SEMCLOUD: USING SEMANTICS TO IMPROVE AUTOMATION ON A CLOUD

by

NAGA KRISHNA GOLLAPUDI

(Under the direction of John A. Miller)

ABSTRACT

A critical enabler of mass scale software deployment on the cloud will be the ability to compose applications from different providers to create composed applications. While a number of composite applications are prevalent as manually created mashups, we contend that more automation is needed for composing applications and mediating among them. However, cloud application descriptions currently lack semantic descriptions, making the task of providing any automation infeasible. We present a framework called SemCloud, which uses a specification called SAWADL (Semantically Annotated Web Application Description Language) to provide automated composition and mediation of cloud applications. In our framework, all the applications are semantically described using SAWADL and an extended Artificial Intelligence(AI) graph planner is used to suggest compositions based on the user's goals. The planner also provides support for data mediation. We present an evaluation over real world cloud applications. Based on the annotations, SemCloud was able to automatically generate output for mashups or applications, except for any user interface(UI) code.

INDEX WORDS: Cloud computing, Semantics, Composition, WADL, SAWADL, REST, AI Graph Planner

SEMCLOUD: USING SEMANTICS TO IMPROVE AUTOMATION ON A CLOUD

by

NAGA KRISHNA GOLLAPUDI

B.Tech., Acharya Nagarjuna University, 2006

A Thesis Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2010

SEMCLOUD: USING SEMANTICS TO IMPROVE AUTOMATION ON A CLOUD

by

NAGA KRISHNA GOLLAPUDI

Approved:

Major Professor: John A. Miller

Committee: Hamid R. Arabnia
Shelby H. Funk

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
July 2010

DEDICATION

To Padma, Mohan Rao, Srijata, Srini and Yamini

ACKNOWLEDGMENTS

I am heartily thankful to my major professor, John A. Miller, and my thesis lead, Kunal verma, for encouragement, guidance and support from the initial to the final level which enabled me to develop an understanding of the subject. Lastly, I offer my regards to all of those who supported me in any respect during the completion of my thesis.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The ability to create and use software in the cloud has gathered momentum. This is largely due to the growing ubiquity of the Internet and the growing popularity of cloud based development platforms such as AppEngine (www.appengine.google.com), Force.com (www.salesforce.com) and Azure (www.microsoft.com/windowsazure). Even prior to the advent of cloud computing, many resources such as Facebook, iLike, BestBuy, etc. have made it easier to access data and URLs to combine the data as a mashup and create innovative new Web applications. Usual issues of creating a scalable web application had to be taken care of such as catering to any number of requests occuring for the application and maintaining minimal variations in response time. Now cloud computing has made it easier by handling the Quality of Service (QoS) for a given application. A cloud provider periodically checks the capability of servers allocated to the current usage per application and if this ratio surpasses a threshold, then a new server is added to the load balancer by copying an instance of the software into it. Thus recently, the focus is shifting from monolithic cloud applications from a single provider to powerful composite cloud applications. For example, Force.com allows users to create new applications by leveraging other resources such as Facebook and Google Apps. As the number of cloud applications proliferate, there will be a number of significant challenges:

Will there be a sophisticated way to search for applications on the cloud? The Google App search engine uses the PageRank algorithm [22]. On a search result a user has to spend time to verify each result. These algorithms do word matching to produce the results. A better approach is to compare the semantic meaning of the application and check for compatible input/output parameters.

How will users find the relevant building blocks for their composite cloud applications? The Programmable Web (www.programmableweb.com) published a list of steps on creating a mashup from scratch using independent APIs (Application Programming Interfaces). Five steps summed up the workflow in which "Sign-up for an API" is a crucial one. A developer has to choose apt API calls to create a mashup. For example, to create a mashup which takes in ArtistName as input and provides videos of that artist as output, one has to decide among 25+ Music APIs and 40+ Video APIs (statistics as per Programmable Web).

Will there be frameworks for rapidly composing composite cloud applications from non-trivial building blocks? A number of mash-up editors such as Yahoo Pipe (www.pipes.yahoo.com) and Google Mash-up editor (now replaced by AppEngine) have tried to provide a framework for composing applications. However, most of the editors were restricted either in choice of application providers or in any level of automation for creating composite cloud applications.

Will there be any (semi-)automatic data mediation possible across cloud applications from different sources? Data mediation, will result in failed execution of a composition process if they are not properly maintained. Even when a plan for a process is generated for a composition, it has to be executed successfully. Different components in the process may not use the same domain terminology. For example, UserName in one domain can be UserID in another.

We believe that all these questions must be answered for the cloud to become a primary medium for enterprise software. We also believe that unless cloud-applications are formally described, it will not be possible to create a comprehensive framework for creating cloud software applications. To that effect, we provide a specification for semantically describing Web applications called Semantically Annotated Web Application Description Language (SAWADL), which we created by extending an existing specification called Web Application Description Language [7] (WADL). SAWADL is based on the ideas of Battle and Benson [28] who proposed a bridge called SBWS (Semantic Bridge for Web Services). SBWS is a Java tool which wraps a set of Web Service operations described by a WSDL [27] or WADL [7] document to create SPARQL SELECT or CONSTRUCT queries. In [28] paper, they provide SPARQL endpoints to the services

with existing REST (Representational State Transfer Protocol) or SOAP (Simple Object Access Protocol) end points. In our case, we have added semantics to the resource descriptors, WADLs [7], and modified the parser to grab the semantic information about the resource. Each application is annotated in a single SAWADL file describing input to and output from the application, as well as preconditions to be satisfied before and effects that occur post execution.

All the annotated SAWADL files are given to an extended AI (Artificial Intelligence) graph planner which is built using Graph Plan [1] , an AI planning algorithm. The graph planner takes the intial state and goal state from a process descriptor and automatically generates the control flow for the process. We use the extension of graph plan proposed by Wu [29] who extended the graph plan by taking the structure and semantics of the input and output parameters into consideration in the planning algorithm. Wu [29] focuses on composition of SOAP-based web services. Previous efforts related to Web service composition considered various approaches, and included use of HTN [4] [5] , Golog [8] [9] [10], classic AI planning [11], rule based planning [13] [14], model checking [15] [17] [18], theorem proving [20] [21] [24] [25], etc. Wu [29] proved that his extension to the graph planner was a good choice to overcome both process and data heterogeneities which were not given equal treatment in previous approaches. Once a control flow is generated for the web process, our interpreter, written in Java, executes the process by using Apache HTTP Client software. The interpreter fetches the first operation from the solution set, generates a URL and fetches the response using HTTP (Hypertext Transfer Protocol) methods. This response is provided to the next inline operation for input. This action continues until the solution set is totally parsed and all the intermediate parameters are saved. Once the solution set is empty, the interpreter generates the output from the Map of parameter and value pairs saved at each level.

This thesis is organized as follows. Chapter 2 gives background on cloud computing, mashups, semantic annotations and cloud resources. Chapter 3 describes the SemCloud Architecture with a suitable diagram. In chapter 4, all the necessary semantic annotations required to describe a resource are explained along with examples. Chapters 5 and 6 explain in detail, the core functionality of SemCloud Composition and SemCloud Data Mediation, respectively. Chapter 7

explains about the execution of the control flow for a process along with a couple of scenarios, loop pattern and parallel actions. Chapter 8 demonstrates SemCloud with a practical mashup. Chapter 9 discusses conclusions and future work.

CHAPTER 2

BACKGROUND

2.1 CLOUD COMPUTING - A HISTORY REPEATED

The history of computers started as centralized computing with time sharing. In those days, if a computer job needed to be done then one had to go to the operator, submit the job and pay the cost. Evolving technology gradually brought computing into the home. Individual computing devices have made life easier and made the job of the operator obsolete. Central server-based computing evolved into distributed computing. The advent of cloud computing is taking us back to the centralized model of computing. With cloud computing all the resources exist in a central place and the hardware is provided as a service (HaaS) with someone taking responsibility for all hardware management. That person will also manage the software and provide it as service, Software as a Service (SaaS). Cloud computing can be defined as a specialized form of standard distributed computing. The vendor of cloud based resources takes the responsibility for the performance / reliability / scalability of the computing environment. From an application developer's point of view, this can be a tremendous advantage, as procuring, maintaining, tuning, monitoring and scaling hardware and software to meet the demands of growth is both difficult and expensive. The work in this thesis focuses more on the user's perspective of cloud.

Five advantages, amongst many, of cloud computing are defined as follows:

1. Reduction of computing costs in organizations.

2. Data and applications can be accessed from anywhere via mulitple devices like terminals, mobile, netbooks.

3. Centralized and agile with practically no down time.

Table 2.1: Mash up tools and support to three types of components

Tool	DA	AL	UI
Yahoo Pipes	Yes	Yes	No
Google Mashup Editor	Yes	Yes	Yes
Popfly	Yes	Yes	Yes

4. Improved security as compared to scattered network.

5. Comparitively easy to manage as it puts everything together in a very organized manner.

A few disadvantages will include loss of control, relying on network connections and external charges for computing services.

2.2 MASHUPS

The Computer Dictionary defines a Mashup as a mixture of content or elements . This aptly reckons the feature of utilizing independently available resources to create a composite resource. Components used for mashups can be of three types, namely Data (DA), Application Logic (AL) and User Interface (UI). DA provides structured data. Web Services can be listed as AL components as they perform some logic to return outputs. Finally, UI components require a user to interact with software. Apart from knowing about the components to create a mashup, one needs to know about the component's interface and its extensibility. Table 2.1 provides information about the mashup tools regarding these three component types. A composition model for mashups needs several distinct characteristics. Obviously, one of which is the component type. Another characateristic is orchestraction style. Different styles such as Flow-based, Event-based, Layout-based can be used in creating a mashup. The final characteristic is data mediation. Data between two components can be exchanged directly or via variables.

2.3 Semantic Annotations

Discovery and composition of resources depend on adequate information. To automate composition, information needs to be machine readable. This led to the advent of semantic technologies in Web Services for composition. Many flavors of semantic annotations like Resource Description Framework (RDF) [**?**], Web Ontology Language (OWL) [19], etc., are being used to express content. These successes motivated us to use semantic technologies to annotate resources and work on automated composition in the cloud.

2.4 Cloud Resources

A resource in our context can be an API Method, an application or a custom built method inside an application. Major differences between an API and an application is their response and user interaction. Structured responses are received upon API calls, while this is not guaranteed for applications. Applications which do not have a REST [12] or SOAP [16] interface have Graphical User Interfaces (GUI) filled with design patterns, style sheets, validation scripts, etc. As composition on a high level is defined as executing a workflow comprised of several resources, at each stage the response of a certain resource has to be passed on to next inline resource. So, having resources with structured response like XML [23], JSON [26], RSS [2], etc., is necessary for automated composition. Applications are built for users. So, it is fair to mention that a single step response is unassured. To get the output from an application, a user has to go through a specific set of graphic screens interacting at each level. On the other hand, API calls result in a single step response without any user interaction. All the above mentioned specifications allow us to predict the extent of automation that can be achieved.

Composed cloud applications can be categorized into three categories based on the resources they are built with. The first category of applications are solely built on existing API methods. These applications also can have any number of external API calls to fulfill the application's goals. The second category of applications are built using a mixture of API calls and custom

methods. When pre-existing API methods do not fulfill the necessary requirements, custom built methods are created. The final category of applications are solely built using custom methods. Each of these categories have to be handled differently based on their responses.

2.5 WHY DO WE NEED SEMCLOUD?

Mash up editors like Yahoo Pipes (www.pipes.yahoo.com), Google Mashup Editor, etc. provide a GUI for human interactive mashup of applications with predefined responses. The responses in the case of Yahoo Pipes (www.pipes.yahoo.com) are defined by the functions used such as FetchCSV, FetchFeed, etc. These editors can not automate the process for two reasons. First, there is no machine readable content semantically describing a resource. Second, user input is required at each stage of process. Consider the Facebook mashup app named Shelftalkers hosted on Google App engine. A summary of this mashup is given below:

1. Login to Facebook.

2. Ask user to enter a music artist name.

3. Get album names for this artist.

4. Ask user to select an album from the list.

5. Get audio streams for this album.

At each step of this process, user input is required which makes complete automation impossible. Shelftalkers uses the Bestbuy API to get album names and iLike API to get audio streams. A developer has to understand the behavior of the operations used from different APIs. Consider a modified Shelftalkers mashup summarized below:

1. Login to Facebook.

2. Get Music from the user's facebook profile.

3. Get album names respective to one musician.

4. Get audio streams for costliest album of this musician.

In this modified mashup, user input is not required at any stage except at the intial stage via login. Even though structured response is expected from the API calls, they are not defined in these

editors. So, neither of these cases can be built using current mashup editors. SemCloud proposes ways to handle both of these situations.

CHAPTER 3

SEMCLOUD ARCHITECTURE

The SemCloud Architecture, fig:3.2, contains four major components - Discovery, Annotator, Composer and Data Mediator. In this thesis, the last three components will be covered to good extent leaving short descriptions about Discovery. The Annotator is the base for SemCloud. The rest of the componenets will fulfill their goals only if annotations of the resources are done properly. The Annotator proposes a way to describe a cloud resource by adding semantic information. Annotations to resource descriptors are written by developers with necessary domain knowledge. Tools can be used to automatically generate semantically annotated descriptor file for a resource. The Composer utilizes the annotated descriptors and a modified AI graph planner to sketch a process. The Data Mediator provides necessary transformations during execution of the process.

Current day cloud computing does not provide these features. For example, Google App engine alone has 10,000+ apps categorized to some extent using groups such as News, Sports, etc. If these apps are annotated and their semantic descriptions are stored in an XML database, then discovery of a resource can be simplified and automated by matching user requirements with content in the descriptions. Two pieces of work will greatly assist SemCloud. First, resources should be annotated by their developers and should create a SAWADL file. Second, cloud providers must share these descriptors in their databases. If both these steps become common practice, SemCloud can be implemented successfully.

A high level description of SemCloud composition is given in fig:3.1. Action specifications along with initial state, goal state and goal conditions are given to a planner. The planner generates a plan adding compatible actions to the intial state, then modifying the state until it reaches the goal state. There are three possible methods of executing a generated plan. One method is by pipelined

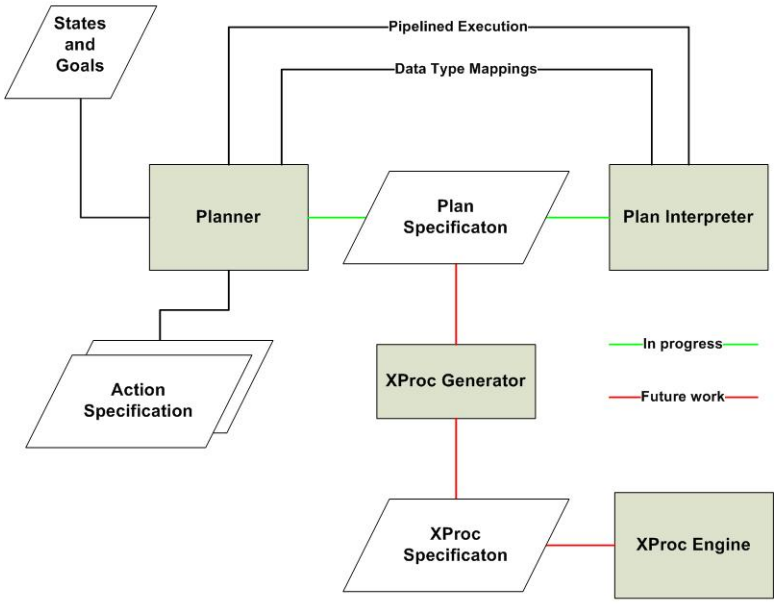

Figure 3.1: SemCloud Composer Description

execution, in which the plan interpreter gets a set of action (plan) and a data type mapper. The data type mapper has the relations between states. SemCloud currently has pipelined execution. The second method of execution is to create an XML specification for the plan and for the data type mapper before giving it to the plan interpreter. This removes the direct dependency between the planner and interpreter. SemCloud currently does not have this feature, but it is work in progress. The third method of execution is by generating an XProc specification using an XProc generator which can be executed using an XProc engine. This feature is written as future work.

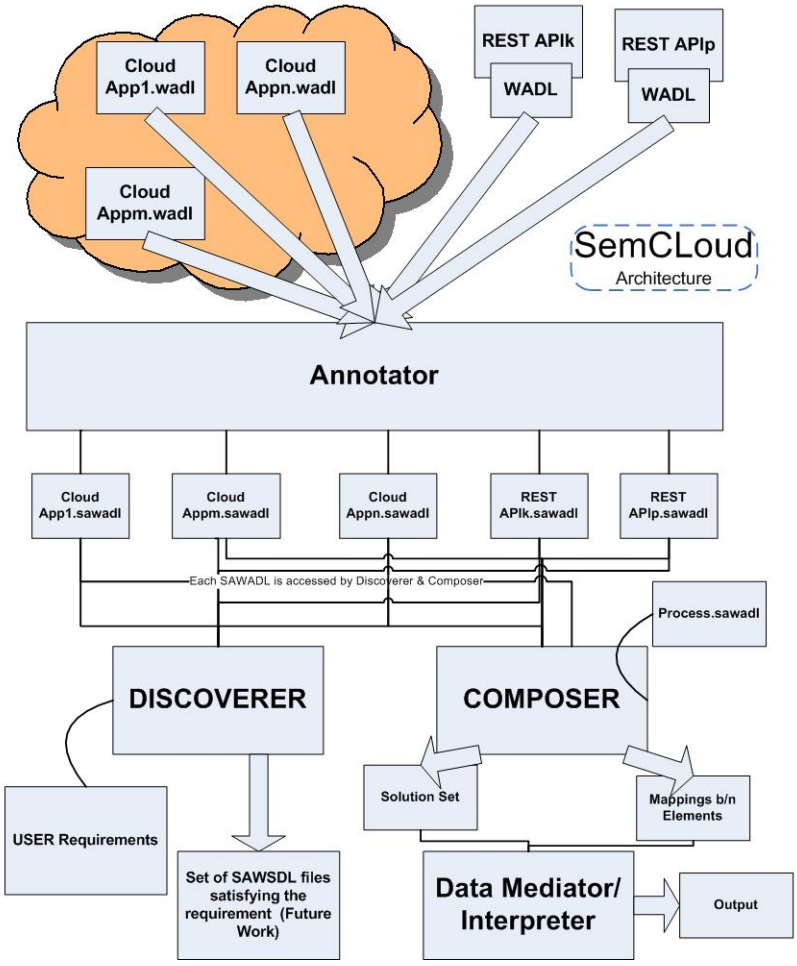

Figure 3.2: SemCloud Architecture Diagram

CHAPTER 4

ANNOTATION - SAWADL

Marc Hadley of Sun Microsystems authored a specification called as WADL (Web Application Description Language) to provide a machine process-able description of HTTP-based Web applications. As SemCloud is a proposal to add semantics to HTTP based cloud applications, we extend the existing well known WADL specification and name it as SAWADL (Semantically Annotated WADL). The idea of utilizing the existing WADL end points to provide semantic information was provided by Battle and Benson [28] who use a tool to translate the WADL end points to SPARQL end points. For SemCloud, extensible elements of WADL are used to provide semantic information for an application. These semantic annotations also relate to XML schema.

4.1 MODELREFERENCE

A modelReference attribute can be attached to any extensible element in a WADL or a XML schema. SAWADL identifies modelReference only to Application, Resource, Representation (request or response), Params (request or response), XML schema elements and XML schema attributes. The value of the modelReference attribute is a set of zero or more URI's which identify concepts in a semantic model. ModelReference to SimpleType elements is trivial, while modelReference to ComplexType elements is achieved by Bottom Level Annotation and/or Top Level Annotation.

4.1.1 ANNOTATING APPLICATION WITH MODEL REFERENCE

These annotations help to categorize applications using a semantic model. When provided for cloud applications, these annotations will help cloud providers to add applications automatically

```
<xs:schema
  targetNamespace="http://www.example.org/ns/sawadl"
  xmlns="http://www.example.org/ns/sawadl"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:wadl="http://wadl.dev.java.net/2009/02">
  <xs:simpleType name="listOfAnyURI">
    <xs:list itemType="xs:anyURI"/>
  </xs:simpleType>
  <xs:attribute
    name="modelReference" type="listOfAnyURI" />
  <xs:attribute
    name="liftingSchemaMapping" type="listOfAnyURI" />
  <xs:attribute
    name="loweringSchemaMapping" type="listOfAnyURI" />
</xs:schema>
```

Figure 4.1: SAWADL Schema

```
 <wadl: application name="LyricsFly"
   sawadl:modelReference="Ontology#Music"/>
 <resources>
 <resource path="lyricsfly"
  sawadl:modelReference="Ontology#GetLyricsForSoundTrack">
</resources
```

Figure 4.2: Application and Resource Annotation - modelReference

```
<xsd:complexType name="albumtype">
 <xsd:sequence >
  <xsd:element name="soundtrack"
     maxOccurs="10" type="sampletype"/>
  <xsd:element name="regularPrice"
     nillable="false"
     sawadl:modelReference="Ontology#AlbumPrice"/>
   <xsd:element name="asin" type="xsd:string"
     nillable="false"
     sawadl:modelReference="Ontology#ASIN"/>
            </xsd:sequence>
</xsd:complexType>
<xs:attribute name="quantity" type="xs:integer"
    sawadl:modelReference = "Ontology#Quantity"/>
<param name="artist" type="xsd:string"
    sawadl:modelReference = "Ontology#SoloMusicArtist" >
```

Figure 4.3: Other modelReference annotations

to categories such as News, Communication, Games, etc. (category names taken from Google App Engine).

4.1.2 Annotating Resource with modelReference

As each SAWADL in our scenario has only one resource, these annotations provide the semantic meaning for intended behavior of the application. Resource annotations can be helpful in situations of application discovery.

4.1.3 Other modelReference annotations

All the above mentioned annotations assist in various operations of the SemCloud framework. As complexType elements cannot be described either with Param or with Grammars in WADL, the only way to incorporate this type is by a XSD [3] schema added to a request or response elements using the representation tag.

```
<param name="artist" type="xsd:string"
 sawadl:modelReference="Ontology#SoloMusicArtist"
 sawadl:liftingSchemaMapping="UserNameCase.xslt">
<param name="artist" type="xsd:string"
 sawadl:modelReference="Ontology#SoloMusicArtist"
 sawadl:loweringSchemaMapping="RDFOnt2UserName.xml">
```

Figure 4.4: Param Element - SchemaMappings

4.2 SCHEMAMAPPING

LiftingSchemaMapping and LoweringSchemaMapping help to resolve mismatches between the semantic model and the structure of inputs and outputs. Schema mappings assist in data mediation during composition of applications. Consider an application is that produces an output in a format that differs from the input format required by the application in the next stage of the plan. In such cases schema mappings will help to lift/translate the output into a semantic model, convert it into an appropriate format and then lower/translate the data to be used by the next application.

4.2.1 ANNOTATING PARAM WITH SCHEMAMAPPINGS

Consider an application which needs a parameter, username, to be in certain case as input. To denote this feature, a transformation liftingSchemaMapping is added to param tag. It lifts UserName to a semantic model using an XSLT. The resulting concept is then converted to the appropriate form inside the semantic model which is then lowered as input to the application.

4.2.2 OTHER SCHEMAMAPPING ANNOTATIONS

These annotations are present outside of a given SAWADL, for example in an XSD. SAWADL has only one way to import external documents, i.e., through XSD Schema elements. The grammars element of SAWADL holds all the necessary schema documents for a given application.

```
<xs:element name="topalbums" type="albumType"
  sawadl:liftingSchemaMapping="OrderItem2Ont.xslt"/>
<xs:simpleType name="albumname"
  sawadl:modelReference = "Ontology#AlbumName"
   sawadl:liftingSchemaMapping="Response2Ont.xslt">
</xs:simpleType>
<xs:element name="start"
  sawadl:loweringSchemaMapping="RDFOnt2Request.xml">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="status" type="xs:integer" />
    <xs:element name="tx" type="item"
      sawadl:modelReference="Ontology1#Lyrics"/>
    </xs:sequence>
   </xs:complexType>
</xs:element>
```

Figure 4.5: SchemaMappings - Other Elements

CHAPTER 5

COMPOSITION

5.1 SEMCLOUD COMPOSITION

For the purpose of composition, this model focuses only on the abstract representation of the resources. Wu [29] proposed this model and successfully implemented it for composition of SOAP Web Services. We adopt the same extension of GraphPlan [1] to use it for cloud resources. Most classic AI planning problems are defined by the STRIPS representational language, which divides its representation scheme into three components, namely, states, goals, and actions. Our model extends STRIPS as the representational language. Extended state has semantic data types ensuring the availability of data before any action is invoked. So, each state of the process has semantic data types (SDTs) and semantic status flags (SSFs). Each status flag is one atomic statement with a URI in a controlled vocabulary. Ternary logic is used when describing status flags. Thus, the truth values of any status flag are True, False or Unknown, i.e., any unmentioned status flag is Unknown. On the other hand semantic data types are membership statements in Description logic of a class or union of classes in an ontology. SDTs represent availability of data and they need good expressive power to compare related messages, such as those of sub-class relationships.

The composition process starts by instantiating the initial state and goal state from a process descriptor. The process descriptor in our case is another SAWADL file with one resource having one method. The request element of this file has the initial state data types of the process and the response element of this file has the goal state data types. The precondition expression from the semantic annotation will be taken as the initial state status flags, while the effect expression will be taken as the goal state status flags. After the initial and goal states are determined, actions are generated for all the available resources. This step will be modified in future when there is a

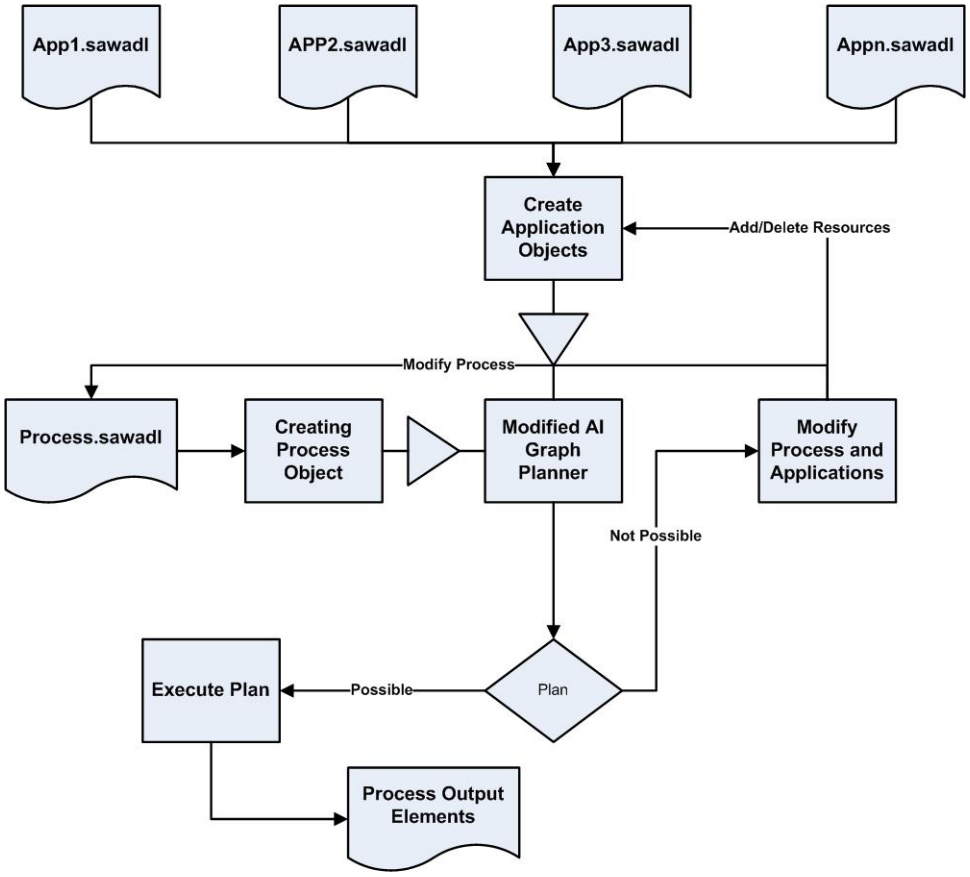

Figure 5.1: SemCloud Composition Process Diagram

XML database. For now, a list of candidate resources which can participate in the composition are provided to the system via descriptor files. The next section summarizes how an action effects the current state.

5.1.1 Process Generation

As said before, the initial state is generated from the process description. Now, from this initial state a process needs to be generated to reach the goal state, which is also generated from the process

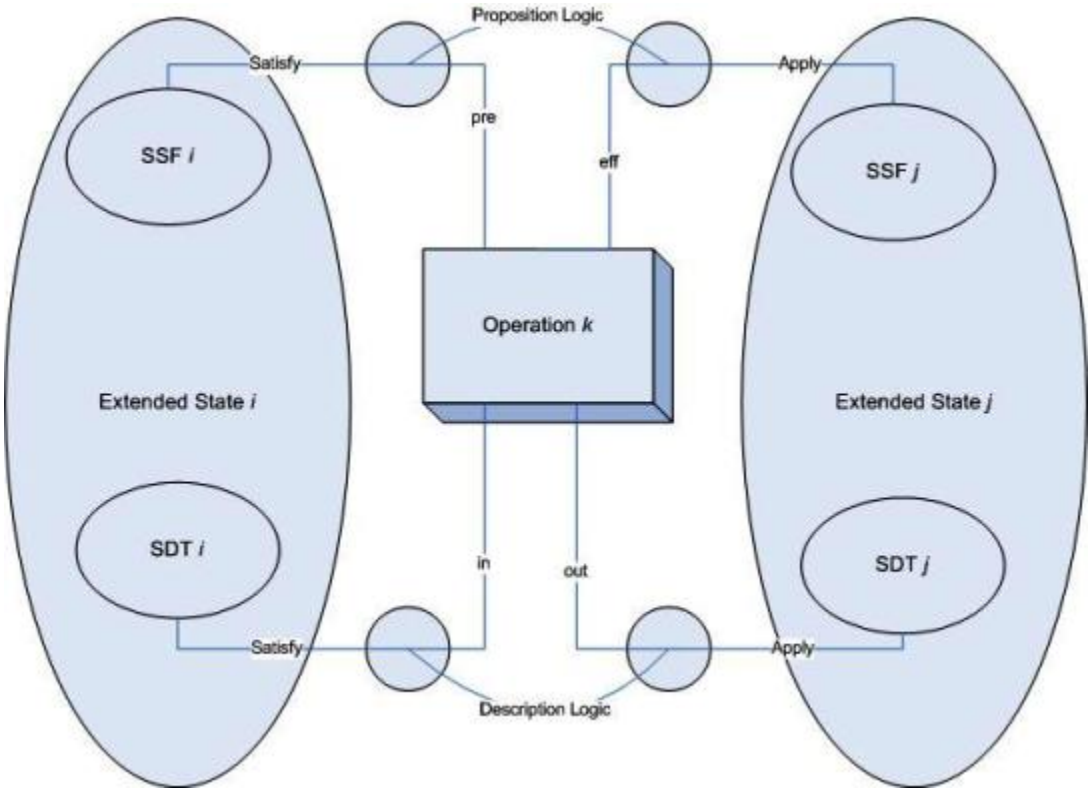

Figure 5.2: SemCloud Operators Diagram

description. Any resource (Res) needs to get a satisfactory cerificate from the *S*atisfy operator against the current extended state.

The steps followed by the Satisfy operator for a given resource:

1. Fetch the preconditions for the resource.

2. Check if these preconditions are satisfied by the current state.

3. If satisfied, fetch the input data types of the resource.

4. If all the necessary input data types are available in this state, i.e., input data types to this resource are semantically matched to data types present in this state,this resource satisfies the current state.

As mentioned before, semantic matching is done using the modelReference attribute of the data types. A score is generated for each comparision and a cut off is set to generate matched data. Once Res gets a satisfactory certificate, it is added to the process and the next extended state is created by using the *a*pply operator.

The steps followed by the Apply operator to add the resource and create the next state are listed below:

1. Add the output data types of the resource to state data types.

2. Modify the status flags with respect to positive effects and negative effects of the resource (Res)

3. Any status flag not in the effect of Res is left unchanged.

After each set of successful Satisfy and Apply operators, the Satisfy operator issed to compare the new state to the goal state. Once the goal state is reached, the Composer generates a plan by backtracking from the goal state to the initial state, thus producing a sequence of operations (REST Services) representing the generated process.

CHAPTER 6

DATA MEDIATION

6.1 WHY DO WE NEED IT?

Given the vast variety of categories of resources existing on a cloud, it is extremely difficult to build resources with data types having the same syntactic form and semantic meanings. Data mediation assists run time execution of the proposed web process. Components of a web process will have different domain specifications. We consider data mediation as critical for generating such executable web processes. For example, consider that a data type SDT1 with schema SXSD1 is the output of action A1 and data type SDT2 with schema SXSD2 is the input to action A2 which succeeds action A1. To build a successful automation of a web process consisting of these two actions, SDT1 should be transformed into SDT2 without any human interaction.

6.2 HETEROGENEITIES

Now let us consider the types of heterogeneities [30] that can exist between these two data types - semantic, syntactic and structural. Environments such as XML address the syntactic heterogeneities. Semantic and structural have to be taken care of. Semantic heterogeneties between output of one action and input of next action means that terms with different names may refer to the same concept, or terms with same name may refer to different concepts. Nagarajan [30] explains in detail about semantic interoperability of web services which can also be realized with any resource. These heterogeneities are reduced when semantically annotated descriptions of the resources are used. That is, modelReferences (classes or concepts in an ontology), the data types are compared during the generation of the web process. So, two different terms can mean the same

to the composer when their modelReferences are exact matches. This makes sense only when there is a single ontology covering both domains. If there are two ontologies for both actions, then there should be ontological mappings between them.

Structural heterogeneties occur when SXSD1 and SXSD2 are different, but SXSD2 can be generated from SXSD1. To put simply, consider that A1 provides a complete name of an individual as output, but A2 needs the first name only. Following the SAWSDL approach, these differences are dealt with using schema mappings. The output of A1 is upcast into an ontology using 'liftingSchemaMapping' inside the SAWADL of A1, converted into the required format for A2 inside the ontology and then downcasted using 'loweringSchemaMapping' inside the SAWADL of A2.

6.3 SemCloud Data Mediation

The composer in the SemCloud architecture does the background work of creating mappings between data types of all the actions listed in the process. XPaths of the data types are used to create mappings. XPath validation is an important feature of the composer. It utilizes this validation in two steps. Step one is to compare the current extended state to the goal state. When this comparison fails, the second step of comparing all the available actions to the current extended state is carried out.

Consider that there is a source message (M1) and a target message (M2). The basic idea is to traverse M2 in top down approach and try to fill up each node by using the data from M1. Typically web resources have params (simple types) as input and structured responses (XML, JSON, etc.) as output. This helps in the way that simple types are taken from an XML and passed on to the resource. The composer generates a map with one XPath as a key and a list of all known matches to this XPath as the value. The creation of this map is described by the Fig:6.1. The match is formed using semantic knowledge of the data types in both XPaths. In a given instance of generating the next extended state during composition, if there are two match possibilities for an XPath in current state, then a Context-based ranking algorithm is used to generate a score. Wu [29] explains the context-based ranking algorithm in detail.

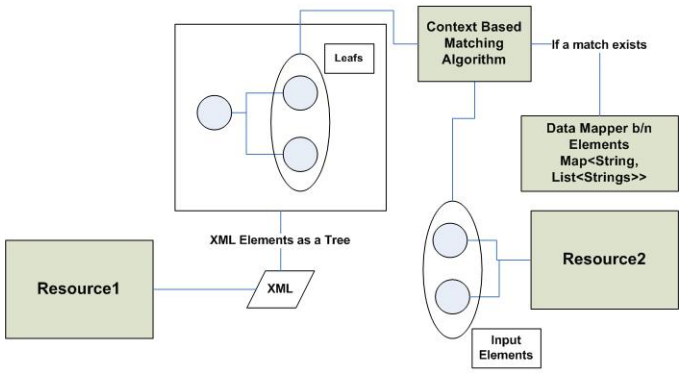

Figure 6.1: Data Mapping Creation

6.4 JSON TO XML TRANSLATION

SemCloud for now is an XML based environment and it supports the JSON environment. Thus, if a resource has a JSON response, it is tranformed into XML. The indication of the resource having a JSON response is provided in the response element of the SAWADL file. An example of our translator is given below.

```
   "id": "742086493",
   "music":
      "data": [

            "name": "Usher",
            "category": "Musicians",
            "id": "6564142497"

      ]
```

Figure 6.2: Javascript Object Notation (JSON) Snippet

```
<?xml version="1.0" encoding="UTF-8"?>
<o>
 <id type="string">742086493</id>
<music class="object">
<data class="array">
<e class="object">
<category type="string">Musicians</category>
<id type="string">6564142497</id>
<name type="string">Usher</name>
</e></data></music></o>
```

Figure 6.3: XML Translation

CHAPTER 7

PROCESS EXECUTION

7.1 EXECUTOR

Traditional workflow execution engines have several characteristics such as maintaining the control flow, assuring data flow, handling exceptions, etc. The SemCloud workflow execution engine handles a subset of them. It can handle Loop and Parallel Split workflow patterns [6]. At a given state during execution, multiple actions can get triggered. For example, in the ordered list of actions given from the Composer to the Executor, there may be an instance where more than one action has to be executed per state, that is, response of one state goes to more than one action which in turn result in multiple responses which have to be considered for further states. For these actions a list of temporary response files are generated. Also, execution of one action is totally independent of the other given the fact that input to both these actions is from a single file.

The loop pattern is better explained with an example. Consider an action with a list of orders as a response and is succeeded by another action which adds a single order to its company database. In such scenarios, an action has to be repeated several times to fulfill the process requirement. This feature is incorporated by using minOccurs and maxOccurs attributes of XML elements. If maxOccurs of an element from the first action has a value greater than 1 or unbounded, while its successor action requires an matching input element with maxOccurs equal to 1, then the action has to be undergo a loop pattern.

Data flow is maintained using files. Invocation of a resource is done using Apache Http Client. The Executor is not pre-tuned to one specific resource. It is generic in the sense that every relevant detail about a resource is fetched from the SAWADL descriptor. Consider the SAWADL file shown in Fig:7.3

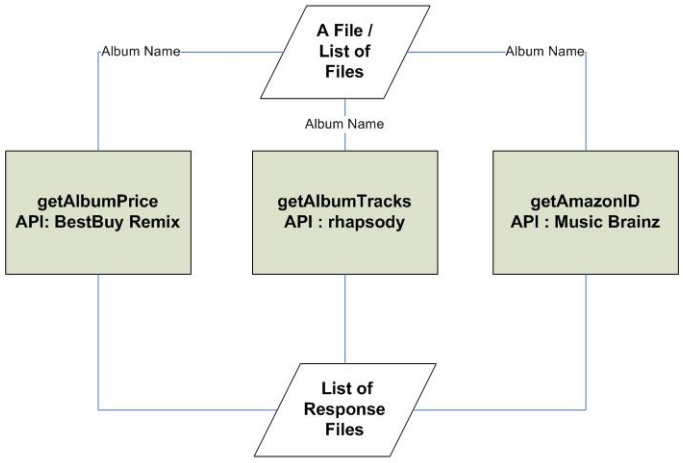

Figure 7.1: Parallel Split Workflow Pattern

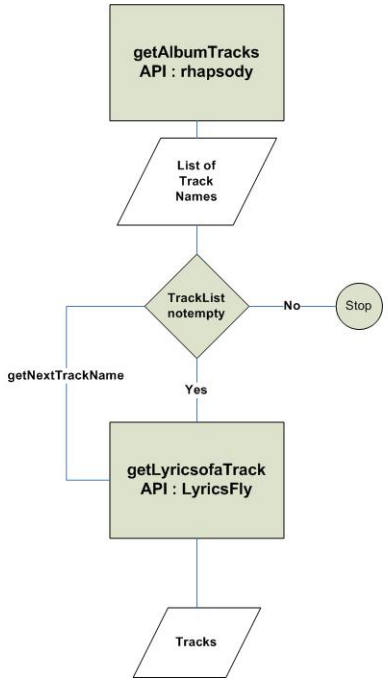

Figure 7.2: Loop Workflow Pattern

```
<resources base="graph.facebook.com" protocol="https" count="1"
                    paramname1="fields" paramvalue1="id,music">
  <resource path="friendfbId" param="true">
       <method name="POST" id="getDetailsofFriend">
        <request>
          <param name='access_token'  type='xsd:string' style="query"
             sawadl:modelReference="Ontology1#FaceBookAPIKey"/>
          <param name='friendfbId' maxOccurs="1" type='xsd:string'
           style="query"
           sawadl:modelReference="Ontology1#Person"/>
        </request>
        <sawadl:precondition expression="selectedFBFriend"/>
        <sawadl:effect expression="retrievedArtistname"/>
        <response status="200">
          <representation mediaType="application/json"
             element="o"/>
  </response></method></resource></resources>
```

Figure 7.3: FaceBook.SAWADL

This SAWADL file describes an action called as getDetailsofFriend (method name in Fig:7.3) with input params, friendfbid and accesstoken (params of request element in Fig:7.3). This action gets the favorite artist for given facebook ID. As this resource has a REST interface, a URL has to be generated to invoke the action and get the response. We built a SAWADL parser to get the required information from a given SAWADL description file. The Executor uses this parser and generates the URI -

```
https://graph.facebook.com/[friendfbId]?fields=id,music&access_token=[]
```

Protocol, base URI and any additional parameters that will go with the URI are provided by the resources element in Fig:7.3. Param attribute, true in this scenario, of the resource element says that the path=friendfbId is an input parameter to this resource. Appendix C explains in detail about URI construction and the next section explains about using ontology for secured content of this URI. Then this URI is given to the Apache HTTPClient to retrieve the response.

The Executor receives a map and a solution set from the Composer. The map has the mappings between all mandatory XPaths, i.e., data types, while the solution set is an ordered list of

actions to fulfill the goal. Mappings are of two types - copy and transform. The copy type simply says that both data types represented by the respective XPaths are identical. The transform type says that one XPath has to tranformed into the other using a certain medium which will most likely be an ontology. SemCloud currently deals with identical XPaths, copy. The Executor gets the first action from the ordered list and creates a URI from its respective SAWADL and fetches the response. The initial phase of execution takes manual entry of necessary inputs. Now, this response is stored in a temporary file and passed on to next stage. As a second step it fetches the next action. Now this action requires, say two inputs Input1 and Input2 with XPath values InX1 and InX2. The Executor starts with InX1 and fetches the list of mapped XPaths. It then searches the available temporary response file for any of the XPaths from the list. Prediction is that this search will have return values; the Composer will not put forward this plan if this prediction in not true. The value of InX1 is passed on appropriately. This cycle goes on through until it completed the solution set. Ideally, the Goal state should be reached on the final action in the solution set. In most cases, the goal state is built using intermediate elements. Keeping this in mind, at each step the Executor saves the elements. Once the solution set is successfully executed, then all the saved elements with values along with outputs of the final action are used to generate the final state instances of this process.

7.2 Security using Ontology

SemCloud utilizes ontology to reduce security risks while handling sensitive information like API keys, passwords, etc. This thesis proposes that semantic descriptions need to be created and given access. Most of the information about a resource can be described in a SAWADL. It is restrictive enough to not display information which can attract false access via web crawlers. Consider that a resource needs a secured key as input to get a response. Its descriptor does say that it needs a secured key using a param - APIKey. Now consider the execution process. All the inputs required are taken from either the previous states or from the initial state. Secured keys will not be generated in intermediate states. This ruins the whole point of security. API keys cannot be generated by

```
<owl:NamedIndividual rdf:about="&shelftalkers;LyricsFlyAPIKey">
  <rdf:type rdf:resource="&shelftalkers;Key"/>
  <owl:topDataProperty>42cbcce797c8ec06f-temporary.API.access
  </owl:topDataProperty>
</owl:NamedIndividual>
```

and the entry for this input inside a SAWADL file is shown below:

```
<request>
 <param name='APIKey'  type='xsd:string' style="query"
    sawadl:modelReference="Ontology1#LyricsFlyAPIKey"/>
 <param name='artist' type='xsd:string' style="query"
    sawadl:modelReference="Ontology1#SoloMusicArtist"/>
 <param name='track' type='xsd:string' style="query"
    sawadl:modelReference="Ontology1#TrackName"/>
</request>
```

Figure 7.4: Security Using Ontology

a machine. Providers create GUI processes so that a user interacts with a resource in creating the API key. If this is not the case, then every web crawler will have this APIs key accessing content at will. So, they are provided in initial state. The values of these data types are stored in an ontology which will not shown to public. A secured data type is linked to ontololgy individuals via a modelReference. An example of such an individual (instance) is shown below,

Now that a framework is established for security, a list of these params is made. Whenever in amidst of a workflow, if this param is encountered then its value is retrieved from ontology.

CHAPTER 8

PRACTICAL SCENARIO

8.1 MASHUP PROPOSAL

In order to show all the proposed concepts in this thesis, we have chosen an extended ShelfTalkers mashup. The outline of this mashup is that for a given facebook Id, fetch the favorite artist and generate lyrics for all the tracks in the most popular album along with its price and Amazon id. All the files related to this mashup are given in Appendix B. From an architectural point of view, when SemCloud gets this mashup to fulfill, SAWADL annotations of all existing resources should be provided along with the task. As Discovery in SemCloud is not completed, each SAWADL file of resource has to be provided. The overall task of this mashup is detailed in a process descriptor file given Fig:8.1.

8.2 GENERATED PROCESS

This process SAWADL file has preconditions, effects, input data types (request params) and output data types (response params). SemCloud parses this file and creates the intial state with status flags (preconditions) and data types (request params), goal state with status flags (effects) and data types (response params). All the SAWADL files of the resources are also parsed to create action objects, respectively with input data types, output data types, preconditions and effects. As initial status flag (selectedFBFriend) matches the precondition of the action getDetailsofFriend [B.1]. and also the current state has the required data types, thus this action is added to the plan. This action is not repeated because the count of available data types in the current state and the required input data types of the action is equal to 1. If this is not the case, then the action should be repeated. The new

```
<resources base="">
  <resource path="shelftalkers">
      <method name="POST" id="consumer">
       <request>
       <param name='friendfbId' maxOccurs="1" type='xsd:string'
         sawadl:modelReference="Ontology1#Person"/>
        <param name='apiKey' type='xsd:string' style="query"
         sawadl:modelReference="Ontology1#BestBuyAPIKey"/>
        <param name='access_token' type='xsd:string' style="query"
         sawadl:modelReference="Ontology1#FacebookAPIKey"/>
        <param name='api_key' type='xsd:string' style="query"
         sawadl:modelReference="Ontology1#LastFMAPIKey"/>
        <param name='i' type='xsd:string' style="query"
         sawadl:modelReference="Ontology1#LyricsFlyAPIKey"/>
       </request>
       <sawadl:precondition expression="selectedFBFriend"/>
       <sawadl:effect expression="retrievedLyricsforSoundTrack
            ^ retrievedAlbumDetails ^ retrievedASIN"/>
       <response status="200">
        <param name='artist' maxOccurs="1" type='xsd:string'
         sawadl:modelReference="Ontology1#SoloMusicArtist"/>
        <param name='albumname' maxOccurs="1" type='xsd:string'
         sawadl:modelReference="Ontology1#AlbumName"/>
        <param name='price'  maxOccurs="1" type='xsd:string'
         sawadl:modelReference="Ontology1#AlbumPrice"/>
        <param name='asin'  maxOccurs="1" type='xsd:string'
         sawadl:modelReference="Ontology1#ASIN"/>
        <param name='trackname'  maxOccurs="unbounded"
         sawadl:modelReference="Ontology1#TrackName"/>
        <param name='lyrics'  maxOccurs="unbounded"
         sawadl:modelReference="Ontology1#Lyrics"/>
      </response></method></resource></resources>
```

Figure 8.1: Process.SAWADL

state has data types from initial state and the output data types of the action getDetailsofFriend. It has the status flags, (selectedFBFriend, retrievedArtistname), after applying the effects of the action getDetailsofFriend. These steps are repeated until the current state matches the goal state.

When the goal state is reached, SemCloud creates a plan by backtracking from the goal state to the initial state by determing which actions result in the semantic data types of that state. The plan comprises a set where each entry pertains to a state with either a single action of a list of actions.

The following plan is generated by SemCloud for this process.

1. Action Name : getFavArtist

Input Data Type model References : Person, FacebookAPIKey

Output Data Type model Reference : SoloMusicArtist

Is Action Repeated : false

2. Action Name : getTopAlbum

Input Data Type model References : Person, LastFMAPIKey

Output Data Type model Reference : AlbumName

Is Action Repeated : false

********** Parallel Actions Start**********

3. Action Name : getAmazonId

Input Data Type model References : MBID (Music Brainz ID)

Output Data Type model Reference : ASIN (Amazon Identification Number)

Is Action Repeated : false

4. Action Name : getAlbumDetails

Input Data Type model References : AlbumName, BestBuyAPIKey, SoloMusicArtist

Output Data Type model Reference : AlbumPrice

Is Action Repeated : false

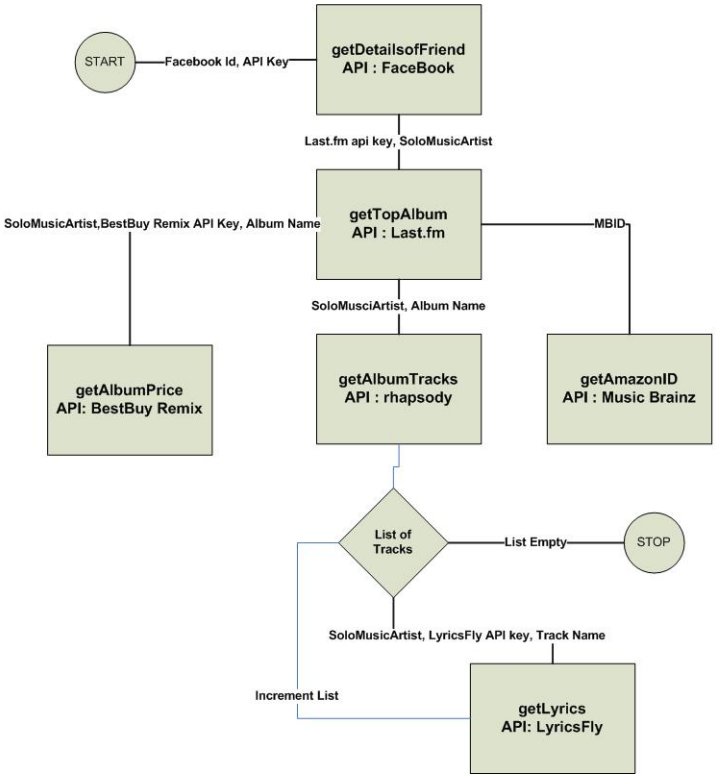

Figure 8.2: Generated Plan with Resources Added

5. Action Name : getAlbumTracks

Input Data Type model References : SoloMusicArtist, AlbumName

Output Data Type model Reference : Tracks

Is Action Repeated : false

********** Parallel Actions End**********

6. Action Name : getLyrics

Input Data Type model References : TrackName, SoloMusicArtist

Output Data Type model Reference : Lyrics

Is Action Repeated : true

8.3 Process Execution

As shown in the fig:8.1, when input friendfbid = krishnagollapudi and all API keys used in this mashup are provided, SemCloud Executor takes the plan and the data type mappings as inputs. Data type mappings provide the dependency of elements among the actions in the plan. As explained before, the plan comprises of a set where each entry pertains to a state with either a single action of a list of actions. So when the executor fetches one entry of the plan, it can be one action or a list of actions. The executor fetches the first entry, a single action getDetailsofFriend, and generates the respective URL as explained in section 7.1 using the input elements. It then invokes the action and stores the response in a temporary file. The executor fetches the input elements of the next entry, a single action getTopAlbum, from the plan. From the data type matcher, all the matching data types of each input is retrieved and these matching data types are searched for values in the temporary response file. A certain match is expected as the composer has listed these actions next to each other. The executor takes these data types along with values to generate a URL, fetch the response and store it in a temporary response file.

Now, the next entry of the plan has a set of actions, [getAmazonId, getAlbumDetails, getAlbumTracks], to be implemented using a parallel-split workflow pattern. The temporary response file from the previous action does not change until all the actions from this entry are executed. After execution of these actions, a list of response files are created instead of one. Until this point no action in the plan has repetition. The next entry in the plan, getLyrics, needs to be repeated over the list of tracks provided as output from the action, getAlbumTracks. This repetition results in another set of response files.

Observe that the required output of the process, Fig:8.1, is a list of parameters summed up by elements, both intermediate and final to the plan. So, the response file/files at each stage of execution is/are saved and is/are parsed to get the values of the output parameters. The output XML file generated for this process is Fig:8.3.

```
<output>
<artistname>usher</artistname>
<albumname>8701</albumname>
<price>19.99</price>
<asin>B00005LKGT</asin>
<trackname index="1">Intro-lude</trackname>
<trackname index="2">U Remind Me</trackname>
<trackname index="3">I Don't Know</trackname>
<trackname index="4">Twork It Out</trackname>
<trackname index="5">U Got It Bad</trackname>
<trackname index="6">If I Want To</trackname>
<trackname index="7">I Can't Let U Go</trackname>
<trackname index="8">U Don't Have to Call</trackname>
<trackname index="9">Without U</trackname>
<trackname index="10">Can U Help Me</trackname>
<lyrics index="1">Query not found!</lyrics>
<lyrics index="2">Yo, I ain't seeing you
 ... To be continued</lyrics>
<lyrics index="3">Check this out, yeah, yeah
... To be continued</lyrics>
<lyrics index="4">Yo Check this here
... To be continued</lyrics>
<lyrics index="5">You Got It Badzzz
... To be continued</lyrics>
<lyrics index="6">Everytime I look up
... To be continued</lyrics>
<lyrics index="7">Query not found!</lyrics>
<lyrics index="8">Query not found!</lyrics>
<lyrics index="9">Query not found!</lyrics>
<lyrics index="10">Query not found!</lyrics>
</output>
```

Figure 8.3: Generated Output XML

CHAPTER 9

CONCLUSION AND FUTURE WORK

9.1 CONCLUSION

This thesis presents SemCloud to annotate and compose cloud resources along with a data mediation paradigm. SemCloud presents SAWADL, an approach to annotate resources on the cloud. SemCloud takes the benefits from SAWSDL such as modelReference and SchemaMappings and also tries to completely describe a resource. A SAWADL file also has information like the time between requests for a resource, any additional parameters in the URL, etc. SemCloud also presents an automatic approach to compose resources on cloud. It addresses process heterogeneities and data heterogeneities by using a modified graph planner and data mediator. Speaking about the modification done to the AI graph planner, semantic data types are added to a state to make sure that data is available for a resource before it gets executed. This modification of AI graph planner is taken from Wu [29]. Currently, SemCloud supports Sequence, Parallel-split, Merge and Loop patterns depending on the SAWADL descriptions. The Parallel-split pattern of execution is explained in the section 7.1. When there are two or more actions to be executed in the same level of the plan, the input data types from previous actions are given to each action in the sequence. All the output data types resulting during parallel execution of the actions are provided to next action in the plan. Now, coming to the loop pattern, when an action has multiple instances of the same data type as output, but the next action in the plan accepts single instance of the same data type, then this action has to be repeated. During the plan generation, maxOccurs attribute of all elements are compared. When maxOccurs of the output data type is unbounded or greater than 1, while the input data type of next action has maxOccurs equal to 1, the later action is noted as a repeated action in the plan. Data mediation is done using a plan interpretor built on Apache Http Client. The Data exchange

language in this plan interpretor is XML. Though, SemCloud supports JSON responses. A JSON response in translated into XML for further processing. A context-based ranking algorithm [29] is used to select the best match for a given element.

9.2 Future Work

Future work to this thesis includes doing research in Discovery part of SemCloud. More work needs to be done to create an executable process for the plan generated by composer. If an executable process is generated in XProc w3.org/TR/xproc or in Python (www.python.org), then a user can take this process and execute it instead of an interpreter running the show. As explained in Appendix C, resources are having different styles of URL's. Work needs to done in standardizing the URL notations.

9.3 References

[1] Blum, A.; and Furst, M. (1997) *Fast Planning through Planning Graph Analysis*, Artificial Intelligence, 90:281-300.

[2] Winer, D. (2003) *Really Simple Syndication.*, http://cyber.law.harvard.edu/rss/rss.html.

[3] Henry, S.; Beech, D.; and Maloney, M. (2004) *XML Schema Definition*, http://www.w3.org/TR/xmlschema-1/.

[4] Sirin, E. (2004) *HTN Planning for Web Service Composition using SHOP2*, Web Semantics, 1(4):377-396.

[5] Sirin, E.; Parsia, B.; and Hendler, J. (2005) *Template base composition of semantic web services*, AAAI fall symp on agents and the semantic web, 85-92.

[6] William, V.; Ter, H.; Kiepuszewski, B.; and Barros, A. (2003) *Workflow Patterns*, Distributed and Parallel Databases, 14(3):5-51.

[7] Hadley, M. (2009) *Web Application Description Language*, Sun Microsystems.Inc, http://www.w3.org/submission/wadl/.

[8] Narayanan, S.; and McIlraith, S. (2002) *Simulation, verification and automated composition of Web service*, In proceedings of the 11th International World Wide Web Conference.

[9] McIlraith, S.; and Son, T. (2002) *Adapting Golog for composition of Semantic Web Services*, Knowledge Representation and Reasoning, 1(8):482-496

[10] McIlraith, S.; Son, T.; and Zeng, H. (2001) *Semantic Web Services*, IEEE Intelligent Systems, 16(2):46-53

[11] Rao, J. (2006) *A Mixed Initiative Approach to Semantic Web Service Discovery and Composition : SAP's Guided Procedures Framework*, The IEEE Intl Conf on Web Services.

[12] Fielding, R. (2000) *Representation State Transfer*, Univ. of California.

[13] Ponnekanti, S.; and Fox, A. (2002) *A Developer Toolkit for Web Service Composition*, The 11th World Wide Web Conference.

[14] Medjahed, B.; Bougettaya, A.; and Elmagarmid, A. (2003) *Composing Web Services on the Semantic Web*, VLDB Journal, 12(4):333-351.

[15] Kuter, U. (2005) *A Hierarchial Task-Network Planner based on Symbolic Model Checking*, ICAPS, 300-309.

[16] Mitra, N.; and Lafon, Y. (2007) *Simple Object Access Protocol*, http://www.w3.org/TR/soap/.

[17] Traverso, P.; and Pistore, M. (1990) *Automated Composition of Semantic Web Services into Executable Processes*, The 3rd International Semantic Web Conference, 380-394.

[18] Pistore, M. (2005) *Automated Synthesis of Composite BPEL4WS Web Services*, IEEE Intl Conf of Web Services.

[19] Deborah, L.; and Harmelen, F. (2004) *Web Ontology Language*, http://www.w3.org/TR/owl-features.

[20] Waldinger, R. (2001) *Web Agents Co-operating Deductively*, First International Workshop on Formal Approaches to Agent-Based Systems, 1871:250-262.

[21] Lammermann, S. (2002) *Runtime Service Composition via Logic-Based Program Synthesis*, Department of Microelectronics and Information Technology.

[22] Page, L. (2002) *Page Rank Algorithm*.

[23] Bray, T.; Paoli, J.; and Maler, E. (2008) *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, http://www.w3.org/XML/.

[24] Rao, J.; Kungas, P.; and Matskin, M. (2003) *Application of Linear Logic to Web Service Composition*, The First Intl Conf of Web Services, 3-9.

[25] Rao, J.; Kungas, P.; and Matskin, M. (2004) *Logic-based Web services composition from service description to process model*, Intl Conf on Web Services, 446-453.

[26] Crockford, D. (2006) *Java Script Object Notation*, http://json.org.

[27] Christensen, E.; Curbera, F.; Weerawarana, S.; and Meredith, G. (1990) *Web Service Description Language*, http://www.w3.org/TR/wsdl/.

[28] Battle, R.; and Benson, E. (2007) *Bridging the Semantic Web and Web 2.0 with Representational State Transfer (REST)*, Web Semantics: Science, Services and Agents on the World Wide Web. 6(1):61-69

[29] Wu, Z.; Ranabahu, R.; Gomadam, K.; Sheth, P. A.; and Miller, J. A. (2007) *Automatic Composition of Semantic Web Services using Process and Data Mediation*, Proceedings of the 9-th International Conference on Web Services, 453-461.

[30] Nagarajan, M.; Verma, K.; K.; Sheth, P. A.; and Miller, J. A. (2006) *Semantic Interoperability of Web Services - Challenges and Experiences*, IEEE International Conference on Web Services, 373-382

[31] Narendra, K. S.; and Parthasarathy, K. (1990) *Identification and Control of Dynamical System using Neural Networks*, IEENN, 1(1):4-27.

APPENDIX A

DIFFERENT TYPES OF APIS

SemCloud has proposed an architecture where resources can be reutilized. One of the key resources provided for composition are APIs. However not all APIs are tuned for automatic composition. APIs with REST interface are apt for this scenario. For example, in the practical scenario iLike is a JavaScript API. That means a user has to write specific JavaScript commands in a GUI for response. So, API types have to be learnt before giving them to the composer. This back log has made us to stop the mashup at fetching lyrics for tracks.

APPENDIX B

MASHUP FILES

B.1 FACEBOOK.SAWADL

The ontology file used for this mashup is given in Fig:B.13. This resource provides details for a given facebook Id. It has a REST Interface which has JSON response. SemCloud translates this response into XML for processing. The access token is the API key given by the user. The output XML of this resource is defined by an XSD schema, facebookprofile.xsd Fig:B.2. The SAWADL file of this resource in given in Fig:B.1.

B.2 BESTBUY.SAWADL

This resource provides details for a given album name of an artist. It has a REST Interface which has XML response. The apikey param is the API key given by the user. The output XML of this resource is defined by an XSD schema, bestbuy.xsd Fig:B.4. The SAWADL file of this resource in given in Fig:B.3.

B.3 LASTFM.SAWADL

This resource provides most popular album names of an artist. It has a REST Interface which has XML response. The api key param is the API key given by the user. The output XML of this resource is defined by an XSD schema, lastfm.xsd Fig:B.6. The SAWADL file of this resource in given in Fig:B.5.

```
<?xml version="1.0"?>
<application targetNamespace="facebook"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://wadl.dev.java.net/2009/02 wadl.xsd"
 xmlns:sawadl="http://www.w3.org/2002/ws/sawadl/spec/sawadl#"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:Ontology1="shelftalkers.owl"
 xmlns:fb="facebook"
 xmlns="http://wadl.dev.java.net/2009/02">
<grammars>
  <include
   href="facebookprofile.xsd"/>
</grammars>
 <resources base="graph.facebook.com" protocol="https"
    count="1" paramname1="fields" paramvalue1="id,music">
  <resource path="friendfbId" param="true">
    <method name="POST" id="getDetailsofFriend">
     <request>
      <param name='access_token'  type='xsd:string'
      style="query" sawadl:modelReference="Ontology1#FaceBookAPIKey"/>
      <param name='friendfbId' maxOccurs="1" type='xsd:string'
      style="query" sawadl:modelReference="Ontology1#Person"/>
       </request>
<sawadl:precondition expression="selectedFBFriend"/>
<sawadl:effect expression="retrievedArtistname"/>
         <response status="200">
           <representation mediaType="application/json"
             element="fb:o" maxOccurs="1"/>
 </response></method></resource></resources>
```

Figure B.1: FaceBook.SAWADL

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:Ontology1="shelftalkers.owl" targetNamespace="facebook"
xmlns="facebook"
xmlns:sawadl="http://www.w3.org/2002/ws/sawadl/spec/sawadl#"
elementFormDefault="qualified">
<xsd:element name="o" type="UserProfileType"/>
<xsd:complexType name="UserProfileType">
<xsd:sequence>
  <xsd:element name="id" type="xsd:int" nillable="false"/>
  <xsd:element name="music" type="music" nillable="false"/>
</xsd:sequence>
</xsd:complexType>
<xsd:complexType name="music">
<xsd:sequence>
<xsd:element name="data" type="data"/>
</xsd:sequence>
</xsd:complexType>
<xsd:complexType name="data">
 <xsd:sequence>
 <xsd:element name="e" type="e"/>
 </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="e">
 <xsd:sequence>
 <xsd:element name="category" type="xsd:string"/>
 <xsd:element name="id" type="xsd:int"/>
 <xsd:element name="name" type="xsd:string"
  sawadl:modelReference="Ontology1#SoloMusicArtist"/>
</xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

Figure B.2: facebookprofile.xsd

```
<?xml version="1.0"?>
<application targetNamespace="shelftalkers"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://wadl.dev.java.net/2009/02 wadl.xsd"
 xmlns:sawadl="http://www.w3.org/2002/ws/sawadl/spec/sawadl#"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:impl="bestbuy"
 xmlns:Ontology1="shelftalkers.owl"
 xmlns="http://wadl.dev.java.net/2009/02">
<grammars>
  <include
    href="bestbuy.xsd"/>
 </grammars>
<resources base="api.remix.bestbuy.com/v1"
  protocol="http" count="1" paramname1="show"
  paramvalue1="name,regularPrice">
 <resource path="products(artistName=[artist]&amp;
      type=Music&amp;name=[albumname])" param="include">
  <method name="POST" id="GetAlbumDetails">
 <sawadl:precondition expression="retrievedAlbums"/>
 <request>
 <param name='apiKey' type='xsd:string'
  style="query" sawadl:modelReference="Ontology1#BestBuyAPIKey"/>
 <param name='artist' type='xsd:string'
  style="query" sawadl:modelReference="Ontology1#SoloMusicArtist"/>
 <param name='albumname' type='xsd:string'
  style="query" sawadl:modelReference="Ontology1#AlbumName"/>
  </request>
  <sawadl:effect expression="retrievedAlbumDetails"/>
  <response status="200">
   <representation mediaType="application/xml"
    element="impl:products" maxOccurs="1"/>
 </response></method></resource></resources></application>
```

Figure B.3: BestBuy.SAWADL

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="bestbuy" xmlns:Ontology1="shelftalkers.owl"
xmlns:sawadl="http://www.w3.org/2002/ws/sawadl/spec/sawadl#"
targetNamespace="bestbuy" elementFormDefault="qualified">
<xsd:element name="products" minOccurs="1" maxOccurs="1"
type="product"/>
<xsd:complexType name="product">
<xsd:sequence>
 <xsd:element name="name" type="xsd:string"
  nillable="false" sawadl:modelReference="Ontology1#AlbumName"/>
 <xsd:element name="regularPrice"
   nillable="false" sawadl:modelReference="Ontology1#AlbumPrice"/>
</xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

Figure B.4: bestbuy.xsd

B.4 LYRICSFLY.SAWADL

This resource provides lyrics for a given track name of an artist. It has a REST Interface which has XML response. The i param is the API key given by the user. The output XML of this resource is defined by an XSD schema, lyricsfly.xsd Fig:B.8. The SAWADL file of this resource in given in Fig:B.7.

B.5 RHAPSODY.SAWADL

This resource provides track names in a given album name of an artist. It has a REST Interface which has XML response. This resource does not require authentication. The output XML of this resource is defined by an XSD schema, rhapsody.xsd Fig:B.10. The SAWADL file of this resource in given in Fig:B.9.

```
<?xml version="1.0"?>
<application targetNamespace="shelftalkers"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://wadl.dev.java.net/2009/02"
 xmlns:sawadl="http://www.w3.org/2002/ws/sawadl/spec/sawadl#"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:albums="albums"
 xmlns:Ontology1="shelftalkers.owl"
 xmlns="http://wadl.dev.java.net/2009/02">
<grammars>
   <include
     href="lastfm.xsd"/>
 </grammars>
<resources base="ws.audioscrobbler.com" protocol="http"
 count="1" paramname1="method"
 paramvalue1="artist.gettopalbums">
 <resource path="2.0">
  <method name="POST" id="getTopAlbumsforanArtist">
 <sawadl:precondition expression="retrievedArtistname"/>
 <request>
  <param name='api_key'  type='xsd:string'
  style="query" sawadl:modelReference="Ontology1#LastFMAPIKey"/>
  <param name='artist' type='xsd:string'
 style="query" sawadl:modelReference="Ontology1#SoloMusicArtist"/>
  </request>
 <sawadl:effect expression="retrievedAlbums ^ retrievedMBID"/>
 <response status="200">
 <representation mediaType="application/xml"
     element="albums:lfm" maxOccurs="1"/>
</response></method></resource></resources></application>
```

Figure B.5: LastFM.SAWADL

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="albums" xmlns:Ontology1="shelftalkers.owl"
xmlns:sawadl="http://www.w3.org/2002/ws/sawadl/spec/sawadl#"
targetNamespace="albums" elementFormDefault="qualified">
<xsd:element name="lfm" minOccurs="1"
   maxOccurs="1" type="lfmtype"/>
<xsd:complexType name="lfmtype">
 <xsd:sequence>
  <xsd:element name="topalbums" type="albumtype"/>
 </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="albumtype">
<xsd:sequence>
 <xsd:element name="album" type="album"/>
</xsd:sequence>
</xsd:complexType>
<xsd:complexType name="album">
<xsd:sequence>
 <xsd:element name="name" type="xsd:string"
  nillable="false" sawadl:modelReference="Ontology1#AlbumName"/>
 <xsd:element name="playcount" type="xsd:int"
 nillable="false" sawadl:modelReference="Ontology1#PlayCount"/>
 <xsd:element name="mbid" type="xsd:string"
 nillable="false" sawadl:modelReference="Ontology1#MBID"/>
</xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

Figure B.6: lastfm.xsd

```
<?xml version="1.0"?>
<application targetNamespace="lyricsfly"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://wadl.dev.java.net/2009/02"
 xmlns:sawadl="http://www.w3.org/2002/ws/sawadl/spec/sawadl#"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:poc="http://www.sws-challenge.org/schemas/rnet/POC"
 xmlns:impl="lyrics"
 xmlns:Ontology1="shelftalkers.owl"
 xmlns="http://wadl.dev.java.net/2009/02">
<grammars>
 <include
  href="lyricsfly.xsd"/>
 </grammars>
<resources base="api.lyricsfly.com/api/api.php"
  protocol="http" wait="5000">
  <resource path="">
  <method name="POST" id="getLyrics">
  <sawadl:precondition expression="retrievedTracksinAlbum"/>
  <request>
  <param name='i'  type='xsd:string'
  style="query" sawadl:modelReference="Ontology1#LyricsFlyAPIKey"/>
  <param name='a' type='xsd:string'
  style="query" sawadl:modelReference="Ontology1#SoloMusicArtist"/>
  <param name='t' type='xsd:string'
  style="query" sawadl:modelReference="Ontology1#TrackName"/>
  </request>
  <sawadl:effect expression="retrievedLyricsforSoundTrack"/>
 <response status="200">
 <representation mediaType="application/xml"
  element="impl:start" maxOccurs="1"/>
 </response></method></resource></resources></application>
```

Figure B.7: LyricsFly.SAWADL

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="lyrics" xmlns:Ontology1="shelftalkers.owl"
xmlns:sawadl="http://www.w3.org/2002/ws/sawadl/spec/sawadl#"
targetNamespace="lyrics" elementFormDefault="qualified">
<xsd:element name="start"  minOccurs="1"
  maxOccurs="1" type="starttype"/>
<xsd:complexType name="starttype">
<xsd:sequence >
<xsd:element name="status" type="xsd:int">
<xsd:element name="tt" type="xsd:string"
sawadl:modelReference="Ontology1#TrackName"/>
<xsd:element name="tx" type="xsd:string"
sawadl:modelReference="Ontology1#Lyrics"/>
</xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

Figure B.8: lyricsfly.xsd

B.6 MUSICBRAINZ.SAWADL

This resource provides amazon Id for a given music brainz Id. It has a REST Interface which has XML response. This resource does not require any authentication. The output XML of this resource is defined by an XSD schema, musicbrainz.xsd Fig:B.12. The SAWADL file of this resource in given in Fig:B.11.

```
<?xml version="1.0"?>
<application targetNamespace="rhapsody"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://wadl.dev.java.net/2009/02"
 xmlns:sawadl="http://www.w3.org/2002/ws/sawadl/spec/sawadl#"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:poc="http://www.sws-challenge.org/schemas/rnet/POC"
 xmlns:impl="tracks"
 xmlns:Ontology1="shelftalkers.owl"
 xmlns="http://wadl.dev.java.net/2009/02">
<grammars>
  <include
  href="rhapsody.xsd"/>
 </grammars>
<resources base="rhapsody.com" protocol="http">
  <resource path="artist/albumname/data.xml" param="rest">
   <method name="POST" id="getAlbumTrackslist">
 <sawadl:precondition expression="retrievedAlbums"/>
  <request>
  <param name='artist' type='xsd:string'
 style="query" sawadl:modelReference="Ontology1#SoloMusicArtist"/>
 <param name='albumname' type='xsd:string'
  style="query" sawadl:modelReference="Ontology1#AlbumName"/>
  </request>
 <sawadl:effect expression="retrievedTracksinAlbum"/>
 <response status="200">
  <representation mediaType="application/xml"
    element="impl:album"  maxOccurs="unbounded"/>
</response></method></resource></resources></application>
```

Figure B.9: Rhapsody.SAWADL

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="tracks" xmlns:Ontology1="shelftalkers.owl"
xmlns:sawadl="http://www.w3.org/2002/ws/sawadl/spec/sawadl#"
targetNamespace="tracks" elementFormDefault="qualified">
<xsd:element name="album" minOccurs="1"
  maxOccurs="unbounded" type="albumtype"/>
<xsd:complexType name="albumtype">
<xsd:sequence>
 <xsd:element name="tracks" type="tracks"/>
</xsd:sequence>
</xsd:complexType>
<xsd:complexType name="tracks">
<xsd:sequence>
 <xsd:element name="track" type="track"/>
</xsd:sequence>
</xsd:complexType>
<xsd:complexType name="track">
<xsd:sequence>
<xsd:element name="name@" type="xsd:string"
sawadl:modelReference="Ontology1#TrackName"/>
<xsd:element name="play-href" type="xsd:string"/>
<xsd:element name="data-href" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

Figure B.10: rhapsody.xsd

```
<?xml version="1.0"?>
<application targetNamespace="shelftalkers"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://wadl.dev.java.net/2009/02"
 xmlns:sawadl="http://www.w3.org/2002/ws/sawadl/spec/sawadl#"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:mb="http://musicbrainz.org/ns/mmd-1.0#"
 xmlns:Ontology1="shelftalkers.owl"
 xmlns="http://wadl.dev.java.net/2009/02">
<grammars>
 <include
  href="musicbrainz.xsd"/>
 </grammars>
<resources base="musicbrainz.org/ws/1/release"
 protocol="http" count="1" paramname1="type" paramvalue1="xml">
<resource path="mbid" param="true">
 <method name="POST" id="getAmazonId">
<sawadl:precondition expression="retrievedMBID"/>
<request>
 <param name='mbid' type='xsd:string'
  style="query" sawadl:modelReference="Ontology1#MBID"/>
</request>
<sawadl:effect expression="retrievedASIN"/>
<response status="200">
<representation mediaType="application/xml"
element="mb:metadata" maxOccurs="1" ns ="true"/>
</response></method></resource></resources></application>
```

Figure B.11: MusicBrainz.SAWADL

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:Ontology1="shelftalkers.owl"
xmlns="http://musicbrainz.org/ns/mmd-1.0#"
xmlns:sawadl="http://www.w3.org/2002/ws/sawadl/spec/sawadl#"
targetNamespace="http://musicbrainz.org/ns/mmd-1.0#">
<xsd:element name="metadata" minOccurs="1"
 maxOccurs="1" type="releasetype"/>
<xsd:complexType name="releasetype">
<xsd:sequence>
 <xsd:element name="release" type="asintype"/>
</xsd:sequence>
</xsd:complexType>
 <xsd:complexType name="asintype">
  <xsd:sequence>
  <xsd:element name="title$" type="xsd:string"
  nillable="false" sawadl:modelReference="Ontology1#AlbumName"/>
  <xsd:element name="asin$" type="xsd:String"
  nillable="false" sawadl:modelReference="Ontology1#ASIN"/>
</xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

Figure B.12: musicbrainz.xsd

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF>
<rdf:RDF xmlns="http://www.owl-ontologies.com/shelftalkers.owl#"
 xml:base="http://www.owl-ontologies.com/shelftalkers.owl"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
 xmlns:owl="http://www.w3.org/2002/07/owl#"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 <owl:Ontology rdf:about="http://www.owl.com/shelftalkers.owl"/>
 <owl:DatatypeProperty rdf:about="&owl;topDataProperty"/>
    <owl:Class rdf:about="&shelftalkers;ASIN"/>
    <owl:Class rdf:about="&shelftalkers;Album"/>
    <owl:Class rdf:about="&shelftalkers;AlbumName"/>
    <owl:Class rdf:about="&shelftalkers;AlbumPrice"/>
    <owl:NamedIndividual rdf:about="&shelftalkers;BestBuyAPIKey">
        <rdf:type rdf:resource="&shelftalkers;Key"/>
      <owl:topDataProperty>5rsgru446</owl:topDataProperty>
    </owl:NamedIndividual>
    <owl:NamedIndividual rdf:about="&shelftalkers;FaceBookAPIKey">
        <rdf:type rdf:resource="&shelftalkers;Key"/>
      <owl:topDataProperty>2227|2.oiIQL8w</owl:topDataProperty>
    </owl:NamedIndividual>
    <owl:Class rdf:about="&shelftalkers;FacebookProfile"/>
    <owl:Class rdf:about="&shelftalkers;Friend"/>
    <owl:Class rdf:about="&shelftalkers;Key"/>
    <owl:NamedIndividual rdf:about="&shelftalkers;LastFMAPIKey">
        <rdf:type rdf:resource="&shelftalkers;Key"/>
      <owl:topDataProperty>7846457829</owl:topDataProperty>
    </owl:NamedIndividual>
    <owl:Class rdf:about="&shelftalkers;Lyric"/>
    <owl:Class rdf:about="&shelftalkers;Lyrics"/>
    <owl:NamedIndividual rdf:about="&shelftalkers;LyricsFlyAPIKey">
        <rdf:type rdf:resource="&shelftalkers;Key"/>
      <owl:topDataProperty>e98ce75e917e02458</owl:topDataProperty>
    </owl:NamedIndividual>
    <owl:Class rdf:about="&shelftalkers;MBID"/>
    <owl:Class rdf:about="&shelftalkers;Person"/>
    <owl:Class rdf:about="&shelftalkers;PlayCount"/>
    <owl:Class rdf:about="&shelftalkers;ProductDetails"/>
    <owl:Class rdf:about="&shelftalkers;SoloMusicArtist"/>
    <owl:Class rdf:about="&shelftalkers;SoundTrack"/>
    <owl:Class rdf:about="&shelftalkers;SoundTrackSample"/>
    <owl:Class rdf:about="&shelftalkers;Stream"/>
    <owl:Class rdf:about="&shelftalkers;TrackName"/>
    <owl:Class rdf:about="&shelftalkers;UserId"/>
    <owl:Class rdf:about="&shelftalkers;UserName">
        <rdfs:subClassOf rdf:resource="&shelftalkers;UserId"/>
    </owl:Class>
</rdf:RDF>
```

Figure B.13: Shelftalkers.owl

APPENDIX C

RESOURCE URL

Each resource has a unique style of writing it's URI. For example, few resources like MusicBrainz treats each MBID as a new resource replicating the style of REST interfaces. On the other hand LyricsFly needs artist name as a parameter in the URI. Resources like Facebook specify the contents seen in the response along with the type of response in the URL.

MusicBrainz URL :

```
http://musicbrainz.org/ws/1/release/MBID
```

BestBuy URL :

```
http://api.remix.bestbuy.com/v1/products(artistName=[name]&type=[type])
        ?&apiKey=[apikey]&show=name,regularPrice
```

Facebook URl :

```
https://graph.facebook.com/facebookid?&access_token=[Token]
```

Necessary steps have to be taken for different styles of resource URls as a simple HTTP-Client request does not suffice. Consider the first case with MusicBrainz. Here is the part of its descriptor.

```
<resources base="musicbrainz.org/ws/1/release" protocol="http"
   count="1" paramname1="type" paramvalue1="xml">
<resource path="mbid" param="true">
```

So, our parser goes through this descriptor and senses that this resource has a resource path which is a param in this SAWADL. It also has an additional parameter, type=xml, to go with the URL. The Count attribute specifies the number of additional parameters.