific textual content if necessary.

ID and IDREF(S). The base types ID, IDREF and IDREF(S) are also supported by our tool, and are declared as usual literal genes with some added integrity constraints (e.g., uniqueness for ID attributes). The generation of values of these types is discussed in Section 3.4.

3.2.2. Structured genes

While text fragments form the content of XML documents, elements and attributes are the building blocks that form the structure of such documents. We now discuss how we extend the XML Schema notation for specifying these and other forms of complex XML content.

As discussed in Section 2, element genes are specified an element label, a PDF that determines the number of elements in an instance, and one or more genes that define the content of each instance. More precisely, each instance of an element gene is a sequence of $n$ elements (where $n$ is sampled from the probability distribution), all of which having the given label and whose contents are instances of the respective content genes. Following the XML Schema
Figure 6. Specifying random structures in a gene.

convention, the genes defining the content of an element are given in either a simpleType or a complexType construct; also, such genes can be declared explicitly or can be “imported” by means of referencing to a named gene (see details below). For example, the author element gene in Figure 5(d) uses the my_author gene, whose declaration is not shown, while the title element gene explicitly provides a literal gene to be used when producing its instances.

Genes for defining attributes are declared in a similar fashion, with the added restriction that their content must be specified by a simpleType construct, which in TSL is restricted to having literal genes only. An instance of an attribute gene is a (possibly empty) list of literals, separated by blank spaces; the probability distribution determines the length of the list.

Structured genes declared within a complexType declaration can contain a mix of genes for elements, attributes, as well as literal genes; thus, an instance of such a gene is a sequence of elements, possibly mixed with text. Note that the attributes of an element are declared within a complexType construct in XML Schema, as well as in TSL. For an example, consider the gene for price elements in Figure 5(d), which contains the specification of an attribute (currency). (The tox-value construct is used for declaring a gene that always return a constant – CDN in the example – and can be used for defining both literal and structured genes.) The content of the book element depicted in Figure 1(a) is an instance of the gene in Figure 5(d).

Note that ToXgene supports all element content models defined in XML [12]; namely: simple content (literal values only), element content (elements only), mixed content (a mix of both elements and literals), and empty content.

Named genes. In XML Schema, types can be named (e.g., the complex type my_book in Figure 5(d)) or anonymous (e.g., the type for price elements in the same figure). ToXgene allows the use of named types as well, and, of course, all annotations given to a type are
3.3. Irregular and recursive XML content

In this section we discuss genes that were introduced to capture other aspects of “realistic” data that are not easily expressible by probabilistic means only.

3.3.1. XML content with irregular structure

One of the main advantages of XML is its ability to represent semi-structured data (i.e., data with irregular structure) [2], which arise naturally in a variety of applications. For instance, in the hypothetical bookstore scenario, some books may have editors instead of authors; also, some reviews may be anonymous while others may contain the name or email address of the users who wrote them. While XML Schema provides notation for allowing irregular structures in a document (e.g., one can specify that some elements are optional), it does not allow one to specify how irregular a document is. In this section we describe two ways of describing irregular XML content in TSL.

We start with the choice and group genes discussed in Section 2.2; which in TSL are denoted by tox-alternatives and tox-option constructs, respectively. For example, Figure 6 shows the TSL notation for the gene in Figure 4. Note that the named genes t1 and t2 could specify different structures (i.e., different element genes), different PCFG’s for the text (e.g., longer or shorter sentences), or both. An instance of a choice gene is obtained by choosing at random one of its groups, based on their probability mass, and instantiating all genes in the chosen group.

The other control gene we introduce conditions the execution of genes to the value of a boolean expression, using an if-then-else construct. This allows, for instance, the specification

Figure 7. Specification and instantiation of recursive element genes.

preserved when it is referenced by name. Naming types is convenient when one wants to reuse the type definitions in several places.
that in a book order, a discount element gene should be instantiated whenever the order contains 10 or more books. We postpone the discussion of these conditions to Section 3.5.2, after we introduce the language for defining the boolean expressions in the next section.

3.3.2. Recursive elements

ToXgene also allows the generation of recursive elements. Before we give the notation for specifying recursive content, let us define some terms. A recursive element is an element that has an ancestor whose label is identical to that element. The level of a recursive element is the number of ancestors of that element that have the same label; for completeness, we say the level of an element that has no ancestors with identical label is zero. The recursive fan-out of an element of label $e$ and level $l$ is the number of its descendants with label $e$ that occur at level $l + 1$.

Recursive elements are specified by annotating an element with three probability distributions that specify: the total number of elements to be generated (specified as before); the maximum level of any recursive descendant; and the recursive fan-out of the elements. Figure 7 illustrates the notation. In that example we define a gene for up to 500 person elements (specified by the maxOccurs="500" in the outer person element gene), each of which might have up to 5 other person elements (specified by the maxOccurs="5" in the inner person element gene) as their descendants; the maximum level of any recursive element is given by $r1$. Note that the level distribution determines the depth of the element that is produced by the gene. Figure 7(b) shows one possible instance of that gene.

Note that, by requiring all distributions to have a maximum value, we avoid potential infinite loops when processing recursive element genes.

3.4. Content sharing and integrity constraints

In most real datasets, data items are related to one another in several different ways; e.g., in our bookstore example, the ISBN of a book that appears in a book order must also appear in the book catalog. Furthermore, all other details of the book in that order (e.g., its title) must agree with the data in the catalog. We call this data replication content sharing because parts of the contents of some book elements will be shared by some order elements. Also, most real datasets must conform to application-specific constraints; for instance, in our bookstore example, no ISBN value can be assigned to more than one book, and the total price of an order must be the sum of all individual book prices in that order. This section discusses control genes for modeling content sharing as well as constraints in ToXgene.

Note that neither content sharing nor integrity constraints are restrictions on the domains of individual types; i.e., the fact that ISBN values must be unique does not change the domain of the ISBN type. Instead, these constraints apply to the output of the template as a whole; that is, one can view a template as a probabilistic model whose domain contains only those (collections of) documents that satisfy the given constraints.

In ToXgene, both content sharing and ensuring integrity constraints are achieved by means of defining virtual datasets, which hold one or more instances of an arbitrary element gene in an “auxiliary memory”. The contents of a virtual dataset are not output to XML documents.
directly; instead, they can be accessed through other control genes that define queries defined in a simple yet relatively powerful language that we discuss below. Conceptually, all virtual datasets are “materialized” before any output is produced by ToXgene, and they remain unchanged until the end of the data generation session. Thus, content sharing is achieved by defining queries over the same virtual dataset in different element genes; since the virtual data does not change, the instances of these genes can share content by simply copying virtual data items. To ensure integrity of the data produced by ToXgene, we require all integrity constraints to be defined over virtual datasets. Since all virtual data is materialized a priori, we can guarantee that all integrity constraints are satisfied before actually generating any output. We note that the genes describing the contents of a virtual dataset can contain control genes that query other virtual datasets.

All virtual datasets are declared using tox-list annotations in a template; for simplicity, we will refer to virtual datasets simply as lists from now on. Each list has a unique name, used for identifying where the data items come from in a query. Like any other XML content, the elements in a list are ordered. The queries are specified by control genes, called iterators, that access elements in a list (either sequentially or in random order) and query expressions that manipulate the values of those elements.

### 3.4.1. Queries

For clarity, we will refer to the elements in a list as “virtual elements” in the sequel. A query defines a mapping from an ordered sequence of virtual elements into a sequence of XML elements that form the content of one or more documents that are output by ToXgene. As mentioned above, each query is specified by an iterator that defines an access method over a sequence of virtual elements and one or more genes that determine the actual content produced by the query. Each iterator specifies a path expression and (optionally) a selection condition that together determine the sequence of virtual elements in that iterator. The contents of the virtual elements are accessed and manipulated by query expression control genes, declared using tox-expression constructs in a template. ToXgene supports expressions arithmetic and simple string operations, as well as common aggregate functions. We discuss iterators, query expressions, and path expressions in the remainder of the section.

An iterator is a gene that accesses an ordered list of elements selected by a path expression (and possibly some selection conditions), one at a time. Each iterator has a pointer to a current element, much like database cursors [16]; each time the pointer advances to the next element, all genes declared within the iterator are instantiated. We define two kinds of iterators for sequential access, called tox-scan and tox-foreach, and one iterator for random access, called tox-sample. Sequential iterators access the selected elements in the same order as they appear in the lists that contain them; random iterators, on the other hand, access the selected elements in random order, possibly allowing repetition.

Instance of iterators are sequences of XML elements that are obtained as follows. Each instance of a tox-scan or a tox-sample iterator is computed by advancing the current element pointer once and instantiating all genes declared within that iterator. An instance of a tox-foreach iterator, on the other hand, is computed by visiting all elements that match the path expression (and satisfy the selection criteria) and, for each such element, instantiating all genes.
declared in the iterator. One can define tox-sample iterators in which the same virtual element is allowed to be sampled more than once. In our bookstore example, this would be useful for allowing some books to have many reviews, for instance.

While tox-scan and tox-foreach iterators can be used interchangeably in some cases, they are designed for two different purposes. The tox-foreach iterator is designed for the cases where all virtual elements are to be used in the content of a single element. For instance, assume we create a list with various books for the purpose of ensuring the constraint that each book has a unique ISBN, and that we want to have all books grouped into a "catalog" element; this can be achieved by defining an element gene for the catalog element containing a tox-foreach iterator defined over the list of virtual book elements. The tox-scan, on the other hand, is intended for the cases where the virtual elements are to be used in the content of several different elements. For instance, suppose now that we want to model batch updates to the bookstore catalog by generating several XML documents, each containing from one to 10 new books (note we need to use a list to ensure uniqueness of the ISBN values, as discussed below). This can be achieved by defining a gene for collections of documents (Section 3.6) in which the genes defining their root elements contains a tox-scan iterator over the list of virtual books. Each time a new document is produced, the iterator will advance to “unused” virtual books, thus ensuring that no book appears in more than one document.

We note that iterators can be nested arbitrarily (by having the path expression defining the nested iterator refer to the name of another iterator), thus allowing the use of most common data manipulation operations, such as Cartesian products, joins, and nest/unnest, etc. as we discuss below. Also, although iterators are designed for querying virtual datasets that are populated by ToXgene, they can also be populated by loading existing data, thus allowing the use of some real data in ToXgene’s output as well as reusing existing synthetic data. Finally, iterators may be assigned names which are used for disambiguating queries, as we discuss below. All references to a named iterator are prefixed by the $ symbol.

Path expressions. A path expression is a sequence of labels of the form \([r[/l_1/\ldots/l_n]/!]\), where the (optional) \(r\) is either a list or an iterator name, each \(l_i\) is an element label and the (optional) ! symbol denotes any textual data. A path expression \(r/l_1/\ldots/l_n\) defines a (unique) ordered sequence of elements or text values that can be reached by traversing a path that matches \(l_1/\ldots/l_n\), starting from an initial element \(e\): if \(r\) is an iterator name, then \(e\) is the current

```
EXP :: ATOM | EXP OP EXP | (EXP)
ATOM :: constant | literal | gene | [path_expression] |
AGG [path_expression]
OP :: + | - | * | / | % | #
AGG :: MIN | MAX | COUNT | SUM | AVG | LEN | CONCAT | DISTINCT
```

Figure 8. Grammar for query expressions.
element in that iterator; on the other hand, if $r$ is a list name, then $e$ is a virtual element that contains all elements in that list.

Query expressions. The query expressions define the actual mapping of virtual elements into the actual content that is produced by ToXgene, and may involve arithmetic operations, basic string manipulation, as well as aggregate operations. For simplicity, we require that each query expression evaluates to a literal value. The operands of each query expression are one of: a constant, an instance of a literal gene computed on the fly, the result of a path expression, the result of an aggregate function over elements matched by a path expression, or the result of some other expression. The grammar defining ToXgene's query language is given in Figure 8. Expressions are declared in tox-expr genes inside a template; next, we discuss each kind of operand in turn.

Numeric and date constants are given in the usual way, and text constants are required to be enclosed by single quotes to allow parsing. Literal genes are referenced in an expression by their names prefixed with a ~. Path expressions used as operands in a query expression are required to return a single literal value, and are evaluated as discussed above. If a path expression does not start with either a list name or a reference to an iterator (that must be an ancestor of the query node in the template), ToXgene evaluates the path expression using the current element of the closest iterator gene that is an ancestor of the gene containing the expression. Aggregate functions return values that are computed by processing one or more elements that are specified by a path expression. The aggregate functions provided by ToXgene are the usual MIN, MAX, COUNT, SUM and AVG; the string operations LEN and CONCAT, which return the length of a string argument and a string which is the concatenation of its arguments, respectively; and DISTINCT, which returns a list containing all distinct values in its argument.
Figure 10. A query over a list of virtual books.

ToXgene allows expressions involving arithmetic and string operations (% denotes the modulo operator, while # denotes string concatenation). Our tool has a simple type system that allows static and dynamic type checking as well as casting (e.g., numbers are cast into strings if used in a concatenation expression); we omit details for lack of space.

Examples. Figure 9 shows the definition of a list of 200 virtual books, each containing an ISBN and a title, as well as from 1 to 5 author_id elements, which are in turn sampled (without repetition) from another list (not shown). Note that the restriction on the number of author identifiers per book is specified in the gene for author_id elements, and not in the iterator gene it contains. Note also that the list specification in Figure 9 requires that all book ISBNs are unique; we discuss how this is enforced below.

Figure 10 shows a query that returns a sequence of book elements copied from the list in Figure 9. Note that the tox-foreach iterator is nested within the tox-scan iterator, which means that the list of author_id elements is recomputed each time the scan iterator advances to another virtual book. Note also that using the tox-scan iterator allows different numbers of books in different runs of ToXgene (the actual number of books is sampled from [100,200] with uniform probability), as well as ensures that the books appear in the same order as they are defined in the list. On the other hand, the nested tox-foreach iterator copies all author identifiers in each virtual book into a separate author element in the result of the query. Finally, note the string expression 'author'#[!] in the query above which adds the prefix 'author' to each author identifier stored in the list.
**Selection conditions.** As discussed above, each iterator accesses a sequence of virtual elements matched by a given path expression; we now discuss `where` clauses that can be used for specifying selection criteria for those elements. ToXgene supports boolean expressions involving the following predicates: the unary `EXISTS`, which evaluates to true if the result of a given query is non-empty; and the binary predicates `EQ`, `DIF`, `LT`, `LEQ`, `GT` and `GEQ`, standing for the usual comparison predicates =, ≠, <, ≤, >, ≥, respectively. The evaluation of an `EXISTS(Q)` predicate is done by evaluating the query `Q` and checking whether its result is non-empty. A binary predicate `OP(P, Q)` evaluates to true depending on how the result of query `P` compares to the result of query `Q`. By default, each path expression inside a query in a `where` clause is evaluated against each virtual element that is matched by the path expression defining the iterator. However, one can define path expressions that refer to lists or other (named) iterators, as usual.

ToXgene supports a variant of the comparison predicates above in which the queries `P` and `Q` may evaluate to a sequence of literals as opposed to a single value. In this cases, each query must be prefixed by one of `ANY` or `ALL` keywords, indicating whether the predicate must hold for at least one or for all pairs of values taken from the results of `P` and `Q`. For instance, suppose all books in the bookstore are categorized by genre; one could generate a list of all romance-lover customers by joining the list of orders and the list of books, and selecting those orders where all books are in the romance genre.

Figure 11 shows how to nest iterators and `where` clauses to effectively define an iterator over the `join` of lists `L1` and `L2`, based on the equality of `[L1/a/x]` and `[L2/b/y]` values. Note that the equality condition is specified as a `where` clause in the nested iterator. Furthermore, note that other selection conditions can be used as well; for example, the outermost iterator restricts all `[L1/a/x]` to be greater than 10.

### 3.5. Integrity constraints

In this section we discuss the specification of several common kinds of integrity constraints in ToXgene, and a few other control genes that are introduced for this purpose.
3.5.1. Uniqueness and key constraints

One common kind of constraint in most real datasets is requiring certain “fields” to serve as “keys” (i.e., contain unique identifiers for the objects that contain them). For example, in our bookstore application the ISBN of each book uniquely identifies that book and is a natural candidate for being a key. Keys are not required to be atomic; for instance, a key for the collection of user reviews might be formed by a user’s email address and an ISBN of a book (i.e., disallowing users to review the same book twice). Studying and reasoning about natural notions of keys is a classical problem in data management, and XML is no exception [13]. Below we discuss ToXgene’s support for generating XML data that satisfy various kinds of keys, starting with the generation of unique values to be used in those keys.

There are two ways of generating unique identifiers in ToXgene. The first method (which is used in many hard-coded synthetic data generators) consists of using a sequence of monotonically increasing integer numbers (possibly prefixed by some keyword). To this end, ToXgene provides a built-in generator for number sequences, and, as discussed above, one can use string concatenation to use such numbers as prefixes or suffixes to predefined keywords. Another method supported by ToXgene is to specify a combination of elements that must be unique among all virtual elements in that list. This constraint is enforced by repeatedly generating instances of the gene in the list until the desired number of instances is met and the uniqueness constraint is satisfied. This is done in Figure 9, which specifies that the ISBN of each book must be unique. While the second method allows random key values and may be desirable, e.g., when testing indexing mechanisms, it suffers from two drawbacks. First, detecting duplicates in the list requires either a linear scan of all elements already in the list, or using an indexed data structure for keeping all used keys until all data is generated (this is the method chosen in ToXgene). Second, the time required to generate all elements in the list depends on the probability of the literal gene (producing the values required to be unique) returning repeated values.

We now briefly discuss the generation of documents satisfying global or local XML keys [13]. A global XML key must hold in the entire document, and can be enforced simply by having generating unique values to used as keys as discussed above. For instance, the ISBN values in the list defined in Figure 9 could be used as global keys for books. A local XML key, on the other hand, must hold only in the context of a given element. For example, requiring that the same author identifier does not appear more than once within a book element can be defined using a local key. This particular constraint can be enforced in ToXgene as follows. Consider again the query in Figure 10 over the list of virtual books defined in Figure 9, and assume that all author identifiers in author_list are unique. Since the author identifiers are obtained by sampling without repetition in Figure 9, it follows that the result of the query satisfies the local key described above.

ID and IDREF(S) attributes. ID and IDREF(S) attributes are mechanisms for defining references among elements in a document as follows. ID attributes must uniquely identify the elements that bear them, while IDREF(S) attributes must refer to existing ID attributes.
Ensuring uniqueness of ID attributes is done similarly as discussed above for keys. Ensuring that IDREF(S) attributes refer to existing ID attributes, on the other hand, is achieved by element sharing (i.e., define the values of the IDREF(S) attributes as queries over the list of unique ID values).

3.5.2. Functional dependencies and more

In this section we discuss mechanisms for generating other forms of constraints and correlations in XML documents. We start with functional dependencies, which are classical constraints in databases. In general terms, a functional dependency \( x \rightarrow y \) holds in a dataset if, every pair of data items that agree on their value of \( x \) also agree on their value of \( y \). For instance, one possible functional dependency in our bookstore scenario could be to require publishers not to publish a book with a title that has been used by another publisher (i.e., the book title functionally determines the book publisher).

ToXgene supports the generation of documents satisfying a functional dependency \( x \rightarrow y \) in two ways. First, if there are just a few possible values for \( x \) and \( y \) above, one can use an if-then-else control gene as follows. The “if” part of the gene contains a boolean predicate defined as in a where clause (recall the previous section), while the “then” and the optional “else” part each contain a group gene (recall Section 3.3.1). We note that if-then-else genes can be nested arbitrarily, thus allowing the implementation of a “case of” construct. Another way of enforcing a functional dependency \( x \rightarrow y \) is to create separate lists with all unique values for \( x \) and \( y \) separately, and use a third list to store, for each value of \( y \), a set of values of \( x \). Then, each time an instance of \( x \) is generated, we use a query to obtain the associated value of \( y \).

Structural dependencies. Another form of correlation in XML content is what we call a structural dependency, which conditions the occurrence of certain elements (or attributes) to the occurrence (or absence) of other elements (or attributes) in the document. For example, a structural dependency in our hypothetical bookstore application would be to require that all book reviews that contain the email address of their author also contain the date in which the review was written. Another plausible dependency would be to require books to have either authors or editors, but not both. These kinds of structural constraints can be described using choice and group genes as discussed in Section 3.3.1).

Another form of structural dependency is when the value of certain elements or attributes determines the structure of the document. For example, shipping addresses in orders destined to Canada should contain an element labeled province, while all other addresses contain states; that is, the name of the country determines the structure of the element describing an order. Such kinds of constraints can be easily specified using an if-then-else control gene.

The distinction IDREF and IDREFS attributes is that the former must be atomic while the latter may be multi-valued.
Other constraints. As discussed in Section 2, a common kind of constraint in most real datasets is that certain values are the result of some computation on other values; e.g., the total price of an order is the sum of the prices of all books in that order. Such constraints are enforced in ToXgene by materializing all “input” data in a list, and using a query to compute the desired result. In the example above, one would materialize a list with all orders, each having several books with their prices; the actual orders would contain “copies” of the books (and their prices) as well as another element with the sum of those prices.

Another form of specifying constraints in lists is by adding boolean predicates as those used in the specified as where clauses (recall the previous section) that must be satisfied by all elements in the list. For instance, in the bookstore example, one might want to require that the total price of all orders are higher than a minimum value which is higher than the cheapest book in the catalog. Another interesting example comes from the TPC-H benchmark, in which customer identifiers are sequentially generated but sparsely distributed to avoid insertions to always occur at the “end” of the index space. The actual constraint is that only the first 8 out of each set of 32 integer can be used to form a key (i.e., 1, 2, 3, 4, 5, 6, 7, 33, 34, ...); this constraint can be expressed as where="LT([customer/id]%32,8)" in ToXgene.

As a final remark, we note that all control genes discussed here are orthogonal to each other and can be mixed with the use of queries and iterators, thus providing great flexibility in the specification of both structure and content for the synthetic XML data.

3.6. Miscellany

We conclude our overview of the ToXgene template language by discussing how documents are specified in a template and how our tool can use existing data (synthetic or otherwise) for generating new synthetic data.

Document genes. An XML document is essentially an element (called the root element of the document). Thus, in a template that contains several element gene declarations, one must specify which gene should be used as the root of the documents to be produced. For this purpose, we provide a tox-document gene that specifies a name for the document to be produced as well as a gene for the root element of the document. A single TSL template may contain many document genes.

ToXgene provides several ways of arranging data into XML documents. For instance, in our hypothetical bookstore application, one could have the entire catalog as a single document or as a collection of documents, each describing a single book. These cases are illustrated by Figures 12(a) and 12(b), respectively. The document names in a collection are formed by appending the value of a counter to the name specified in the template.

We note that the processing of the genes in a document collection is done in the same way as in a single document. Thus, if we use the same initial random seed in two runs of ToXgene on the templates in Figures 12(a) and 12(b), the ith book in the first document will correspond to the ith document in the collection. Other variations in which each document contains a random number of book elements are possible as well.
<tox-document name="books"
<element name="books">
  <complexType>
    <element name="book" type="book" minOccurs="200" maxOccurs="200"/>
  </complexType>
</element>
</tox-document>

(a) Single document, multiple books.

<tox-document name="book" copies="200"
<element name="book" type="book"/>
</element>

(b) Collection of documents, each with a single book.

Figure 12. Declaration of XML documents in ToXgene.

Reuse of existing data. Using existing data might be necessary to increase the usefulness of a synthetic dataset. One typical example is when testing a system that must interface to other systems, say by formulating queries to be processed remotely. In order to obtain meaningful answers, such queries must refer to some data items residing in the remote systems. Another example is using real data for increasing the readability of the synthetic data; for instance, using non-gibberish text for customer names or email addresses, which might result in a less distracting testing of the system's interface.

ToXgene addresses the issues above in two ways. First, it allows the loading of existing data into a list which can be queried as before. We note that a gene for such a list must be specified (despite the fact that it will not be instantiated), to provide the necessary information for type checking and casting. Also, ToXgene enforces integrity constraints over such lists in the same way it does over lists it generates. Second, ToXgene provides built-in literal genes that use real values for several common kinds of data (e.g., country and city names, Internet domains, etc.), and allows the user to provide her own literal genes, as discussed in the next section.

4. ToXgene architecture

In this section we discuss the architecture of our current implementation of ToXgene, focusing on how it can be used by other applications. We implemented the tool in Java 2, primarily aiming at achieving increased portability and at leveraging the existing support for XML processing in this platform.

Figure 13 gives a very high level description of ToXgene’s architecture and illustrates how user applications interface to the data generation engine. The ToXgene API provides an interface between user applications and the three main modules in our tool: the Template Parser, which converts the templates in TSL into genes; the Gene Processor, which implements the algorithms for instantiating the genes; and the ToXgene Reporter, which provides reporting messages concerning the execution of the template, as well as errors and warnings. As depicted in Figure 13, ToXgene’s command line interface (CLI) module is implemented as an independent application that accesses the data generation engine through the API as well.
We note that ToXgene’s API allows one to develop special-purpose data generators in which one or more templates are hard-coded.

The Template Parser provides essentially one service: compiling an input template into a sequence of genes for lists, documents, and collections of documents. A gene for a document collection can be used in a way to produce one document at a time, instead of all of them at once. ToXgene’s parser validates input templates against a DTD for TSL and performs extensive type checking on query expressions; thus, all syntactic errors and several kinds of type errors are detected at compile time. The actual data generation is done by the Gene Processor, which essentially implements the algorithm described in Section 2. A salient feature of our implementation is that the actual instances of the genes in a template are written to a buffer provided by the application using ToXgene, and several kinds of buffers can be used (in fact, any `PrintStream` Java object can be used). For instance, one can write the synthetic data into files by using file buffers (as is done by the CLI module), or keep the synthetic data in memory-resident buffers.

The Query Processor module evaluates both query expressions and `where` clauses discussed in the previous section; this module triggers the materialization of all lists required for a given query that are not yet materialized. Another important feature of ToXgene is that it can use a persistent object manager (POM) for storing and accessing lists on disk; effectively, this allows ToXgene to generate synthetic datasets that are larger than the main memory of the host machine. Our current implementation uses the PoBoy\(^*\) [24], which provides a simple implementation of all collection classes defined in Java, and thus can be used without any modification of ToXgene’s original code.

As mentioned in Section 1, ToXgene allows user applications to provide their own synthetic data generators to be used instead of the literal genes already provided. This is done by adding

\(^*\)At the time of writing, the PoBoy library is only available from ToXgene’s website.
an entry to ToXgene’s list of literal genes; each entry in this list has a unique name for the type as well as a full path to a Java class that must implement a simple interface for literal genes. Once a new type is registered, it can be referenced like any other type inside a template.

We conclude this section with a discussion about memory management. Essentially, ToXgene uses memory for three main purposes: storing lists, enforcing integrity constraints over lists, or providing buffer storage for the POM. The amount of memory used for buffering by PoBoy can be set by the application, and remains constant throughout the entire data generation session. On the other hand, the amount of memory used for storing lists and unique values depends on the number of instances in those lists, and may vary over time, as new lists are materialized in memory, or deleted whenever they are no longer needed. In an attempt to avoid wasting resources, we keep track of all genes that reference each list declared in the template; a list is only materialized when the first gene referencing that list is instantiated. Also, we free the memory used by a list whenever all genes that refer to that list are processed. This simple memory management mechanism is particularly useful when one must combine the virtual elements in several lists into a larger one (e.g., when generating data conforming to functional dependencies as discussed in Section 3.5.2).

5. Validation

This section argues about the effectiveness of ToXgene. First, we summarize experimental results carried out to validate our tool by comparing its output against that of hard-coded data generators in two established benchmarks. Next, we describe two successful applications of ToXgene: a comprehensive XML benchmark suite, and a simulator engine for complex web services.

5.1. An experimental validation

In an attempt to verify whether our tool could reproduce industry-grade benchmarking data, we modeled the synthetic data in the TPC-H [33] and XMark [28] as ToXgene templates, and compared the execution of the query workloads in those benchmarks against our data and the data produced by each hard-coded generator. (We provide a brief summary of the results obtained which is complementary to the discussion in Barbosa et al. [7].) The rationale for using these benchmarks is as follows. Although TPC-H is a relational benchmark, it was chosen because it defines several non-trivial integrity constraints over its dataset; XMark was chosen because it was the most complex XML benchmark at the time we started the development of ToXgene.

We built two datasets for each benchmark: one using ToXgene and the other using the hard-coded data generator provided with the given benchmark. (For TPC-H, we convert our XML data into a relational database in a straightforward way.) We ran the query workloads against each dataset, measuring the execution time and the size of the result of each query. We compared the results on each dataset considering four scenarios defined by the following orthogonal criteria: query execution time (short versus long), and query result size (small versus large).
The TPC-H benchmark models a business application and contains data about customers, suppliers, parts and orders; several non-trivial integrity constraints are defined over the database. This experiment was done with 100MB TPC-H databases; the one generated with ToXgene was divided into 6 XML documents, corresponding to roughly 400MB of space. DB2 V7.1 was used to run the queries and its XML Extender [21] was used to load the XML documents into an empty database created using the standard TPC-H tools.

The XMark benchmark consists of an auction dataset, stored as a single XML document. The data describes items for auction, persons (buyers or sellers), etc.; several references among elements are defined as ID and IDREF(S) attribute (e.g., each auction corresponds to a particular item and has a person as the seller). The XMark experiment was done with 100MB documents, and the queries were run using Kweelt [26]. All queries were rewritten in a way that each element (or text value) in the answer was printed in a separate line; the length of the answer in lines was used as the measure of query size.

The results of both experiments above can be summarized as follows. First, the data produced by ToXgene did not violate any integrity constraints in the respective schemas. Second, both the running times and the query sizes on the data produced by ToXgene were very close to those obtained using the data produced by the respective hard-coded data generator. On average, the ratios of query execution times (defined as the time taken on ToXgene’s data divided by the time taken on the hard-coded generator’s data) were 0.97 and 0.99 for the TPC-H and XMark experiments, respectively. The average ratios of query result sizes (computed similarly as the time ratios) were 1.2 and 0.88 for the TPC-H and XMark experiments, respectively. Finally, we note that the data generation time using ToXgene for both datasets was in the order of tens of minutes (58min for TPC-H and 25min for XMark), which we find very reasonable.

5.2. The XBench XML benchmark suite

ToXgene is the data generator of the XBench family of benchmarks [34], which models several kinds of application domains, divided in two major dimensions: data-centric versus text-centric and single-document versus multi-document. In typical data-centric (DC) applications, XML documents are used for representing fairly structured data, such as the book catalog we described in our examples. Text-centric (TC) applications, on the other hand, deal with hypertext content written in XML, such as conference proceedings in a digital library. The second dimension in XBench deals with how the data is represented in a single document (SD) or in multiple documents (MD).

XBench defines four benchmarks with different datasets and query workloads, covering all DC/SD, DC/MD, TC/SD and TC/MD domains. The design of the datasets in each of the classes was done by performing a statistical analysis of some representative datasets; e.g., the TPC-W benchmark was the starting point in the DC/SD domain, while a corpus of Reuters news articles was used for modeling the TC/MD domain. ToXgene templates were designed for each dataset in the application domains above; workloads of varying sizes are defined in XBench, ranging from 10MB (small) to 10GB (huge). The sizes of the multi-document datasets range from a few dozen (26 in the smallest TC collection) to millions (2.5M in the largest DC collection). Each XBench workload defines queries, updates and bulk-loading operations. The
queries are subdivided into: core queries (e.g., exact match of path expressions); text-centric queries (e.g., reconstruction of document fragments); and data-centric (e.g., join queries). For more details about the data generation on XBench as well as results on some existing XML DBMSs, we refer the reader to Yao et al. [34].

5.3. A simulator for complex web service executions

ToXgene has also been used for producing XML payloads in a simulator for complex web services, which, at the time of writing, is under development as part of a research project at the University of Toronto and IBM [9]. One of the goals of the project is to allow the debugging of a complex web service in isolation, by simulating the execution of the other services that exchange data with that service. A noteworthy feature of this simulator is that it uses a conceptual model of the entire application which describes each individual service as well as data dependencies among them (e.g., if service \( s_1 \) places a book order to service \( s_2 \) on behalf of customer \( c \), \( s_2 \)’s response refers to \( c \) as well as the books in that order). These kinds of constraints are captured by element sharing, as discussed in Section 3.4.

In this particular project, web services are described using the Business Process Execution Language for Web Services (BPEL4WS) [4] which builds on the Web Services Description Language (WSDL) [14] for specifying the types of the XML messages exchanged by the services. WSDL specifications are semi-automatically translated into TSL templates by the simulator; user input is required for specifying data dependencies among services and the volume of data to be generated in a simulation. Essentially, each template defines several document collections consisting of the messages that are exchanged among services. The simulator interfaces to ToXgene through the API, and obtains an instance of a given document collection whenever the corresponding service is required to “send” a message to some other service.

6. Related work

Probabilistic models, such as Bayesian networks, are useful tools for representing statistical patterns in real world domains [25]. Moreover, such models have been used to describe relational databases [17] as well as XML documents [20]. While previous work has focused on learning the models from existing databases and on answering probabilistic queries, we use these models for providing a flexible and declarative way of generating realistic synthetic data. It is worth noting that existing models are not capable of representing integrity constraints; TSL, on the other hand, is a rich language for specifying probabilistic models of data that supports integrity constraints.

As it is often the case, most previous synthetic data generators for XML are hard-coded tools resulting from benchmarking efforts; examples are XMach-1 [10] and XMark [28]. Unlike these generators, ToXgene is a declarative, general purpose tool that supports the generation of multiple collections of documents, with varying structure, size and complexity. As discussed above, ToXgene is capable of closely reproducing the XMark dataset; although we have not considered XMach-1 in our work, it would be straightforward to model that benchmark as a TSL template.
Aboulnaga et al. [3] describe a data generator that is capable of producing XML documents with both random content and random structure. While that tool accepts a few parameters that describe the shape of the resulting XML tree (e.g., average number of children per element), it does not provide enough control over the synthetic documents it produces (e.g., one cannot specify which element or attribute labels can be used). ToXgene, on the other hand, uses templates that provide a conceptual description of the data to be generated, i.e., our tool is intended for those cases where the user knows a schema for the data to be generated and requires the data to conform to that schema. As discussed in Section 3.3, ToXgene allows the specification of random structures as well as random content in the the documents it produces in a controlled way. Also, our tool can generate fairly more complex XML content than that of Aboulnaga et al. [3], including elements with mixed content, attributes, non-gibberish text, and different numerical and date values. Finally, ToXgene supports several different probability distributions.

Another general purpose tool for generating random XML data is the IBM XML Generator [22], which is also template-based (templates are annotated DTDs). There are many limitations to that generator compared to ToXgene, however. First, it allows one to limit the maximum depth of the document tree or the number of IDs and IDREFs that are present in the document, but it does not allow one to specify actual values for these properties. Second, that tool does not allow the definition of different probabilities of occurrence on a per element basis, nor the generation of literals of different datatypes.

Finally, ToXgene differs from both general-purpose tools above in the following ways. First, ToXgene allows content sharing, which is a requirement for generating correlated content as we discussed in Section 3.6. Second, ToXgene allows the specification of common integrity constraints over the data it produces, thus allowing the generation of consistent ID, and IDREF(S) attributes. (We note that the XMark data generator supports these features by using several identical streams of random numbers, thus without requiring the materialization and management of temporary data; on the other hand, the XMark generator cannot be used for generating data with user-defined structure and content.) Third, ToXgene can (re)use existing XML content, synthetic or otherwise, to produce more useful synthetic XML datasets. Fourth, ToXgene can use literal genes provided by the user, and thus, be customized to specific application domains for which the built-in literal genes are insufficient. Finally, our tool can be used as a component in other applications, as we discussed in Section 4.

7. Conclusion

In this article we introduced ToXgene, which is a declarative synthetic data generator for XML. Our tool contributes to the development and testing of XML-based applications, as well as to the development of XML benchmarks, by allowing the developers to focus on specifying what data is needed, as opposed to a method for producing such data. The data generation model in ToXgene is based on XML Schema’s rich type system augmented with probabilistic generative models describing both the structure and the content of the synthetic documents. As discussed in the article, ToXgene is a powerful data generator capable of reproducing complex benchmarks defining non-trivial integrity constraints as well as complex XML constructs. Our
current implementation of the tool allows it to be used as a component in other applications as well as the generation of large datasets (larger than the main-memory of the host machine). ToXgene has been widely used by researchers and practitioners since its first release.

As future research, one interesting problem we identify is extracting ToXgene templates from existing collections of documents. The motivations for doing so are twofold. First, such templates capture considerable information about the data they describe, and thus are valuable metadata on their own. Second, this would, at least to some extent, automate the design of data generators for realistic synthetic data. Note that this approach would be particularly useful for gathering information about real datasets, which tend to contain sensitive or confidential data, without having to disclose that data. In terms of future development of our tool, we consider the implementation of a template compiler as an interesting option. In this scenario, ToXgene could be used in the development and tuning of templates, which would then be “packaged” as efficient special-purpose applications.

ACKNOWLEDGEMENTS

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada, and by an IBM PhD Fellowship.

REFERENCES


