Coding with Unique Identifiers and How to Test for them with Schematron

Why and how we use unique identifiers in XML

Most of our projects demand that we compile *prosopography* data, that is, a standard list of unique identifiers for people, places, books, and other named entities. We might compile a prosopography list in a `header` element holding metadata in a project file, or perhaps in a separate file altogether if our project contains multiple XML files and we just want one document to store all the detailed prosopography information relevant anywhere and everywhere throughout our project. Storing a list of prosopography entries involves creating something like a contacts list of everyone you know with their contact information and unique identification information. In XML prosopography lists, we apply the `@xml:id` attribute to hold a unique string of text, different from any other `@xml:id` in your project, designed to identify a particular person, place, or any named thing that you need to distinguish from other named things. In Relax NG, the `@xml:id` is coded to have an `xsd:ID` datatype, which requires a unique string of text for each distinct `@xml:id` and permits no duplicates. The `xsd:ID` datatype permits text only or a combination of text and numbers, but it must not be a string of numbers alone and it must not contain any white space.

A common standardized format for prosopography data is defined by TEI Guidelines, specifically Chapter 13: Names, Dates, People, and Places which provides helpful examples and useful things to think about when distinguishing among proper names. The Digital Mitford project's Site Index file offers several lists combined together in one file, lists of historical people, fictional characters, real and fictional places, books, journal publications, works of art, and more, and it serves as a kind of backbone for all of the project XML files. Each XML file representing a writing by or about the nineteenth-century author Mary Russell Mitford, say, a letter, a poem, a play, or some other kind of text, makes reference to named entities, often several of the kinds we have mentioned, and we are coding them in a way to help keep track of particular people, fictional characters, places, books, etc. whenever they appear (in whatever form they appear) in our files. Think about why we need to do this. One person might be called several different names in different places, so Mitford sometimes affectionately referred to her father as “Drum” or “Papa” while another text in our project might refer to him more formally as “Dr. Mitford” or “George Mitford”. The same person might have several nicknames, and in the case of women, their surnames might change in marriage, but we still need a way to track these names and trace them to the same person when Mitford knew them before and after they were married. Here is an example of an entry for Mary Ann Harvey or Mary Ann Davenport from the Digital Mitford site index:

```
<person xml:id="Davenport_MA" sex="2">
                    <persName>
                        <surname type="maiden">Harvey</surname>
                        <surname type="married">Davenport</surname>
                        <forename>Mary</forename>
                        <forename>Ann</forename>
                    </persName>
                   <occupation>actor</occupation>
                   <note type="bio" resp="#lmw">English actor (1759-1843).</note>
</person>
```

We could add other `<persName>` elements inside this entry if we discover that Mary Ann Harvey used a pseudonym or fake name, or had a nickname. We add as much or as little information as we can find and as seems necessary to our project, and our entries are always in a state of development as we keep working on our project. In the body of a Mitford letter, if she mentions that her father has seen Mary Davenport perform in a play, we encode Mary's name like this:

```
<p><persName ref="#Mitford_Geo">Drum</persName> saw <persName ref="#Davenport_MA">Mary Davenport</persName>
         perform in <title ref="#TwelfthNight_Shkspr">Twelfth Night</title> last week.</p>
```

Because we are only permitted to use an `@xml:id` attribute value *once* in our entire project, we place a hashtag on all attributes (like `@ref`, `@wit`, `@corresp` that *refer* or *point to* that `@xml:id`. The use of the hashtag permits us to invoke the identifying string as often as we need to, and it permits us to pull up more information about a given name by pointing to information in our site index file.

How to write Schematron to check hashtagged attributes against `@xml:id` values

Since it is very easy to mistype an attribute value as we are coding our project files, we can either embed our `@xml:id` values in a Relax NG file, or write Schematron rules to help us check their values against our standard list, wherever we have placed it in relation to our project files. We can also write Schematron rules to make sure we remember to include a hashtag at the start of each value when it is in a pointer `@ref` or other attribute that points to an `@xml:id`. That way we can point to those identifiers as many times as we need to in the document. We show you how to do with an example from the Emily Dickinson Fascicle 16 project.

The Dickinson team prepared a list of published editions of Dickinson's poems with `@xml:id` attributes and detailed bibliography information about each, and they compiled each entry inside a TEI `<listWit>` element or a list of witnesses. Each entry on the list holds distinct identifying information about a different *witness* that produced a distinct published representation of Emily Dickinson's manuscript poems. The witnesses produced *variants* or different readings of the same document, and coding these variant readings in the lines of poetry helped the Dickinson team to study how published editions normalized or otherwise altered Dickinson's poems in the different published editions. To mark the different *variants* (or different versions of words and phrases and punctuation) within particular lines in each poem, and the Dickinson team referred to the source of each variant with a hashtagged `@wit` attribute pointing to the `@xml:id` in the `<listWit>` up in their `teiHeader` element. Here is an abbreviated view of their list of witnesses:

```
<listWit>
   [. . .]
      <witness xml:id="poems3">
        <bibl>
            <title>Poems, Third Series</title>
            <author>Emily Dickinson</author>
            <editor>Mabel Loomis Todd</editor>
            <pubPlace>Boston</pubPlace>
            <publisher>Little, Brown and Company</publisher>
            <date>1896</date>
            <ref target="http://catalog.hathitrust.org/Record/100654113">Hathi Trust Digital Library</ref>
        </bibl>
```

```
    </witness>
    <witness xml:id="ce">
        <bibl>
            <title>The Poems of Emily Dickinson: Centenary Edition</title>
            <author>Emily Dickinson</author>
            <editor>Martha Dickinson Bianchi and Alfred Leete Hampson</editor>
            <pubPlace>Boston</pubPlace>
            <publisher>Little, Brown and Company</publisher>
            <date>1930</date>
        </bibl>
    </witness>
    <witness xml:id="fh">
        <bibl>
            <title>Final Harvest: Emily Dickinson's Poems</title>
            <author>Emily Dickinson</author>
            <editor>Thomas H. Johnson</editor>
            <pubPlace>Boston</pubPlace>
            <publisher>Little, Brown and Company</publisher>
            <date>1961</date>
        </bibl>
        [. . .]
    </witness>
</listWit>
```

Schematron can be used to constrain the writing of `@xml:id` values in this list to meet project specifications. For example, the Dickinson team will want to make sure their `@xml:ids` do not begin with hashtags:

```
<pattern>
     <rule context="@xml:id">
            <report test="starts-with(., '#')">
                xml:id attributes must not begin with a hashtag!
            </report>
        </rule>
    </pattern>
```

The Dickinson team uses an `@wit` attribute with a hashtag in the body of the poem to point to *one or more* identified published editions. Often a particular variant is displayed in three or four published editions, and when that is the case, the project team separates each distinct hashtagged identifier with a white space, like this: `<rdg wit="#ce #poems2 #fh">`. These point to `@xml:id` values defined up in the TEI header and its `<listWit>` as: df16, fh, and poems. It is very easy to mistype these ids, so we need a good schema rule to ensure they are correct. Here is how we prepared our rule for the `@wit` attribute

```
<pattern>
                <rule context="@wit">
                    <let name="tokens" value="for $w in tokenize(., '\s+') return substring-after($w, '#')"/>
                    <assert test="every $token in $tokens satisfies $token = //tei:TEI//tei:listWit//@xml:id">
                        Every reading witness (@wit) after the hashtag must match an xml:id defined in the list of witnesses in
                    </assert>
                </rule>
            </pattern>
```

This complex rule permits us to use white space as a separator, so we can refer to multiple published editions that represent a particular variant in the text.

The rule accomplishes several things:

1. The `<let>` element: This defines a *variable* in Schematron, and gives it an `@name` (`"tokens"`) which we can quickly refer to with a dollar-sign in front of it, as `$token`. We can define a variable inside a rule to make it *local* (in which case the parser only "knows" about it and reads it within the context of a particular rule), or we can define it as a *global* variable by setting the `<let>` element above the `<pattern>` elements in the Schematron hierarchy so the variable can be invoked everywhere.

   Note: We make global variables when we need to write Schematron rules that point to other files, to see if a value of an attribute matches an `@xml:id` defined in a separate project file, for example. But variables can be used to hold any complex pattern that you want to invoke in a `rule @context` or in an `@test` on an `assert` or `report` element.

2. Dealing with multiple values: First, in our variable, we tokenized our `@wit` attribute on white space, and that created multiple values or token. So if we *do* use one or more white spaces in an `@wit` attribute, we use those white spaces as a dividing point: we separate the value into **"tokens"**: so `<rdg wit="#ce #poems2 #fh">` would be tokenized into three pieces. Our language, `for $w in tokenize(., '\s+')`, defines a separate variable *for each one of these tokens*, since we need to look at them one by one. For each of these, we need to cut off the leading hashtag, so we do one more thing: return the `substring-after($w, '#')`. This creates three tokens in this format:
   - token 1: ce
   - token 2: poems2
   - token 3: fh

3. Now our assert test needs to do something more, so it can deal with a situation in which there's only one token **or** multiple tokens. We can't just test all the tokens at once against each `@xml:id` because Schematron needs to look at them one at a time: first ce, then poems2, then fh. For that one-at-a-time handling, we use this syntax: `<assert test="every $token in $tokens satisfies $token = //tei:TEI//tei:listWit//@xml:id">` The work of this is done by the structure: `every [singular] in [plural] satisfies [a test you design for the singular value]`.

   If the Dickinson team members move their `<listWit>` to separate file of prosopography lists, like the site index of the Mitford project, they would define a variable with another `let` statement pointing to a file in its location relative to the Dickinson project files. The file can be served on the web and pointed to with an absolute web address, or referenced by a relative address as we do here:

```
<let name="si" value="doc('Dickinson_listIds.xml')//@xml:id"/>
    <pattern>
                <rule context="@wit">
                    <let name="tokens" value="for $w in tokenize(., '\s+') return substring-after($w, '#')"/>
                    <assert test="every $token in $tokens satisfies $token = $si">
                        Every reading witness (@wit) after the hashtag must match an xml:id defined in the list of witness
                    </assert>
                </rule>
            </pattern>
```

   Defining the variable `$si` above a `<pattern>` element makes it a *global variable* in Schematron, which means it can be referenced in multiple `pattern` elements. We could also define it *inside* the `<rule>` if we only need to represent the variable within this particular rule. Defining variables as filepaths in Schematron is a convenient way to make your schema rules cross-check values between multiple files. Here it permits us to use one file as a way to validate the attribute values in use on any project file associated with this Schematron file.