A Query Engine for Web Service Compositions

Nicoleta Preda

The sources for this class are to be found at the address: Sources

The vision is to create a framework where adding a new Web service requires a minimum effort. To this purpose, Web services will be described in XML documents that conform to the same predefined schema. The description should contain the information about how to compose Web calls (URL) and the signature of the function (input and output parameters). An example of a document describing a Web service is given in Section 2.

For uniformity, the call results of every Web service should be translated to tuples that conform to the signature of the respective Web service. In this way, an application aiming at combining and integrating data from a set of Web services will reason in terms of tuples conforming to the signature of the respective Web services (as opposed to dealing with the actual XML documents). Section 3 shows an example of a transformation function (.xsl script) for the Web service described in Section 3.

Contents

1 Which Web Service APIs?

To get familiar with Web service APIs, you can either search for Web services using a Web browser and/or you can examine the Web site: `http://www.programmableweb.com`. Search for instance APIs from the domains: music, movies, and books.

For this class, you need to select one Web service API and to provide descriptions for at least 3 Web services. Please choose a highly available API. For instance, you may consider the Music Brainz Web service API (version 2). Ideally, you should consider 3 Web services that can be composed in order to answer more complex queries.

2 A language for describing Web Services

.

In our vision a Web Service (WS) is a function that consists of a name, input parameters and output parameters. A WS call requires values (bindings) for the input parameters and returns values for the output. The signature of a WS is an expression of the form $f^{adornment}(x_1, x_2, \ldots x_n)$ where f is the name of the function, $x_1, x_2, \ldots x_n$ are the input and the output parameters (variable names) and ^{adornment} is a sequence of letters i, and o, corresponding to every parameter in the definition and which specifies the kind of the respective parameter: whether is an input or an output. **Important:** We consider that the values of the input parameters can be changed by the function. Hence, the input parameters are also output parameters. We say that they are input-output parameters.

Example. As example, we will consider a Web service from a deprecated API. Let's consider the service that given the name of an artist, returns the `id` of that artist, as defined at the Web site.The call to this service for the input singer Enya is the URL: `http://musicbrainz.org/ws/1/artist/?name=Enya`. Try it in a browser.

Let `mb_getArtistInfoByName` be the name that we give to that service. Note that the function receives as input the the name of the artist and returns their id, the date of birth and the date of death. Abstractly, the signature of this WS can be defined as follows `mb_getArtistInfoByName`^{iooo}`(?artistName, ?artistId, ?beginDate, ?endDate)`.

The WS description. We describe abstractly a WS using a WS document. For our WS let the document be `mb_getArtistInfoByName.xml`. The abstract description of the WS can be encoded as follows:

mb_getArtistInfoByName.xml: Function Signature

```
<prefix name="w" value="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
<prefix name="y" value="http://mpii.de/yago/resource/"/>
   <!-- variables in the head: the order matters:
        the first variables should be the input variables, followed by the output
            variables
        also, for the input variables the order should match the order in the URLs of
            the calls -->
   <headVariables>
       <variable type="inout" name="?artistName"/>
       <variable type="output" name="?artistId"/>
       <variable type="output" name="?beginDate"/>
       <variable type="output" name="?endDate"/>
   </headVariables>
```

The document `mb_getArtistInfoByName.xml` should also store the information that is necessary to generate the calls (the URLs) to the Web service. We note that such a URL consists of a concatenations of constant strings and values of the attributes. For instance, in our examples it consists of a constant part (`http://musicbrainz.org/ws/1/artist/?name=`) followed by the value of the input variable `?artistName`. This metadata can be encoded, as follows:

mb_getArtistInfoByName.xml: descriptions of the calls from

```
<call>
 <part type="constant" value="http://musicbrainz.org/ws/1/artist/?name="/>
 <part type="input" variable="?artistName"  separator="+" />
</call>
```

Finally the `.xml` description should link to the file with the transformation function.

mb_getArtistInfoByName.xml: pointer to the transformation function

```
<transformation file="mb_getArtistInfoByName.xsl"/>
```

3 XSLT Transformation Function

.

mb_getArtistInfoByName.xsl

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Created by Clement on 090524 -->

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:mb="http://musicbrainz.org/ns/mmd-1.0#">
<xsl:template match="/">
<RESULT>
    <xsl:for-each
    select="*[local-name()='metadata']/*[local-name()='artist-list']/*[local-name()='artist']">
          <RECORD>
             <ITEM ANGIE-VAR='?artistName'><xsl:value-of select="mb:name"/></ITEM>
             <ITEM ANGIE-VAR='?artistId'><xsl:value-of select="@id"/></ITEM>
             <ITEM ANGIE-VAR='?beginDate' ><xsl:value-of
                 select="mb:life-span/@begin"/></ITEM>
             <ITEM ANGIE-VAR='?endDate'><xsl:value-of
                 select="mb:life-span/@end"/></ITEM>
  </RECORD>
    </xsl:for-each>
</RESULT>
</xsl:template>
</xsl:stylesheet>
```

4 Java code

Online, on the Web page of this course you may find the following resources:

- `WS-Evaluation` - a directory that has the following structure:
  - `ws-definitions` - a directory that contains function descriptions and the transformation functions that are associated to them. For a function (WS) called f, the directory will contain (1) a file labeled with f*.xml* that consists of the xlm description of the Web service operation, (2) a file f*.xsl* representing the XSL transformation function that is to be applied on the call results in order to produce tuples.
  - for each function f described in `ws-definitions`, a directory with the name of the function. In turn, each directory had two subdirectories:
    - `call_results` a directory that will cached results (the Java program will save here the call results)
    - `transf_results` a directory that will contain for each call result `call_results`, an `.xml` file with the results of the xls transformation.

  Nota Bene: For every new WS that you introduce in the system, you have to create a directory `WS-Evaluation/<name_of_your_WS>`, and its two sub-directories: `call_results` and `transf_results`.

- `ws-execution`: a Java project that is able to perform the following operations:

1. load and parse WS descriptions;
2. execute calls for loaded WSs $\rightarrow$ the call results are stored in a file, on the disk depending of by WS's name and of the values of the inputs;
3. given a WS and a call result, it can invoke the transformation function $\rightarrow$ the transformation results are stored in a file on the disk depending of by WS's name and of the values of the inputs;
4. a parser that "reads" the results from the XML-like encoding produced by the transformation functions. Important: note that there is no need to re-write this parser for each WS. If your transformation functions produce XML documents with the same structure as the transformation function that I give as example, then this parser can be used to "read" those results as well.

5 Web Service Compositions

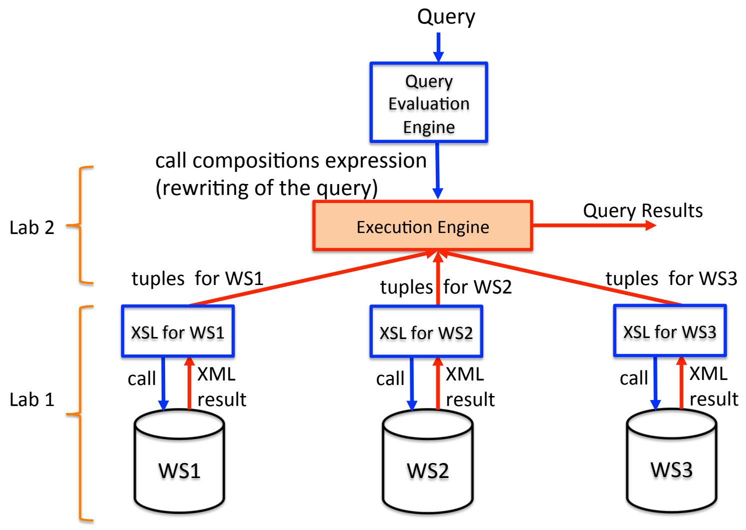

Figure 1: Overview of the Query Evaluation Engine

The goal is to develop an execution engine that is able to execute call composition plans. The algorithm will take as input an expression representing a call composition. The execution engine could be used for instance for executing the plans produced for some given used quert by some query reviewing algorithm. Figure 1 shows overall architecture of the query evaluation engine.

Let's consider again the WS given as example in the last class:

$$\mathsf{getArtistInfoByName}^{iooo}\ (\mathsf{?artistName,\ ?artistId,\ ?beginDate,\ ?endDate})$$

We know that we can count on the transformation function to give transform the results of call results into tuples of the relation getArtistInfoByName. Assume now that, two other WS are defined, namely:

$$\mathsf{getAlbumByArtistId}^{ioo}(\mathsf{?artistId,\ ?beginDate,\ ?albumId,\ ?albumName})\text{ and}$$
$$\mathsf{getSongByAlbumId}^{iooo}(\mathsf{?albumId,\ ?singerName,\ ?songTitle,\ ?year})$$

Intuitively, a call composition is a conjunctive query consisting of an ordered sequence of atoms. As example, consider the following expression:

$$\mathsf{getArtistInfoByName}^{iooo}(\text{"}\mathsf{Frank\ Sinatra}\text{"},\ \mathsf{?id,\ ?b,\ ?e})$$
$$\#\ \mathsf{getAlbumByArtistId}^{ioo}(\mathsf{?id,\ ?b,\ ?aid,\ ?albumName})$$
$$\#\ \mathsf{getSongByAlbumId}^{iooo}(\mathsf{?aid},\ \text{"}\mathsf{Frank\ Sinatra}\text{"},\ \mathsf{?title,\ ?year})$$

The first expression ($\mathsf{getArtistInfoByName}^{iooo}(\text{"}\mathsf{Frank\ Sinatra}\text{"},\ \mathsf{?id,\ ?b,\ ?e})$) denotes a single function call. The second expression ($\mathsf{getAlbumByArtistId}^{ioo}(\mathsf{?id,\ ?b,\ ?aid,\ ?albumName})$)

denotes a set of function calls, one for each input $?id$ obtained as answer of the first expression. The third expression (getSongByAlbumId^{iooo}(?aid, "Frank Sinatra", ?title, ?year)) denotes a set of function calls, one for each input ?aid.

Definition 1 (Workflow (Plan)) *A workflow is a sequence of function calls. Each argument in the workflow has to be either (1) a constant symbol that appears in Q or (2) a variable.*

The workflow is *admissible* if, for every variable, the first occurrence of the variable in the call sequence is in an o-position of a function call. Let us look again at our sample plan above. The plan is admissible because every i-argument of a function call is either a constant or has been bound (was given a value) by a previous function call.

Execution. The execution of the expression is left to right, step by step. At each step we join the results of two expressions function calls and obtain a new table. For our example, we have 3 function call expressions, hence we will have two joins. We first compute the results for

getArtistInfoByName^{iooo}("Frank Sinatra", ?id, ?b, ?e)
# getAlbumByArtistId^{ioo}(?id, ?aid, ?albumName)

This will lead to a table partialResults1^{oooo}("Frank Sinatra", ?id, ?b, ?e, ?aid, ?albumName). Note that every variable in occurring in the two expressions appears once in the head of the new table. Then, we have to join:

partialResults1^{oooo}("Frank Sinatra", ?id, ?b, ?e, ?aid, ?albumName)
# getSongByAlbumId^{iooo}(?aid, "Frank Sinatra", ?title, ?year)

The result is a table: partialResults2^{oooo}("Frank Sinatra", ?id, ?b, ?e, ?aid, ?albumName, "Frank Sinatra", ?title, ?year).

Joins. The first call will result in a (small) table of tuples. This table will provide the inputs for the function calls of the second expression. Hence we have a join between the results of the first calls and the results provided by the second call. A join should also be performed when the same variable is marked as output variable for two different Web calls of a composition. For instance, assume that the WS getAlbumByArtistId returns also the birthday of the artist. Then the call composition

getArtistInfoByName^{iooo}("Frank Sinatra", ?id, ?b, ?e)
# getAlbumByArtistId^{ioo}(?id, ?b,?aid, ?albumName)

is equivalent to

getArtistInfoByName^{iooo}("Frank Sinatra", ?id, ?b, ?e)
# getAlbumByArtistId^{ioo}(?id, ?b1,?aid, ?albumName)
# ?b=?b1

Selections. Note that in the third first and in the third expression, a constant ("Frank Sinatra") occurs on the place of an input-output and on the place of an output variable. In both cases, a selection should be performed. The algorithm should select only the tuples where the variable on that position are equal to ("Frank Sinatra").

6 Assignment

Implement an execution engine for call compositions. The execution engine receives as input an expression such as

$$\mathsf{getArtistInfoByName}^{iooo}(\text{"}\mathsf{Frank\ Sinatra}\text{"},\ \mathsf{?id},\ \mathsf{?b},\ \mathsf{?e})$$
$$\#\ \mathsf{getAlbumByArtistId}^{ioo}(\mathsf{?id},\ \mathsf{?aid},\ \mathsf{?albumName})$$

The engine should be able to

1. parse the expression $\rightarrow$ get the list of functions to be called (for simplicity, you can use # to separate calls. It's will be easier to parse the expression)
2. execute the calls in the order described by the composition
3. output the results