Making BPEL flexible

Daniela Feldkamp
University of Applied Sciences
Northwestern Switzerland, School of Business
daniela.feldkamp@fhnw.ch

Nishant Singh
Indian Institute of Technology
Kanpur
nssingh@iitk.ac.in

Abstract

The Business Process Execution Language (BPEL) is a process modeling language which uses standard control constructs to define a process flow. But today enterprises have to be flexible and adaptable to cope with increasing change, uncertainty and unpredictability. Automating agile business processes is still a challenge as they are normally knowledge intensive, little automated but compliance relevant. Service-oriented architecture (SOA) inherently enables flexibility and adaptivity through choreography of services where each service can select and invoke any other web services. Web services are basic building blocks for building online processes. In this paper we introduce an approach combining BPEL, Business rules and semantic web technologies to achieve adaptivity during runtime.

Introduction

Adaptivity and changes are the most important challenges of business (Hammer & Champy 1996), because they are able to cope up with increasing changes, uncertainty and unpredictability in the business enviroment (Henbury 2007). To meet these challenges, over the last years Business Process Management(BPM) and Service-Oriented Architecture (SOA) have evolved as key technologies. SOA's basic idea is to provide a set of web services, which can be used by anyone at anytime (Chow & Medley 2007). Additionally,the separation of business process logic from business application became common practise. Business process automation increases with workflow management systems. But, this was an appropriate approach to create and automate well-structured processes, which assumed, that task move from one resource to another. But, involving people in processes leads to the challenge of supporting highly variable, unstructured and complex tasks. Especially supporting knowledge-intensive tasks needs us to deal with exceptional situations, unforeseeable events and unpredictable situations (Hinkelmann, Probst, & Thönssen 2006).

SOA inherently enables flexibilty and adaptivity through choreography of services where each service can select and invoke any other web service. Hence, combining technologies of SOA- and BPM will achieve more flexibility in orchestrating tasks.

The Business Process Execution Language for Web Services (BPEL) supports the specification for coordination and composition of web services (Alonso *et al.* 2004). It provides standard control constructs, like switch or sequences, to define a process flow. It is useful for structured process flows, but it constrains flexibility. Hence, modeling knowledge-intensive processes for every possible variants of cases can lead to a complex process model, which is hard to maintain. Sometimes, this modeling is even impossible, if the tasks are mutually depending on each other.

To achieve more flexibility we provide an approach to model these parts more abstractly in the design time and define the flow of these parts during runtime regarding the case. The abstract parts are modeled using business rules and semantic technologies, which leads to flexible selection and invocation of tasks during run time.

Hence, in the next section, modeling Appraoch, we discuss the business rule paradigm and SOA paradigm (basically BPEL). Thereafter, we propose our approach describing modeling of variable processes and how to execute the process model. In the following section, Run Time Approach, we describe the business rule engine RHEA, which would present the run time execution of the approach. Thereafter, in Conclusion, we conclude with a summary and the significance of our approach.

Modeling Approach

As mentioned in the introduction section, BPEL is a standard for orchestrating web services which lays down the foundation for inherently flexible SOAs, but is incapable of modeling unstructured processes. Thus, we lay down an approach which uses business rules for overcoming the shortcomings of BPEL.

We provide a brief description of BPEL and business rules. Thereafter, we will propose our approach

in contrast to certain proposed approaches with similar objectives.

BPEL

BPEL stands for Business Process Execution Language. As the name suggests it provides a platform to structure business processes and deploy it as web services. It provides XML based structure to describe processes. Some key notions are as follows (Alonso *et al.* 2004)

Process Information - <bpel:process> tag encloses the description of the whole process. It contains information about the process like the name of the process, process namespace etc.

Imports - <bpel:import> tag has the information of the WSDL files, filenames and namespaces, which have been referred in the BPEL process.

Partnerlinks and variables - <bpel:partnerlink> gives the name of the partner link and its partnerLinkType. The partnerLinkType associates the ports, which in turn associate the variables to this partnerlink. <bpel:variable> gives the information of the variables which include messageType which is mapped to a WSDL message.

Flow - <bpel:flow> tag encloses the flow of the process. It encloses information of link between activities(<bpel:link>), copy activities (<bpel:copy>), receive and reply activities(<bpel:receive> and <bpel:reply>) and finally the flow tree which consists of iterative encapsulated constructs like sequences, split etc. (e.g. <bpel:sequence>). To be more explicit <bpel:flow> tag only encloses the constructs at the highest level. All the iterative lower level constructs are enclosed within the scope of the tags of their parent constructs. The constructs which are to be run in parallel can be modeled by the split construct. All conditional branches can be modeled by IF-DO construct and all the sequential activities can be modeled by sequence construct.

Business Rules

Business Rules are explicit statements to govern business behavior (Ross 2003), (von Halle 2002). Several classification schemata for Business rules formalization exist, but since it is well understood and comprehensive, we follow the classification of Barbara von Halle to explain the different kinds of rules. The rule classification depicts business rules into the following main components:

Terms A term is a phrase which has a specific meaning for the business in some context.

Facts A fact is a statement expressing relationships between two or more terms.

Rules Rules are declarative statements that apply logic or computation to information values.

Barbara von Halle splits rules into five sub-classes. Amongst others she defines action enabling rules as rules which trigger another rule or a process (step), if the condition holds (von Halle 2002).

Relating Business Rules and Tasks

Geminiuc from Oracle proposes a concept for combining business rules with BPEL to support decision points (Geminiuc 2006), which has the main goal to decouple the interpretation and execution of rules in application independent services. Another approach focuses on integration of BPEL with different rule-based systems, by providing a service bus (ESB) as middleware for the service-oriented architecture (Rosenberg & Dustdar 2005). These approaches support decision points and not to achieve flexibility in process orchestration and choreography.

Additionally, our work is related to business rule approaches. For instance, one approach using action enabling rules is represented by (Beer *et al.* 2007). In their approach ECA (Event, condition, action) rules are used to provide users with tailored messages related to their current situation (context). In (Bider *et al.* 2006) an approach is represented to activate knowledge using rules for planning.

For the structured parts we want to use BPEL. As, BPEL only provides standard control constructs, which can lead to complex process models, we use business rule approaches to achieve flexibility during runtime. Because, we want to avoid extending the BPEL standard, we implement a web service as done by (Geminiuc 2006), which provide the additional functionalities for process orchestration and execution during run time.

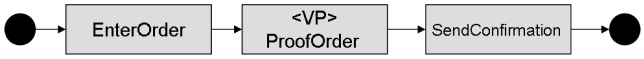

Figure 1: Relation variable process to activity pool

For example in figure 1, we assume that the task *ProofOrder* is a task containing several sub tasks which can be hardly modeled during design time for every contrivable case. Instead of modeling all cases, we replace these parts by using the object type *variable process*. This object type is related to a pool of sub tasks. These sub tasks are determined at run-time and hence avoids strictly modeling them. These tasks are related to action enabling rules.

Figure 2 shows, which sub tasks are related to the variable process. All sub tasks are related to action enabling rules. These action enabling rules select at run time, the activities that have to be executed depending on the actual case. If a condition (defined in the IF-clause) of a rule is satisfied the sub task (specified in the DO-part) is invoked.

For clearer understanding we reduce the number of possible sub tasks, shown in figure 2, in the task pool. Figure 3 shows the shortend example. The whole process is invoked by sending an order by a user. If the

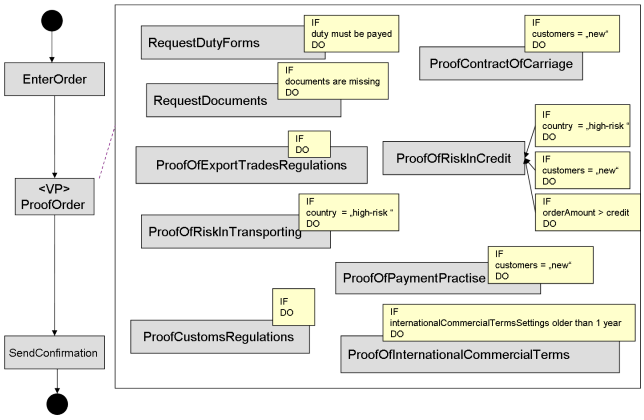

Figure 2: Relation of variable process to tasks pool

customer is new he has to be added to the information system and a limit has to be set on his credit. If the order amount is lower than his limit, he will get a confirmation otherwise the order has to be rejected. This example can be modeled in the traditional way. But to make the formalization and the process execution clearer we assume that this is a variable process. We define four activities:

Activity A - CheckAndSetCreditVolume This activity checks and sets the high of the credit limit of the customer.

Activity B - RejectOrder RejectOrder is invoked, if the order has to be rejected.

Activity C - AddCustomerToSystem If a customer is not registered in the information system, he has to be added.

Activity D: ApplyOrder ApplyOrder is executed, if the order can be applied.

We assume, that a new customer has submitted an order. Being new, she is not registered in the system, she has to be added to the system and her credit limit has to be set. After invoking activity A, C needs to be executed. They can be modeled as a sequence, because this has to be done by every new customer in every case.

Formalization

For the static part of the process model we use BPEL and its standard control constructs to define process flow. For the variable parts we use semantic web technologies.

In order to be validated and executed, the business rules and processes have to be represented in a language with well-defined semantics. Current business rule systems only have simple formalism with weak semantics for representing facts and terms. Extending the expressiveness towards ontologies has the advantage of higher expressiveness and the chance to use inferences. As a consequence, since the procedural knowledge must be highly integrated with declarative knowledge, a rule

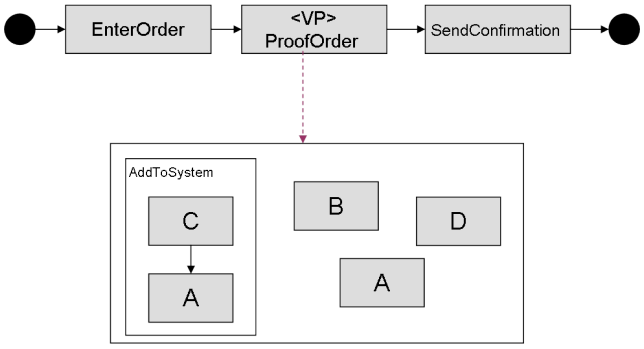

Figure 3: Brief example

language must be available in which all rule types can be expressed and which can be integrated with ontologies. Therefore, we use OWL and SWRL to express terms, facts and business rules, while the process models are represented in OWL-S.

To model variable processes we use the following phases:

1. Creation of a domain ontology, which is used to represent the domain knowledge.
2. Representation of tasks of the variable process part in OWL-S.
3. Creation of rules.

Stage 1 - Creation of a domain ontology. To be able to present the domain specific knowledge a domain ontology is created. Figure 4 illustrates the domain ontology for the short example. In this case we have two concepts: a customer and an order. The customer is described by her name, address, status (storing that the customer is new or registered), lastCheck, representing the last time his credit limit was checked and his credit limit. The order has the property amount. To illustrate that a customer has a specific order, both concepts are related by the object property hasOrdered.

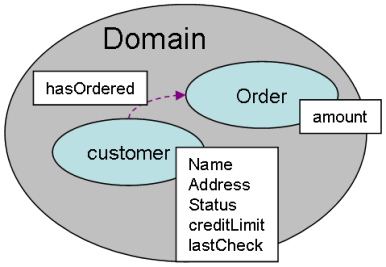

Figure 4: Domain Ontology

Stage 2 - Represent of tasks. The tasks of the variable process part are modeled using OWL-S. The

tasks combined to a variable process part are represented as OWL-S atomic processes. If during build time sequences or other control constructs can be identified, which order has to be fulfilled for every case, these atomic processes can be combined into a compositeProcess

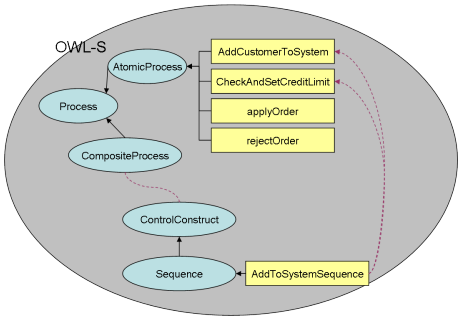

Figure 5: Process Ontology

Figure 5 represents OWL-S for the brief example. Regarding the four activities, we have added four instances to the concept atomic process: *AddCustomerToSystem*, *CheckAndSetCreditLimit*, *applyOrder* and *rejectOrder*. Every time, if a customer is new, he must be added to the system and after it, his credit limit has to be set. This sequence is represented by *AddToSystemSequence*, which is an instance of a controllConstruct Sequence.

Stage 3 - Creation of rules. After these two steps the action enabling rules can be created.

For the example we have created four rules. One of the rules is depicted in figure 6 which is applicable for an instance where only the task *applyOrder* is invoked (for the case that the customer has the status regular, and the amount of the order is less than or equal to his credit limit).

```
customer(?x)  and
hasOrdered(?x, ?o) and
amount(?o, ?a) and
creditLimit(?x, ?cl)  and
status(?x, "regular")  and
swrlb:lessThanOrEqual(?a, ?cl)
  →  atomicProcess(applyOrder)
```

Figure 6: Action enabling rule for triggering applyOrder

During runtime, if the condition is true that the status is regular and the amount of the order is less than or equal to his credit limit, the atomic process (applyOrder) is invoked.

Combination BPEL and variable process

As we mentioned before, we still use BPEL for the structured process part. So, every BPEL engine can invoke the process. The unstructered parts are orchestrated and invoked by a web service, called RHEA, which is implemented during the FIT-Project[1].

Run time approach

In order to adapt process execution at runtime - taking into account the actual case - the web service RHEA has been implemented, which provides the following features:

- Integration of ontological inferences with rule execution
- Invocation of rule engine for context-adaptive process execution by workflow-engine.

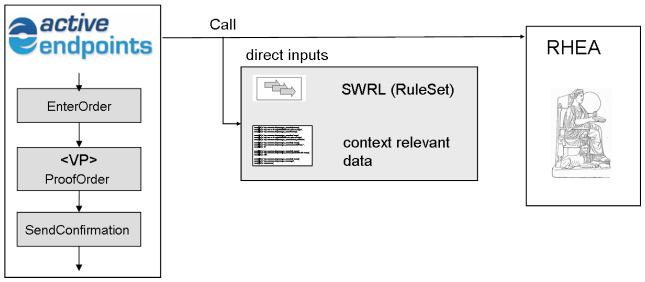

Figure 7: Invocation of RHEA

We use ActiveBPEL[2] to evaluate our approach. Figure 7 shows the relation between the process executed by ActiveBPEL and RHEA, which is invoked, if the variable process should be executed. The invocation of RHEA needs three input parameters. One input contains the rules, one an URI to the ontologies and the context relevant data must be given. These inputs are necessary knowledge to execute rules.

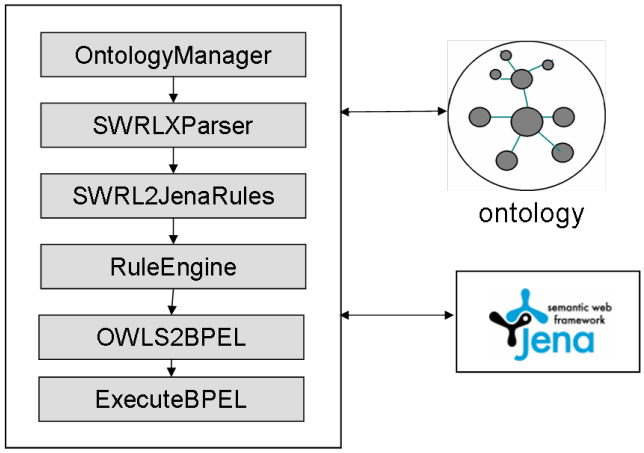

Figure 8: Architecture of RHEA

[1]FIT Project http://www.fit-project.org/

[2]ActiveBPEL, http://www.active-endpoints.com/active-bpel-engine-overview.htm

The overall architecture of RHEA is shown in figure 8. For RHEA we use the open source framework Jena2[3]. This framework is on the one hand used for parsing ontologies and on the other hand the provided rule-based inference engine is extended for our approach.

RHEA consits of several components, which are used in specific order. First, the knowledge has to be added to the rule engine. Second the rule engine can be executed. If rule are fired, the last two components parse the process ontology, create and invoke the sub BPEL process.

OntologyManager This component parses the domain ontology using Jena2 and adds the context relevant data as instances and property values.

SWRLXParser This component parses the SWRL file and transfers the rule to an internal structure.

SWRL2Jena Therefore, to use the rule engine of Jena2, we transfer our internal rule structure to a format Jena2 uses.

RuleEngine The rule engine executes the rules and throws the result regarding the context specific data.

OWLS2BPEL This component parses the OWL-S and exports the data to BPEL.

ExecuteBPEL This component constructs the required WSDL file and executes the generated BPEL file.

In the following sections, we describe the components in more detail.

OntologyManager

The OntologyManager adds context relevant data of the current case to the domain ontology. The context relevant data (mentioned in figure 7 as context relevant data) is given by an array of RDF-triples.

```
// Set Order_123 as instance of Order
data[0][0] = nameSpace + "Order_123";
data[0][1] = rdfNameSpace + "type";
data[0][2] = nameSpace + "Order";

//Set type=commercial
data[1][0] = nameSpace + "Order_123";
data[1][1] = nameSpace + "amount";
data[1][2] = "4.500";

//Set order as orderOf customer Paul
data[2][0] = nameSpace + "Paul";
data[2][1] = nameSpace + "hasOrdered";
data[2][2] = nameSpace + "Order_123";
```

Figure 9: Context relevant data

Figure 9 shows a triple, which specifies the *Order_123* of customer *Paul*, by using RDF triples. The first part defines *Order_123* as an instance of *Order*. The second triple specifies the *amount* and the last triple relates the order to the customer *Paul*.

The OntologyManager adds this data to the domain ontology by parsing the ontology and searching for the URIs. For the first triple, the manager searches for the concept with the URI <namespace> Order and adds the instance *Order_123* to this concept. After it, the *amount* is set, by adding the value at the data property of *Order_123*. After all, the *Order_123* is related to the customer Paul, by using the object property hasOrdered.

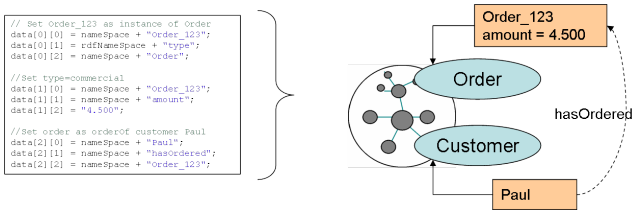

Figure 10: Added context relevant data to domain ontology

SWRLXParser and SWRL2JenaRules

Both parsers, SWRLXParser and SWRL2JenaRules, are used to parse the SWRL rules and add these rules to the knowledge base of the rule engine. We have adapted the rule engine provided by the Jena2-Framework. The reasoner supports rule based inference over RDF graphs and provides forward, backward chaining and a hybrid execution model(Reynolds 2007). Hence, Jena2 uses another format for expressing rules. We transform SWRL to the format used by Jena2. The atoms of the SWRL file are transferred into triples. To express *a person x hasOrdered the order o* the SWRL-atom is defined as follows:

```
<swrlx:individualPropertyAtom
swrlx:property="hasOrdered">
<ruleml:var>x</ruleml:var>
<ruleml:var>o</ruleml:var>
</swrlx:individualPropertyAtom>
```

This fragment is transformed using the RHEA parsers to the following Jena2-expression:

```
(?x hasOrdered ?o)
```

The header and the body of a rule are separated in Jena2 using the symbol "'-¿"'. So, first all body-atoms are transfered, the narrow is added and after everything, the header is created.

Figure 11 illustrates a complete transformed rule.

RuleEngine

After setting the knowledge, the rule engine can now be invoked. For RHEA we use forward chaining RETE engine provided by Jena2. We have extended the classic

[3]Jena2: http://jena.sourceforge.net/

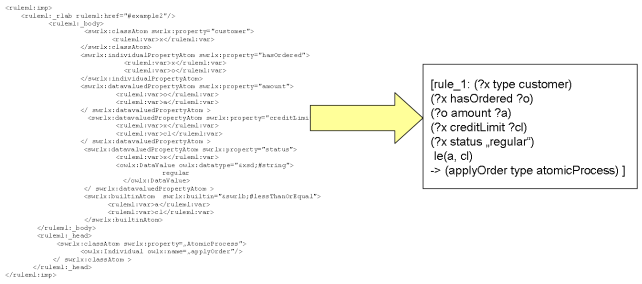

Figure 11: Transformation of SWRL to Jena2

RETEEngine by a functionality which stores the fired consequences for an execution in a list. If a condition holds, an instance of an atomic or composite process is fired and added to the list. According to the example shown in last sections, if the Order of Paul is less than or equal to his credit limit, the atomic process applyOrder is fired.

OWLS2BPEL

BPEL and OWL-S have a very similar process structure, and the similarity lying in the constructs which lay the framework for designing a process. There is a formulation of conversion from BPEL to OWL-S (Aslam *et al.* 2006). Here, in lines to the goals of RHEA we propose a formulation of conversion of OWL-S processes to BPEL processes.

A complete conversion of OWL-S to BPEL requires parsing the OWL-S grounding class and creating the WSDL to be referred by the BPEL process for execution. This is explained in the next section. Assuming the WSDL has been created, here we explain the export of OWL-S process information to BPEL.

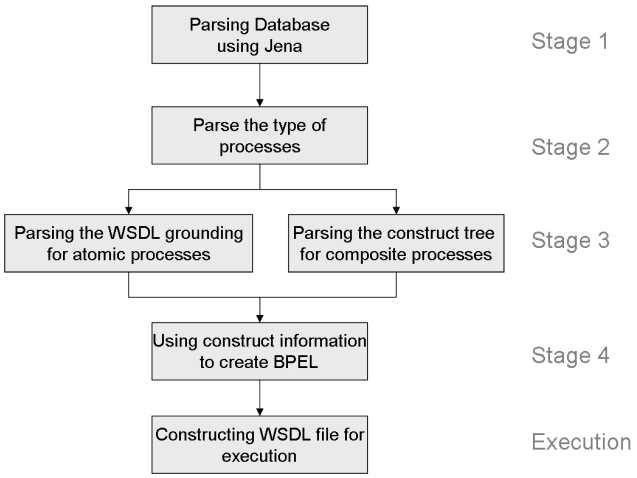

Figure 12: Stages of OWL-S to BPEL

The approach is divided in stages which are as follows:

Stage 1 - Parsing the OWL-S ontology using Jena2 and storing it as RDF triplets in a Database. In this stage using the `DBConnection` object type we obtain a connection and then pass this connection as a parameter to the ModelMaker object type which invokes the `createModel` function with the OWL-S file as parameter and hence stores the ontology in the form of RDF triplets in the database. This database can be queried using SPARQL query language.

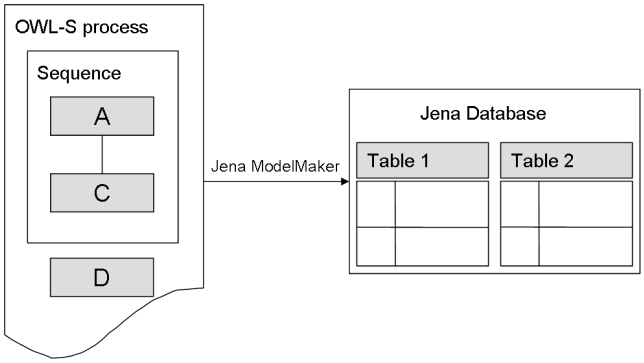

Figure 13: Stage 1: OWLS file to JenaDB

Stage 2 - Parsing the type of processes. In this stage we use the following SPARQL queries :

First query `SELECT ?t\n`
```
WHERE {
        ?t <&rdf;#type>
<&owls;#AtomicProcess> }
```

Second query `SELECT ?t\n`
```
WHERE {
     ?t <&rdf;#type>
     <&owls;#CompositeProcess> }
```

From the first query we get the atomic processes and store it in a table named *atomicprocesses*. For example, there would be three entries namely A, C and D. We also store the name of the table and namespace entries for the processes. Similarly, for the Composite processes we make a table named *CompositeProcesses* with entries for each composite process and their respective *TableName*. Here we will have a single entry named *CompositeProcess_1*. This structure is depicted in figure 14.

For each composite process we create a table, which stores the information of the construct tree. Refer to figure 15 for structure details.

Stage 3/PART-1 - Parsing the WSDL Grounding for each atomic processs. Each process table for the atomic processes namely AtomicProcess_i (for ith atomic process) consists of fields namely *WSDLPartName*, *WSDLPartType* and *ParentStruct*. For in-

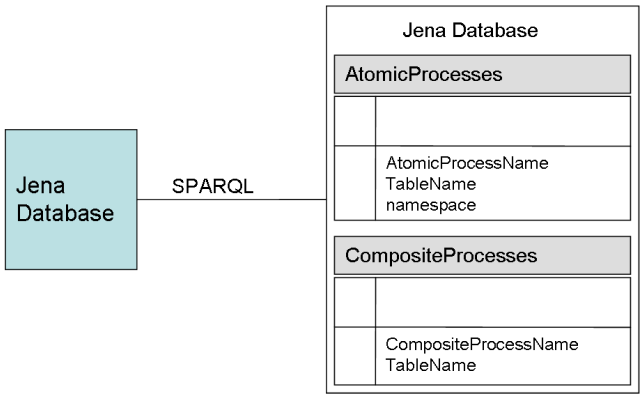

Figure 14: Stage 2 : Table structure for each composite and atomic process

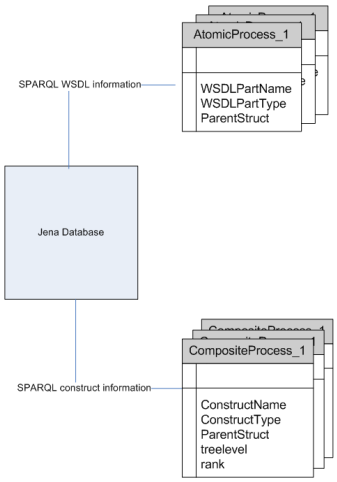

Figure 15: Querying for processes and table

stance, we assume, we have a RDF triplet after parsing the WSDL Grounding through Jena2 as follows:

```
<grounding:wsdlgrounding A>
<grounding.owl#wsdlDocument>
<C>
```

When we query for a document for the above grounding namely A, we get C. Now, we store the information in the respective table with A as the *ParentStruct*, *grounding:wsdlDocument* as the *WSDLPartType* and C as the *WSDLPartName*. Similarly with appropriate SPARQL queries we extract information and store it in a MySQL database.

Later on this information is used through SQL queries to generate appropriate tags and partner links which has been later explained in the document. The queries used extract the WSDL information by first identifying the *atomicProcessGrounding* for the given *AtomicProcess*. This is done by querying by keeping the predicate as *owlsProcess*. The *ParentStruct* for this information is kept as the *ProcessName*.

We also query with *hasInput* and *hasOutput* as the predicate to extract information of *Inputs* and *Outputs*. These are also stored with the *ParentStruct* as the *ProcessName*. Thereafter the *wsdlDocument*, *wsdlInputMessage* and *wsdlOutputMessage* are identified by keeping appropriate predicates. They have *wsdlGrounding* as the *ParentStruct*. wsdlGrounding also acts as the ParentStruct for objects queried with predicates of *wsdlOperation*. Thereafter, this object serves as the *ParentStruct* for *wsdlPartTypes wsdl:operation* and *wsdl:portType*, which symbolize the *porttype* and *operation* information.

Similarly proceeding *Input* and *Output Message maps* are also extracted. Additionally, Binding information is extracted. Once all this information is made available in the tables they are later queried through SQL to construct copy tags, partnerlinks and other relevant tags.

Stage 3/PART-2 - Parsing the construct tree for each composite process. Each process table consists of fields namely *ConstructName*, *ConstructType*, *ParentName*, *treelevel* and *rank* ,figure 15. *ConstructName* depicts the name of the construct, *ConstructType* represents the type of the construct i.e. sequence, split, until or others. *ParentName* represents the name of the parent construct. *treelevel* represents the depth of this construct in the tree and *rank* represents the rank of this child among the children of the parent construct.

Hence the parsing of the tree is done through a recursive procedure, where each recursion terminates at a construct with construct type as *Perform*. *Perform* is the construct type which implies that the node is an atomic process or a composite process and not a construct and hence there is no sub tree to be traversed.

At first the *composedOf* property is checked, which gives the construct with treelevel 0.

The entry of this construct has the parentName as the processName. Thereafter the ProcessTree func-

tion is called for this construct. It checks the components property for constructs other than *ControlConstructBag*, *ControlConstructList*, *If-Then-Else*, *Until and While*. For ControlConstructList and Bag, the first and rest properties are checked and the recursion is called for both the constructs returned, with the treelevel unchanged.

The *first* has a rank entry preceding the rank entry for *rest*. For Until and While the until-process and while-process properties are checked, respectively and the recursion is called with treelevel incremented and rank as 0. For If-Then-Else, Then and Else properties are checked and next recursion invoked with incremented treelevel and each having the rank as 1. Thus the recursive traversal takes place and terminates at a leaf which implies a Perform construct type. An example run is depicted in figure 16.

Stage 4 - Using all the parsed data and putting in the BPEL framework. First, we construct the process tag. For this each atomic process has to be associated with a namespace. This information is stored in the table atomicprocesses. At the time of exporting this data is used to construct the process tag.

Thereafter, we construct the partnerlink and variable tags. For this, we need to parse the WSDL grounding information for each atomic process beforehand. Hence, we create a table respective to each atomicprocess, figure 15 and store all the WSDL information like InputMessageMap, OutputMessageMap, operation, porttype and others.

This information is stored in the table for each atomic process in the fields WSDLPartName, WSDLPartType and ParentStruct. These are the object, predicate and subject respectively in the RDF triple, for a grounding class property to the related atomic process. This is extracted for tag construction. Each atomic process in a given composite process is a partner process and each is assigned a partnerLink and PartnerLinkType which have a common basename e.g. PartnerLink_1 and PartnerLinkType_1. For getting the variable names for each partnerProcess we use grounding:wsdlInputMessage and grounding:wsdlOutputMessage. All of this information is made available in each atomic process table and is retrieved using SQL queries.

Thereafter, we generate the flow tags, which comprise of link tags, copy tags, receive and reply tags and construct tags. The Receive and Reply tags require processname, as we generate one partner link (named on the process) and use standard names for portTypes and operation, so that it can be added later in the WSDL.

As for the link tags, there are always four link tags generated with the source-target pair as ProcessName-CopyF, CopyF-MainConstruct, MainConstruct-CopyL and CopyL-ProcessName, where MainConstruct is the subject of composedOf property and CopyL and CopyF are the first and last copy tags generated. MainConstruct symbolizes the construct which envelops the

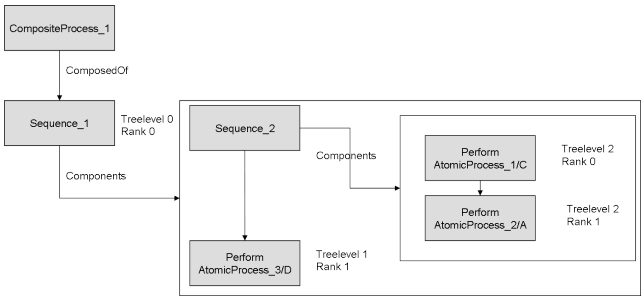

Figure 16: Parsing the construct tree

whole process model. So all the inputs required by the processmodel are made available to the MainConstruct by CopyF copy tag, which copies all the inputs from the Receive process.

Similarly, all the outputs are made available to the Reply tag through the CopyL copy tag which copies the outputs form the MainConstruct to the variables of the Reply process. Thus the links are between the Receive and CopyF, CopyF and MainConstruct, MainConstruct and CopyL, CopyL and Reply and they are appropriately named as above.

The copy tags are of two types. The ones from TheParentPerform to the atomic processes and the others from the process to process. Hence checking the values of the fromProcess property for *TheParentPerform* we can generate the copy tags of the first kind. And for the other copy tags we need to query for fromProcess as the Atomic Process being considered and then check the TheVar and toParam properties to get the copy tags.

This information is already available in the respective atomic process tables, as it was parsed while extracting the wsdl information. To be explicit we extracted the binding information at that stage. Now to extract the binding information we query the table for each atomicProcess with the WSDLPartType as fromProcess and wsdlPartName as the name of the perform construct which executes the atomic process in question. Now as the result set we get all the processes which are providing inputs to that specific atomic process. Now, with each of these process names we query with them as the ParentStruct. We choose wsdl:TheVar and wsdl:ToParam as our predicates.

Result of these queries is the pair of variables to be used in the copy tag. Similarly, we get all the input copy tags for each process. The copy tags for the outputs are generated by taking care of as each output has to be bound to an input for a process. So, when all the processes input copy tags are generated the output tags get generated simultaneously. Now, the tags which are left are the end tags, the CopyL and CopyF tags. CopyL tags are generated in the same way. Only the destination of these copy tags is mentioned as TheEnd. As for the CopyF tags, these are generated by keeping

the fromProcess object as the ParentPerform.

The construct tags are generated by using the information in the table of the composite process and the following mapping between the OWL-S constructs and BPEL constructs:

- OWL-S sequence to BPEL sequence
- OWL-S split to BPEL split etc.

Now, once we have the tags generated we order them under the flow tag. First,come the link and receive and reply tags. Thereafter, the copy tags from TheParentPerform and the tags to TheEnd. These two copy tags are though generated under the scope of a <bpel:assign> tag.

These tags are easily generated by putting the copy tag and the respective link tags under the scope of each assign tag. Thereafter, we generate the construct tags top to bottom by using the information stored earlier. At each perform we generate the invoke tag and the corresponding copy tags to that process. This way we fulfill the flow structure.

Once all the stages have been performed the BPEL file is created. But, this is still unexecutable without the WSDL file needed for reference. This is discussed in the following section.

ExecuteBPEL

For the BPEL generated to be executed we need to create the WSDL file, which comprises of all the information we have added like partnerLinks and partnerLinkType. This is the only information we have added as all the other information is already grounded in the already existing files for each atomic process.

Thus, as these are the values we have generated, we have the liberty of naming them. Hence, we have used a common base name for each partnerLinkType we have added. For instance, we have atomic processes named *A*, *C* and *D*. We have stored each one's WSDL information in a table named AtomicProcess_1, AtomicProcess_2 and AtomicProcess_3.

We have associated with them namespaces ns1, ns2 and ns3 and if two of them have same namespaces, we have given them the same namespaces and recorded the data in the atomicproceses table. Thus, now each of the above process is a partnerProcess and hence requires a partnerLink. Therefore, they are assigned the link-type pair as, PartnerLink_1-PartnerLinkType_1 and so on. These partnerlinks are also stored in the parent table which has all the data which might be retrieved later. These include namespaces and partner links and partner link types. These partner link types are stored alongside the tablename and the atomicprocessname. Hence, we know the corresponding data for each atomic process.

At the time of WSDL construction we add these partnerLinkType tags with these standardized names. The attributes of portType and operation come form the atomic process grounding information grounding:portType and grounding:operation respectively,

AtomicProcesses		
AtomicProcessName	TableName	Namespace
A	AtomicProcess_1	ns1
C	AtomicProcess_2	ns2
D	AtomicProcess_3	ns3

Table 1: Database entries for table atomicprocesses for the example

CompositeProcesses	
CompositeProcessName	TableName
CompositeProcess_1	CompositeProcess_1

Table 2: Database entries for table compositeprocesses for the example

which has already been parsed into the table for the concerned atomic process.

Hence, with this information added we have WSDL grounding for all the BPEL process information.

Thus, with the WSDL and BPEL files in place we are ready to execute the generated BPEL process.

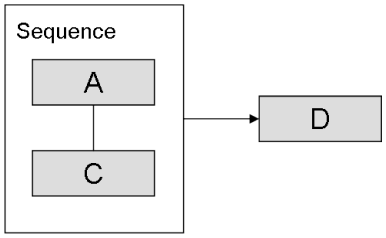

Figure 17: The OWL-S process

Considering the example specified in figure 3, if we assume Paul is a new customer and the credit is greater than the set limit, then RHEA provides us with the OWL-S process as depicted in figure 17.

Hence, given the OWL-S process we parse it using the Jena2 API (Stage 1). This is depicted in figure 15.

Thereafter, we proceed to Stage 2, which is parsing the type of processes and creating appropriate tables. Here we have three atomic processes namely A, C and D and one composite process say CompositeProcess_1. Thus, the tables and the entries are as shown in Table 1 and Table 2. Later in this stage we also create separate tables for each atomic and CompositeProcess which are filled later in stage 3 and 4.

Moving on with stage 3, in this stage we parse the construct tree for the composite process. The recursive algorithm mentioned above yields the field values for table CompositeProcess_1 as depicted in figure 17. To get an idea Some of these values are depicted in Table 3.

Moving on to stage 4, in this stage we fill the tables of each atomic process with the grounding information and later generate tags using the construct tree

Construct Name	Construct Type	Parentstruct	Tree level	rank
Sequence_2	Sequence	Sequence_1	1	0
Perform_D	Perfrom	Sequence_1	1	1
Perform_A	Perfrom	Sequence_2	2	0
Perfrom_C	Perfrom	Sequence_2	2	1

Table 3: Database entries for table CompositeProcess_1 for the example

and grounding information. Here in figure 18. we depict the data used for generating respective BPEL tags and the operations done to get the data required for constructing each tag. Please note that the <bpel:construct> used in the figure is not a real tag and is a generic notation for construct tags like <bpel:sequence>, <bpel:split> etc.

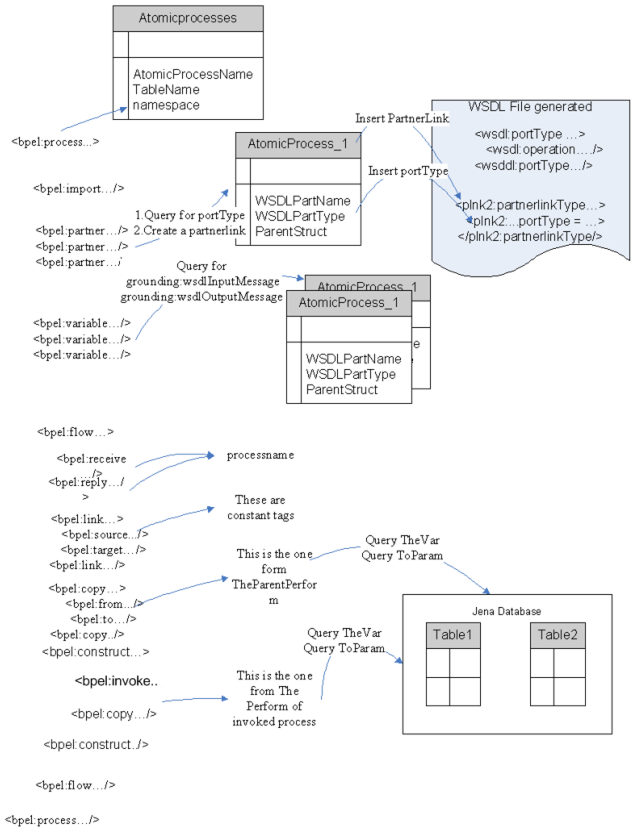

Figure 18: The BPEL tag generation, operations used and WSDL generated(it is a depiction)

Finally, for execution we generate the appropriate WSDL file as depicted in figure 18. It requires us to query the portType and add it to the generated partnerLinkType tag. We also need to add new portTypes and operation for the partnerLinkType generated for Receive and Reply processes.

Conclusion and Future works

In this paper we have shown how unstructured process parts within the predefined control constructs of BPEL can be modeled, by replacing the unstructured parts using a new modeling object type.

During runtime, we have represented an approach of combining the static and dynamic parts by using BPEL, semantic technologies and business rules.

We have first explained how we can model these dynamic parts using semantic web technologies. After it we have shown, how this approach can be executed.

This approach leads to more flexible and adaptable processes, because the final process flow is determined during runtime and existing BPEL process flows can be easily adapted by calling the web service RHEA.

Further work must be done because of the return values of the executed sub tasks. If the relevant data changes because of the execution of a sub task, RHEA must be invoked again, because with the changed values, other tasks can be invoked. In the approach we presented in this paper, we only invoke RHEA once.

Another problem can be recognized as the use of OWL-S for our approach. We only use the two concepts AtomicProcess and CompositeProcess, so we do not use the whole standard of OWL-S. We also need to extend the conversion of OWL-S to BPEL to support exception handling and fault handling constructs.

References

Alonso, G.; Casati, F.; Kuno, H.; and Machiraju, V. 2004. *Web Services, Concepts, Architectures and Applications*. Berlin, Heidelberg: Springer-Verlag.

Aslam, M.; Auer, S.; Shen, J.; and Herrmann, M. 2006. Expressing business process model as owl-s ontologies. In *2nd International Workshop on Grid and Peer-to-Peer based Workflows (GPWW 2006)*, 400–415.

Beer, T.; Rasinger, J.; Höpken, W.; Fuchs, M.; and Werthner, H. 2007. Exploiting e-c-a rules for defining and processing context-aware push messages. In *RuleML*, 199–206.

Bider, I.; Johansson, L.; Perjons, E.; and Striy, A. 2006. Activation of knowledge in an integrated business process support/knowledge management system. In *PAKM*, 13–24.

Chow, L., and Medley, C. 2007. Bpm and service-oriented architecture teamed together: A pathway to success for an agile government.

Geminiuc, K. 2006. A service-oriented approach to business rules development. http://www.oracle.com/technology/pub/articles/bpel_cookbook/geminiuc.html.

Hammer, M., and Champy, J. 1996. *Business Reengineering, Die Radikalkur für das Unternehmen*, volume 7. Frankfurt/New York: Campus Fachbuch.

Henbury, C. 2007. Two definitions of agility. http://www.cheshirehenbury.com/agility/twodefinitions.html.

Hinkelmann, K.; Probst, F.; and Thönssen, B. 2006. Agile process management framework and methodolgy. In Abecker, A.; Seth, A.; Mentzas, G.; and Stojanovic, L., eds., *Semantic Web Meets e-Government*.

Reynolds, D. 2007. Jena2 inference support. http://jena.sourceforge.net/inference/.

Rosenberg, F., and Dustdar, S. 2005. Business rules integration in bpel, a service-oriented approach. In *Proceedings of the 7th International IEEE Conference on E-Commerce Technology (CEC 2005)*.

Ross, R. 2003. *Principles of the Business Rules Approach*. Addison-Wesley.

von Halle, B. 2002. *Business Rules Applied, Building Better Systems Using the Business Rules Approach*. New York: Wiley and Sons.