



Universal XML/XML-Schema Utility UXU

Prepared by Frank Middleton

01/30/06

XML/XML Schema Utility

- Create an XML Schema and TMap from an existing XML file using heuristics to arrive at the structure (001 not required)
- Canonical Schemas including converting Salami Slice to Russian Doll style (001 not required)
- Provides round trip XML Schema to 001 TMap conversion with space and style preserving comments (001 analyzer required for the trip back)
- Round trip XML to 001 OMap with support for built-in XML datatypes such as xs:date and xs:time (001 analyzer required)
- Automatically generate an OMap and XML file from a TMap/Schema with randomly generated data that exercises and permutes all the elements of the Schema (001 analyzer required)
- RDBMS Schema import/export (001 analyzer required)
- e-business applications – full copy of 001 required

Introduction

The utility is based on and built with a technology originally developed for NASA that combines the best of object oriented and functional programming to provide a compiled code development environment as fast, rich, and flexible as Python and Perl, with the performance and robustness of a strongly typed compiled runtime, with the additional benefit of unprecedented reuse. It is based on class-diagram like maps of types (TMaps¹) and maps of functions (FMaps). The utility presently generates only TMaps and creates schemas from TMaps. It will eventually create FMaps from behavioral XML specifications such as SOAP and ebXML.

For XML to Schema and Schema to Canonical Schema, no copy of the base technology (marketed by Hamilton Technologies, Inc. under the name 001) is required, nor is there any need to learn it or its syntax. For advanced features such as Schema to XML and all RDBMS functions, a component of 001 known as the analyzer is required, but again no knowledge of 001 or its syntax is required.

To gain maximum advantage, a full copy of 001 is recommended. This allows one to edit XML Schemas using the remarkable TMap editor and syntax that can make a 1000 line XML Schema completely and usefully visible on a 1600*1200 pixel display, using color to outline structure. You can use FMaps to model and define the behavior of any application and the analyzer to generate code for and build complete systems with fully executable models.

The 001 constructs TMaps and FMaps allow you to specify the structure of a system. At runtime, the constructs OMap (Object Map) and EMap (Execution Map) describe the execution space and time, respectively. The UXU utility also provides a general utility for converting XML documents to OMaps (and subsequently, to and from RDBMS tables) and vice-versa. An API is

¹ TMap, FMap, EMap, OMap, 001, 001AXES, are all trademarks of Hamilton Technologies, Inc.

available to make all the UXU functionality available to applications.

Getting Started

You can start with an existing XML Schema (xsd) or XML document, and create a canonical XML Schema, as well as a TMap. In order for the heuristics to work, an XML document must contain an instance of every element, and, in the case of <choice>, at least one instance of every possible selection. If the schema uses unlabeled structures (e.g., <choice maxOccurs="unbounded">) then there must be permuted instances, otherwise it is impossible to tell the difference between this and <sequence maxOccurs="unbounded">. For example:

```
<c1>
  <e1/>
  <e2/>
  <e2/>
  <e1/>
</c1>
```

is likely an an unbounded choice of e1 and e2 of element c1. However, we do not recommend the use of unlabeled structures. They save little space and make the XML much harder to follow.

UXU automatically detects tree-like constructs, and creates a special recursive schema type for them. One common mistake is to use recursive unbounded sequences where a tree would be far more appropriate, and the visual representation of a TMap shows this kind of error quite dramatically. UXU won't (yet) convert a recursive <sequence> to a tree, but the TMap editor makes it very easy to do so. 001 supports trees as a native type structure.

If you start with an 001 TMap, you can round trip to an XML Schema and back, create XML from any OMap corresponding to the TMap/Schema (don't forget to analyze the TMap!), randomly create a suitable OMap, and even recreate the TMap/Schema from the XML, which you can also import into an OMap. Comments will survive the round trip with no loss of format and white space, and all of this with no special processing if you stick with basic 001 types (well, a bit of magic for *char* and *TreeOf*, but nothing unusual by XML Schema standards).

Issues and resolutions:

001 Structures vs. XML Schema structures

The only native 001 structure not supported directly by XML Schema is *TreeOf*. The Schema generator will automatically create a suitable *complexType* for *TreeOf*, and detect any similar *TreeOf* pattern when converting from Schema to TMap, and will attempt to detect *TreeOf* patterns when constructing a Schema from an XML document.

XML Schema semantics includes types and elements, each of which are in separate namespaces.

The converter mostly ignores this issue, which is only a problem in salami style schemas when the names of types and elements are the same, which would lead to definitions (in 001) like

```
label(label.)
```

meaning that *element* label is of *type* label. Normal XML Schema usage won't mix "ref=" (i.e.,

referring to the *element*) and “*type=*” (referring to the *type*), so the *element* definition will “swallow up” the *type* definition, eliminating the ambiguity. If not, the *element* definition will result in a TMap subtree:

```
label (tupleof)
  xs p1_label (label.)
```

and the *type* definition will result in a subtree:

```
label (tupleof)
  ...structure
```

which cannot analyze. This special case should never occur if XML Schema conventions are followed, and the TMap would not then include the former subtree at all.

001 Datatypes not native to XML Schema

The 001 datatype *char* is supported by unconditionally inserting a `<simpleType name="char">` into all generated schemas where *char* is a *restriction* of XML type *string*. 001 types *boolean*, *int*, *nat*, *rat*, *str*, *short* all map into native XML Schema types and vice versa. No other 001 primitive types are presently supported.

XML Schema types not native to 001

As noted above, XML has a separate namespace for types. The 001 namespace is extended by prefixing the element name with “xs”. It is also convenient in 001 to use built in types (such as *STR*) since 001 persistence is only available for a restricted set of such types. In order to preserve the XML names (i.e., element and type), the solution adopted is to prefix the type with “xs” as well. The second “xs” is required as a delimiter during OMap to XML conversion. For example:

Schema	TMap
<code><element mydate type="xs:date"/></code>	<code>xs mydate xs date(str)</code>
XML	OMap
<code><mydate>2005-10-22<mydate></code>	<code>xsmysdatexsdate:2005-10-22</code>

or optionally (with the *parent* “-p” argument) to convert the simple XML type into a complex 001 type and the reverse, as follows:

```
mydate (tupleof)
  xs date (str)
```

XML types *date*, *time*, and *duration* are supported in this fashion, however, *duration* types are not detected by the XML to Schema converter since the probability of mis-characterizing an arbitrary string as a duration is too great.

Salami style schemas

This results in single node subtrees, for example, *label(str)*. If the *consolidate subtrees* (-j) option is enabled, unique references to such subtrees will swallow them up. The consolidation process

effectively converts Salami style to Russian Doll style, and round tripping the process will create a Schema quite different to the original, although the same XML documents will work with either.

Attributes

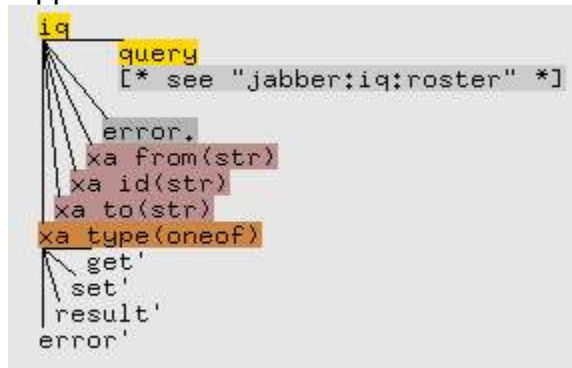
Attributes refer to XML elements (name-value pairs) where the values are encoded inside the opening label. For example, this XML fragment is from RFC 3921:

```
<iq type="get" from="romeo@montague.lit/pda">  
  <query xmlns="jabber:iq:roster"/>  
</iq>
```

The schema for this element is as follows:

```
<xs:element name='iq'>  
  <xs:complexType>  
    <xs:sequence>  
      <xs:any namespace='##other' minOccurs='0' />  
      <xs:element ref='error' minOccurs='0' />  
    </xs:sequence>  
    <xs:attribute name='from' type='xs:string' use='optional' />  
    <xs:attribute name='id' type='xs:NMTOKEN' use='required' />  
    <xs:attribute name='to' type='xs:string' use='optional' />  
    <xs:attribute name='type' use='required'>  
      <xs:simpleType>  
        <xs:restriction base='xs:NCName'>  
          <xs:enumeration value='error' />  
          <xs:enumeration value='get' />  
          <xs:enumeration value='result' />  
          <xs:enumeration value='set' />  
        </xs:restriction>  
      </xs:simpleType>  
    </xs:attribute>  
    <xs:attribute ref='xml:lang' use='optional' />  
  </xs:complexType>  
</xs:element>
```

Attribute names live in yet another namespace; here the 001 namespace is extended with prefix "xa". Thus this example is mapped into:



and the OMap to XML and XML to OMap converters will work correctly. Note that XML Schema requires all attributes to be declared **last**. A round trip from TMap to XSD to TMap will result in the attributes being moved, to the end (left) of the structure if necessary. This includes comments. If the attribute is also a built in XML type like *date*, the resulting TMap node would be

```
xa mydateattribute xs date(str)
```

Unlabeled XML Schema Bookkeeping nodes

001 requires all bookkeeping nodes to be labeled with a token that is both the type and the name. The converter creates unique node labels from the names of parents of unlabeled schema nodes, and prefixes them with "xs". This technique allows for somewhat transparent round-tripping. Unfortunately, XML Schema has many ways of representing the same structure, but the TMap to XSD conversion can in practice only produce one. The following tables show 4 examples of different XML Schemas (including attributes) that seem to produce the same XML. However, if the application (this is how the OMap generator works) follows the following algorithm, the data patterns will be different:

- An OsetOf with a OneOf child outputs exactly one element for each choice,
- An OsetOf with a TupleOf child repeats each child *n* times and each instance randomly picks a choice for OneOf elements of the TupleOf. (*n* defaults to 3 but can be changed with the -m switch).

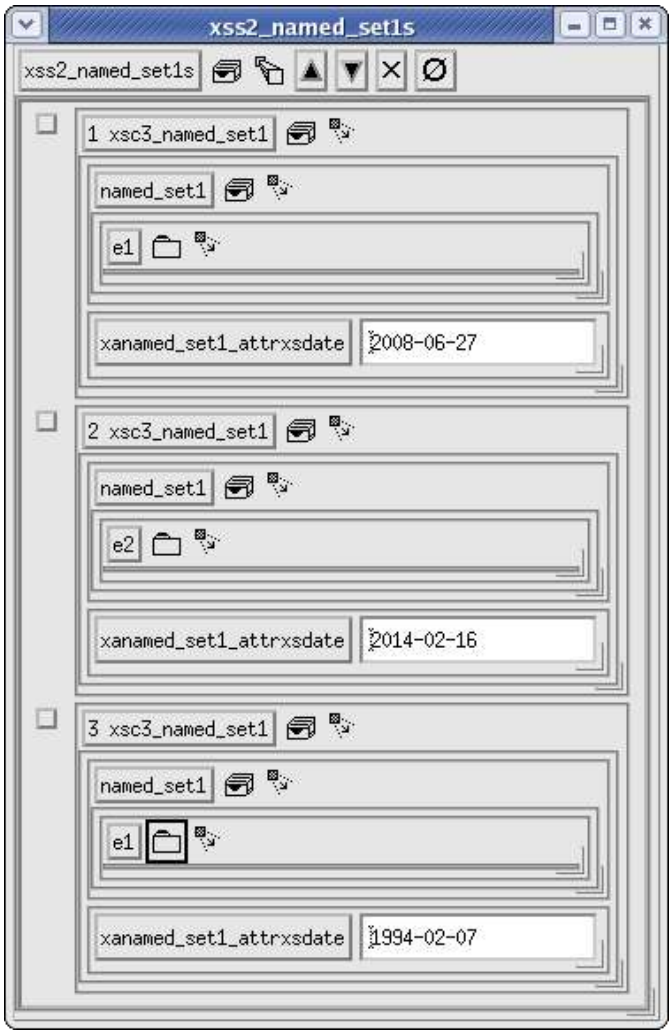
In the following examples, e1 and e2 are both defined s type state, which is in turn could be defined as

```
<xs:complexType name="state">  
  <xs:choice>  
    <xs:element name="mytime" type="xs:time"/>  
    <xs:element name="mydate" type="xs:date"/>  
  </xs:choice>  
</xs:complexType>
```

or as a TMap:

```
state(oneof)  
  xs mytime xs time(str)  
  xs mydate xs date(str)  
done(str)
```

Hidden Nodes Example 1

SCHEMA	TMap
<pre><xs:element name="named_set1" minOccurs="0" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:choice> <xs:element name="e1" type="state"/> <xs:element name="e2" type="state"/> </xs:choice> </xs:sequence> <xs:attribute name="named_set1_attr" type="xs:date"/> </xs:complexType> </xs:element></pre>	<pre>xs s2_named_set1s(osetof) xs c3_named_set1(tupleof) named_set1(oneof) e1(state,.) e2(state,.) xa named_set1_attr xs date(str)</pre>
XML	OMap
<pre><named_set1 named_set1_attr="2008-06-27"> <e1> <...> </e1> </named_set1> <named_set1 named_set1_attr="2014-02-16"> <e2> <...> </e2> </named_set1> <named_set1 named_set1_attr="1994-02-07"> <e1> <...> </e1> </named_set1></pre>	 <p>The screenshot shows a graphical representation of an OMap structure. The root node is 'xss2_named_set1s'. It contains three instances of 'xsc3_named_set1'. Each instance contains a 'named_set1' object. The first instance has 'e1' as a child and an attribute 'xanamed_set1_attrxsdate' with the value '2008-06-27'. The second instance has 'e2' as a child and the attribute value '2014-02-16'. The third instance has 'e1' as a child and the attribute value '1994-02-07'.</p>

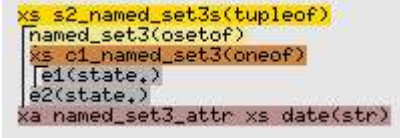
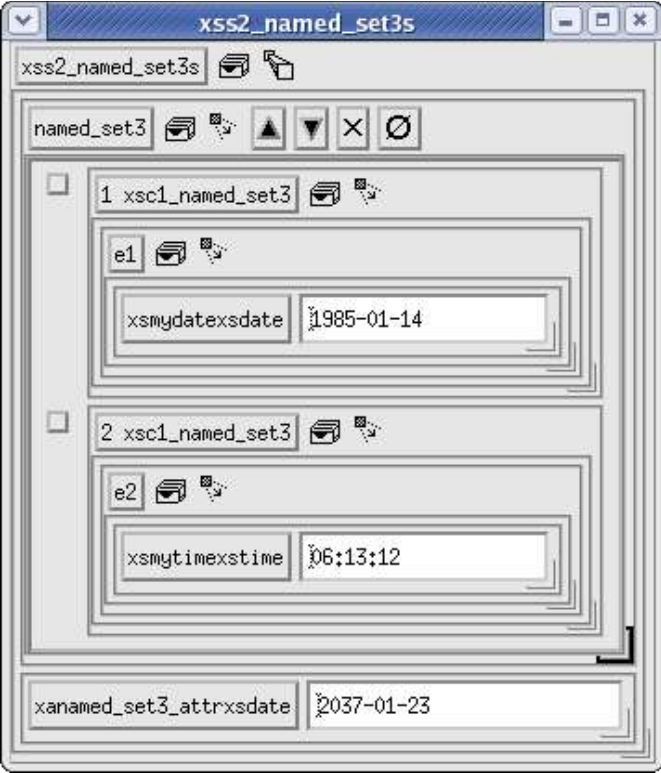
The OMap output is from the runtime object editor, a graphical representation of an OMap (OMaps are efficient binary structures in memory). Note that an extra placeholder (*xs c1_named_set1*) is required to provide a structure to store the attribute value in parallel with the OneOf.

Hidden Nodes Example 2

SCHEMA	TMap
<pre> <xs:element name="named_set2" minOccurs="0" maxOccurs="1"> <xs:complexType> <xs:sequence minOccurs="0" maxOccurs="unbounded"> <xs:choice> <xs:element name="e1" type="state"/> <xs:element name="e2" type="state"/> </xs:choice> </xs:sequence> <xs:attribute name="named_set2_attr" type="xs:date"/> </xs:complexType> </xs:element> </pre>	
XML	OMap
<pre> <named_set2 named_set2_attr="1994-02-07"> <e1> <mytime>05:12:17</mytime> </e1> <e2> <mydate>1996-06-09</mytime> </e2> </named_set2> </pre>	

Note that the XML elements are identical. Only the pattern is different (3 elements vs. 2). In this case an additional placeholder for the attribute is required (*xs s6_named_set2*) as well as one for the unlabeled *<xs:choice>*.

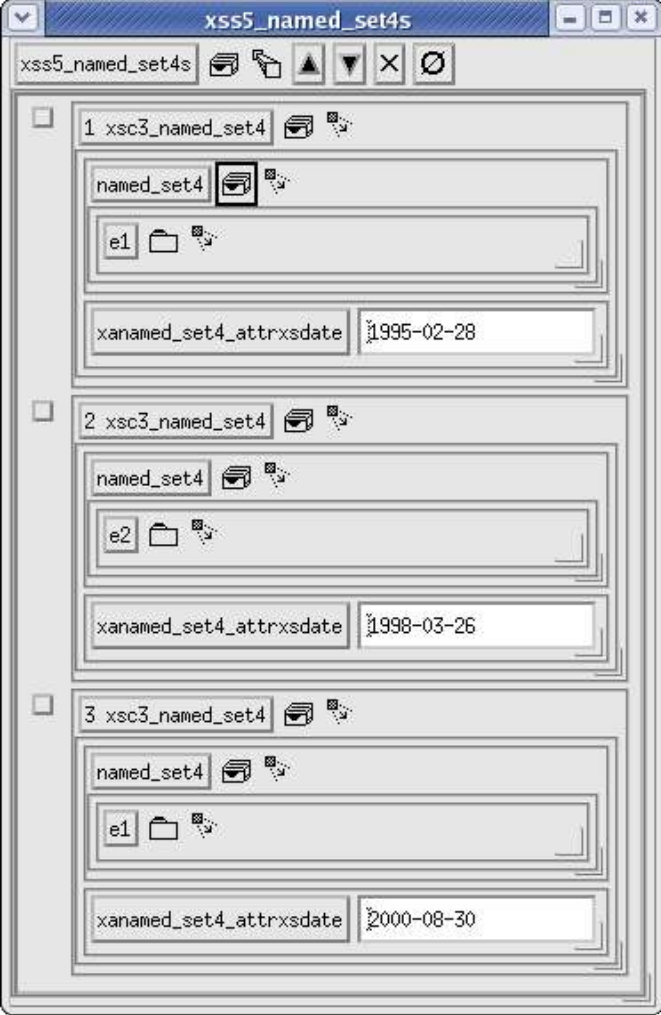
Hidden Nodes Example 3

SCHEMA	TMap
<pre> <xs:element name="named_set3" minOccurs="0" maxOccurs="1"> <xs:complexType> <xs:choice minOccurs="0" maxOccurs="unbounded"> <xs:element name="e1" type="state"/> <xs:element name="e2" type="state"/> </xs:choice> <xs:attribute name="named_set3_attr" type="xs:date"/> </xs:complexType> </xs:element> </pre>	
XML	OMap
<pre> <named_set3 named_set3_attr="2037-01-23"> <e1> <mydate>1985-01-14</mydate> </e1> <e2> <mytime>06:13:12</mytime>` ` </e2> </named_set3> </pre>	

Note that the XML and TMap/OMap are indistinguishable from that of example 2, although the schemas differ in that the *maxOccurs="unbounded"* occurs in a *sequence* vs. a *choice*.

As these examples show, it may be impossible to tell which of these 4 structures are implied when attempting to guess a schema from an XML file unless there are enough elements present to disambiguate the structures.


Hidden Nodes Example 4

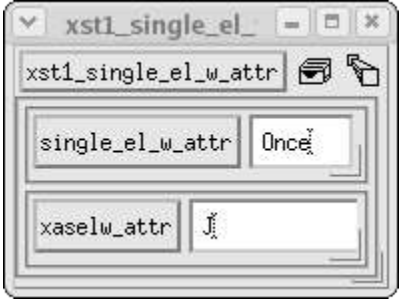
SCHEMA	TMap
<pre> <xs:element name="named_set4" minOccurs="0" maxOccurs="unbounded"> <xs:complexType> <xs:choice> <xs:element name="e1" type="state"/> <xs:element name="e2" type="state"/> </xs:choice> <xs:attribute name="named_set4_attr" type="xs:date"/> </xs:complexType> </xs:element> </pre>	<pre> xs s5_named_set4s(osetof) xs c3_named_set4(tupleof) named_set4(oneof) e1(state,) e2(state,) xa named_set4_attr xs_date(str) </pre>
XML	OMap
<pre> <named_set4 named_set4_attr="1995-02-28"> <e1> <...? </e1> </named_set4> <named_set4 named_set4_attr="1998-03-26"> <e1> <...> </e1> </named_set4> <named_set4 named_set4_attr="2000-08-30"> <e2> <...> </e2> </named_set4> </pre>	

Looks rather like example 1, not surprisingly since the only difference is a superfluous `<xs:sequence>`. The ability to insert any number of superfluous schema elements makes it impossible to guarantee perfect round-tripping. Inevitably superfluous schema elements will be lost.

Other Special Cases

XML Schema has special structures for an attribute-only node, or a primitive element with attributes. The equivalent structures are shown below:

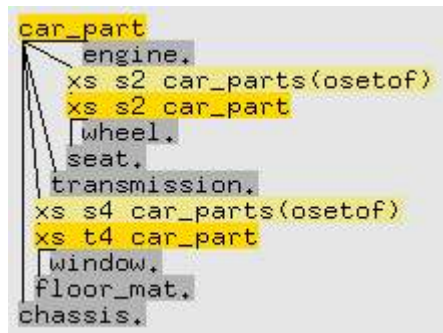
Attribute Only Node	
Schema	TMap
<pre><xs:element name="attr_only"> <xs:complexType> <xs:attribute name="o_attr" type="xs:date"/> </xs:complexType> </xs:element></pre>	<pre>attr_only(tupleof) xa o_attr xs date(str)</pre>
	OMap
	
XML	
<pre><attr_only o_attr="1976-07-28"/></pre>	

Primitive node with attribute	
Schema	TMap
<pre><xs:element name="single_el_w_attr" minOccurs="0" maxOccurs="1"> <xs:complexType> <xs:simpleContent> <xs:extension base="xs:string"> <xs:attribute name="selw_attr" type="char"/> </xs:extension> </xs:simpleContent> </xs:complexType> </xs:element></pre>	<pre>xs t1_single_el_w_attr(tupleof) [single_el_w_attr(str) xa selw_attr(char)</pre>
	OMap
	
XML	
<pre><single_el_w_attr selw_attr="J">Once</single_el_w_attr></pre>	

Note that the attribute is associated with a sibling if the parent is a hidden node (starts with "xs ") or with the parent if it is not hidden. If someone adds an extra node to the *Primitive node with attribute* example, UXU would apply the attribute to both, but the schema would not be valid unless the "xs " is also removed.

Round Trip issues with Hidden Nodes

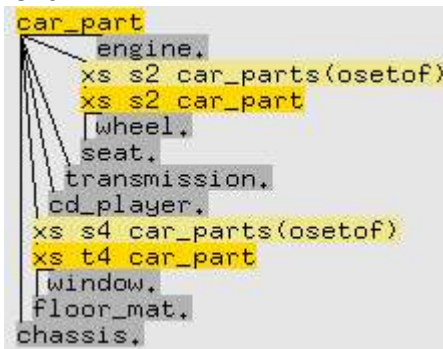
Consider:



Which reversibly produces the following XML Schema fragment

```
<element name="car_part">
  <sequence>
    <element engine type="engine" minOccurs="1" maxOccurs="1"/>
    <sequence minOccurs="1" maxOccurs="unbounded">
      <element wheel type="wheel" minOccurs="1" maxOccurs="1"/>
      <element seat type="seat" minOccurs="1" maxOccurs="1"/>
    </sequence>
    <element transmission type="transmission" minOccurs="1" maxOccurs="1"/>
    <sequence minOccurs="1" maxOccurs="unbounded">
      <element window type="window" minOccurs="1" maxOccurs="1"/>
      <element floor_mat type="floor_mat" minOccurs="1" maxOccurs="1"/>
    </sequence>
  </sequence>
</element>
```

But suppose we add a new element



The new schema will be

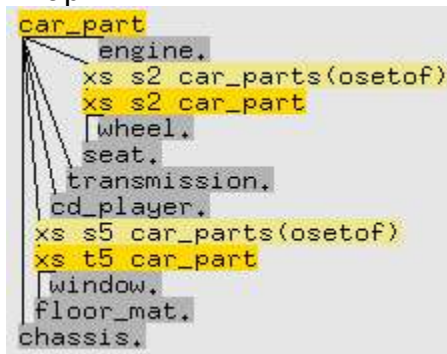
```
<element name="car_part">
  <sequence>
```

```

<element engine type="engine" minOccurs="1" maxOccurs="1"/>
<sequence minOccurs="1" maxOccurs="unbounded">
  <element wheel type="wheel" minOccurs="1" maxOccurs="1"/>
  <element seat type="seat" minOccurs="1" maxOccurs="1"/>
</sequence>
<element transmission type="transmission" minOccurs="1" maxOccurs="1"/>
<element cd_player type="cd_player" minOccurs="1" maxOccurs="1"/>
<sequence minOccurs="1" maxOccurs="unbounded">
  <element window type="window" minOccurs="1" maxOccurs="1"/>
  <element floor_mat type="floor_mat" minOccurs="1" maxOccurs="1"/>
</sequence>
</sequence>
</element>

```

If we round trip, we'll get a new TMap



so any FMaps that relied (e.g.) on "window" being in the "xs s4 car parts" would be in trouble.

The obvious, and simple way to avoid this problem is to only add new elements to the end. The OMap runtime is completely compatible with this approach, so old objects can seamlessly migrate to new versions with added fields.

Creating TMaps and Schemas from an existing XML document.

Some XML Schemas produce non-deterministic XML (for example the OsetOf OneOf described above) so the schema generator may produce a consistent but perhaps unintended result. However, it is unlikely that any application would ever generate XML like this without a schema, so in this case the issue is moot. The only known practical problem is the case when a pair of elements reference a type using two different names. Suppose the original schema has a type called child, and two elements reference it as follows:

```

<complexType name="child">
  <sequence>
    <element name="grandchild1" type="xs:string"/>
    <element name="grandchild2" type="xs:string"/>
    <element name="etc" type="etc"/>
  </sequence>
</complexType>
<element parent>
  <sequence>
    <element name="child" type="child"/>

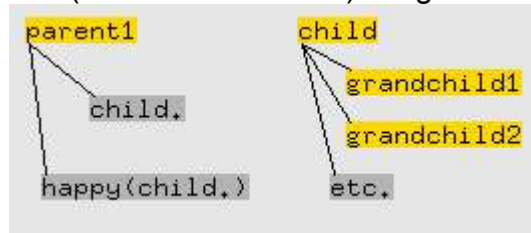
```

```
<element name="happy" type="child"/>
</sequence>
</element>
```

The XML file will not reference the type in any way (since schema type names are not visible in an XML document), so the XML to Schema converter has no way of knowing that they map to the same type (this is why schemas are needed). In this example, an XML instance might be:

```
<parent1>
  <child>James</child>
  <happy>Fred</happy>
  <etc>
  ...
</etc>
</parent1>
```

UXU uses a heuristic working backwards through the paths to match up trees which are subsets of each other, and looks for hints (such as references) to figure out which tree is the one to use.



which, in this example, coincidentally happens to be correct. Applications developed without using schemas would be unlikely to use different labels for each type instance, so this may be a solution looking for a problem.

Other than this, given that there are enough instances in the XML document to determine its structure, the heuristics can accurately construct an XSD from XML, including analyzing patterns that look like trees, and strings that look like dates and times. Presently *xs:duration* will be converted to 001 type *str* since an instance looks like "P1996Y11M03D12H56M02S" where any of the parts can be missing, and deterministically parsing for this is not worth the effort. Is "P02S" a string or a 2 second duration?

Using the UXU utility

- Solaris or Linux command-line executable
- Embed using 001 Libraries and API

TMap to XML Schema

```
$ $HTI_EXE:export_treesdb.exe test ty
$ uxu -r -o test.xsd test.ty
```

The first command exports a type map called *test* to *test.ty.treesdb.omap*

The second line converts *test.ty.treesdb.omap* to *test.xsd* after reversing (-r) the order so that the xsd file top to bottom is equivalent to the standard right to left reading of a TMap.

```
$ uxu -r -o test.xsd -t test ty
```

is a preferred alternative syntax

XML Schema to TMap

```
$ uxu -s -j -i test.xsd test.ty
$ $HTI_EXE:import_treesdb.exe test ty
```

Create *test.ty.treesdb.omap* from *test.xsd*, joining (-j) single references to types (converting Salami to Russian Doll) and sorting (-s) subtrees by locality of reference.

```
$ uxu -s -j -i test.xsd -t test ty
```

is a preferred alternative syntax

XML to Schema and TMap

```
$ count=`wc -l test.xml`
$ uxu -j -s -c $count -i test.xml -o test.xsd test.ty
```

Similar to XML Schema to TMap. For very large source files, the -c flag makes UXU output an estimated time to completion once a second. On a 330MHz Sparc it takes about 10 minutes to process about 500,000 lines of XML.

TMap to OMap and XML

```
$ uxu -f /usr/local/fortune/datfiles/fortunes -m 3 -xml -o test.xml -t test om
```

Uses rand() to instantiate a TMap, using random lines from a text file (-f) in fortune format for strings, using *./test/db.tmap2* (-t test) to create *om.test.omap* and *test.xml* as if copying an XML file (-xml). Structures are filled out as follows:

OsetOf: (<element ... maxOccurs="unbounded"/>) with OneOf children (children of <choice>) are automatically filled with one element for each choice, otherwise *m* instances of each child.

OneOf: (<choice>) other than when occurring as child of OsetOf are selected randomly

TreeOf: children are created in the following 3*3 (*m*m*) pattern

```
1.
1.2.
1.2.1.
1.2.2.
```

1.2.2.1
1.2.3.
1.3.

XML to OMap

```
$ count=`wc -l test.xml`
```

```
$ uxu r -c $count -i test.xml -t test om
```

inputs *test.xml* and creates *om.test.omap*, using *./test/db.tmap2* as a schema.

OMap to XML

```
$ uxu -o test.xml -t test om
```

outputs *test.xml* from *om.test.omap*, using *./test/db.tmap2* as a schema.

API

Requires instantiated TMap xmldb with options filled in. See *run_conversion.op* for examples of the API for each function. All functions will be available as a layered type, which makes extraordinary power to the application available at little effort.

Command Line Switches

Usage UXU [-d <dir>] [-i <input>] [-j] [-f filename] [-n] [-o <output>] [-p] [-r] [-s] [-t tmapname] [-u] [-xml] [-xsd] [path to OMap]

<dir> is 'R' or 'L' for OMap to XML conversion only. 'R' starts with visually the rightmost (top) node and is the default

- i input file (.xml or .xsd)
- f <filename> fortune formatted file of strings for random insertion to test OMap
- j eliminate single definedas uses (replaces ref-"xyzyy" with xyzyy, inline)
- m number of instances of unbounded entities to create
- n suppresses the expansion of <include>
- o output file (.xsd or .xml)
- p create parent (tupleof) style structures for XML built in datatypes
- r will reverse the trees in the output treesdb (schema) OMap before writing it.
If -r is the only switch, it will read in the treesdb (schema) OMap first, reverse it, and then write it out again.
- s sort subtrees by locality of reference
- t TMapname used to construct OMap filename as well.
Must be "...treesdb" for schemas if present.
- u execute *export_treesdb* or *import_treesdb* when appropriate
- xml forces xml input mode regardless of file suffix,
- xsd similarly forces xsd input mode.