An Algorithm for Merging a simpleType Dependency Chain

Roger L. Costello
April 2011

Introduction

This simpleType does not depend on another simpleType:

```xml
<xsd:simpleType name="English-language-family-name">
    <xsd:restriction base="xsd:string">
        <xsd:minLength value="2" />
        <xsd:maxLength value="100" />
        <xsd:pattern value="[a-zA-Z'\s.-]+" />
    </xsd:restriction>
</xsd:simpleType>
```

The simpleType can be understood on its own, without referring to another simpleType. In words it says, “An English language family name consists of the lower- and upper-case letters, apostrophe, space, period, and dash; a name has a length of 2 to 100 characters.”

The minLength, maxLength, and pattern facets constrain the simpleType.

On the other hand, this simpleType does depend on another simpleType:

```xml
<xsd:simpleType name="BostonAreaSurfaceElevation">
    <xsd:restriction base="elev:EarthSurfaceElevation">
        <xsd:minInclusive value="0" />
        <xsd:maxInclusive value="120" />
    </xsd:restriction>
</xsd:simpleType>
```

It cannot be understood in isolation. To understand the constraints on BostonAreaSurfaceElevation it is necessary to examine its base type, elev:EarthSurfaceElevation. Furthermore, elev:EarthSurfaceElevation may have a base type, which will require understanding it, and so forth. The entire dependency chain must be understood. Each simpleType may be in different schema documents and in different namespaces.

If a simpleType is at the bottom of a long dependency chain, and the simpleTypes are scattered across multiple schema documents, then understanding the constraints on it can be challenging.
The purpose of this article is to describe an algorithm for merging all the constraints in a dependency chain; thus, creating a simpleType that can be understood on its own.

**Example: the simpleType named BostonAreaSurfaceElevation**

```xml
<xsd:simpleType name="BostonAreaSurfaceElevation">
  <xsd:restriction base="elev:EarthSurfaceElevation">
    <xsd:minInclusive value="0"/>
    <xsd:maxInclusive value="120"/>
  </xsd:restriction>
</xsd:simpleType>
```

is rendered to this

```xml
<xsd:simpleType name="BostonAreaSurfaceElevation">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="0"/>
    <xsd:maxInclusive value="120"/>
  </xsd:restriction>
</xsd:simpleType>
```

Observe that it can be understood on its own.

**“And” Facets versus “Or” Facets**

The enumeration facets in this simpleType are or-ed together:

```xml
<xsd:simpleType name="Color">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="red"/>
    <xsd:enumeration value="green"/>
    <xsd:enumeration value="blue"/>
  </xsd:restriction>
</xsd:simpleType>
```

In words it says, “The value of Color may be red, green, or blue.”

Likewise, pattern facets are or-ed together, e.g.: 
A value of ISBNType must conform to the first pattern, the second pattern, or the third pattern.

enumeration facets and pattern facets are or-ed together.

The other facets are and-ed together. This simpleType says that the minimum integer value is 0 and the maximum integer value is 120:

"And" Pattern Facets in a Dependency Chain

The pattern facets within a simpleType are or-ed together. But the pattern facets across simpleTypes are and-ed together.

Example: simpleType "B" is the base of simpleType "A":

"And" Pattern Facets in a Dependency Chain

The pattern facets within a simpleType are or-ed together. But the pattern facets across simpleTypes are and-ed together.

Example: simpleType "B" is the base of simpleType "A":
The pattern facets are "and-ed" together. The value of "A" is an integer and it must consist of 1-5 digits and 1-3 digits.

**Rendering a Standalone simpleType that is a Merge of its Dependency Chain**

As shown above, some facets are or-ed together, some facets are and-ed together, and pattern facets are sometimes or-ed together and sometimes and-ed together. That variability makes processing difficult.

Recall that the purpose of this paper is to describe an algorithm for merging all the constraints in a dependency chain, thus rendering a simpleType that can be understood on its own.

**Rule:** in my rendering all the facets in the rendered simpleType are and-ed together.

That uniformity will make processing easier.

A rendered simpleType will have at most one enumeration facet. For example, this simpleType

```xml
<xsd:simpleType name="Color">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="red"/>
    <xsd:enumeration value="green"/>
    <xsd:enumeration value="blue"/>
  </xsd:restriction>
</xsd:simpleType>
```

is rendered as

```xml
<xsd:simpleType name="Color">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration>
      <xsd:value>red</xsd:value>
      <xsd:value>green</xsd:value>
      <xsd:value>blue</xsd:value>
    </xsd:enumeration>
  </xsd:restriction>
</xsd:simpleType>
```

The pattern facets within a simpleType are combined into one facet by or-ing the values together. For example, this simpleType

```xml
<xsd:simpleType name="Color">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration>
      <xsd:value>red</xsd:value>
      <xsd:value>green</xsd:value>
      <xsd:value>blue</xsd:value>
    </xsd:enumeration>
  </xsd:restriction>
</xsd:simpleType>
```
The pattern facets across simpleTypes are juxtaposed. For example, this dependency chain

```xml
<xsd:simpleType name="A">
    <xsd:restriction base="B">
        <xsd:pattern value="[0-9]{1,5}" />
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="B">
    <xsd:restriction base="xsd:integer">
        <xsd:pattern value="[0-9]{1,3}" />
    </xsd:restriction>
</xsd:simpleType>
```

is rendered as

```xml
<xsd:simpleType name="A">
    <xsd:restriction base="xsd:integer">
        <xsd:pattern value="[0-9]{1,5}" />
        <xsd:pattern value="[0-9]{1,3}" />
    </xsd:restriction>
</xsd:simpleType>
```

Since all facets are and-ed together, a value for simpleType "A" must conform to the first pattern and the second pattern.

Example: here is an example of rendering a simpleType that contains a variety of facets.

```xml
<xsd:simpleType name="Color">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="red" />
        <xsd:enumeration value="green" />
    </xsd:restriction>
</xsd:simpleType>
```
simpleType Forms

There are 7 forms of simpleTypes:

1. The simpleType has a base attribute

```xml
<xsd:simpleType name="BostonAreaSurfaceElevation">
    <xsd:restriction base="elev:EarthSurfaceElevation">
        <xsd:minInclusive value="0"/>
        <xsd:maxInclusive value="120"/>
    </xsd:restriction>
</xsd:simpleType>
```

2. The simpleType has a nested simpleType

```xml
<xsd:simpleType name="BostonAreaSurfaceElevation">
    <xsd:restriction>
        <xsd:simpleType>
            <xsd:restriction base="xsd:integer">
                <xsd:minInclusive value="-1290"/>
                <xsd:maxInclusive value="29035"/>
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:restriction>
</xsd:simpleType>
```
3. **The simpleType is a list and has an `itemType` attribute**

```xml
<xsd:simpleType name="A">
  <xsd:list itemType="B" />
</xsd:simpleType>
```

4. **The simpleType is a list and has a nested simpleType**

```xml
<xsd:simpleType name="A">
  <xsd:list>
    <xsd:simpleType>
      <xsd:restriction base="xsd:integer">
        <xsd:minLength value="10"/>
        <xsd:maxLength value="20"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:list>
</xsd:simpleType>
```

5. **The simpleType is a union and has a `memberTypes` attribute**

```xml
<xsd:simpleType name="maxOccurs_type">
  <xsd:union memberTypes="unbounded_type xsd:nonNegativeInteger"/>
</xsd:simpleType>
```

6. **The simpleType is a union and has nested simpleTypes**

```xml
<xsd:simpleType name="___">
  <xsd:union>
    <xsd:simpleType>
      ...  
    </xsd:simpleType>
    <xsd:simpleType>
      ...  
    </xsd:simpleType>
  </xsd:union>
</xsd:simpleType>
```

7. **The simpleType is a union and has a `memberTypes` attribute and nested simpleTypes**

```xml
<xs:restriction value="0"/>  
<xs:maxInclusive value="120"/>
</xs:restriction>
</xs:simpleType>
```
Determining the Base Type for a simpleType from its Dependency Chain

Below is a dependency chain. The simpleType "A" has a base type "B" which has a base type "C" (i.e., the dependency chain is A -> B -> C).

```xml
<xsd:simpleType name="A">
    <xsd:restriction base="B">
        ...
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="B">
    <xsd:restriction base="C">
        ...
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="C">
    <xsd:restriction base="xsd:integer">
        ...
    </xsd:restriction>
</xsd:simpleType>
```

simpleType "C" is at the top of the dependency chain, and its base type is xsd:integer. Therefore, the base type for simpleType "A" is xsd:integer and it is rendered as

```xml
<xsd:simpleType name="A">
    <xsd:restriction base="xsd:integer">
        ...
    </xsd:restriction>
</xsd:simpleType>
```

It is tempting to infer that the base type will always be the base type of the simpleType at the top of the dependency chain. However, that is not always the case.
**Example:** in the dependency chain $A \rightarrow B \rightarrow C$ the `simpleType "B"` is a list type:

```xml
<xsd:simpleType name="A">
    <xsd:restriction base="B">
        ...
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="B">
    <xsd:list itemType="C" />
</xsd:simpleType>

<xsd:simpleType name="C">
    <xsd:restriction base="xsd:integer">
        ...
    </xsd:restriction>
</xsd:simpleType>
```

The base type of `simpleType "A"` is a list type. The `simpleType "A"` is rendered as this:

```xml
<xsd:simpleType name="A">
    <xsd:restriction>
        <xsd:simpleType>
            <xsd:list>
                <xsd:simpleType>
                    <xsd:restriction base="xsd:integer"/>
                </xsd:simpleType>
            ...
        </xsd:list>
    </xsd:restriction>
</xsd:simpleType>
```

In words it says, “The base type of `simpleType "A"` is a `simpleType` that is a list type; the items in the list are constrained integers.”

Now for the algorithm. Be sure you understand all of the above before reading the algorithm.

**Rendering Algorithm**

```java
render (simpleType A) {
    First, create a sequence from the dependency chain, e.g., [A, B, C], where A is the `simpleType` to be rendered, its base type is B, which has base type C.
    sequence = type-dependency-sequence (A)
```
If the length of sequence is 1 and A is not a union type then return A.

If A has form #1 (see simpleType Forms above) and there is list simpleType (L) in the dependency chain then render L and use it as the base type for A; merge A’s facets with the simpleTypes up to, but not including, L.

```xml
  <xsd:simpleType name="A">
    <xsd:restriction>
      render (L)
      merge-facets ([A .. up to but not including L])
    </xsd:restriction>
  </xsd:simpleType>
```

If A has form #1 (see simpleType Forms above) and there is no list simpleType in the dependency chain then A’s base type is the base type of the simpleType at the top of the dependency chain. Merge A’s facets with all the simpleTypes up the dependency chain.

```xml
  <xsd:simpleType name="A">
    <xsd:restriction base="base-type(list(last()))">
      merge-facets ([A ..])
    </xsd:restriction>
  </xsd:simpleType>
```

If A has form #2 (see simpleType Forms above) then create a new sequence by pulling out the nested simpleType and modifying A so that its base type is the extracted simpleType. Render this new sequence.

Example: Convert this one simpleType

```xml
  <xsd:simpleType name="A">
    <xsd:restriction>
      <xsd:simpleType>
       <xsd:restriction base="B">
        ...
       </xsd:restriction>
     </xsd:simpleType>
    ...
  </xsd:restriction>
</xsd:simpleType>
```

into two simpleTypes

```xml
  <xsd:simpleType name="A">
    <xsd:restriction base="random()">
      ...
    </xsd:restriction>
  </xsd:simpleType>
```

```xml
  <xsd:simpleType>
    ...
  </xsd:simpleType>
```
Succinctly: convert the sequence \([A, B, C]\) to \([A, X, B, C]\) and then render the latter sequence.

If \(A\) has form \#3 (see \textit{simpleType Forms} above) then render the base type of the list's \texttt{itemType} and nest it within the list.

```xml
<xsd:simpleType name="A">
  <xsd:list>
    render([B...])
  </xsd:list>
</xsd:simpleType>
```

If \(A\) has form \#4 (see \textit{simpleType Forms} above) then render the nested \texttt{simpleType} and nest it within the list.

```xml
<xsd:simpleType name="A">
  <xsd:list>
    render(nested simpleType)
  </xsd:list>
</xsd:simpleType>
```

If \(A\) has form \#5, \#6, or \#7 (see \textit{simpleType Forms} above) then render the nested \texttt{simpleTypes} and render the \texttt{simpleTypes} referenced by \texttt{memberTypes}; nest all these rendered types inside \texttt{union}.

```xml
<xsd:simpleType name="A">
  <xsd:union>
    for each \texttt{simpleType}, \(X\), within the union
    render \((X)\)
    for each \texttt{simpleType}, \(Y\), within \texttt{memberTypes}
    render \((Y)\)
  </xsd:union>
</xsd:simpleType>
```

merge-facets(simpleType+ list) {

merge-enumeration-facets(list)
merge-fractionDigits-facets(list)
merge-length-facets(list)
}
merge-maxExclusive-facets(list)
merge-maxInclusive-facets(list)
merge-maxLength-facets(list)
merge-minExclusive-facets(list)
merge-minInclusive-facets(list)
merge-minLength-facets(list)
merge-pattern-facets(list)
merge-totalDigits-facets(list)
merge-whiteSpace-facets(list)

merge-enumeration-facets(simpleType+ list)

Get the first simpleType in the list. If it has any enumeration facets then render them as one enumeration facet in the form described in Rendering a Standalone simpleType that is a Merge of its Dependency Chain. If it doesn’t have any enumeration facets then recurse: merge-enumeration-facets(list[2..])

merge-pattern-facets(simpleType+ list)

Get the first simpleType in the list. If it has any pattern facets then render them as one pattern facet with the patterns or-ed together, as described in Rendering a Standalone simpleType that is a Merge of its Dependency Chain. Then merge-pattern-facets(list[2..])

merge-maxInclusive-facets(simpleType+ list)

Get the first simpleType in the list. If it has a maxInclusive facet then render the maxInclusive facet unaltered. If it doesn’t have a maxInclusive facet then recurse: merge-maxInclusive-facets(list[2..])

Merging the other facets is identical to merging maxInclusive.