KTH V17P01 Programming Web Services: Project

Fannar Magnusson (fannar@kth.se)
Thorsteinn Thorri Sigurdsson (ttsi@kth.se)

March 6, 2017

1 Introduction

This project focuses on writing two kinds of matchers for matching output of one web service with input of another web service. The matching can be syntactic, where we use the Edit Distance algorithm to find linguistic similarity between words, or it can be semantic, where we use ontology to find matching elements.

2 Parsing

We were provided with WSDL files that we needed to match. Our first task was to parse those files, flatten them out and store all the basic types each operation has. We started out by trying to use XPath for parsing the WSDLs. It has a low level syntax and gives you a good control of navigating through elements and attributes in the document. However, we were having trouble with it when we were trying to parse broken documents or complex documents with namespaces. We ran into the most trouble dealing with the fact that some of the given WSDL files had namespace prefixes while others didn't. Eventually we decided to switch over to using the Predic8 library[1] for doing the parsing. Predic8 has a lot of built in functions for retrieving schemas, services, elements and attributes from WSDL files and is easy to use.

When we run the matchers we start by going through all the WSDL files and store their service name, ports, portTypes, operations, messages, elements and sub-elements in Java container objects. It then depends on the matcher how we compare those elements and calculate the matching degree.

2.1 Invalid WSDL files

We were not able to parse all of the given WSDL files. Some of them were broken, and some of them were not WSDL files at all, but rather HTML

[1] `http://www.membrane-soa.org/soa-model/`

files (e.g. `BangoDirectBillingAPIProfile.wsdl`).

3 Syntactic Matching

After we have parsed the WSDLs and stored all elements and sub-elements in containers, we loop through all services and their output operations. For each output operation we compare it to every input operation for each of the services. More specifically, we compare the elements of these operations and calculate their matching degree using the `EditDistance.getSimilarity()` function. If we get a distance that is equal or higher then 0,8 we have a match and store it.

Finally when we have looped through all the service containers and stored all matches, we generate a `SyntacticOutput.xml` file using JAXB from the `WSMatching` parent object that stores all the matched elements and other relevant information such as the WSScores and OpScores.

4 Semantic Matching

Like with the syntactic matcher we begin by parsing the documents with the parser and then run the comparer. The only difference between the matchers is how we compare the elements and calculate the matching degree. For the semantic matcher we don't consider sub-elements. We simply retrieve the message parts, find their names and types, and lookup their ontology reference from the document's schema.

The Predic8 library did not seem to be able to extract the `sawsdl:modelReference` attribute from the SAWSDL files. We therefore used XPath to fetch the attributes. The attributes are of the format

```
sawsdl:modelReference="http://127.0.0.1/ontology/books.owl#Author
```

We fetch this attribute and use the string on the right side of the hashtag (in this case "Author") as the input for the `getMatchingDegree()` function.

With the message parts ontology references, we can compare two parts/elements with the help of the `MyOntManager` and `Reasoner` classes that we were provided with. We simply create `OWLClasses` from the ontology references and then use the OntManager and Reasoner to check if the classes are equal (matching degree 1,0), are sub-classes of each other (matching degree 0,8 or 0,6) or if there is a relation between them (matching degree 0,5). If we get a matching degree that is equal or higher than 0,5 we have a match and store it. We used the Protege[2] tool to explore the ontology and verify that the semantic matcher was giving correct results.

Finally, like with the syntactic matcher, we loop through the container and use JAXB to generate a `SemanticOutput.xml` file.

[2] `http://protege.stanford.edu/download/protege/4.0/`

5 Output XML

The output of the program are two XML documents, one for the syntactic matcher and one for the semantic matcher, that conform to the given schema, `Output.xsd`.

An example of the output from the semantic matcher can be seen in figure 1.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<tns:WSMatching xmlns:tns="http://www.kth.se/ict/id2208/Matching"
    xmlns:ns2="http://www.w3.org/2001/XMLSchema-instance"
    ns2:schemaLocation="http://www.kth.se/ict/id2208/Matching Output.xsd ">
    <tns:Matching>
        <tns:OutputServiceName>LuxuryhotelService</tns:OutputServiceName>
        <tns:InputServiceName>AgentmovementService</tns:InputServiceName>
        <tns:MatchedOperation>
            <tns:OutputOperationName>get_LUXURYHOTEL</tns:OutputOperationName>
            <tns:InputOperationName>getFuturePosition</tns:InputOperationName>
            <tns:OpScore>0.6</tns:OpScore>
            <tns:MatchedElement>
                <tns:OutputElement>_LUXURYHOTEL</tns:OutputElement>
                <tns:InputElement>Agent</tns:InputElement>
                <tns:Score>0.6</tns:Score>
            </tns:MatchedElement>
        </tns:MatchedOperation>
        <tns:WsScore>0.6</tns:WsScore>
    </tns:Matching>
    ....
```

Figure 1: Example of an output file from the semantic matcher