Search-WS Description Language

Introduction
This is not a formal definition of the Search-WS Description Language (as such a thing does not yet exist), rather, it is a document intended to generate discussion of the ideas contained herein.

The objective of the Search-WS Description Language is to allow searching of disparate on-line databases through a simple interface. In essence, it describes how to:

1. Create a url that will be used to obtain a response from a database.
2. Extract useful content from the response.
3. Repeat 1. and 2. until done.

To achieve this you need a description of the database written using the Search-WS Description Language and a programming interface that will read the description and execute the operations it describes.

The current incarnation is very incomplete and is offered as as-is in the hope that giving it a public viewing will elicit some helpful criticism.
A minimal example

The following example Search-WS Description describes a web page and how to extract the MODS records it contains.

```
01 <sws>
02  <op>
03   <request href="http://some.host/mods-records.xml"/>
04   <response>
05    <set name="items">
06     <xpath select="/mods"/>
07    </set>
08  </response>
09 </op>
10 </sws>
```

A Search-WS description comprises a series of Operations (the `<op>` element) each of which is comprised of a Request part and a Response part (the `<request>` and `<response>` elements.)

The Request part describes how to construct a URL and the Response part describes how to extract useful content from the HTTP response.

The above example describes one operation. There will be a single HTTP request to the url contained in the href attribute in line 03. The response section uses an XPATH expression (line 06) to extract the MODS records from the HTTP response. The MODS records are placed in a variable called “items” (line 05.)

The expectation of the writer of the description is that the HTTP response document is a valid XML document (otherwise the XPATH expression will fail to retrieve any records.)

A Search-WS Description on its own is no use without a programming interface that will perform the operations and make available the content parsed from the responses. The following example of Perl code, shows how such a programming interface would work with the above description.

```
01 use SWS;
02 my $sws = new SWS (filename => 'description.sws');
03 $sws->search ();
04 foreach $mods ($sws->var ('items')) {
05  # use the MODS records in $mods
06 }
```

Line 03 instructs the API to perform the operations defined in the description. The search method returns once all the operations have been performed and all responses have been parsed. The var method call in line 04 gives access to the “items” variable which contains the MODS records parsed from the response.

An equivalent C++ code snippet might be as follows.
The Request element

The href attribute of the request element is actually a url template. Character sequences bracketed with curly braces are substituted with user supplied data. The following example shows how a user supplied query might be inserted into a url of a database with a simple search interface.

```
<request href="http://some.host/search?q={query}"/>
```

The above example defines a url with a parameter named “query”. If the user supplied data for “query” is “pride and prejudice”, then the resulting url retrieved by the API would be as shown below.

http://some.host/search?q=pride+and+prejudice

A more complex example is shown below. In this example the href attribute contains no templates. Instead the form sub-element of the request element shows that we are interacting with a CGI script that is driven by an HTML form.

```
<request href="http://copac.ac.uk/"
<form action="/wzgw" method="get">
  <param name="au"/>
  <param name="ti"/>
  <param name="any"/>
  <param name="form" value="qs"/>
  <param name="fs" value="Search"/>
</form>
</request>
```

The form element mimics the HTML form driving the CGI script. The param elements mirroring the input elements of the HTML form. Using the previous example, if the user supplied data for the “au” parameter was “austen” and that for the “ti” parameter was “pride and prejudice”, the the resulting URL created by the API would be as follows:

http://copac.ac.uk/wzgw?au=austen&ti=pride+and+prejudice&form=qs&fs=Search

Note that as no user data was supplied for the “any” parameter it has not been inserted into the url. Also note that the “form” and “fs” parameters have taken their values from data supplied in the “value” attributes of the “param” elements.
The Response element

The response element contains a series of sub-elements that describe how to parse data from the HTTP response and save that data in variables.

Variables

All data parsed from the HTTP responses are stored in variables. The variables are arrays of strings and new values can be appended to the end of the array as more data is retrieved. Three actions have been defined to manipulate the variables: “set”, “append” and “increment”. These actions are defined with the `<set>`, `<append>` and `<increment>` elements respectively. Each of the elements has a “name” attribute which names the variable that is to be modified.

We have already seen the “set” element used in the first example above:

```xml
<response>
  <set name="items">
    <xpath select="/mods"/>
  </set>
</response>
```

Should the response document have contained ten MODS records, then after after the response has been parsed the “items” variable will contain ten values, each value being a single MODS record. Any values previously stored in the “items” variable are lost as the result of the “set” action.

The append action is used to append new values to the variable (i.e. previously stored values are retained.) The following example is used to append new values to the “items” variable (it is not an error to append values to an otherwise empty variable.)

```xml
<response>
  <append name="items">
    <xpath select="/mods"/>
  </set>
</response>
```

If the set or append elements have the optional “value” attribute then the content of the attribute is used to modify the variable (and no data is parsed from the response for the action.)

```xml
<set name="page_number" value="1"/>
```

The above example sets the “page_number” variable to the single value “1”.

The “increment” action is special in that it only modifies the first value stored in a variable and that value is treated as an integer. It is used as shown below.

```xml
<increment name="page_number"/>
```
As a result of the above increment action the “page_number” variable will have one added to its first value. Should the variable have contained multiple values, then only the first value is changed. An optional increment can be specified with the “increment” attribute:

```xml
<increment name="record_number" increment="10"/>
```

**Parsing**

Currently there are two ways of parsing data from the response, either through an XPATH expression or through a regular expression. (It is intended to define other methods in the future.)

**Parsing with xpath**

The xpath element has the attributes shown in the following table. Only the select attribute is mandatory.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Content</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>select</td>
<td>XPATH path</td>
<td>The path used to extract elements from the response.</td>
</tr>
<tr>
<td>string-value</td>
<td>“yes” or “no”</td>
<td>If this attribute is set to “yes” then the content of the matching element is returned as a string.</td>
</tr>
</tbody>
</table>

**Parsing with regexp**

The regexp element has the attributes shown in the following table. Only the regexp attribute is mandatory.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Content</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>regexp</td>
<td>regular expression</td>
<td>A Perl regular expression.</td>
</tr>
<tr>
<td>occurrence</td>
<td>integer</td>
<td>If this attribute is not specified then the regular expression is repeatedly matched against the response, potentially parsing multiple values from the response. If it is specified then only the occurrence’s value is returned as a value.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Content</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>sub-var</td>
<td>integer</td>
<td>If the regular expression contains bracketed constructs then the value returned is that matched by the sub-var'th brackets. If the regular expression contains brackets and the sub-var attribute is not specified, then the value returned is that of the first bracketed construct. If there are no bracketed constructs, then the entire string matched by the regular expression is returned.</td>
</tr>
<tr>
<td>ignore-case</td>
<td>“yes” or “no”</td>
<td>Defines whether or not the regular expression is case sensitive.</td>
</tr>
</tbody>
</table>

**Parsing examples**

The following two examples show how to extract from an SRU response the number of records found by a query.

```xml
<set name="number_of_records">
  <regexp regexp="&lt;zs:numberOfRecords&gt;(.*?)&lt;/\"/>
</set>

<set name="number_of_records">
  <xpath select="/zs:searchRetrieveResponse/zs:numberOfRecords" string-value="yes"/>
</set>
```

Note that the angle brackets in the regular expression have to be escaped to survive XML parsing and that the regular expression uses the Perl minimal match operator “*?”.  

```xml
<append name="items">
  <xpath select="/modsCollection/mods"/>
</append>
```

**Iterations and Conditions**

It might be that repeated HTTP requests need to be made to retrieve the desired number of, say, records. The Description Language allows for this with its `<while>` element. The following example shows a “while” loop that repeatedly requests records from an hypothetical server until the rec_num variable is equal to 10. As records are parsed from the HTTP responses they are accumulated in the “records” variable.

```xml
<while test="rec_num != 10">
  <op>
    <request href="http://some.host/search?resultset=1&amp;rec={rec_num}"/>
  </response>
  <xpath select="/modsCollection/mods"/>
  <increment name="rec_num"/>
</while>
```
The above example assumes several things:

• that the rec_num variable has already been initialised to an appropriate value.
• each request returns a single MODS record.
• that the response is a valid well-formed XML document with a “mods” root element.

An operation can be conditionally performed with the `<if>` element. In the following example an operation is only performed if some records were found as a result of a previous operation.

```xml
<if test="number_of_records &gt; 0">
<op>
...
</op>
</if>
```

Note that the `<while>` and `<if>` elements can contain any number of operations as sub-elements (the operations being performed sequentially.)

**The test attribute**

When a variable is named within a test expression it always the first value stored within the variable that is evaluated. Moreover it is currently only evaluated as an integer. Only simple integer comparisons are allowed using the operators: <, >, <=, >=, != and =.

The test attribute can check how many values are stored in a variable using the following special syntax.

```xml
<while test="records.size != 10">
...
</while>
```

Appending “.size” to the name of a variable in an test evaluates to the number of values stored within the variable.

**Future developments**

**Parameter semantics**

In the case where several databases are needed to be searched together we need to consider the semantics of the request url parameters more carefully. Two databases may both offer searching by author and title, but have very different naming conventions for the HTML form fields. This means we need to be able to indicate the semantics behind the various fields within the form. We might write the request parts of the operations for the two database as follows.
If the API is told that the “author” is “austen” and that the “title” is “pride” then the following two urls would be constructed.

http://host.xxx/query?au=austen&ti=pride
http://host.yyy/search?name=austen&title=pride&func=search

Databases with a single field interfaces need special consideration as all search terms need to be channelled into the one field.

http://host.zzz/search?q=austen+pride

It is not proposed that a formal systems of semantics will be defined. It is left to the Description authors to work out what semantics are appropriate to their own use cases.

**CQL queries**

SRU uses CQL to carry its query in the url. While simple curly bracketed parameter substitution will suffice for very simple CQL queries, it will not do for anything more than a simple keyword search. Therefore it is proposed that CQL/SRU be handled in the following manner:

```xml
:request href="http://host.xxx/">
 <form action="/query" method="get">
   <param name="au" semantics="author"/>
   <param name="ti" semantics="title"/>
 </form>
 </request>

:request href="http://host.yyy/">
 <form action="/search" method="get">
   <param name="name" semantics="author"/>
   <param name="titl" semantics="title"/>
   <param name="func" value="search"/>
 </form>
 </request>

:request href="http://host.zzz/">
 <form action="/search" method="get">
   <param name="q" greedy="yes"/>
 </form>
 </request>
```
Additional information will need to be provided to the API to map the user’s query term semantics to CQL indexes.

**Parsing mechanisms**
Currently only regular expression and XPATH paths may be used to parse the responses. It may be useful to allow other methods, such as:

- CSS descriptors.
- Line and byte/character offsets.

**Pre- and post-conditions**
Allow operations to specify `<precondition>` and `<postcondition>` sub-elements. A post-condition might be that the number of items found is greater than 0. Should a pre or post-condition fail then all processing is aborted there and then. A pre-condition is tested before any url request is made and a post-condition is tested after a response has been parsed.

**Error handling**
All processing should stop on any error condition. Error conditions may be, amongst other things:

- An HTTP error.
- An XPATH error.
- A regular expression that doesn’t compile.
- A variable is referenced as an integer in a while or if test attribute, yet the variable contains data that is not convertible to an integer.

**Code examples**
A Perl module implementing many of the ideas outlined above can be found from the Search-WS committee page within the OASIS web site.

```perl
01 use SWS;
02
03 my $sws = new SWS (filename => 'copac.sws');
04
05 my %query = (06  'author' => 'essery',
07  'keywords' => '"midland railway"',
08 );
09
10 my @items = $sws->search (@query);
11 print "found ", scalar (@items), " items\n";
12 print "item[0] = ", @items[0], "\n";
13
14 foreach my $item ($sws->var_names) {
15    print "$item = ", $sws->var ($item), "\n";
16 }
```

– 9 –
In the above example, line 03 creates a new instance of the SWS class and will read in the description contained in the copac.sws file.

Lines 05-08 is the query we are going to send to Copac. This defines two query terms with semantics author and keywords.

Line 10 calls the SWS::search() function which performs all the work of:

- constructing and fetching all the urls in all the operations defined within the description.
- parsing the responses to retrieve the records.

The return value of SWS::search() function is the “items” variable (if it has been set by the Description.)

Lines 14-16 prints a list "variables" parsed from the response.

The Description used by the above example code might be as follows:

```xml
<sws>
<op>
  <request href="http://copac.ac.uk/">
    <form action="/wzgw" method="get">
      <param name="au" semantics="author"/>
      <param name="ti" semantics="title"/>
      <param name="any" semantics="keywords"/>
      <param name="form" value="qs"/>
      <param name="fs" value="Search"/>
    </form>
  </request>
</op>
<response>
  <set name="numberOfItems">
    <regexp regexp="&lt;span id="num Hits"&gt;([0-9]+)\&quot;">01</regexp>
  </set>
  <set name="sessionID">
    <regexp regexp="/wzgw\?id=(^[^&]+)" occurance="1"/>
  </set>
  <set name="resultSetName">
    <regexp regexp="&amp;amp;rsn=([0-9]+)" occurance="1"/>
  </set>
  <set name="isSorted">
    <regexp regexp="; sorted by:"/>
  </set>
  <set name="sortOrder">
    <regexp regexp="; sorted by: ([^&lt;]+)"/>
  </set>
</response>
<if test="numberOfItems &gt; 0">
  <op>
    <request href="http://copac.ac.uk:8080/wzgw?rsn={resultSetName}&amp;amp;format=XML+-+MODS&amp;amp;id={sessionID}&amp;amp;fs=Download+records"/>
    <response>
```

– 10 –
<set name="items">
  <xpath select="/modsCollection/mods"/>
</set>
</request>
</op>
</if>
</sws>

Note that the Perl module takes no notice of XML namespaces. I'm not sure if this is an bug or a feature.