Model-based Big Data Analytics-as-a-Service: Take Big Data to the Next Level

—

Annex A: Compiling the Procedural Model into the Deployment Model

Claudio A. Ardagna, Valerio Bellandi, Michele Bezzi, Paolo Ceravolo, Ernesto Damiani, Cedric Hebert

1 Executable Workflows

The last step of MBDAaaS methodology is based on a compiler that semi-automatically transforms a *procedural model* (OWL-S service composition and description) into a *deployment model* that is ready to be executed on the target Big Data Platform. After defining the internal operation of the compiler, we show the translation of an OWL-S service composition into a deployment model implemented as an Oozie workflow. Other deployment models can be supported by adapting the sub-processes of our compiler to the target model.

Deployment models are executable, platform-dependent models that specify how procedural models are instantiated and configured on a target platform, using components relevant for the analytics to be performed. They define configurations for architecture deployment and drive analytics execution in real scenarios. A deployment model $\overline{G}$ is an instance of a procedural model. We recall that the procedural models are *platform-independent models* that formally and unambiguously describe how services must be configured and composed.

1.1 Compiler

The transformation in Definition 5.1 is implemented by a compiler that takes as input the OWL-S service composition, the OWL-S service description of all services and information on the target platform, and produces as output a technology-dependent workflow that can be executed by the workflow engine available on the target platform.

- *C.A. Ardagna, V. Bellandi, P. Ceravolo and E. Damiani are with the Dipartimento di Informatica, Università degli Studi di Milano, Milano, Italy. M. Bezzi and C. Hebert are with Security Research, SAP Labs France, Sophia Antipolis, France. Ernesto Damiani is also with CINI - Consorzio Interuniversitario, Nazionale per l'Informatica, Rome, Italy and EBTIC, Khalifa University, Abu Dhabi, UAE).*
E-mail: claudio.ardagna,@unimi.it, valerio.bellandi@unimi.it, michele.bezzi@sap.com, paolo. ceravolo@unimi.it, ernesto.damiani@kustar.ac.ae, cedric.hebert@sap.com


```
<process:CompositeProcess rdf:about="tdm:CompositeProcess00001">
  <process:composedOf>
    <process:Sequence>
      <process:components>
       <process:ControlConstructList>
        <list:first> <process:Perform>
          <process:process rdf:resource="tdm:CleaningService"/>
        </process:Perform> </list:first>
      <process:components>
       <process:ControlConstructList>
        <list:first> <process:Perform>
          <process:process rdf:resource="tdm:AnonymizationService"/>
        </process:Perform> </list:first>
      <process:ControlConstructList>
        <list:first> <process:Perform>
          <process:process rdf:resource="tdm:ClusteringService"/>
        </process:Perform> </list:first>
       </process:ControlConstructList>
      </process:components>
    </process:Sequence>
  </process:composedOf>
</process:CompositeProcess>
```

Fig. 1. OWL-S workflow

The compiler represents the cornerstone of our MBDAaaS methodology. It consists of two main sub-processes, namely, *structure generation* and *service configuration*, each focusing on a specific part of the transformation as follows.

Structure generation. The compiler parses the OWL-S service composition and identifies the process operators composing it. Different operators can be managed including:

1) *Sequence* $\odot$ implementing a sequence statement where s_i is executed before s_j ($s_i \odot s_j$).
2) *Alternative* $\otimes$ implementing a conditional statement where either s_i or s_j is executed ($s_i \otimes s_j$).
3) *Parallel* $\oplus$ implementing a parallel statement where both s_i and s_j are executed at the same time ($s_i \oplus s_j$).
4) *Loop* μ implementing a loop statement, which can be either implemented as a separate statement (μs_i) or as a degeneration of the sequence.

Upon identifying the operators, the compiler loads the driver of the language used for deployment model speci-

fication and generates an empty skeleton of the Big Data pipeline to be executed, which corresponds to the structure of the procedural model. The structure, in addition to operators, contains an empty service for each service in the procedural model. Table 1 shows the mapping between OWL-S operators and corresponding operators in two popular workflow languages: Apache Oozie, specifically defined for Big Data, and WS-BPEL, originally defined for web service composition.

Service configuration. After generating the workflow structure, for each empty service s_i', *i)* the corresponding service s_i in the procedural model is identified, *ii)* service s_i is instantiated using the OWL-S service description, information on the Big Data execution platform, and the language selected for specifying the deployment model. We recall that the reference to a WSDL file in the OWL-S service description is used to identify the real service instance to be added in s_i' and executed on the target platform.

1.2 From OWL-S service compositions to Oozie workflows

Structure generation and service configuration subprocesses in Section 1.1 are implemented using Oozie workflows.

1.2.1 Structure generation

We use a SAX parser to parse the OWL-S service composition by means of an event-driven online algorithm and generate the corresponding Oozie workflow structure. An OWL-S service composition, modeled using the process model sub-ontology and the operators in Table 1, defines `process:CompositeProcess` as the root of the composite process. It then defines the inputs of the process with element `process:hasInput` and the corresponding outputs with element `process:hasOutput`. It finally describes the real service composition using element `process:composedOf`. The latter element specifies how services are composed and corresponding inputs/outputs are linked among them. In particular, it includes the specific operator (e.g., element `process:sequence`), which in turn contains the specific component services (element `process:components`) to be composed. For instance, Figure 1 shows an example of sequence (element `process:ControlConstructList`), where three services are composed in a sequence on a pair basis. The first service, *DataCleaningServices*, is wrapped within an element `list:first` and composed with a sequence of services *AnonymizationService* and *ClusteringService* which are wrapped within element `list:rest`.

The result is an OWL-S document that gets translated in an Oozie workflow according to the control constructs in Table 1. First of all, element `workflow − app` is added as root of the Oozie workflow. Then, for each service in the OWL-S service composition, an element `action` is added within the root. Each of the generated elements has an attribute *name*, and two elements `ok` and `error` modeling the execution flow. Figure 2 presents an Oozie workflow that corresponds to the OWL-S composition in Figure 1. In our example, three elements `action` have been added at the root level. The first action has name *DataCleaningService*

```
<workflow−app name='Process1' xmlns='uri:oozie:workflow:0.1'>
  <start to='DataCleaningProcess' />
  <action name='DataCleaningProcess'>
    <java>
      <job−tracker>${jobTracker}</job−tracker>
      <name−node>${nameNode}</name−node>
      <prepare><delete path='${outputDir}'/></prepare>
      <configuration>
        <property>
          <name>mapred.reduce.tasks</name>
          <value>300</value>
        </property>
      </configuration>
      <main−class>${MainClass}</main−class>
      <arg>${normalization}</arg>
      <arg>${datasetURI}</arg>
    </java>
    <ok to='AnonymizationProcess' />
    <error to='kill' />
  </action>
  <action name='AnonymizationProcess'>
    <java>
      <job−tracker>${jobTracker}</job−tracker>
      <name−node>${nameNode}</name−node>
      <prepare><delete path='${outputDir}'/></prepare>
      <configuration>
        <property>
          <name>mapred.reduce.tasks</name>
          <value>300</value>
        </property>
      </configuration>
      <main−class>${MainClass}</main−class>
      <arg>${normalizedDatasetURI}</arg>
      <arg>${hashing}</arg>
      <arg>${IPAddress}</arg>
    </java>
    <ok to='ClusteringProcess' />
    <error to='kill' />
  </action>
  <action name='ClusteringProcess'>
    <java>
      <job−tracker>${jobTracker}</job−tracker>
      <name−node>${nameNode}</name−node>
      <prepare><delete path='${outputDir}'/></prepare>
      <configuration>
        <property>
          <name>mapred.reduce.tasks</name>
          <value>300</value>
        </property>
      </configuration>
      <main−class>${MainClass}</main−class>
      <arg>${anonDatasetURI}</arg>
      <arg>${K}</arg>
      <arg>${maxIterations}</arg>
    </java>
    <ok to='end' />
    <error to='kill' />
  </action>
  <kill name="kill">
    <message>Task failed, error message[${wf:errorMessage()}]</message>
  </kill>
  <end name='end'/>
</workflow−app>
```

Fig. 2. Example of an Oozie workflow

and is connected to *AnonymizationService* using element `ok`. The second action has name *AnonymizationService* and is connected to *ClusteringService* using element `ok`. The third action has name *ClusteringService* and is connected to the end of the workflow. A default service (*kill_job*) is modeled using element `kill` and represents an unexpected end of the workflow due to faults/errors.

1.2.2 Service configuration

Service configuration acts on each element `action` and configures it according to the information included in the OWL-S sub-ontologies Profile and Grounding. The

TABLE 1
Control constructs considered by the Compiler

Operators	OWL-S Constructs	BPEL Constructs	Oozie Constructs
Sequence $\odot$	⟨process:Sequence⟩ ... ⟨/process:Sequence⟩	⟨sequence ...⟩ ... ⟨/sequence⟩	–
Alternative $\otimes$	⟨process:Choice⟩ ... ⟨/process:Choice⟩	⟨pick ...⟩ ... ⟨/pick⟩ ⟨switch⟩ ... ⟨/switch⟩	⟨switch⟩ ... ⟨/switch⟩
Parallel $\oplus$	⟨process:Split⟩ ... ⟨/process:Split⟩	⟨flow ...⟩ ... ⟨/flow⟩	⟨fork name="forking"⟩ ... ⟨/fork⟩
Loop μ	⟨process:Iterate⟩ ... ⟨/process:Iterate⟩	⟨while ...⟩ ... ⟨/while⟩	⟨action name="loop"⟩ ... ⟨/action⟩

first element specifies, for each action, the language used to implement the service itself (e.g., `Java` or `Spark`). Then, different elements specify how to execute the job including: *i)* `job − tracker`, to trace and manage the job, *ii)* `name − node`, where the job is executed, *iii)* `prepare`, to cleanup the directory for job execution, *iv)* `configuration`, the configurations of the job to be executed, *v)* `main − class`, the class to be executed, *vi)* a set of `arg`, arguments to be given as input to the job.

Example 1.1. *Figure 2 shows an example of an Oozie workflow corresponding to the OWL-S workflow in Figure 1 and compatible with the Oozie workflow engine. The workflow in Figure 2 is composed of three activities (elements* `action`*) in a sequence that execute* i) *data cleaning*, ii) *anonymization, and* iii) *clustering. As indicated by the element* `start`*, the workflow begins by executing* DataCleaningProcess*, which receives as input two parameters (elements* `arg`*), the first with the type of normalization to be executed (${normalization}) and the second with the target dataset (${datasetURI}). Upon a successful execution of* DataCleaningProcess, AnonymizationProcess *is executed (second element* `action`*). It receives as input the output of* DataCleaningProcess *(${normalizedDatasetURI}), the anonymization operation (${hashing}) and the field(s) over which the anonymization must be performed (${IPAddress}). Upon a successful execution of* AnonymizationProcess, ClusteringProcess *is finally executed (third element* `action`*). It receives as input the results of* AnonymizationProcess *(${anonDataset}), and parameter(s) for clustering (${K}, ${maxIterations}).*

There is, however, a subtlety to consider: the workflow in Figure 2 is not completely specified. In fact, the activities in the workflow refer to templates that need to be instantiated with parameters in the OWL-S workflow[1] *For instance, let us consider the reference to* AnonymizationProcess*. in Figure 2. The corresponding activity is completely specified unless the variables associated with the parameters anonymization operation ($hashing) and target ($IPAddress). When $hashing and $IPAddress are filled in with real values, the workflow is completely specified and ready to be executed*[2]*.*

1. In case a parameter value is missing, it is prompted to the user that can provide it.

2. In case no workflow engine is available on the target platform, these parameters are directly added in the call to the corresponding library (through a command line interface).

```
@RestController
public class OozieController implements BeanFactoryAware {

...

@RequestMapping(value="/owls2oozie", method=RequestMethod.POST)
public Oozie owls2oozie(@RequestBody Owls owls) throws ParserConfigException
{
    printWorkflow(owls.getOwls());
    return new Oozie(processOWLS(owls.getOwls()));
}

...
}
```

Fig. 3. REST Controller

1.2.3 Technical Details and Implementation of the Oozie Compiler

The Oozie Compiler receives as input an OWL-S service composition and returns as output an Oozie workflow (XML file). It follows the Springs approach that builds on RESTful web services; each activity, including HTTP request handling, is managed by a specific controller. Figure 3 shows the OozieController, which handles POST requests for the target */owls2oozie* by returning a new instance of the class. We note that controllers are identified by the *@RestController* annotation.

The *@RequestMapping* annotation ensures that HTTP requests to */owls2oozie* are mapped to the *owls2oozie()* method. Furthermore *@RequestBody* binds the value of the body into the class OWL-S with owls parameter of the *owls2oozie()* method. The implementation of the method body creates and returns a new Oozie object with content attributes based on the transformation of OWL-S to Oozie Workflow.

Our solution uses Spring4's new *@RestController* annotation, which marks the class as a controller where every method returns a domain object instead of a view. It is shorthand for *@Controller* and *@ResponseBody* rolled together.

At this point the Oozie object must be converted to JSON. Thanks to Springs HTTP message converter support, we do not need to do this conversion manually. Because Jackson 2 is on the classpath, Spring's MappingJackson2HttpMessageConverter is automatically chosen to convert the Oozie instance to JSON. Transformation of OWL-S to OOZIE Workflow start from the method called *printWorkflow()*.

```
try {
    InputStream owlsIs = IOUtils.toInputStream(owls, "UTF−8");
    Model model = Rio.parse(owlsIs, "", RDFFormat.RDFXML);
    Repository db = new SailRepository(new MemoryStore());
    db.initialize();
```

Fig. 4. Piece of code of a RDF parser

```
try (RepositoryConnection conn = db.getConnection()) {
    conn.add(model);

    TupleQuery tupleQuery = conn.prepareTupleQuery(
      + PREFIX rdf:<http://www.w3.org/1999/02/22−rdf−syntax−ns#> \n
      + PREFIX process:<http://www.daml.org/services/owl−s/1.2/Process.owl#>
      + SELECT ?main ?typemain ?component \n
      + WHERE { \n
      + ?main process:composedOf ?component. \n
      + ?main rdf:type ?typemain. });

    TupleQueryResult result = tupleQuery.evaluate();
```

Fig. 5. Example of SPARQL SELECT query

For interaction with RDF our solutions use Eclipse RDF4J to create RDF models, Eclipse RDF4J is a Java API for RDF and permits to create, parse, write, store, query and reason with RDF data in a highly scalable manner. The RDF Model API is the core of the RDF4J framework. It provides the basic building blocks for manipulating RDF data in Java. In this section, we introduce these basic building blocks for manipulation of OWL-S. First of all the *printWorkflow()* method receives OWL-S string as input and returns the corresponding workflow, the method parse RDF of composed service and then create SPARQL query to generate Oozie Workflow.

In figure 4 is shown a piece of code that permits to evaluate a SPARQL SELECT query and returns a TupleQueryResult, which consists of a sequence of BindingSet objects. Each BindingSet contains a set of Binding objects. A binding is a pair relating a variable name (as used in the query's SELECT clause) with a value.

Java InputStream's are used for reading byte based data, our compiler parses an RDF document and collects all the parsed statements in a Java Collection object (specifically, in a Model object) using: *Rio.parse(owlsIs, "", RDFFormat.RDFXML)*.

Subsequently, the compiler creates a new Repository based on SailReporitory class.

SailRepository is a repository that operates directly on top of a Sail object(s) for storage and retrieval of RDF data, the behavior of a repository is determined by the Sail(s) that it operates on; the repository supports OWL semantics. It permits to open a connection to the database, add the model; then we can perform queries.

Using the piece of code exposed in fig.5 we can execute a SPARQL SELECT query and it returns a *TupleQueryResult*, which consists of a sequence of *process:composedOf* our OWL-S.

The compiler uses the *TupleQueryResult* to iterate over all results and get each individual item for *CompositeProcess* and type of components. Using this solution we retrieve values by name rather than by an index. The names used should be the names of variables as specified in the query.

```
<owl:Class rdf:ID="Process">
  <rdfs:comment> The most general class of processes </rdfs:comment>
  <owl:disjointUnionOf rdf:parseType="daml:collection">
    <owl:Class rdf:about="#AtomicProcess"/>
    <owl:Class rdf:about="#SimpleProcess"/>
    <owl:Class rdf:about="#CompositeProcess"/>
  </owl:disjointUnionOf>
</owl:Class>
```

Fig. 6. Process Ontology

```
<service:Service rdf:ID="SparkKmeanService">
    <service:supports>
      <grounding:WsdlGrounding rdf:ID="SparkKmeanGrounding"/>
    </service:supports>
    <service:describedBy>
      <process:AtomicProcess rdf:ID="SparkKmeanProcess"/>
    </service:describedBy>
    <service:presents>
      <profile:Profile rdf:ID="SparkKmeanProfile"/>
    </service:presents>
  </service:Service>
```

Fig. 7. Process Ontology

We expect that our process ontology (fig. 6) to serve as the basis for specifying a wide array of services, services are modeled as processes. Furthermore, we distinguish between three types of processes: *i)* atomic, *ii)* simple, and *iii)* composite; each of these is described below.

- *The atomic processes* can be directly called (by providing the appropriate messages), have no subprocesses, and execute in a single step, from the perspective of the service requester. Figure 7 shows an example of an atomic process based on k-means classification algorithm.
- *Simple processes* are not invocable and are not associated with a grounding, but, like atomic processes, they are conceived of as having single-step executions. In the former case, the simple process is realizedBy the atomic process; in the latter case, the simple process expandsTo the composite process.

- *Composite processes* are decomposable into other processes; their decomposition can be specified by using control constructs such as SEQUENCE and IF-THEN-ELSE, which are discussed below. Such a decomposition normally shows, among other things, how the various inputs of the process are accepted by particular subprocesses, and how its various outputs are returned by particular subprocesses. A COMPOSITEPROCESS must have a composedOf property by which is indicated the control structure of the composite, using a CONTROLCONSTRUCT. Each control construct, in turn, is associated with an additional property called components to indicate the ordering and conditional execution of the subprocesses (or control constructs) from which it is composed.

In the process upper ontology, we have included a minimal set of control constructs that can be specialized to describe a variety of services. This minimal set consists of

- *Sequence*, a list of Processes to be done in order, the components of a Sequence to be a List of process components which may be either processes or control constructs.
- *Split,* the components of a Split process are a bag of process to be executed concurrently. No further specification about waiting or synchronization is made at this level.
- *Split,* the components of a Split process are a bag of processes to be executed concurrently. No further specification about waiting or synchronization is made at this level.
- *Split + Join*, SPLIT is similar to other ontologies'use of Fork, Concurrent, or Parallel. Here the process consists of concurrent executions of a bunch of process components with barrier synchronization. With SPLIT and SPLIT-JOIN, we can define processes that have partial synchronization (e.g., split all and join some sub-bag).
- *If-Then-Else*, this class is a control construct that has properties *ifCondition*, *then* and *else* holding different aspects of the IF-THEN-ELSE. Its semantics is intended as "Test If-condition; if True do Then, if False do Else." (Note that the class CONDITION, which is a placeholder for further work, will be defined as a class of logical expressions.)
- *Repeat-While and Repeat-Until*, these classes specializes the IF-THEN-ELSE class where the ifCondition is the same as the untilCondition and different from the REPEAT-WHILE class in that the else (compared to then) property is the repeated process. Thus, the process repeats until the untilCondition becomes true.