ToXgene Template Specification Language

Denilson Barbosa
Department of Computer Science
University of Toronto
dmb@cs.toronto.edu

Version 2.1 - March 2003

1 Introduction

ToXgene is a template-based generator for large collections of synthetic XML documents, under development at the University of Toronto as part of the ToX (Toronto XML Engine) project. This document describes the basic functioning of our tool and the ToXgene Template Specification Language (TSL).

TSL templates are essentially XML Schema [1] specifications annotated with specifications of probability distributions and `CDATA` content descriptors, used for generating both attributes and elements. Like XML Schema specifications, TSL templates are XML documents. For increased readability, we present the XML Schema notation of TSL in our examples using `typewriter face`. TSL-specific annotations are attributes or elements added to a schema document; in either case, annotations are prefixed by the string **tox-** and presented using a **different font**.

1.1 Definitions

Def. 1 - Type: A *type* is a data type, as defined in [1]: a type is one of a `simpleType` or a `complexType`.

Def. 2 - Instance: An *instance* of a type t is some XML content that conforms to t. Instances of a `simpleType` are character data (`CDATA`) literals, while instances of a `complexType` are either XML elements, `CDATA` literals, or a mix of both.

Def. 3 - Gene: A *gene* is the specification of some XML content that conforms to a type.

1.2 Anatomy of a template

A TSL template is an XML document whose root is a **tox-template** element, as specified by the following DTD:

```
<!ELEMENT tox-template (tox-distribution|simpleType|complexType|tox-list|
  tox-document)*>
```

These are the main main declarations in a template. We cover each of them throughout this document.

- **tox-distribution**: defines a random number generator according to the given distribution (see Section 2).

- `simpleType`: defines a gene for literals (i.e., `CDATA` values) conforming to the specified simple type (as specified in [1]).
- `complexType`: defines a gene for document fragments that conform to the specified complex type (as specified in [1]).
- **tox-list**: defines a main memory repository for values that can be shared during the creation of the actual XML documents (see Section 8).
- **tox-document**: defines the generation of one or more XML documents of a given type (see Section 6).

1.3 Installation instructions

- **Platform requirements:** ToXgene should run on all Java platforms. It has been developed and tested under Linux and Solaris, using the following:
  - JRE compatible with JDK 1.4.1;
  - Xerces 1.4.4;
  - PoBoy 1.1 (persistent object manager).
- **Installation instructions:**
  1. Download and unzip the file
  2. Set the following environment variables:
     - `TOXGENE_HOME` should point to the full path of the directory where you unzipped the package.
     - `XERCES_HOME` should point to the full path of the directory where you copied the Xerces jarfile.
  3. Add `$(TOXGENE_HOME)/bin` to your `PATH` variable.
  4. To test your installation, execute the script toxgene. You should get the response shown in Figure 1.

1.4 Executing ToXgene

Templates are processed in two steps: validation and generation. The validation process starts with the parsing of the template document, which is done by the Xerces XML parser using the ToXgene DTD. Syntactical and some semantic errors are caught at this time and reported by the Xerces parser automatically. The next step in the validation detects other semantic errors, such as references to non-existent datatypes or path expressions in lists (see Section 8). After the validation steps are done, ToXgene moves to data generation mode, when the actual processing of the template occurs.

ToXgene takes several command options that specify how it behaves. Invoking ToXgene without arguments gives a list of valid options (see Figure 1), most of which are self describing. We next discuss some of these options:

- `-v`: when executed in verbose mode, ToXgene outputs several messages as the template is being parsed. This mode is helpful for developing and debugging templates.

```
ToXgene 2.1 - (c) 2001 by University of Toronto and IBM Corporation

Usage:  ToXgene <options> <template file>
where possible options include (<default value>):
-c Check template only; do not generate files <false>
-h Print this message
-v Verbose mode <false>
-w Omit warning messages <false>
-n Add a new line after each element close tag <false>
-t Show elapsed time <false>
-s v Use "v" as initial random seed
-o [p[f][[s]]] Use a persistent object manager <false>:
                 p is the path where to store temporary files
                 f is the fraction of the memory used for buffering
                 s is the size (in kilobytes) of the memory buffers
                 the default values are:  p=".", f=0.5 and s=8KB
```

Figure 1: ToXgene output listing invocation instructions.

- `-n`: this option makes ToXgene break lines after each element closing tag in the output documents. This options makes ToXgene's output more human readable and thus, also helps in debugging the templates.
- `-s` v: uses v as master random seed. This option is useful if one wants to repeat a workload from a given template.
- `-o [`p`[`f`[`s`]]]`: this option tells ToXgene to use a persistent object manager (POM) to store the intermediate content when processing the templates. This permits ToXgene to generate XML data that is larger than the memory of the machine being used (as is the typical situation in most benchmarks). For details, refer to Section 11.

2 Specifying value distributions

Value distributions are declared by **tox-distribution** elements. User defined multinomial distributions are specified with the aid of **enumeration** subelements as shown below:

```
<!ELEMENT tox-distribution (enumeration*)>
<!ATTLIST tox-distribution
    name ID #REQUIRED
    type (constant | exponential | geometric | lognormal
          normal | uniform | user-defined) ’uniform’
    minInclusive CDATA #REQUIRED
    maxInclusive CDATA #REQUIRED
    mean CDATA #IMPLIED
    variance CDATA #IMPLIED>

<!ELEMENT enumeration EMPTY>
<!ATTLIST enumeration value CDATA #REQUIRED
    tox-percent CDATA #REQUIRED>
```

The attributes of a **tox-distribution** element are defined as follows:

- **name:** uniquely identifies the distribution within the template;

- **minInclusive:** specifies the minimum value that is drawn from the distribution;
- **maxInclusive:** specifies the maximum value that is drawn from the distribution;
- **type:** specifies the type of the distribution; it can be one of:
  - **constant:** specifies a constant distribution where the value drawn is always **minInclusive**;
  - **exponential:** specifies an exponential distribution, with expected value specified by attribute **mean**;
  - **geometric:** specifies a geometric distribution, with mean value specified by attribute **mean**;
  - **lognormal:** specifies a log-normal distribution, with expected value specified by attribute **mean** and variance specified by attribute **variance**;
  - **normal:** specifies a normal distribution, with expected value specified by attribute **mean** and variance specified by attribute **variance**;
  - **uniform:** specifies a uniform distribution in the interval (default);
  - **user-defined:** specifies a arbitrary user-defined distribution given by **enumeration** elements (see below).

User-defined distributions are specified by **enumeration** elements, each containing the following attributes:

- **value:** numeric outcome (the following must hold: **minInclusive** $\leq$ **value** $\leq$ **maxInclusive**).
- **tox-percent:** probability mass associated with the value.

2.1 Examples:

Example 1. Definition of an exponential distribution:

```
<tox-distribution name="n-pars" type="exponential" minValue="1" maxValue="5" mean="2"/>
```

■

Example 2. A user-defined distribution:

```
<tox-distribution name="discount" type="user-defined" min="0" max="30">
  <enumeration value="0" tox-percent="50"/>
  <enumeration value="5" tox-percent="25"/>
  <enumeration value="10" tox-percent="15"/>
  <enumeration value="30" tox-percent="10"/>
</tox-distribution>
```

■

3 Specifying `simpleType`s and constants

A `simpleType` is a specialization of a base type (e.g., string, integer, etc.). Each `simpleType` *inherits* the properties of the base type it builds upon. The following base types are supported in TSL:

- Numerical types: integer, unsigned integer, long, unsigned long, int, unsigned int, short, byte, non-negative integer, positive integer, non-positive integer, negative integer, decimal, float and double;

- String;
- Date;
- `ID`, `IDREF` and `IDREFS`.

Types are either *named* or *anonymous*. A named type is declared once, independently of elements or attributes, and is referenced by its name. The same named type can be used by multiple elements or attributes in a template. Anonymous types are declared within element or attribute specifications, and cannot be reused.

This the general notation for specifying simple types in TSL:

```
<simpleType name="type_name">
  <restriction base="base type">
    <facet1/>
    ...
    <facetn/>
  </restriction>
</simpleType>
```

Of course, the `name` attribute is only used for defining named simple types. Each **facet** element refines the definition of the type; the properties specified in a facet have to conform with the canonical properties of the base type being refined. Important note: *all* facets are optional; if no facets are provided, the default values for the corresponding base type are used.

The generation of instances of `ID`, `IDREF` and `IDREFS` types is covered in Section 8. We now give more details on the various facets that specify the generation of instances of each of the base types.

3.1 Numerical types

An instance of a numerical type is a string representation of a number, extracted from the interval specified by [`minInclusive`, `maxInclusive`]. Instances of a numeric type are either randomly selected from the interval or sequentially computed as discussed below. XML Schema defines various numeric data types, some of which have infinite domains (e.g., the integers). ToXgene, however, imposes bounds on the maximum and minimum values allowed for each numeric type supported[1]. Table 3.1 shows the maximum and minimum values for each of the numeric types supported by our tool.

These are the facets used for defining numerical types:

- `minInclusive`: defines the minimum value allowed for instances of the type. The new value has to be greater than or equal to the default value for the base type.
- `maxInclusive`: defines the maximum value allowed for instances of the type. The new value has to be smaller than or equal to the default value for the base type.
- `minExclusive`: defines a value for which all allowed instances of the type have to be *strictly greater* than. This value has to be greater than or equal to `minInclusive` + S, where S is the smallest absolute number that can be represented by the machine (e.g., 1 for integer datatypes).
- `maxExclusive`: defines a value for which all allowed instances of the type have to be *strictly smaller* than. This value has to be greater than or equal to `maxInclusive` - S, where S is the smallest absolute number that can be represented by the machine (e.g., 1 for integer datatypes).

[1]It is natural to do so, since we can represent only finitely many numbers on digital computers anyway.

Base type	minInclusive	maxInclusive	Example
`integer`	`MINLONG`	`MAXLONG`	123
`unsignedInt`	0	4294967295	123
`long`	`MINLONG`	`MAXLONG`	123
`unsignedLong`	0	`MAXLONG`	123
`int`	-2147483648	2147483647	123
`short`	-32767	32768	123
`byte`	-127	128	123
`nonNegativeInteger`	0	`MAXLONG`	123
`positiveInteger`	1	`MAXLONG`	123
`nonPositiveInteger`	`MINLONG`	0	123
`negativeInteger`	`MINLONG`	-1	123
`decimal`	`MINFLOAT`	`MAXFLOAT`	123.45
`float`	`MINFLOAT`	`MAXFLOAT`	123.456
`double`	`MINFLOAT`	`MAXFLOAT`	123.456

Table 1: Supported numerical datatypes. `MAXLONG` and `MINLONG` represent respectively the largest and smallest integer number that can be represented by the (Java virtual) machine ToXgeneis being executed. Similarly, `MAXFLOAT` and `MINFLOAT` represent the largest and the smallest floating point numbers that can be represented. These values may vary among different implementations of the Java virtual machine. Please refer to the documentation of your local Java runtime environment for details.

- **tox-format**: specifies formatting instructions for writing the actual instances of the type. For numerical values, we use the formatting conventions of the `NumberFormat` Java class (for more details, see [3]).
- **tox-number**: this is ToXgene's main annotation for specifying numeric values. It provides the following attributes:

```
<!ELEMENT tox-number EMPTY>
<!ATTLIST tox-number
    minInclusive CDATA #IMPLIED
    maxInclusive CDATA #IMPLIED
    tox-distribution IDREF #IMPLIED
    sequential (yes|no) 'no'
    format CDATA #IMPLIED>
```

  - **tox-distribution**: specifies a distribution used to generate instances of the type. If none is provided a uniform distribution on the interval [**minInclusive**, **maxInclusive**] is used.
  - **sequential**: specifies whether the numbers are sequentially generated; the default value for this attribute is `no`. Both increasing and decreasing sequences are allowed. The increment is specified by providing a constant distribution; an increment of +1 is used if no distribution is specified. Consecutive instances of the type form an increasing sequence starting at **minInclusive** if the increment is positive, or a decreasing sequence starting at **maxInclusive** otherwise.
  - **format**: same as **tox-format** above; this one takes precedence.
  - **minInclusive**: same as above; this one takes precedence.

  - **maxInclusive**: same as above; this one takes precedence.

Example 3. Anonymous `simpleType` for positive integers in the interval [1,1000]. See Section 4 for more details on element declarations:

```
<element name="my_element">
  <simpleType>
    <restriction base="positiveInteger">
      <minInclusive value="1"/>
      <maxInclusive value="1000"/>
    </restriction>
  </simpleType>
</element>
```

■

Example 4. Named `simpleType` for sequentially generated integer numbers. Consecutive instants of the type form the sequence 100, 98, 96,...:

```
<tox-distribution name=”c1” type=”constant” minInclusive=”-2” maxInclusive=”-2”/>
<simpleType name="sequential">
  <restriction base="Integer">
    <tox-number sequential=”yes” tox-distribution=”c1” maxInclusive value=”1000”/>
  </restriction>
</simpleType>
```

■

3.2 Date types

Date literals are also generated from an interval. There is only one base `date` type, whose default `minInclusive` and `maxInclusive` values are January 1st, 1 A.D. and January 1st, 3000 A.D., respectively. The same facets used for specifying numerical types are also defined for date types, except for **tox-number**, which is replaced by:

- **tox-date**: specifies a random date generator and has the following attributes: **tox-distribution**, `minInclusive`, `maxInclusive` and **format**, with the same usage as defined for numerical types:

```
<!ELEMENT tox-date EMPTY>
<!ATTLIST tox-date
    start-date CDATA #REQUIRED
    end-date CDATA #REQUIRED
    tox-distribution IDREF #IMPLIED
    format CDATA #IMPLIED>
```

Instances of date types are generated by adding a random number of days d to the start date of the specified interval $[\texttt{minInclusive}, \texttt{maxInclusive}]$. d is randomly chosen from the given distribution; if no distribution is specified, d is uniformly chosen from the interval $[0, \mathsf{days}(\texttt{maxInclusive}, \texttt{minInclusive})]$, where $\mathsf{days}(a, b)$ returns the number of days between the dates a and b. In any case, $d + \texttt{minInclusive} \leq \texttt{maxInclusive}$.

Important note: all date constants in a template, such as `minInclusive` values, are specified using the format `YYYY-MM-dd`. Output formatting instructions follow the standards in the `DateFormat` class in the Java language [2].

Type	Description
`text`	A textual string generated according to the TPC-H benchmark [5] rules.
`word`	A word, extracted from the list of words used in the XMark benchmark [4].
`xmrk_text`	A textual string generated according to the XMark benchmark rules.
`fname`	A first name, extracted from the list of first names in the XMark benchmark.
`lname`	A last name, extracted from the list of last names in the XMark benchmark.
`city`	A city name, extracted from the list of cities in the XMark benchmark.
`province`	A province name, extracted from the list of provinces in the XMark benchmark.
`country`	A country name, extracted from the list of countries in the XMark benchmark.
`domain`	An internet domain name, extracted from the list of domains in the XMark benchmark.
`email`	An email address of the form x.y@z, where x is an instance of `fname`, y is an instance of `lname` and z is an instance of `domain`.
`gibberish`	A random string in {a, ..., z, ..., A, ..., Z, 0, ..., 9} ∪ {#,^,*,(,),_,-,=,$,+,{,},[,],?,.,/,~, }.

Table 2: Predefined `tox-string` types.

Example 5. The following example specifies a type for date literals, using `dd/MM/yyyy` as format pattern:

```
<simpleType name="x_date">
  <restriction base="date">
    <minInclusive value="1998-01-01"/>
    <maxInclusive value="2001-28-12"/>
    <tox-format value=”dd/MM/yyyy”/>
  </restriction>
</simpleType>
```

■

3.3 String types

ToXgene comes with several predefined string generators, as listed in Table 2; the default string type is gibberish. String literals are defined by the following facets:

- `minLength`: defines the value for the minimum length allowed for instances of the type.
- `maxLength`: defines the value for the maximum length allowed for instances of the type. If necessary, instances are truncated to fit the maximum length specified.
- `pattern`: specifies a pattern to which the instances of the type have to conform. ToXgene uses this pattern when generating the instances of the type. ToXgene supports a subset of the pattern specification language in [1].
- **tox-string**: this is the main annotation for specifying string types, and has the following structure:

```
<!ELEMENT tox-string EMPTY>
<!ATTLIST tox-string
    type CDATA 'gibberish'
    minLength CDATA #IMPLIED
    maxLength CDATA #IMPLIED
    tox-distribution IDREF #IMPLIED>
```

- **type**: specifies the type of the strings to be generated; the default is gibberish. The basic string that come with ToXgene are listed in Table 2. User-defined string types can also be used (see Section 10).
- **minLength**: same as above; this one takes precedence;
- **maxLength**: same as above; this one takes precedence;
- **tox-distribution**: specifies a distribution for determining the length of instances of this type; an uniform distribution in the interval [`minLength`, `maxLength`] is used if no distribution is provided.

The default values for `minLength` and `maxLength` are 1 and 200, respectively.

Example 6. A named `simpleType` for string literals based on a pattern:

```
<simpleType name="isbn type">
  <restriction base="string">
    <pattern value="[0-9]{10}"/>
  </restriction>
</simpleType>
```

■

3.4 Constants

Constants are also instances of a `simpleType` and in ToXgene are declared using **tox-value** elements. Constants can be used for specifying the content of both elements and attributes.

```
<!ELEMENT tox-value (#CDATA)>
```

Example 7. Declaration of an integer constant:

```
<simpleType name="my_constant">
  <restriction base="integer">
    <tox-value>10</tox-value>
  </restriction>
</simpleType>
```

■

4 Specifying attributes and elements

In this section we discuss the specification of attributes and elements that have `simpleType` content. These are the building blocks for `complexType` specifications, covered in Section 5. The notation for specifying elements and attributes is as follows:

```
<!ELEMENT element ((simpleType|complexType|tox-expr*)?)>
<!ATTLIST element
    name CDATA #REQUIRED
    type CDATA #IMPLIED
    minOccurs CDATA '1'
    maxOccurs CDATA '1'
    tox-distribution IDREF #IMPLIED
    tox-recursionLevels IDREF #IMPLIED
    tox-reset (yes|no) 'no'
    tox-omitTag (yes|no) 'no'>

<!ELEMENT attribute (tox-expr|simpleType)>
<!ATTLIST attribute
    name CDATA #REQUIRED
    type CDATA #IMPLIED
    tox-minOccurs CDATA '1'
    tox-maxOccurs CDATA '1'
    separator CDATA #IMPLIED
    tox-distribution IDREF #IMPLIED>
```

Elements and attributes are specified by a name, a type, and cardinality constraints. The type of an element or attribute can be a named type (see previous section) or an anonymous type; elements can have either `simpleType`s or `complexType`s; attributes can have only `simpleType`s. The **tox-expr** annotation is discussed in Section 7.

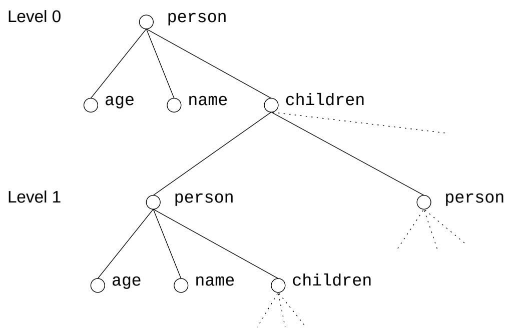
These are the attributes used in the definition of elements:

- `name`: specifies the name of the element.
- `type`: provides information about the type for the content of the element. If not specified, the element declaration must contain an anonymous type specification. When provided, it must be either a named type (simple or complex) previously declared, or a base type. When a a base type is specified, the content of the element is generated using the canonical specifications of the type (see Section 3).
- `minOccurs`: specifies the minimum number of occurrences for the element. Valid values are non-negative integers.
- `maxOccurs`: specifies the maximum number of occurrences for the element. Valid values are non-negative integers or `unbounded`.
- **tox-distribution**: specifies a distribution used to determine the actual number of occurrences of the element at generation time. If omitted, an uniform distribution in the interval $[\texttt{minOccurs}, \texttt{maxOccurs}]$ is used. Recall that each distribution is also defined in terms of a $[\texttt{min}, \texttt{max}]$ interval, which might conflict with the interval defined for the element. In such cases, the interval specified for the distribution takes precedence and warning messages are issued by ToXgene at template validation time.
- **tox-recursionLevels**: specifies a value distribution that governs the depth of the recursion tree when generating recursive content.
- **tox-reset**: when set to `yes`, the genes for all descendant elements and/or attributes are reset each time an instance of the element is produced. For example, resetting a **tox-number** specification for sequentially incremented numbers restarts the numeric sequence specified by the annotation.

As for attribute declarations, `name` and `type` have exactly the same use as above, thus we focus on the remaining attributes next. As far as XML Schema is concerned, an attribute is simply a $\langle \mathrm{name}, \mathrm{value} \rangle$

```
<element name="person" maxOccurs="500"
  tox-recursionLevels="r1">
  <complexType>
    <element name="name" type="name_type"/>
    <element name="age" type="age_type"/>
    ...
    <element name="children">
      <complexType>
        <element name="person" maxOccurs="5"/>
      </complexType>
    ...
  </complexType>
</element>
```

(a) Specification of a recursive gene.



(b) Structure of resulting XML elements.

Figure 2: Specification and instantiation of recursive element genes. Note the reference to distribution `"r1"` that determines how many recursive levels will be generated.

pair; the value of the attribute is a string, and can be a list of *words*. In ToXgene, each *word* is viewed as an "occurrence" of a simple type: an attribute whose value is a single string has 1 occurrence, while an attribute whose value is a list has many occurrences. The **tox-minOccurs**, **tox-maxOccurs** and **tox-distribution** are used to determine the number of occurrences for attributes, and behave similarly to the attributes `minOccurs`, `maxOccurs` and **tox-distribution** for element declarations. The `separator` attribute specifies which character is used to separate the *words* in a list-valued attribute; its default value is "`,`".

Example 8. Specification of elements using named and anonymous types:

```
<simpleType name="my_int">
  <restriction base="integer">
    <tox-number minInclusive="5" maxInclusive="100"/>
  </restriction>
</simpleType>
...
<element name="capacity" type="my_int"/>
<element name="age" type="nonNegativeInteger"/>
```

■

Example 9. Specifying anonymous types in attribute declarations:

```
<attribute name="address">
  <simpleType>
    <restriction base="string">
      <tox-string type=”gibberish” maxLength=”25”>
    </restriction>
  </simpleType>
</attribute>
...
<attribute name="currency">
  <simpleType>
    <restriction base="string">
      <tox-value>CDN</tox-value> <!-- a constant -->
    </restriction>
  </simpleType>
</attribute>
```

■

4.1 Specifying recursive elements

ToXgene also supports the generation of recursive XML content, determined by three parameters: the total number of (recursive) elements to be generated, the number of children per element, and the number of levels in the recursion. Figure 2 illustrates the generation of recursive content with ToXgene.

5 Specifying `complexType`s

Similarly to a `simpleType`, a `complexType` can be named or anonymous. Named types can be used to define the content of different elements in a template. The syntax for specifying complex types is as follows:

```
<!ELEMENT complexType ((tox-alternatives| tox-scan | tox-sample | tox-foreach |
                  tox-if |attribute | element | tox-string | tox-number |
                  tox-value | tox-expr)*)>
<!ATTLIST complexType
    name ID #REQUIRED
    mixed (true|false) ’false’>
```

As in [1], an instance of a `complexType` might contain a `CDATA` string, one or more elements, or a mix of elements and `CDATA` strings (in this case the attribute `mixed` has to be set to `true` in the corresponding type declaration). By definition, only elements with `complexType` can have attributes.

Regarding the notation for specifying a `complexType`, `element`, `attribute`, **tox-string**, **tox-number** and **tox-value** were covered already in previous sections; **tox-scan**, **tox-sample**, and **tox-expr**, which are used for specifying queries over previously defined content, are covered in 8; finally, **tox-if** and **tox-alternatives** define the conditional generation of content and are covered in Section 9.

5.1 Specifying elements with `complexType`

Similarly to simple types, complex types can be named or anonymous and named types can be used to define the content of multiple elements. The following examples summarizes the notation for specifying complex types.

Example 10. Specification of anonymous and named complex type. Note the anonymous complex type for the `price` element has mixed content, defined by the `currency` element and a **tox-number** specification:

```
<complexType name="my_book">
  <attribute name="isbn" type="isbn_type"/>
  <element name="title" type="string">
    <tox-string type="text"/>
  </element>
  <element name="price">
    <complexType mixed="true">
      <element name="currency">
        <simpleType>
          <restriction base="string">
            <tox-value>CDN</tox-value>
          </restriction>
        </simpleType>
      </element>
      <tox-number tox-distribution="c1" format="0.00"/>
    </complexType>
  </element>
</complexType>
...
<element name="book" type="my_book"/>
```

■

6 Specifying documents

A document specification is TSL consists of a name and an element declaration, which defines the root element of the document being generated. The **tox-document** annotation has the following properties:

```
<!ELEMENT tox-document (element)>
<!ATTLIST tox-document
    name CDATA #REQUIRED
    copies CDATA '1'
    starting-number CDATA '0'
    DTD-file CDATA #IMPLIED>
```

These are the attributes that affect the a **tox-document** specification:

- **name:** the name of the document. ToXgene automatically appends a `".xml"` extension to all documents it generates. Documents that have more than one copy (see below) are named by appending an unique integer identifier to the document name (e.g., `book00.xml`, `book01.xml`, etc.).
- **copies:** determines the number of documents to be generated by ToXgene using the same element specification. Each document will contain its own root element (a different instance of the element defining the document). This feature allows the emulation of situations in which the database consists of multiple (similar) documents, each containing a single object, as opposed to having a single document containing all objects in the database. By default, copies are numbered from 0 to the number specified.
- **starting-number:** specifies the initial value for numbering the documents.
- **DTD-file:** specifies a value that is added to the `DOCTYPE` declaration of each document generated.

Example 11. Specification of a document for a book catalog, containing from 100 to 200 instances of the `my_book` type:

```
<tox-document name="books">
  <element name="books">
    <complexType>
      <element name="book" type="my_book" minQtty="100" maxQtty="200"/>
    </element>
  </complexType>
</tox-document>
```

■

7 Specifying expressions

Expressions are the base for defining queries, some simple computations and integrity constraints over the data being generated. The following grammar sketch describes the valid expressions in a template:

```
         EXP : ATOM | EXP OP EXP | (EXP)
        ATOM : CONSTANT | SIMPLE_TYPE | QUERY
          OP : + | - | * | / | % | #
    CONSTANT : string | date | int | float
 SIMPLE_TYPE : ~name
       QUERY : [path_expression] | AGG[path_expression]
         AGG : MIN | MAX | COUNT | SUM | AVG | LEN | CONCAT
```

We now discuss each of the terminal symbols.

- **string:** a CDATA literal enclosed by single quote marks (e.g., `'maple leafs'`);
- **date:** a date, enclosed by single quote marks, in the YYYY-MM-DD format (e.g., `'1998-06-28'`);
- **float:** a floating point number, written using the format `#0.0#` (e.g., `123.45`);
- **int:** an integer number, written using the format `#0` (e.g., `123`);
- **name:** a name of a named simple type declared in the template);
- **path_expression:** a path used for navigating elements in queries (see Section 8).

The operators allowed in an expression are the usual arithmetic operators (`+`, `-`, `*` and `/`), the modulo operator (`%`), and the the string concatenation operator (`#`). Expressions in ToXgene *are* typed. Table 3 shows the resulting types for valid expressions and the cases where automatic type casting is done.

The result of a path expression might be an atomic value (of any of the types supported) or a list of atomic values (see Section 8 for details on how queries are written and evaluated). ToXgene also offers the most common aggregate operations on the results of path expressions, as shown in Table 4.

Expressions are evaluated in the usual left to right manner; the order of evaluation of expressions is as follows: queries and named type references; multiplication, division and modulo; all other operators. Parenthesis can be used to override the default precedence. A named type reference evaluates to an instance of the named simple type.

Example 12. The following example shows how to prefix a sequentially incremented numeric values with a string:

operand 1	operand 2	operator	type
int	int	+,-,*,/,%	int
float	float	+,-,*,/	float
int	float	+,-,*,/	float *
float	int	+,-,*,/	float *
date	int	+,-	date
int	date	+,-	date
ANY	ANY	#	string *[a]

[a]No warning is issued if both operands are of type string.

Table 3: Resulting types for valid operations, according to the operands in ToXgene. Lines marked with a star indicate situations where automatic type casting is done by ToXgene, a warning message at template validation time is issued in such cases. Any other combination of operands and operators result in error. ANY represents any type.

name	Input		Output	
	list/atom	type	description	type
MIN	list	int,float,date	returns the smallest value in the list	unch
MAX	list	int,float,date	returns the largest value in the list	unch
COUNT	list	ANY	returns the number of elements in the list	int
SUM	list	int,float	returns the sum of all values in the list	unch
AVG	list	int,float	returns the average of the values in the list	float
LEN	atom	ANY	returns the length in characters of the operand	int
CONCAT	list	ANY	returns the concatenation of all items in the list, treated as strings	string

Table 4: Aggregate operators in ToXgene.

```
<simpleType name="key">
  <restriction base="positiveInteger">
    <tox-number sequential=”yes”/>
  </restriction>
</simpleType>
...
<element name="book_id">
  <tox-expr value=”’book’#~key”/>
</element>
```

■

7.1 Debugging expression executions

ToXgene offers some debugging information that might help understanding the parsing of expressions when executed in “verbose” mode (-v option in the command line). We recommend you use this mode when writing templates, to make sure your expressions get evaluated the way you expect them to be.

Here is an example of the output of ToXgene in verbose mode:

```
...
Processing tox-expr for:  [a]+([b]+[c])*[c]/[d]
parsed expression:  ([a]+(([b]+[c])*([c]/[d])))
...
```

The following is a fragment of ToXgene's output in verbose mode when processing the `make_orders` template in the TPCH sample:

```
...
Processing tox-expr for:  [PartKey]
parsed expression:  [PartKey]
Processing tox-expr for:  (([PartKey]+([i]*(250+([PartKey]-1)/1000)))%1000)+1
parsed expression:  ((([PartKey]+([i]*(250+(([PartKey]-1)/1000))))%1000)+1)
...
```

8 Specifying lists and queries

In many situations one needs to generate correlated content. For instance, one might want to generate a catalog of books (uniquely identified by their ISBN values) and a collection of reader reviews about these books. Evidently, each review has to refer to an existing book (e.g., include an ISBN value present in the catalog). This is called *element sharing* (the content of some elements is shared with others). The ToXgene approach for element sharing is to first generate all data that will be shared in a "temporary memory" called a **tox-list**, and query this list whenever necessary.

A list declaration is similar to a document declaration in the sense that both include an element specification that is used to generate their content (the document itself or the items in the list). The actual number of elements in a list is dictated by the occurrence pattern for the element declared inside it (i.e., the element's distribution or `minOccurs`, `maxOccurs` values). The following DTD fragment describes the syntax for declaring lists:

```
<!ELEMENT tox-list (element)>
<!ATTLIST tox-list
    name ID #REQUIRED
    unique CDATA #IMPLIED
    where CDATA #IMPLIED
    readFrom CDATA #IMPLIED
    abort (yes|no) 'no'
    dump (yes|no) 'no'>
```

- **name:** uniquely identifies the list within a template;
- **unique:** specifies uniqueness constraints for the elements in the list; see details below;
- **where:** specifies more elaborate integrity constraints over the elements in the list; see details below;
- **readFrom:** specifies a file from which the elements in the list are read;
- **abort:** determines whether the template processing is aborted when elements read from a list violate any of the integrity constraints specified for the list;
- **dump:** determines whether the list should be written to a file for later reuse.

Example 13. Declaration of a list with 100 book "records". Each record contains a unique ISBN and a title of a book:

```
 <tox-list name="book_list" unique="book/isbn">
  <element name="book_rec" minOccurs="100" maxOccurs="100">
    <complexType>
      <element name="isbn" type="isbn_type"/>
      <element name="title">
        <simpleType>
          <restriction base="string">
            <tox-string type="text"/>
          </restriction>
        </simpleType>
      </element>
    </complexType>
  </element>
</tox-list>
```

■

For the moment we discuss how the elements in a list are generated by ToXgene. Section 8.4 describes how a list can be read from a file. Each element in the list is generated independently and validated against the list's integrity constraints; if the element violates any of the constraints it is discarded. This process is repeated until all elements in the list are generated. For instance, for the list specified in Example 13, each time a `book_rec` element is generated, its `isbn` value is compared to those for the books already in the list; if this `isbn` value was assigned to an existing book, the element is discarded; otherwise, the element is added to the list. Of course, the frequency of repeated `isbn` values depends on the specification of the `isbn_type` and the number of instances in the list.

8.1 Querying lists

Queries in ToXgene are specified using cursors and **tox-expr** annotations. Each cursor specifies a *path expression*, relative to a list or to another cursor, which determines a list of elements over which the cursor iterates. As mentioned before, queries are expression, thus declared in **tox-expr** annotations. Each query is bound to a cursor, thus **tox-expr** elements that define queries have to be descendants of some element defining a cursor. By default, a query is bound to its closest ancestor cursor. Cursors in ToXgene work in the usual "one-element-at-a-time" mode: each query returns (possibly a set of) values that are descendants of the current element of the cursor only.

A path expression is a sequence of *labels* separated by '`/`'. A label is one of a list name, an element name, a variable name (referring to a named cursor) or '`!`' (shorthand for `CDATA`). All path expressions in ToXgene are enclosed by square brackets. For example, the path expression `[book_list/book_rec/isbn/!]` returns all isbn values of all book records in `book_list`; `[$a/isbn/!]` returns the isbn of the current element in the cursor named `a` (see example below). All path expressions in ToXgene are *fully* specified: no wildcards are allowed.

There are two types of cursors in ToXgene, differing on the way they access their contents, declared by the following annotations:

```
<!ELEMENT tox-scan ((tox-scan|tox-sample|tox-foreach|tox-expr|attribute|element|
              tox-if|tox-alternatives)*)>
<!ATTLIST tox-scan
    path CDATA #REQUIRED
    name CDATA #IMPLIED
    where CDATA #IMPLIED>

<!ELEMENT tox-sample (((tox-scan|tox-sample|tox-foreach|tox-expr)*)|
              (attribute|element)*)>
<!ATTLIST tox-sample
    path CDATA #REQUIRED
    name CDATA #IMPLIED
    where CDATA #IMPLIED
    tox-distribution IDREF #IMPLIED
    duplicates (yes|no) 'yes'>

<!ELEMENT tox-foreach ((tox-scan|tox-sample|tox-foreach|tox-expr|attribute|element|
              tox-if|tox-alternatives)*)>
<!ATTLIST tox-foreach
    path CDATA #REQUIRED
    name CDATA #IMPLIED
    where CDATA #IMPLIED>
```

- **tox-scan:** specifies a cursor for a sequential scan over a list of elements resulting from evaluating a path expression. The following attributes are used for defining the cursor:
  - **path:** specifies a path expression relative to a list or to an ancestor cursor (see example of nested cursors below) from where the elements of the cursor are extracted;
  - **name:** necessary only when declaring nested cursors within the one being specified (see example of nested cursors below);
  - **where:** specifies conditions for the selection of the elements resulting from evaluating the path expression (see example below).

  A cursor is populated as follows: the path expression is evaluated returning a list of elements (that match the path expression); each element is checked against the conditions in the **where** clause, and those that do not qualify are discarded.
- **tox-sample:** specifies a cursor for sampling from a list of elements resulting from evaluating a path expression. All attributes common to **tox-scan** and **tox-sample** cursors behave in the same way as above (**path**, **name** and **where**). The following attributes are exclusive to **tox-sample** cursors:
  - **tox-distribution:** specifies a distribution in the interval $[0, 100]$, that determines which element of the list is sampled next;
  - **duplicates:** determines whether the same element can be sampled more than once.
- **tox-foreach:** specifies a cursor that is instantiated for each element that matches the given path expression. This cursor is especially useful for producing complex types.

Example 14. The element definition below specifies a query over the `book_list`, defined in Example 13. Note that one can define new random content within a cursor (as in the `author` element specification):

```
 <tox-document name="books">
...
  <element name="book" minOccurs="100">
    <complexType>
      <tox-scan path="[book_list/book_rec]" name="a">
        <attribute name="isbn" type="isbn_type">
          <tox-expr value="[$a/isbn/!]"/>
        </attribute>
        <element name="title" type="string">
          <tox-expr value="[title/!]"/>
        </element>
        <element name="author" maxOccurs="5" type="string">
          <tox-string type="gibberish" maxLength="30"/>
        </element>
      </tox-scan>
    </complexType>
  </element>
...
</tox-document>
```

■

The example above shows how path expression in queries bind to cursors. One can define the path relative to a cursor name, as in **[$a/isbn/!]**. However, explicit naming is required only when declaring nested cursors. By default, each path expression binds to its closest ancestor cursor, thus in the example above, the expressions **[title/!]** and **[$a/title/!]** are equivalent.

Observe that the ISBN value of each book is declared as an element in the `book_list` and as attributes in the `book` element. As shown in the example above, both elements and attributes in a document declaration can include queries. However, list records cannot have attributes: we stress the fact that lists are meant for temporary storage of data, for querying purposes only, and element declarations alone suffice for this purpose.

Example 15. Nested queries. The following example illustrates the specification of queries within list definitions (note the definition of the `authors` element in the new `book_list`) and of nested queries:

```
 <tox-list name=”book_list” unique=”book/isbn”>
  <element name="book_rec" minOccurs="100" maxOccurs="100">
    <element name="isbn" type="isbn_type"/>
    <element name="title">
        <simpleType>
          <restriction base="string">
            <tox-string type=”text”/>
          </restriction>
        </simpleType>
      </element>
    <element name="author_id" type="integer" maxOccurs="5">
      <tox-sample path=”[author_list/author]” duplicates=”no”>
        <tox-expr value=”[id/!]”/>
      </tox-sample>
    </element>
  </element>
</tox-list>
...
<tox-document name=”books”>
...
  <element name="book" minOccurs="100">
    <complexType>
      <tox-scan path=”[book_list/book_rec]” name=”a”>
        <attribute name="isbn" type="isbn_type">
          <tox-expr value=”[isbn/!]”/>
        </attribute>
        <element name="title" type="string">
          <tox-expr value=”[title/!]”/>
        </element>
        <attribute name="authors" type="IDREFS" maxOccurs="unbounded">
          <tox-scan path=”[$a/author_id]”>
            <tox-expr value=”’author’#[!]”/>
          </tox-scan>
        </attribute>
      </tox-scan>
    </complexType>
  </element>
...
</tox-document>
```

■

A few words are worth mentioning here on how the specifications in Example 15 are processed, especially with respect to the `authors` attribute in the `book` element. First, note that the “records” in `book_list` now contain a random number (up to 5) of author identifiers, which are sampled without repetition from another list. Assume that these identifiers are declared to be unique within that list, so we get up to 5 distinct author id’s in each record in `book_list`. As in Example 14, the specification of the `book` element defines a sequential scan over the records in `book_list`, copying both the ISBN and the title in each record. Here, however, we define the `authors` attribute, containing all author identifiers in each book record. This can be done by declaring a nested cursor over the `author_id` elements within each `book_rec` in the list. Note the reference to the outer cursor is made by defining a path expression relative to its name (`$a` in the example). Also, note that `maxOccurs` is set to `unbounded` for the `authors` attribute, meaning that there will be as many items in the list as there items matching `[$a/author_id]` (i.e., author identifiers in the current book record of the outer cursor).

8.2 Specification of `ID`, `IDREF` and `IDREFS` attributes

As illustrated in Example 15, one can use lists with uniqueness constraints and queries to produce attributes of type `ID`, `IDREF` and `IDREFS`. The same mechanisms can be used to enforce “referential integrity” among documents as well. For a complete example, consider the catalog sample, which produces a document containing a book catalog with 500 books and 1000 review documents, each referring to an existing book.

8.3 Where clauses

Other integrity constraints can be specified when generating a list via **where** clauses. Each clause defines multiple predicates involving queries that are checked whenever a new element is generated (or read from a file). Where clauses are used also for specifying selection criteria in cursors, and the same notation applies in both cases:

$$\begin{aligned}
\mathsf{CLAUSE} &: \mathsf{PREDICATE}(\mathsf{QUAL\,EXP}, \mathsf{QUAL\,EXP})\\
\mathsf{PREDICATE} &: \mathtt{EQ} \mid \mathtt{DIF} \mid \mathtt{LT} \mid \mathtt{LEQ} \mid \mathtt{GT} \mid \mathtt{GEQ}\\
\mathsf{QUAL} &: \mathtt{ALL} \mid \mathtt{ANY} \mid \lambda\\
\mathsf{EXP} &: \mathsf{ATOM} \mid \mathsf{EXP\ OP\ EXP} \mid (\mathsf{EXP})\\
\mathsf{ATOM} &: \mathsf{CONSTANT} \mid \mathsf{SIMPLE\_TYPE} \mid \mathsf{QUERY}\\
\mathsf{OP} &: + \mid - \mid * \mid / \mid \% \mid \#\\
\mathsf{CONSTANT} &: \mathbf{string} \mid \mathbf{date} \mid \mathbf{int} \mid \mathbf{float}\\
\mathsf{SIMPLE\_TYPE} &: \sim\!\mathbf{name}\\
\mathsf{QUERY} &: [\mathbf{path\_expression}] \mid \mathsf{AGG}[\mathbf{path\_expression}]\\
\mathsf{AGG} &: \mathtt{MIN} \mid \mathtt{MAX} \mid \mathtt{COUNT} \mid \mathtt{SUM} \mid \mathtt{AVG} \mid \mathtt{LEN} \mid \mathtt{CONCAT}
\end{aligned}$$

The λ symbol above represents the empty string, thus makes the qualifier optional. The definition for EXP and all rules that derive from it are identical to the ones specified in Section 7, we repeat them here for the convenience of the reader. The predicates EQ, DIF, LT, LEQ, GT, GEQ correspond to $=, \neq, <, \leq, >$ and $\geq$, respectively[2].

Example 16. The following list contains sequentially incremented integer values, used as keys for the order relation in the TPC-H database. The where clause in the list specifies that only the first 8 numbers from each group of 32 (i.e., $1, \ldots, 8, 33, \ldots, 40, \ldots$) are used, as required by the TPC-H specification:

```
<tox-list name="order_keys0" where="LT([OrderKey]%32,8)">
  <element name="OrderKey" minOccurs="1000" maxOccurs="1000">
    <simpleType>
      <restriction base="nonNegativeInteger">
        <maxInclusive value="1000"/>
        <tox-number sequential="yes"/>
      </restriction>
    </simpleType>
  </element>
</tox-list>
```

■

Example 17. Join of lists. The following example shows how nested cursors can be used to implement a join operation between lists `L1` and `L2`. Note that the **where** clause in the outer cursor defines a selection criterion (`L1/a/x>100`), while in the inner cursor the **where** clause defines the join condition (`L2/b/y=L1/a/x`):

[2]We use the prefix notation simply because one cannot have angle brackets inside attributes in XML documents (remember ToXgene templates are XML documents).

```
<element name="e1" maxOccurs="unbounded">
  <complexType>
    <tox-scan path="[L1/a]" where="GT([x],100)" name="a">
      <element name="x" type="integer">
        <tox-expr value="[x]"/>
      </element>
      <element name="y" maxOccurs="unbounded">
        <simpleType>
          <restriction base="interger">
            <tox-scan path="[L2/b]" where="EQ([y],[$a/x])">
              <tox-query expr="[y]"/>
            </tox-scan>
          </restriction>
        </simpleType>
      </element>
    </tox-scan>
  </complexType>
</element>
```

■

8.4 Reading lists from files

As mentioned earlier, lists can be “dumped” into files for later reuse (by setting the **dump** attribute to yes on the list declaration). For convenience, the lists are written to XML documents. We stress that although dumping a list to a file produces an XML document, this is not the preferred method for generating documents in ToXgene. The **tox-document** declaration provides more resources for this task such as the ability of generating attributes, to cite just one. Moreover, ToXgene implements a simple garbage collection mechanism which enforces that only lists that are used (directly or indirectly) in a document declaration are populated. Thus, no data at all is generated for templates that do not declare documents. ToXgene’s garbage collector also ensures that the memory used by a list is freed as soon as all queries defined over it are executed.

Evidently, lists can be populated with the contents of XML documents, not necessarily generated by our tool. This means that one can reuse previously generated content, say for growing an existing collection of documents, as well as using real data in the document generation process. One typical example is the use of real country names in documents.

There are a few important things to mention here. First, an element specification, describing the type of the “records” being read has to be declared. The validity checking of path expressions and type inferencing in ToXgene are done before any data generation takes place, thus the element declaration is required for verifying the integrity of the expressions and queries in the template. We note that *only* the elements of interest need to be declared (i.e., the document might contain more elements than those in the list declaration). Second, only elements in the input XML document are considered; attributes are ignored. Finally, integrity constraints over lists read from files are declared in the same way as for lists being generated by ToXgene. List “records” that violate any constraint are discarded; the **abort** attribute indicates whether ToXgene should abort the generation process should a list record read violate an integrity constraints defined for the respective list. At template processing time, ToXgene outputs the actual number of elements that were generated or read for each list:

```
ToXgene V2.1 - (c) 2001 by University of Toronto and IBM Corporation

***** Parsing template:  Done!

Generating 1500 elements in order_keys0:  Done!
Generating 1500 elements in order_keys1:  Done!
Reading list Parts from temp/orders_input.xml:  Done!  2000 elements read.
Generating 1500 elements in order_list0:  Done!
Generating 1500 elements in order_list1:  Done!
Generating 1500 elements in order_list2:  Done!
Reading list priorities_list from input/priorities.xml:  Done!  5 elements read.
Reading list ship_mode_list from input/ship_modes.xml:  Done!  7 elements read.
Reading list ship_instructions_list from input/ship_instructions.xml:  Done!
4 elements read.
```

All ToXgene sample templates include lists that are read from XML documents.

9 Specifying random structures

In this section we describe two useful annotations, used for introducing some controlled randomness in the data generation process in ToXgene:

```
<!ELEMENT tox-if (tox-then, tox-else?)>
<!ATTLIST tox-if
    expr CDATA #REQUIRED>
<!ELEMENT tox-then (tox-alternatives|tox-scan|tox-sample|tox-foreach|tox-if|
                element|attribute)*>
<!ELEMENT tox-else (tox-alternatives|tox-scan|tox-sample|tox-foreach|tox-if|
                element|attribute)*>

<!ELEMENT tox-alternatives (tox-option*)>
<!ATTLIST tox-alternatives
    tox-distribution IDREF #IMPLIED>
<!ELEMENT tox-option ((attribute|element|tox-if|tox-alternatives|tox-expr|
              tox-string)*)> <!ATTLIST tox-option odds CDATA #REQUIRED>
```

- **tox-if, tox-then, tox-else:** these annotations provide IF-THEN-ELSE blocks, which can be nested. The decision on which branch to take is made based on the logical value of the expression defined by the **expr** attribute in the IF statement. Such expression is declared in the same way as a integrity constraint (see Section 8.3). The ELSE block is optional.

- **tox-alternatives, tox-option:** these annotations provide a way to randomly choose one processing path among various options. Each **tox-option** has a content descriptor (e.g., an element declaration) and the odds with which that option will be chosen as follows. Let $p_1, p_2, \ldots, p_n$ be the odds for n **tox-option** elements (an error occurs if $\sum_i p_i > 100$). The $p_1, \ldots, p_n$ values yield the following probability ranges:

$$b_i = \begin{cases} [0, p_i], & i = 1 \\ [\sum_{j=0}^{i-1} p_j, \sum_{j=0}^{i-1} p_j + p_i], & 1 < i < n \\ [\sum_{j=0}^{i-1} p_j, 100], & i = n \end{cases}$$

  For Example, suppose the odds are 30,30,20 (no need to sum up to 100); the brackets will be $[0, 30]$; $[30, 60]$; $[60, 100]$ (the last option is always assigned all probability mass left). The processing of a **tox-alternatives** consists then of simply generating a random number according to the given probability distribution and determining in which bracket this number falls, and thus, which option to process. The content of the options need not be related in any way: one can declare completely different structures and/or genes for describing the XML content to be generated. Also, a **tox-option** might be empty, thus allowing the specification of optional content (see example below).

Example 18. Alternative ways of specifying optional content in ToXgene:

```
<!-- the first method uses a random number generator and an IF-THEN-ELSE construct-->
<simpleType name="rand">
  <restriction base="nonNegativeInteger">
    <maxInclusive="100"/>
  </restriction>
</simpleType>
...
  <tox-if expr="GT(~rand,50)">
    <element name="my_element" type="my_type"/>
  </tox-if/>
...
<!-- the second method uses alternatives/options -->
  <tox-alternatives>
    <tox-option odds="50">
      <element name="my_element" type="my_type"/>
    </tox-option>
    <tox-option odds="50"/>
  </tox-alternatives>
```

■

10 Using user-defined CDATA generators

ToXgene can be extended to allow the use of other CDATA generators than the ones that come with it by default. This section explains how this can be done.

All ToXgene's CDATA generators must implement the following interface:

```
package genes.literals;

public interface Cdata{
  public void setRandomSeed(int seed);
  public String getCdata(int length);
}
```

The first method gives a random seed that should be used by the CDATA generator. This aims at allowing the generation of the exact same content when the same master random seed is specified (recall the `-s` v command line option).

The second method should return a string literal to be used in the data that is output by ToXgene. If `length` is positive, the string to be returned must be exactly `length` characters long. If `length` is -1, the CDATA generator can return a string of any length. This last mode is useful when the user wants to bypass the specification of `minLenght` and `maxLength` values and, thus, avoid truncating strings.

For example, suppose we provide a CDATA generator for email addresses that outputs strings of minimum lenght 5 and maximum length 20. Suppose we use this CDATA generator in a template which specifies `minLength` and `maxLength` to these values. When processing this template, ToXgene will set `length` to a number uniformly distributed between 5 and 20, say 10, and pass it to the CDATA generator. However, it could be the case that the email address generated at that iteration has more than 10 characters and must be truncated.

All registered string types in ToXgene are listed in a file called `setup/cdata.cfg` which is bundled in the `toxgene.jar` file. Each line in this file associates a CDATA type name to a given Java class, and default values for `minLength` and `maxLength` for that type. By default, this file contains the following entries:

```
text;util.cdata.tpch.Text;1;200
xmrk_text;util.cdata.xmark.Text;1;200
email;util.cdata.xmark.Emails;-1;-1
fname;util.cdata.xmark.FirstNames;-1;-1
lname;util.cdata.xmark.LastNames;-1;-1
city;util.cdata.xmark.Cities;-1;-1
country;util.cdata.xmark.Countries;-1;-1
province;util.cdata.xmark.Provinces;-1;-1
domain;util.cdata.xmark.InetDomains;-1;-1
word;util.cdata.xmark.Names;-1;-1
```

Note the -1's specified as `minLength` and `maxLength` for most of the types. For these types, ToXgene will pass -1 as `lenght` for the respective CDATA generators.

When a given string type is referenced in a template, ToXgene will try to load the associated class and immediately after that call `setRandomSeed`. An error will occur if the specified class cannot be found by the Java class loader. One important note here is that ToXgene will create only 1 Cdata generator of each type, thus minimizing the amount of memory used. The gibberish string generator is integrated into ToXgene's code directly.

For convenience, we provide the source code of all default CDATA generators that come with ToXgene in the `src` directory.

11 Using a persistent object manager

ToXgene can make use a Persistent Object Manager (POM) the "temporary" data that must be kept in **tox-list** elements is too large to fit in main memory. As discussed earlier, this feature is activated by passing the `-o [`p` [`f` [`s`]]]` command line option when invoking ToXgene.

The optional parameters p,f and s specify: the path where to store the files containing the temporary data; the fraction of the memory that should be used for buffering; and the size in kilobytes of the buffer pages, respectively. The default values for these parameters are "." (i.e., the files will be placed in the directory where ToXgene is being executed), 50%, and 8KB.

All files used by the POM are placed in the directory specified by p (the default value of p is the directory from where you invoke ToXgene. All such files are called `tg_file_`x, where x is an identifier used by the POM. The value of p is completely irrelevant to ToXgene itself. Thus, one can specify a directory residing in a different file system and take advantage of parallel I/O.

Note that using a POM has several performance implications. Note that ToXgene has no control on how the POM does memory management. In fact, ToXgene competes with the POM for memory for generating new random data and for processing queries. Also, accessing data via a POM requires expensive I/O operations. On top of that, there is the overhead imposed by the POM bookeeping operations as well. The f and s parameters allow the user to adjust the working of the POM in two ways. First, the closer f is to 1, the more buffering will be effective. However, high values of f mean that ToXgene has less "working memory" and thus might not be able to process the template without swapping (done automatically by the Java virtual machine). Similarly, higher s values mean the POM will perform I/O over larger chunks of data.

For queries that perform sequential access to data items (see Section 8), this will mean fewer cache misses due to locality of reference.

ToXgene tries to use memory and execute queries as efficiently as possible. However, the optimal values for f and s are, of course, template dependent. Some “guidelines” for choosing these values are. If your template does mostly sequential scans and performs “simple” queries, use high values for f and s; otherwise, experiment with lower values. We also note that, whenever possible, allowing some extra memory to the Java virtual machine and not using the POM will yield much better overall performance.

Note on uniqueness constraints. As discussed in Section 8, one can specify uniqueness constraints over elements in the lists. Therefore, during the generation of the list, ToXgene keeps an index that contains the keys that have been used. We note that, even when using a POM, this index is kept in primary memory, due to high I/O costs of dynamically maintaining an index on secondary memory. This also applies for when a list is read from a file.

Of cours, one way of generating keys for datasets that do not fit in memory without the uniqueness constraint is by using sequentially incremented numbers. In fact, this is the way used by most benchmarks. In doing so, one saves both memory, by avoiding the materialization of a key index, and also time, since uniqueness is guaranteed by construction.

References

[1] D. C. Fallside. XML schema part 0: Primer - W3C candidate recommendation. Available at `http://www.w3.org/TR/xmlschema-0/`, October 24 2000.

[2] Java SDK - `DateFormat` class documentation. Available at `http://java.sun.com/j2se/1.3/docs/api/java/text/DateFormat.html`.

[3] Java SDK - `DecimalFormat` class documentation. Available at `http://java.sun.com/j2se/1.3/docs/api/java/text/DecimalFormat.html`.

[4] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, Amsterdam, The Netherlands, April 2001.

[5] Transaction Processing Performance Council. *TPC Benchmark H - Decision Support*, 1999. Revision 1.3.0.

Appendix

A Credits and contact information

ToXgene is the data generator for the ToX project, developed at the University of Toronto. This tool was developed by Denilson Barbosa under the supervision of Alberto Mendelzon, both from the University of Toronto, in cooperation with John Keenleyside and Kelly Lyons, from the IBM Toronto Lab.

This work was partly funded by the Department of Computer Science of the University of Toronto, the Natural Sciences and Engineering Research Council of Canada, Bell University Laboratories, IBM Corporation, the IBM Centre for Advanced Studies and the IRIS Network of Centres of Excellence.

Acknowledgments. Of course ToXgene would not have been possible without the use of other people's code. For copyright (and gratitude) reasons, we state that:

A.1 Other ToXgene resources

- ToXgene websites:

  `http://www.cs.toronto.edu/tox/toxgene/`
  `http://www.alphaworks.ibm.com/tech/toxgene`

- ToXgene discussion forum:

  `http://www.alphaworks.ibm.com/forum/toxgene.nsf/main`

- ToXgene related publications:

  – Denilson Barbosa, Alberto Mendelzon, John Keenleyside and Kelly Lyons. **ToXgene: a template-based data generator for XML**. In *Proceedings of the Fifth International Workshop on the Web and Databases* (WebDB 2002). Madison, Wisconsin - June 6-7, 2002.

  – Denilson Barbosa, Alberto Mendelzon, John Keenleyside and Kelly Lyons. **ToXgene: A template-based data generator for XML**. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* – Demo program. Madison, Wisconsin - June 4-6, 2002.

Please report bugs by email to `toxgene-bugs@cs.toronto.edu`

B A DTD for TSL

The following DTD is used to validate the templates by ToXgene.

```
<!ELEMENT tox-template (tox-distribution|simpleType|complexType|tox-list|
                        tox-document)*>

 <!-- ***********************************************************
      XML Schema definitions needed
      *********************************************************** -->

 <!-- Just a copy of what’s in the XML Schema specifications -->

 <!ELEMENT simpleType (restriction)>
 <!ATTLIST simpleType name CDATA #IMPLIED>

 <!-- tox-string, tox-number and tox-value are all "literals", thus can go
 inside a restriction definition -->

 <!ELEMENT restriction ((tox-value | pattern) |
                        ((minLength |maxLength)*,tox-string) |
                        ((minInclusive | maxInclusive | minExclusive |
                          maxExclusive | tox-format)*, (tox-number|tox-date)?) |
                        (tox-scan|tox-sample|tox-foreach))>

 <!-- Here we have ALL 45 XML Schema datatypes! -->
 <!ATTLIST restriction base (string|integer|long|unsignedLong|int|unsignedInt|
                             short|byte|nonNegativeInteger|positiveInteger|
                             nonPositiveInteger|negativeInteger|decimal|
                             float|double|date) #REQUIRED>

 <!ELEMENT pattern EMPTY>
 <!ATTLIST pattern value CDATA #REQUIRED>

 <!ELEMENT minInclusive EMPTY>
 <!ATTLIST minInclusive value CDATA #REQUIRED>

 <!ELEMENT maxInclusive EMPTY>
 <!ATTLIST maxInclusive value CDATA #REQUIRED>

 <!ELEMENT minExclusive EMPTY>
 <!ATTLIST minExclusive value CDATA #REQUIRED>

 <!ELEMENT maxExclusive EMPTY>
 <!ATTLIST maxExclusive value CDATA #REQUIRED>

 <!ELEMENT minLength EMPTY>
 <!ATTLIST minLength value CDATA #REQUIRED>

 <!ELEMENT maxLength EMPTY>
 <!ATTLIST maxLength value CDATA #REQUIRED>

 <!ELEMENT tox-format EMPTY>
 <!ATTLIST tox-format value CDATA #REQUIRED>

 <!-- The tox-percent gives the probability of that particular element being
 chose from the list -->

 <!ELEMENT enumeration EMPTY>
 <!ATTLIST enumeration value CDATA #REQUIRED
                       tox-percent CDATA #REQUIRED>

 <!-- This is an simplified version of the XML Schema definition of element,
 but this will do for our needs -->

 <!ELEMENT complexType ((tox-alternatives|tox-scan|tox-sample|tox-foreach|
                         tox-if|attribute|element|tox-string|tox-number|
                         tox-value|tox-expr)*)>
 <!ATTLIST complexType name CDATA #IMPLIED
                       mixed (true|false) ’false’>

 <!-- These elements are like the XML Schema elements, and are used to define
 data types and tox-files -->

 <!ELEMENT element ((simpleType|complexType|tox-expr*)?)>
 <!ATTLIST element name CDATA #REQUIRED
                   type CDATA #IMPLIED
                   minOccurs CDATA ’1’
                   maxOccurs CDATA ’1’
                   tox-distribution IDREF #IMPLIED
                   tox-recursionLevels CDATA #IMPLIED
                   tox-reset (yes|no) ’no’
                   tox-omitTag (yes|no) ’no’>

 <!ELEMENT attribute (tox-expr|simpleType)>
 <!ATTLIST attribute name CDATA #REQUIRED
                     type CDATA #IMPLIED
                     tox-minOccurs CDATA ’1’
                     tox-maxOccurs CDATA ’1’
                     tox-distribution IDREF #IMPLIED
                     separator CDATA #IMPLIED>

 <!-- ***************************************************************
      ToXgene specific elements
      *************************************************************** -->

 <!-- elements for defining skewed value distributions -->

 <!ELEMENT tox-distribution (enumeration*)>
 <!ATTLIST tox-distribution name ID #REQUIRED
                            type (constant | exponential | geometric |
                            lognormal | normal | uniform | user-defined)
                            ’uniform’
                            minInclusive CDATA #REQUIRED
                            maxInclusive CDATA #REQUIRED
                            mean CDATA #IMPLIED
                            variance CDATA #IMPLIED>

 <!-- These define string literals -->

 <!ELEMENT tox-string EMPTY>
 <!ATTLIST tox-string type CDATA ’gibberish’
                      minLength CDATA #IMPLIED
                      maxLength CDATA #IMPLIED
                      tox-distribution IDREF #IMPLIED>

 <!-- These define numeric literals. This element will be extended in the
 future to add things like different formatting for the numbers, according
 to the XML Schema definitions for numeric datatypes. -->

 <!ELEMENT tox-number EMPTY>
 <!ATTLIST tox-number minInclusive CDATA #IMPLIED
                      maxInclusive CDATA #IMPLIED
                      tox-distribution IDREF #IMPLIED
                      sequential (yes|no) ’no’
                      format CDATA #IMPLIED>

 <!-- These elements define date literals -->
 <!ELEMENT tox-date EMPTY>
 <!ATTLIST tox-date start-date CDATA #REQUIRED
                    end-date CDATA #REQUIRED
                    tox-distribution IDREF #IMPLIED
                    format CDATA #IMPLIED>
```

```
<!-- The tox-value element is used to specify user-defined constants -->
<!ELEMENT tox-value (#PCDATA)>

<!-- A tox list is a repository for literals that are shared by the files
in the template -->

<!ELEMENT tox-list (element)>
<!ATTLIST tox-list name ID #REQUIRED
                   unique CDATA #IMPLIED
                   where CDATA #IMPLIED
                   readFrom CDATA #IMPLIED
                   abort (yes|no) ’no’
                   dump (yes|no) ’no’>

<!ELEMENT tox-alternatives (tox-option*)>
<!ATTLIST tox-alternatives tox-distribution IDREF #IMPLIED>

<!ELEMENT tox-option ((attribute|element|tox-if|tox-alternatives|tox-expr|
                       tox-string)*)>
<!ATTLIST tox-option odds CDATA #REQUIRED>

<!-- These elements implement ToXgene’s query language -->

<!ELEMENT tox-scan ((tox-scan|tox-sample|tox-foreach|tox-expr|attribute|element|
                     tox-if|tox-alternatives)*)>
<!ATTLIST tox-scan path CDATA #REQUIRED
                   name CDATA #IMPLIED
                   where CDATA #IMPLIED>

<!ELEMENT tox-sample (((tox-scan|tox-sample|tox-foreach|tox-expr)*)|
                       (attribute|element)*)>
<!ATTLIST tox-sample path CDATA #REQUIRED
                     name CDATA #IMPLIED
                     where CDATA #IMPLIED
                     tox-distribution IDREF #IMPLIED
                     duplicates (yes|no) ’yes’>

<!-- this element allows the nesting of content -->
<!ELEMENT tox-foreach ((tox-scan|tox-sample|tox-foreach|tox-expr|attribute|element|
                       tox-if|tox-alternatives)*)>
<!ATTLIST tox-foreach path CDATA #REQUIRED
                      name CDATA #IMPLIED
                      where CDATA #IMPLIED>

<!ELEMENT tox-expr EMPTY>
<!ATTLIST tox-expr value CDATA #REQUIRED
                   format CDATA #IMPLIED>

<!ELEMENT tox-if (tox-then, tox-else?)>
<!ATTLIST tox-if expr CDATA #REQUIRED>
<!ELEMENT tox-then (tox-alternatives|tox-scan|tox-sample|tox-foreach|tox-if|
                    element|attribute)*>
<!ELEMENT tox-else (tox-alternatives|tox-scan|tox-sample|tox-foreach|tox-if|
                    element|attribute)*>

<!-- This element defines the actual XML documents that are generated. -->

<!ELEMENT tox-document (element)>
<!ATTLIST tox-document name CDATA #REQUIRED
                       copies CDATA ’1’
                       starting-number CDATA ’0’
                       DTD-file CDATA #IMPLIED
                       pad (yes|no) ’no’>
```