Implementation of a template system with semantic web services

Adan Scotney[⋆] (*a.scotney@bath.ac.uk*)

University of Bath

Abstract. OWL-S supports the concept of composite services. In these services, a data flow is defined to allow execution of multiple services as a single one. However, the processes which make up this composite services are static. Once the flow has been defined, that is final. This tool, developed as part of the ALIVE [1] project, allows one to ground templates, composites where the concrete implementation of a particular process is undefined, to form an invokable composite service.

1 Introduction

OWL-S [2] composite services provide a way of combining multiple services into a single one. Ordinarily, the services which make up the composite are fixed, but using the concept of templates, it is possible to replace the processes which make up a composite, while leaving the data flow in tact.

The concept of a template is simple: a composite service where one or more of the sub-processes are not invokable. These sub-processes are called slots if they have inputs and outputs (bound to inputs and outputs in the outer composite as normal), but no concrete process is pointed to.

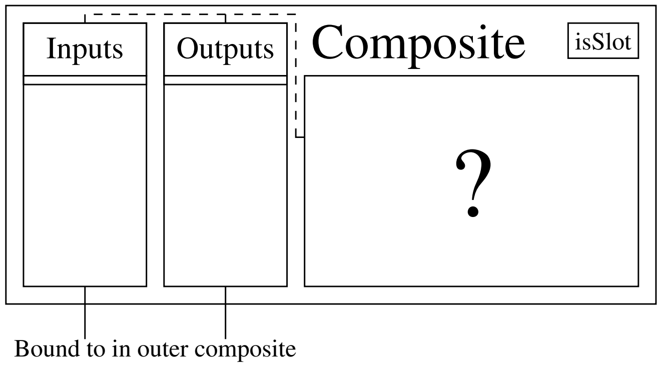

Fig. 1. A slot

In this implementation, slots are themselves composite processes, which are sub-processes of the outer composite. They have inputs and outputs bound to the outer process, a slot marker, and an empty ControlConstruct.

⋆ This work is supported by the IST ALIVE project

When grounding a slot, a Perform is built within the slot process, which contains the grounded process, and bindings are then built to this from the slot inputs and outputs, allowing it to be invoked as if the parameters passed between the slot and outer composite were in fact passed directly to the grounded process.

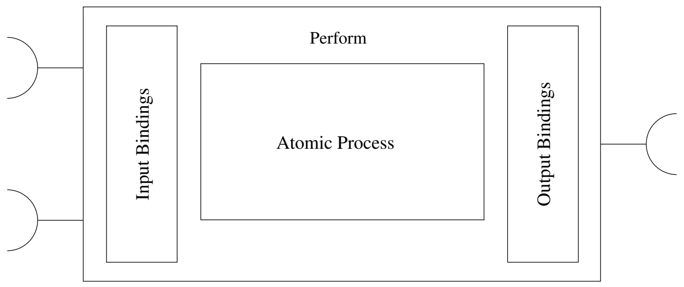

Fig. 2. The constructed perform which is inserted into a slot

In addition to simply grounding templates, the API also provides a mechanism for accessing the ALIVE matchmaker to discover candidates, and another for setting constraints upon the services which can be selected by using ASP (Answer Set Programming) [3].

2 Creating a template

When creating a template, there are three properties which are defined in the included templates ontology (`http://www.ist-alive.eu/ontologies/ALIVE-Template.owl`) that must be used. These are as follows:

– `isSlot` on `CompositeProcess` : Datatype property containing a string which is the identifier of the slot. This must be a valid ASP name (i.e. lower case, no spaces). A composite process which has this property is considered a slot.
– `aspCode` on `Service` : Datatype property containing a string, which is used to select possible services with ASP. See the ASP section below.
– `aspBindings` on `Service` : Datatype property containing a string for user-defined ASP bindings. See the ASP section below.

To create a template, simply create a service as described above, then assert each oh these properties where appropriate. There should only be one `aspCode` and one `aspBindings` on the Service, and at least one `CompositeService` as a sub-process of the main service which is labelled with `isSlot`.

This proved to be the most workable approach, after attempts were made to use atomic processes as slots, directly swapping them out and rebuilding the bindings; and also to make the replacement service and parameters `owls:sameAs` those of the slot.

3 API Use

3.1 API Configuration

The API uses some Java properties which define how it generates answer sets. These are ordinarily contained within the `alivetemplates.properties` file, however there are forms of the `TemplateConstructorImpl` constructor which take a Properties object as a parameter, allowing one to use the API where it isn't possible to determine where the file should be located.

- `clingopath` : Path to the clingo binary.
- `clingoargs` : Arguments to pass when invoking clingo (number of answer sets generated can be limited here).

It is also possible to override the path of the properties file by setting the `aspconfig.file` system property.

3.2 Interface

The API is accessible through an interface, `TemplateConstructor` of which `TemplateConstructorImpl` is the included implementation. Full details of the interface can be found in the Javadocs, however it will be briefly outlined here.

- `List<Process> findTemplateSlots()` Helper method to find all slots.
- `Map<Process, Collection<Match>> getCandidates(List<Process> slotProcesses)` Invokes the matchmaker on a list of slots, and returns a Map from slots to a collection of matches (services which have compatible inputs and outputs).
- `Program getGeneratedASP(Map<Process, Collection<Match>> candidates)` Fetches the answer set program which will be used to select which processes can fit into each slot. This can be used to add additional assertions outside of the API should you wish.
- `List<TemplateAnswerSet> getAnswerSets(Program aspProg)` Runs an answer set program and returns the sets of processes which can be used, along with the original model returned from clingo, allowing one to extract additional information which the API doesn't look for.
- `OWLOntology performReplacement(Map<Process, Process> replacements, URI newServiceURI)` Returns an ontology containing the template as a service grounded with the specified URI.

There are some additional methods which can be found in the Javadocs, but these are the core ones required to use the API.

3.3 Example

A short example of the way in which one may typically use the API.

```
TemplateConstructor builder = new TemplateConstructorImpl(
   new File("test/newcomp.owl").toURI(),
   URI.create("urn:test-composite-ont#MasterService"), matchMaker);

// Run template
List<Process> slotList = builder.findTemplateSlots();
Map<Process, Collection<Match>> candidateMap = builder.getCandidates(slotList);
Program aspProg = builder.getGeneratedASP(candidateMap);
List<TemplateAnswerSet> selectionMaps = builder.getAnswerSets(aspProg);
OWLOntology ont = builder.performReplacement(selectionMaps.get(1).getMapping(),
   URI.create("urn:test-composite-ont#GroundedMasterService"));
// Store result
ont.write(new FileOutputStream(new File("groundedTemplate.owl")),
   URI.create("urn:alive-templates"));
```

4 ASP

4.1 Outline

The templates API allows the programmer to use an answer set program, run through clingo, to generate sets of valid groundings for the the slots. It is possible to generate sets based on the selection of other services (i.e. dependencies), as well as based on the presence of properties on an individual, and in the case of data properties, the value of the property.

ASP code is placed in the data property `aspCode` on the `Service` for the template, and is executed when calling the `getAnswerSets` method. This code will make use of the assertions generated by the API (which are detailed in the next section), and indicates the decision to ground a slot with a particular process using the `selection(slotID, processID)` atom.

All slots must be grounded in all answer sets generated.

To bypass the issue of certain data values being impossible to represent in ASP, URIs and data are hashed. One can look up the originals post execution using `URI lookupASPNameHash(String name)` and `Object lookupASPDataHash(String hash)`.

Slots are referred to by the name specified in the `isSlot` property on their composite process.

There is a provision for generating assertions based on the actual values of data: this has its own section below.

Example

```
1{selection(Slot, Y): candidateProcess(Slot,Y)}1 :-  slot(Slot).
```

This very simple piece of code generates an answer set for each possible replacement for each slot.

4.2 Assertions

The templates API generates a set of assertions which are always present irrespective of any bindings. These state basic facts which can be used to generate valid groundings.

The assertions are as follows:

– `candidateProcess(slot, candidate)` States that `candidate` is a possible grounding for `slot`.
– `slot(X)` States the existence of a slot `X`.
– `service(svc)` States the existence of a Service, `svc`.
– `processOf(process, svc)` States that a Process `process` belongs to a Service `svc`.

4.3 Bindings

The bindings allow a service developer to get the API to generate assertions about properties. The `aspBindings` string contains a representation of a map which instructs it what properties are to be examined.

Entries in the map are separated by new lines, and the entries themselves take the format:
`'atomName': propertyURI`
or
`'atomName': propertyURI {datahash: individualURI(, ...)}`

In the first style, this simply creates an atom with the given name, describing properties with a given URI. This checks all individuals in the knowledge base to see whether they have the given property upon it. If the property is a data value, and the individual has that property on it, the API asserts `atomName(indivdual, dataval)` where `dataval` is a hash of the data value (which can be looked up later if needed as previously detailed). If the property is an object property, the API asserts `atomName(individual, uriOfObject)`.

The second style lets you assign names to data values. Provided the property is a data value, the value of that property on the given individual will be hashed to `datahash`, then all other individuals with that property which have the same value will be hashed to the same name. The same functionality is achievable through ASP, this is just for convenience. This second form still has the same functionality as the first, it is a superset of it.

Example

```
<j.0:aspBindings><![CDATA[
   'hasProfile': http://www.daml.org/services/owl-s/1.2/Service.owl#presents
   'svcName': http://www.daml.org/services/owl-s/1.2/Profile.owl#serviceName \\
 {subservicename: http://aliveservicewrapper.bath.edu/SubtractServiceSimple#SubServiceProfile}]]>
</j.0:aspBindings>
```

In this example, all individuals which have the `hasProfile` property on them will have an assertion made about them e.g.

```
hasProfile(urn_test_composite_ont_MasterService8,urn_test_composite_ont_MasterProfile5).
```

And for `svcName`, the same is true, plus the data value hashing.

```
svcName(urn_test_composite_ont_MasterProfile5,aspdatavaluehash1).
svcName(http_alivetemplates_bath_edu_MultiplyServiceSimple_MulServiceProfile6,subservicename).
svcName(http_aliveservicewrapper_bath_edu_SubtractServiceSimple_SubServiceProfile7,subservicename).
```

4.4 Data value handlers

Using a data value handler allows one to generate assertions in ASP about the actual values of data. A handler is defined by the `ASPValueHandler` interface which has three methods on it which must be implemented by a handler:

- `Collection<URI> typeHandled()`
- `void registerData(Object dataItem, String hash)`
- `void invoke(Program aspProg)`

The handler system works by allowing a user of the API to register handlers, which are then mapped to `rdf:datatypes`. The `typeHandled` method returns a collection of data type URIs that the particular handler can deal with.

When generating ASP assertions using the bindings, the API checks for each instance of a property and hashes it (as usual) but as an extra step, if the data type has one or more handlers associated with it, the `registerData` method for each handler is invoked with the actual data, as well as the hash each item of data has been given.

Once all data has been registered with each appropriate handler, the `invoke` method on each handler is called with the ASP program as a parameter, allowing one to make assertions about the values of the data beyond simple equality.

Two generic handlers are included: `IntegerValueHandler` which generates assertions in the form `isInteger(hash, value)` allowing one to use the functionality provided as part of ASP for dealing with integers natively; and `NumberOrderValueHandler` which asserts an order over all numerical types (though currently only `&xsd;#int`, `&xsd;#float`, and `&xsd;#double` are handled) by a chain of `lessThan(hash, hash)` rules.

One registers value handlers using `TemplateConstructor.addValueHandler(handler)`. Nothing more is required, since each handler carries with it a list of types which it can support (this could be generated dynamically if the handler does not deal with a static list of types).

5 Further Work

None!

References

1. ALIVE Project, http://www.ist-alive.eu/
2. OWL-S Specification, http://www.ai.sri.com/daml/services/owl-s/1.2/overview/
3. Potassco Answer Set Solving Collection, http://potassco.sourceforge.net/