Writing Dynamic Service Orchestrations with DSOL

Leandro Sales Pinto, Gianpaolo Cugola, Carlo Ghezzi
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Milano, Italy
{*pinto,cugola,ghezzi*}@*elet.polimi.it*

***Abstract*—We present the workflow language DSOL, its runtime system and the tools available to support the development of dynamic service orchestrations. DSOL aims at supporting dynamic, self-managed service compositions that can adapt to changes occurring at runtime.**

***Keywords*-Service oriented computing; service orchestration; declarative language**

I. Introduction

Service orchestrations [1] live in a very unstable world in which changes occur continuously and unpredictably. External services invoked by the orchestration may be discontinued by their providers, they may fail, or they may become unreachable or incompatible with the original versions. Furthermore, the orchestration requirements may evolve due to business needs. It is therefore fundamental that orchestration languages and their runtime systems provide simple yet efficient ways to write service compositions that are able to cope with exceptional situations and are also able to easily adapt dynamically to external changes.

The *DSOL* language and its associated runtime system are the result of research [2], [3] to explore an alternative language to current mainstream imperative workflow languages for service orchestrations. DSOL adopts an innovative declarative approach in which an orchestration is modeled in terms of its goals and the actions to reach them. At runtime, DSOL uses planning techniques to determine the actual flow of execution to achieve the orchestration's goals (i.e., which actions to execute and in which order). The same planning techniques are also used in case of faults, to build alternative paths of execution without the need for explicitly programming them.

This eases the job of building flexible orchestrations capable of coping with faults and changes in the external environment, while the modularity and dynamism inherent in such approach also simplifies the steps required to change the orchestration model at runtime.

II. A Walk through DSOL

To introduce the various aspects included in the DSOL orchestration model, we present a simple example scenario. The idea is to design a service to translate an existing Doodle poll into a chosen language. Such a service is built as an orchestration of three existing external services: a service to retrieve information related to the existing Doodle poll, a service to detect the language in which the poll was initially written, and a service to actually translate the text included in the poll. In particular, we consider the following requirements:

1) The composite service shall be invoked with the identification code (id) of the existing Doodle poll to be translated, and the language ($desiredLanguage$) to which the poll should be translated.
2) To detect the language in which the poll was initially written, the system shall use its title.
3) The service shall translate the title, the description and the options of the existing poll and return the translated poll to the client.

The DSOL model of a service orchestration includes different aspects, which are defined separately, possibly by different stakeholders, as shown below.

A. Abstract Actions

Abstract actions are high-level descriptions of the primitive actions available in a given domain, which DSOL uses at runtime as the building blocks to automatically build orchestration plans. They are modeled in an easy-to-use, logic-like language, in terms of their *signature*, *precondition*, and *postcondition*. Listing 1 illustrates the abstract actions that model the poll translator scenario. Note how they reflect all the activities described in the service requirements.

```
action getPoll(Id)
pre: pollId(Id)
post: poll(requestedPoll)

action translatePoll(Poll, PollLanguage, ToLanguage)
pre: poll(Poll), language(Poll, PollLanguage), language(ToLanguage)
post: poll(translatedPoll), language(translatedPoll, ToLanguage)

action detectLanguage(Text)
pre: text(Text)
post: language(Text,detectedLanguage)

seam action getText
pre: poll(requestedPoll)
post: text(requestedPoll.title)

seam action getPollLanguage(Language)
pre: language(requestedPoll.title, Language)
post: language(requestedPoll, Language)
```

Listing 1. The abstract actions for the poll translator example

Although abstract actions are specified in a very abstract way, they are meant to be executed by one or more concrete counterparts that implement it (see Section II-B). Listing 1 also shows the usage of a special kind of abstract action, called *seam action*. Seam actions do not have a concrete implementation and are used in some situations in which is necessary to relate different states of a domain or deduce some new facts. For example, using the seam action *getText*, it is possible to deduce that a poll title is a text, and the action *detectLanguage* could be called using it as an argument.

B. Concrete Actions

Concrete Actions are the actual implementation of abstract actions. They are implemented as Java methods using the ad-hoc annotation `@Action` to refer to the abstract actions they implement. Several concrete actions may be bound to the same abstract action, representing different alternatives to accomplish the same actions.

Concrete actions may be of two different types: *service actions* and *generic actions*. The former are abstract Java methods directly mapped to external services. As an example see Listing 2. The special attribute `service` of the `@Action` annotation is a reference to the external service to invoke. In the example, the abstract action `detectLanguage` will be implemented by invoking the service identified by the reference *languageDetector*. Having just a mnemonic label to reference a service allows us to have a loosely coupled model that could be easily modified, even at runtime. In fact, the actual information about the service (e.g., URI, operation) is specified externally (see Section II-C).

```
@Action(name="detectLanguage", service="languageDetector")
public abstract String detectLanguage(String text);
```

Listing 2. Service action that implements the *detectLanguage* abstract action

Generic actions are ordinary Java methods to be used to perform general operations like retrieving information from a database or pre-processing data between service invocations. Listing 3 illustrates a generic action. This generic action is used to specify which parts of the poll should be translated (title, description and poll), and translates them by calling the *translate* method, which is a service action.

```
@Action(name="translatePoll")
@ReturnValue("translatedPoll")
public Poll translatePoll(Poll poll, String pollLanguage, String targetLanguage){

 Poll translatedPoll = new Poll(poll);
 String title = translate(poll.getTitle(), pollLanguage,
                          targetLanguage);

 String description = translate(poll.getDescription(), pollLanguage,
                                targetLanguage);

 //translate options
 ...
 return translatedPoll;
}
```

Listing 3. Generic action that implements the *translatePoll* abstract action

C. Composed Services

The *to-be* composed services can be defined in two distinct moments. At design time, using a XML file with ad-hoc tags. At runtime, the composed services can be modified using the provided Web interface, or programmatically using the RESTful API exposed for that purpose (see Section III-C).

Listing 4 illustrates the XML definition of two of the services used in our example. The first one represents the service provided by Doodle to retrieve an existing poll and the second one the service provided by Microsoft Translator API to detect the language in which a given text was written. In this example we show the two kind of services supported by DSOL: *SOAP* and *REST*. Both kinds of services must include a *name* and an *id*. The former is used by concrete actions to refer to the service they are linked to. Actually, more than one service can have the same name, and they are used as alternatives[1]. The latter is used to uniquely identify a service description. It is used, for example, to delete programmatically a specific service, if a monitor detects it is unavailable. The meaning of the other attributes present in the service tag is straightforward.

In the case of a REST service, DSOL supports the use of all four HTTP methods (GET, POST, DELETE, and UPDATE). To send and receive messages, *xml* and *json* are supported as media type.

Services may also include arguments that come from the concrete actions they are linked to. Note in the *url* attribute of service *getDoodle*, the usage of $\#\{pollId\}$ representing that this value must be changed at runtime with the actual value of the $pollId$ argument passed to the concrete action. This is useful in the case of RESTful services that usually do not have a formal specification of the method to be invoked.

Concrete actions attached to SOAP services must comply with the expected arguments of the external service. For instance, the service provided by Microsoft Translator API to detect the language of a text expects an argument of type String. So, concrete actions linked to the *microsoftLanguageDetector* service must receive String as an argument. In such a way, services can be automatically invoked without having to generate stub classes, as usually is done for SOAP services.

```
<services>
 <service type="soap"
          name="languageDetector"
          id="microsoftLanguageDetector"
          wsdl="http://api.microsofttranslator.com/V2/Soap.svc"
          operationNamespaceUri="http://api.microsofttranslator.com/V2"
          operation="Detect"/>

 <service type="rest"
          name="poll"
          id="getDoodle"
          url="http://doodle-test.com/api1WithoutAccessControl/polls/#{pollId}"
          method="GET"
          mediaType="application/xml"/>
</services>
```

Listing 4. Services composed in the poll translator example

D. Orchestration Interface

The *orchestration interface* formalizes how the orchestration is exposed as a Web service and it is defined as

[1]Readers are welcome to recall the definition of the service action in Listing 2

a Java interface compliant with the JAX-WS specification, which specifies how Web services are created in Java. The provided annotations are used in mapping Java to WSDL and in controlling how runtime processes respond to Web service invocations. Listing 5 shows the orchestration interface of our example.

```
@WebService
public interface PollTranslator{
@WebResult(name="translatedPoll")
public Poll getTranslatedPoll(@WebParam(name="pollId") String id,
                              @WebParam(name="language") String desiredLanguage);
}
```

Listing 5. The poll translator orchestration interface

The orchestration interface may actually include several methods. For each method, an initial state and a goal must be specified. The *initial state* models the state in which the orchestration starts while the *goal* represents the desired state after executing it. This is actually how the orchestration is built, using planning techniques to find a plan that goes from the initial to the goal state using the provided abstract actions.

The initial state is actually derived from the method signature in the orchestration interface. For instance, in our example, the derived initial state for the method *getTranslatedPoll* is $pollId(id)\ and\ language(desiredLanguage)$. The same JAX-WS annotations used to generate the WSDL are used to generate the initial state, together with the formal names of parameters. The developer can also specify an additional part of the initial state to help in the construction of the plan, e.g., a fact that relates two of the parameters, through the Web interface (see Section III-B).

The goal state is specified by the developer also as part of the orchestration model file or through the Web interface. Since the goal of the orchestration is to have a poll translated into a given language, it is expressed in DSOL as $poll(translatedPoll)\ and\ language(translatedPoll, desiredLanguage)$.

E. Execution

At the time a DSOL orchestration is invoked, its goal, initial state, and abstract actions are used by an internal *Planner* to build an abstract plan of execution, which lists the logical steps through which the desired goal may be reached. Listing 6 illustrates a plan for our example. It includes a list of abstract actions that can lead from the initial state to a state that satisfies the orchestration goal.

```
getPoll(id)
detectLanguage(requestedPoll.title)
translatePoll(requestedPoll,detectedLanguage,desiredLanguage)
```

Listing 6. A possible plan for the poll translator example

This plan is taken and enacted by associating each step (i.e., each *abstract action*) with a *concrete action* that is executed, possibly invoking external services. Note that, while the plan is described as a sequence of actions, the system is free to execute them in parallel, by invoking each of them as soon as their precondition is satisfied.

During the execution of the orchestration, if something goes wrong (e.g., an external service is unavailable), first a different concrete action for the abstract action that failed is tried. If this is not enough to overcome the current situation, the Planner is invoked again to find a different course of actions that could skip the failed step. Furthermore, by comparing the old and the new plan, and considering the current state of execution, the system is able to calculate the set of actions that need to be *compensated* (i.e., undone) as they have already been executed but are not part of the new plan. Compensation actions are defined following the same idea of concrete actions.

This plan-execute-replan process is repeated until one plan is found that successfully reaches the orchestration goal or a plan cannot be built. In the first case, a successful message is sent to the client. Otherwise, the last tried plan is compensated and an exception is thrown.

III. Support Tools

In this section, we describe the tools provided by DSOL to help the development and also the management of orchestrations at runtime.

A. Eclipse Plug-In

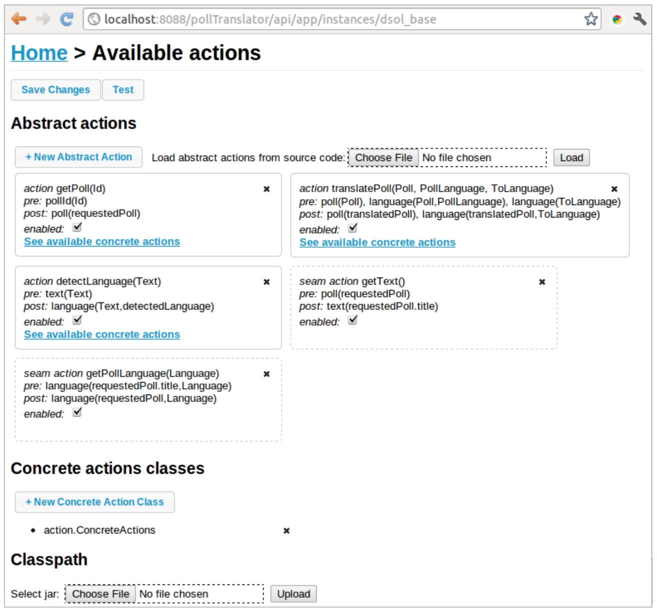
The DSOL Eclipse plug-in helps developers to configure a DSOL orchestration project with an ad-hoc wizard that creates a skeleton of the project, based on the information entered by the developer. This skeleton project includes the configuration files, the source code for the abstract actions, concrete actions and orchestration interface, and also runtime information like the context and the port in which the service must be available when running.

Using the DSOL plug-in it is also possible to run and debug the orchestration from inside Eclipse. Furthermore, it provides additional wizards to help the developer to specify abstract actions, concrete actions and to define the services to be composed.

B. Runtime Web Interface

Modularity, achieved by the loose coupling between abstract and concrete actions and also by how services are dynamically bound to actions, and dynamism, achieved by building the flow of execution at runtime, are inherent in the DSOL approach. Thanks to them, while a DSOL orchestration is running, the runtime system provides a Web interface that can be used to change the orchestration model and its running instances.

Figure 1 illustrates the page to modify the available abstract and concrete actions. Note it is also possible to test if the orchestration goal is still reachable using the new set of abstract action before actually changing the model. Other pages allow to manage initial states and goals, and to add new composed services.



Abstract actions

Load abstract actions from source code:

action getPoll(Id)
pre: pollId(Id)
post: poll(requestedPoll)
enabled: ✓

action translatePoll(Poll, PollLanguage, ToLanguage)
pre: poll(Poll), language(Poll,PollLanguage), language(ToLanguage)
post: poll(translatedPoll), language(translatedPoll,ToLanguage)
enabled: ✓

action detectLanguage(Text)
pre: text(Text)
post: language(Text,detectedLanguage)
enabled: ✓

seam action getText()
pre: poll(requestedPoll)
post: text(requestedPoll.title)
enabled: ✓

seam action getPollLanguage(Language)
pre: language(requestedPoll.title,Language)
post: language(requestedPoll,Language)
enabled: ✓

Concrete actions classes

- action.ConcreteActions

Classpath

Select jar:

Figure 1. Web interface to manage the available actions at runtime

C. RESTful service API

The DSOL runtime also allows the services binding to be changed at runtime in a programmatic manner. To do so, DSOL exposes a set of operations that can be invoked using HTTP requests. For example, a GET request sent to `api/services` will return, in a json format, the list of all available services, including all related information. A POST request sent to `api/services/<service_name>` will include a new service in the group of services referenced by `service_name`. A DELETE request sent to `api/services/<service_name>/<service_id>` removes from the orchestration model the service identified by `service_id`. Such API[2] could be useful, for example, for monitoring tools that can possibly monitor the composed services and manage them automatically to prevent faults or violation of QoS contracts.

IV. Related Work

DSOL was designed as an alternative to traditional orchestration languages such as BPEL, Jolie [4] and JOpera [5]. In general, such languages adopt an imperative style, which limits the ability of the defined service compositions to evolve at runtime and to cope with exceptional situations. In contrast, DSOL adopts a declarative approach to support the definition of flexible and self-adaptive service orchestrations, also able to cope with unexpected behaviors at runtime.

Other researchers followed the idea of adopting a declarative approach. Among those proposals, DecSerFlow [6] is the closest to our work. In DecSerFlow service compositions are defined as a set of actions and the constraints that relate them. Both actions and constraints are modeled graphically, while constraints have a formal semantics given in Linear Temporal Logic (LTL). There are several differences between DecSerFlow and DSOL. First of all, DecSerFlow focuses on modelling service compositions to support verification and monitoring. Conversely, we focus specifically on enacting them. This difference motivates the adoption of LTL as the basic modeling tool, as it enables powerful verification mechanisms but introduces an overhead that can be prohibitive for an enactment tool. The DSOL approach to modeling offers less opportunities for verification but it can lead to an efficient enactment tool. Secondly, DSOL emphasizes re-planning at runtime as a mechanism to support self-adaptive service orchestrations that maximize reliability even in presence of unexpected failures. This is an issue largely neglected by DecSerFlow, as it focuses on specification and verification and it does not offer specific mechanisms to manage failures at runtime.

V. Conclusion

In this paper we presented DSOL, its runtime system and the tools available to support the development and the maintenance of service orchestrations written in DSOL.

DSOL is completely written in Java and easily extensible. It can be found, together with its source code and the example presented in this paper, at http://www.dsol-lang.net.

A video of our tool-set is available at http://www.dsol-lang.net/demo.

Acknowledgment

This work was partially supported by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

References

[1] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, 2005.

[2] G. Cugola, C. Ghezzi, and L. S. Pinto, "Process programming in the service age: Old problems and new challenges," in *Engineering of Software*. Springer Berlin Heidelberg, 2011.

[3] L. S. Pinto, "A declarative approach to enable flexible and dynamic service compositions," in *Proceeding of the 33rd International Conference on Software engineering*, 2011.

[4] F. Montesi, C. Guidi, R. Lucchi, and G. Zavattaro, "Jolie: a java orchestration language interpreter engine," *Electronic Notes in Theoretical Computer Science*, vol. 181, no. 0, pp. 19–33, 2007.

[5] C. Pautasso and G. Alonso, "JOpera: A Toolkit for Efficient Visual Composition of Web Services," *Int. J. Electron. Commerce*, vol. 9, 2005.

[6] M. Montali, M. Pesic, W. M. P. v. d. Aalst, F. Chesani, P. Mello, and S. Storari, "Declarative specification and verification of service choreographies," *ACM Trans. Web*, vol. 4, pp. 3:1–3:62, January 2010.

[2]The complete reference can be found at DSOL website