Project

Mattias Cederlund, mcede@kth.se

Task 1

For Task 1, a program that takes two web services WSDL files and does syntactic matching between the inputs of the first with the outputs of the second service should be implemented. The result of the matching should be exported to an XML file of provided format.

The first thing I had to implement was parsing of WSDL files, for which I downloaded the XML Schema for WSDLs from http://schemas.xmlsoap.org/wsdl/ and used JAXB to create their respective Java classes.

When I had the WSDLs parsed to Java classes I extracted the operations from PortTypes, all the messages and also the Types (containing schemas for referenced types in message parts).

I then proceeded by iterating all operations, found their input and output messages and the message parts. In order to get the parts I sometimes had to find the referenced elements or types in the Types (schema). This was probably the hardest part in the entire task.

In order to parse the schemas I identified three types of references, references to elements, complexTypes and simpleTypes. So I iterated the schema, which I got in DOM-form, and put all elements (and their nested elements if they contained a complexType) into one HashMap, the complexTypes (and its nested elements) and simpleTypes were parsed in a similar way and put in another HashMap.

Then, when a message referred to an element or a type I had the two lookup HashMaps where I easily could find the referred message parameters. Although, I did not match parameters that were more than one reference away, so nested references were not handled.

After matching all message parts with their referenced elements and types I was left with a list of operations containing their name, input parameters and output parameters.

This list was generated for each web service and passed two and two to the matching algorithm. The algorithm took the operations from the first operation and found the best matching operation from the second operation. In order to do so, each input parameter from the first operation were compared to the output parameters of the second operation and the best matching parameters were chosen. When all parameters were matched it would compare the matching to the previously best matched operation and chose the best one. This resulted in a list of matched operations where all input operations were matched to exactly one output operation, while output operations could be matched to many input operations. The same thing applies to the matching of parameters.

As a possible improvement the algorithm could take already matched operations and parameters into consideration, so no input operation (or parameter) were matched to many output operations (or parameters) in the end.

The result were then exported to an XML file. As the output XML Schema was provided, I used JAXB for generating the output as well.

Task 2

For task 2, Operation score should be calculated from the average of the element score and Service score should also be calculated from the average of the Operation score. Since the matching was already in place, adding these scores was trivial. In fact, I implemented Task 1 and Task 2 at the same time. The algorithm needed to calculate the averages in order to find the best matching anyway.

After adding the scores I sorted the output XML file in Service score order.

Task 3

For Task 3, the Web services should be matched using ontology instead of syntactic matching. The first thing I needed to add was the parsing of ontology attributes in the message parts and schemas. As the main parsing structure already was in place, it was just a matter of parsing an additional attribute, which was trivial.

To enable two different matching strategies while reusing most of the code I made the Matcher class (containing the parsing and matching logic) abstract and sub-classed it by creating two classes, SyntacticMatcher and SemanticMatcher. Each of these classes implemented their own method, `getMatchingScore(inputParameter, outputParameter)`, which calculated the matching score accordingly.

The SyntactigMatcher calculated the matching score based on edit distance. The SemanticMatcher used the ontology attributes to get the proper OWLClasses from the OntologyManager. Then they were compared and graded by the matching degrees that were provided.

Running instructions

The project is provided as an Eclipse project, in the file Project.zip. Unzip it and import it to Eclipse by navigating File → Import... → General → Existing Projects into Workspace → Next → Browse → Find the project folder → Ok → Finnish.

In order to make the semantic matching work, you need to provide an absolute path to /data/SUMO.owl. This should be defined in the constant ***OWL_PATH*** of SemanticMatcher.java.

Now you should be able to run the project by running Performer.java. The output generated from the program will be in /xml/Output.xml (syntactic) and /xml/OutputOntology.xml (semantic).