APPROVAL SHEET

Title of Thesis: Automatic Composition and Invocation of Semantic Web Services

Name of Candidate: Mithun Sheshagiri
Master of Science, 2004

Thesis and Abstract Approved: _____

Dr. Marie desJardins
Assistant Professor
Department of Computer Science and
Electrical Engineering

Date Approved: _____

Curriculum Vitae

Name: Mithun Sheshagiri.

Permanent Address: 4813 Fernley Square, Baltimore, MD 21227.

Degree and date to be conferred: Master of Science, 2004.

Date of Birth: February 6, 1979.

Place of Birth: Miraj, India.

Secondary Education: South Indian Education Institute, Mumbai, 1996.

Collegiate institutions attended:

University of Maryland, Baltimore County, M.S. Computer Science, 2004.
Ramrao Adik Institute of Technology, India, B.E. Computer Science, 2000.

Major: Computer Science.

Professional publications:

Mithun Sheshagiri and Marie desJardins
Data Persistence: A Design Principle for Hybrid Robot Control Architectures
KBCS 2002

Mithun Sheshagiri, Marie desJardins and Tim Finin
A Planner for Composing Service Described in DAML-S
Workshop on Planning for Web Services, International Conference on Automated Planning and Scheduling, Trento, July 2003

Subhash Kumar, Anugeetha Kunjithapatham, Mithun Sheshagiri, Tim Finin, Anupam Joshi, Yun Peng, and R. Scott Cost
A Personal Agent Application for the Semantic Web
AAAI Fall Symposium on Personalized Agents, North Falmouth, Nov 15-17, 2002

Mithun Sheshagiri, Norman Sadeh and Fabien Gandon
Using Semantic Web Services for Context-Aware Mobile Applications
Proceedings of MobiSys2004 Workshop on Context Awareness, Boston,

June 2004

Professional positions held:

Senior Software Engineer, Samsung Information Systems America (April '04 -)

Research Associate, Carnegie Mellon University (Feb '04 - April '04)

Research Intern, Hewlett Packard Laboratories (June '03 - Dec '03)

Research Assistant, CSEE Department, UMBC. (Jan. '02 - Jun. '03).

Software Engineer, Patni Computer Systems Ltd., Mumbai. (June. '00 - Aug. '01).

ABSTRACT

Title of Thesis:

Automatic Composition and Invocation of Semantic Web Services

Author: Mithun Sheshagiri, Master of Science, 2004

Thesis directed by: Dr. Marie desJardins, Assistant Professor
Department of Computer Science and
Electrical Engineering

Web Services are viewed as a new paradigm for building distributed web applications. The W3C defines web services as a programmatic interface that enables communication among applications [1]. OWL-S (earlier known as DAML-S) is an initiative that aims to describe web services using semantic web languages like RDF, RDFS and OWL. The motive behind adding semantics is to enable automatic discovery, composition, invocation and execution monitoring. In this thesis, we look at the problem of web service composition – synthesizing complex tasks from a set of atomic (basic) tasks. We look at how a service description can be transformed into planning operators. We present a planner that uses a backward-chaining algorithm for the task of composition. We also discuss an algorithm for automatic invocation. The automatic invocation algorithm takes the plan generated by the planner and determines data required for invocation using queries. Finally, we discuss the architecture of our agent, which is capable of interacting with web services solely based on service descriptions.

Automatic Composition and Invocation of Semantic Web Services

by

Mithun Sheshagiri

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
2004

Dedicated to my Mother

ACKNOWLEDGMENTS

I would like to take this oppurtunity to thank my *Gurus*: Dr. Marie desJardins and Dr. Tim Finin. This thesis wouldn't have been possible without their support. Dr. desJardins, my advisor, has been extremely helpful in shaping my ideas and provided me a great deal of flexibility in carrying out my work. She has shown great patience and has put considerable effort to make herself accessible. I am also indebted to her for taking efforts to improve the quality of this thesis. Dr. Finin has been been very encouraging and I feel lucky to have worked under him. I am also grateful to Dr. Anupam Joshi and Dr. Alan Messer for agreeing to be on my thesis committee. I should also thank Dr. Stuart Williams, Dr. Janet Bruten and HP-Laboratories for providing an opportunity to work as an intern at HP. My ideas on modelling and effects were conceived under Dr. Stuart Williams' wing. I am grateful to my room-mates for keeping me sane during my graduation. Special thanks to Priyang Rathod for doing all my paper-work when I was away from Baltimore. I would like to thank the Graduate Program Director - Dr. Krishna Sivalingam, the administrative staff at the CSEE department and Linda Thomas from the Graduate School for their co-operation. Finally, my parents, especialy my mother deserves special mention: her belief in me is very valuable to me.

TABLE OF CONTENTS

. i

. ii

ACKNOWLEDGMENTS . . . . . **iii**

LIST OF FIGURES . . . . . **v**

Chapter 1 INTRODUCTION **1**

1.1 Background and Motivation 3

Chapter 2 RELATED WORK **6**

Chapter 3 INTERNAL ARCHITECTURE OF THE AGENT . . **10**

3.1 System Architecture for Disconnected Composition and Invocation . . 10

3.2 Using Forward Chaining for Simultaneous Composition and Invocation 11

Chapter 4 OWL-S: SERVICE DESCRIPTIONS USING OWL . . **14**

Chapter 5 PLANNING FOR COMPOSITION **18**

5.1 Planner 19

5.2 Extraction of Operators from Service Descriptions 20

5.2.1 Additional Constraints 20
5.3 Planning Algorithm for Composition 21

Chapter 6 PRECONDITIONS AND EFFECTS 23

6.1 Disconnected Composition and Invocation 23
6.2 Modelling Effects 26
6.3 Adding Semantics to Existing Services 31
6.3.1 Mapping between RDF and XML 34
6.3.2 Realizing Preconditions and Effects 37

Chapter 7 AUTOMATIC INVOCATION 43

7.1 Routines 45
7.1.1 AutoInvoke 45
7.1.2 ExQuery and PartQuery 46
7.1.3 QueryAndMatch 46
7.1.4 Diagnose 48

Chapter 8 CONCLUSIONS 54

REFERENCES 56

LIST OF FIGURES

3.1 **Arcitecture for Disconnected Composition and Invocation** . . 11

3.2 **Architecture for Simultaneous Composition and Invocation** . 12

4.1 **Currency Conversion Service** 17

6.1 **Modelling Effects by Reification** 33

6.2 **Current Web Service Architecture** 33

6.3 **Semantic Web Service Architecture** 34

6.4 **Transformation between RDF and XML** 42

6.5 **Realization of Preconditions** 42

7.1 **The ListOfFlights Class** 50

Chapter 1

INTRODUCTION

Traditionally, the coupling between applications that interact over a network has been very tight. Changes to any application could sometimes lead to the re-engineering of the other applications. Introduction of new applications to the system required an in-depth analysis of the existing system. The tight coupling, along with the fact that applications in the system were developed independently, made the task of designing and maintaining these applications very difficult. An alternative black-box approach was introduced aimed at reducing the tight coupling. In this approach, the internal working of the system is hidden and functionalities of the application is exposed as service descriptions. This approach to building applications ,along with a set of supporting technologies, is referred to as web service technology.

Web Service Description Language (WSDL) is by far the most widely language used for web service descriptions. Web service technology, in its current state, provides a platform and language independent way of building applications. In other words, given a WSDL description for a set of services, the user has the liberty to interact with the service provider using a language of his/her choice. The actual remote invocations can be synthesized as a local API, simplifying the integration of functionalities provided by the service provider. However, there is still a human involved in the loop. The signature of methods (method name, input and output)

convey insufficient information so the task of ascertaining the correct usage of the service is often left to the human. To enable an agent to interact with the service provider, the agent needs to know the correct sequence of the operations that achieve the user's goal (automatic composition) and it needs to know the content of the messages that need to be sent out to the service provider (automatic invocation). WSDL descriptions by themselves do not provide sufficient information to enable an agent to perform the above activities. We use Semantic Web Languages (SWL) to add semantics to WSDL descriptions and capture information that will enable an agent to automatically interact with the service.

The goal of this research has been to enable an agent to interact automatically with a system based solely on service descriptions. This is achieved by using our techniques for automatic composition and invocation. Web service composition can be defined as the task of putting together atomic (basic) services to achieve complex tasks. We present a planner that uses a backward-chaining algorithm for the task of service composition. Web services by nature are non-deterministic; we take this into account and suggest techniques for representing the web service to enable composition.

The service composition orders atomic processes in the right sequence; the interaction is complete only when the processes are invoked/executed. Invocation involves gathering the correct data to be provided as input to each service. We present an algorithm for automatic invocation using two types of queries – ExactQuery and PartQuery. At any stage, if invocation fails, our algorithm tries to diagnose the likely cause of failure, which may be either the faulty execution of a service or incorrect/incomplete information provided by the user.

We also address the issue of adding semantics to existing services. If Semantic Web Services (SWS) will be adopted by the industry, there will be a large number of WSDL/SOAP services in use. Methods of porting these into SWS would be of great

help. WSDL operations are defined using inputs and outputs (IOs) whereas OWL-S represents atomic processes (the equivalent of WSDL operations) in terms of inputs, outputs, preconditions and effects (IOPEs). We look at how IOs can be mapped to IOPEs.

1.1 Background and Motivation

Machine-readable descriptions of existing web service applications are stored in the form of WSDL files. The WSDL file essentially provides information about the operations (methods) that can be invoked, defines the schema for the message content, describes the message exchange pattern and the location of the service. This is analogous to a collection of methods and their signatures along with information about the syntax of the input and the output. The decision to invoke is still done manually, based on textual descriptions that explain the implications of invoking the service. The goal of SWS is to make the existing account of services richer by articulating the implications in SWL rather than text. The SWS lets a machine know when a service should/could be executed and what the effects of executing the service are. This description can, in principle, be used by the agent to perform automatic discovery, automatic composition and automatic invocation.

The potential to delegate tasks to a client that is capable of interacting with various services on the web has several implications. A typical (Business-to-Consumer) B2C scenario involves using online websites for such as shopping and traveling. The consumer's interest lies in choosing the *best* deal from available deals. *Best* could be quantified by the customer as a function of factors such as price, reputation of manufacturer and seller, quality of service of the seller and convenience. Traditionally, there have been two approaches: (1) visit each website individually and interact with it, or (2) use a service that interacts with a fixed number of websites and aggregate

the data for the user. Such services (websites) help the user compare similar services offered by multiple B2C sites. The former approach is a tedious and time-consuming task and involves a high cognitive load for the user. The latter, although it is easier for the user, typically relies on hard-coded logic embedded in wrappers that interact with the B2C sites. As a result, integrating a new B2C site is a substantial engineering task and such systems need to be updated whenever changes are made to the websites that are used as information sources.

Even after choosing the website, the user of a B2C site is bound by the functionalities provided by the site. For example, if a user wants to buy a product, they must also use the shipping service provided by the site selling the product, and provide the mode of payment accepted by the site. An alternative model would be to provide users with more flexibility, allowing them to choose from a variety of services offered by various service providers. Users could choose an online shop for selecting a product, make the payment using a service that corresponds to their preferred mode of payment, and choose their favorite shipping service.

A web-service-based system with rich service descriptions coupled with a user agent offers solutions to some of these problems. The client provides flexibility by relieving the customer of the tedious task of navigating through the site; at the same time, it has the potential to interact with an arbitrary number of web-based businesses, as long as it has the service descriptions for those web sites. Since the client relies solely on the service description to interact with the service, the service description needs to be updated every time the service provider decides to change some part of the service.

We use a B2C based shopping scenario to illustrate our techniques for automatic composition and invocation. We chose the online shopping domain since it is well understood and large online shopping stores like Amazon [9] already provide a web

service-based infrastructure.

As part of this work, we have implemented a planner that composes services. We also discuss our techniques for modelling effects and preconditions. We also present the design of a forward-chaining based service composition and invocation engine. Chapter 3 discusses the design of two agents: a backward-chaining service-composition agent and a forward-chaining service-invocation agent. Chapter 4 discusses the use of OWL-S for service descriptions. In Chapter 5, we discuss our the planner we use for composition. Chapter 6 discusses preconditions, modeling of effects and mechanism to add semantics to existing web services. Chapter 7 explains our algorithm for automatic invocation.

Chapter 2

RELATED WORK

With the increasing popularity of Web Services, several efforts have emerged in the last year that address the problem of service composition. Before we look at these approaches, let us revisit the definition of web services. We earlier introduced the W3C's definition of web service as a programmatic interface that enables communication among applications. A major part of this initiative involves developing standard interfaces that describe the applications and uniform way of accessing them. Only when service providers commit to such standards, others can use it. Developing representations for web services without regard to standards makes it difficult, if not impossible, for others to use it. This goes against the primary objective of web services. We make use of OWL-S, the most widely known Semantic Web-based framework for describing web services. OWL-S has been designed to add semantics on top of WSDL, the *de facto* standard for representing web services.

In our system, we automatically build operators from the web service descriptions in OWL-S and compose complex tasks from scratch. Most of the service composition techniques do not address these two points simultaneously.

SWORD [16] is a model for web service composition. However, it uses its own simple description language and does not support any existing standards like WSDL or OWL-S. Services are modeled using inputs and outputs, which are specified using

an Entity Relationship model. Inputs are classified into conditional inputs and data inputs. Outputs are classified similarly. Conditional inputs/outputs are assertions about entities on which the service operates and the relationships between entities. Data inputs/outputs constitute the actual data (attributes of entities) that the service uses.

A similar framework was developed at IBM Research Laboratories as part of the Web Services Toolkit (WSTK) [2]. A composition engine [18] has been built for services described in WSDL. Although this work describes the use of a planner for composition, constructing operators from the service description is not fully automated. This is primarily because of the absence of a mechanism to capture domain knowledge in WSDL. Our planner makes use of services described in OWL-S. OWL helps us to describe explicit ontologies for capturing domain language. This added knowledge gives our planner greater versatility and helps compose complex services.

The Golog-based system [13] does service composition by using general templates that are modified based on user preferences, yielding the final plan. The templates are not automatically built and constitute part of the plan. Our composer is able to build plans dynamically from scratch and does not rely on templates for composition.

Semantic E-Workflow Composition [6] talks about composition in workflow systems. Workflow is an abstract representation of a process. A workflow is built using components called tasks/activities. Traditionally, appropriate tasks are selected from a workflow repository. This work introduce the notion of using web services as tasks in the workflow. They address the issue of selecting appropriate web services using semantic discovery. They also discuss how web services can be integrated into workflows by syntactic and semantic integration of inputs and outputs. The primary contribution of this work is to provide a tool to assist in manual workflow composition. This work does not deal with OWL-S based semantic web services.

SHOP2 is a Hierarchical Task Network (HTN)-based planner for composing web services [21]. This along with our earlier work on composition [17] were among the first to deal with composition using DAML-S/OWL-S. The SHOP2-based composer requires that each atomic service either produces outputs or effects but not both. This assumption has been made to differentiate between information-gathering services and effect-producing services. By making this differentiation, SHOP2 executes the information-gathering services during plan generation and simulates the execution of services that produce effects. To use SHOP2 for composition, service providers have to describe services that are consistent with the above assumption. In principle, services can be re-designed so that each service either produces only outputs or only effects but not both. Our framework does not the above restriction and re-design is not required in our case.

Paolucci, Sycara and Kawamura in their work [14] make a reference to RETSINA, a planner that makes use of the HTN planning paradigm. This planner is similar to SHOP2 in the way they interleave planning and execution. They claim that by executing the information-gathering services during planning, unexpected situations can be handled by replanning. They make use of OWL-S in their work but do not mention how the RETSINA planner has been modified to use OWL-S descriptions.

SHOP2 and RETSINA-based composers offer simultaneous information-gathering and planning. We think that the advantages of simultaneous composition and invocation in the web services domain is limited. Most services (except the services that constitute the head of the plan) are dependent on the execution of other services and cannot be executed in an arbitrary order. This is because these services need inputs that are available after the execution of other services, specifically, the ones which are closer to the plan head. As a result, simultaneous execution and planning can

be done in a forward-chaining planner or a hybrid planner which starts building the plan from the head and the tail. Secondly, only information-gathering services can be invoked during plan generation. Once a service that produces one or more effects is encountered while building the plan, all services that make use of the information provided (as output/effect) by this service cannot be executed. Given a goal, it is not known in advance if a plan can be built successfully. This results in service invocations that correspond to failed attempts to build the plan. Execution of a service is usually a time-intensive process that involves sending a request and receiving a response over the network. By first building the complete plan, we avoid these additional executions. By committing to a forward-chaining approach, RETSINA and SHOP2 lose their potential benefits which they gained by their ability to do sensing (information gathering) during planning.

Our system first builds the plan and then executes it. If a service does not produce the intended result during execution, the execution engine can request the planner to build a plan using a different service that produces the same required result. We do not lose the ability to replan by committing to a disconnected planning and execution approach. We present the design of a separate invocation engine that executes the plan generated by the planner. In case the execution of the plan fails, the invocation also determines the reason for failure – faulty execution of the service or incomplete information provided by the user.

Chapter 3

INTERNAL ARCHITECTURE OF THE AGENT

3.1 System Architecture for Disconnected Composition and Invocation

Figure 3.1 shows the overall architecture of a Semantic Web Service system along with the internals of the agent that interacts with such a system. The service provider provides descriptions (OWL-S+WSDL). The OWL-S part of the description describes services in terms of IOPEs and are used in building the planning operators for the Planning module. WSDL defines the structure of messages using a schema, the message interaction, the SOAP encoding and the location of the service. Planning operators are synthesized from service descriptions. The user specifies the goal and other information that she/he wants to use to interact with the service (userKB). The composer builds a plan to achieve the user's goal and passes it on to the invocation engine. The invocation engine fetches operators from the plan starting from the head and moving towards the tail. It gathers information by querying the KB (userKB+outputKB). The client stub is generated using Apache Axis [19] from the WSDL file. The client stub contains methods that correspond to service invocations. The invocation engine invokes these methods to do the actual invocation.

3.2 Using Forward Chaining for Simultaneous Composition and Invocation

The alternative approach would be to use forward-chaining and move towards the goal. Forward chaining has the advantage of maintaining the complete description of the state of the system at all intermediate states [4]. This feature enables simultaneous planning and invocation. However, it suffers from the lack of direction in reaching the goal. As a result, the planner has to deal with a very large search space.

Let us assume that the agent has service descriptions to perform various tasks. For instance, services used to buy a book are clustered together. We refer to each individual group as a "service-group." A forward-chaining algorithm starts from the initial state and moves towards the goal state. In a B2C site, the initial state is the user's knowledge known to the user. If the agent were to search for services

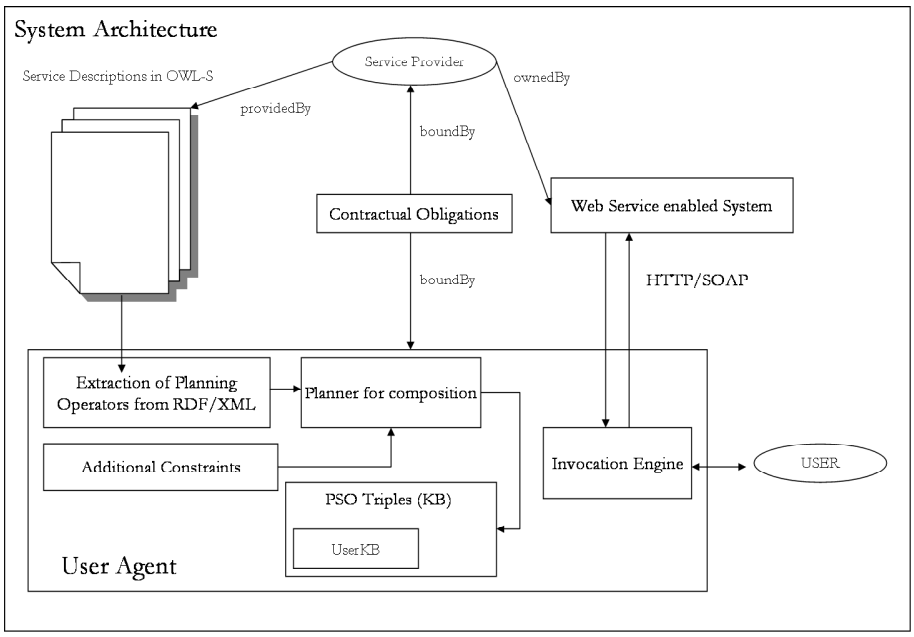

FIG. 3.1. **Arcitecture for Disconnected Composition and Invocation**

that take some combination of user information (initial state) as the input, its search space will include a large number of services from each service-group. The problem is compounded by the fact that the agent executes all of these services in its forward-chaining algorithm. Compartmentalizing can be used to address this problem. Each service-group is stored in distinct KBs. At any given time the planner has access to service from a single service-group. When the user specifies a goal, the most appropriate service-group is chosen and the planner uses a service in this service-group to build the plan. Service-group independent services – services that can be used across groups (for example, a third pary credit card authorization service) – should be treated differently and considered while building all plans, across service-groups.

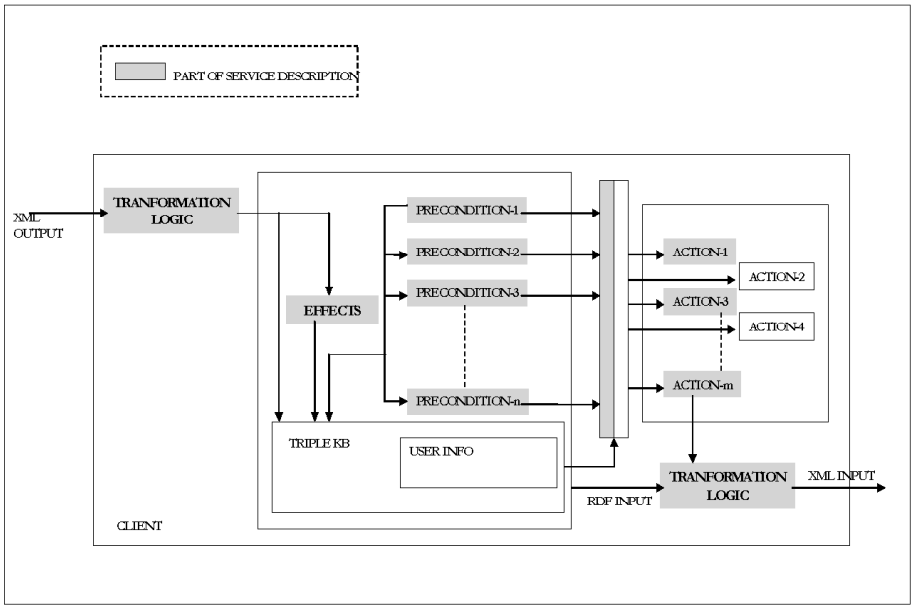

FIG. 3.2. **Architecture for Simultaneous Composition and Invocation**

Figure 3.2 shows the architecture of an agent that does simultaneous composition and invocation. The forward-chaining algorithm is implicit in its working. The agent invokes a service and receives the output. The output is transformed into effects or

preconditions or both. Effects manipulate data structures in the client. Preconditions act as guards for actions. The realization of new preconditions permits the client to take one or more actions. These actions could be invocation of services or operations internal to the client. It can be the case that the realization of a single precondition can make more than one service, belonging to different service-groups, executable. This is where compartmentalization of service-groups can be used by exposing the agent to the service-group that achieves the user's objective. When the realization of a precondition allows the execution of multiple services in the same service-group and if all these services produce different outputs, they are executed one after the other. The precondition might enable the execution of more than one service that achieves the same output. In such cases, the agent, on behalf of the user, chooses one service and executes it. The agent might make use of user preferences to pick a service.

Chapter 4

OWL-S: SERVICE DESCRIPTIONS USING OWL

OWL-S is an ontology in OWL for describing services. The long-term aim of OWL-S is to help service providers describe services to enable automatic discovery, composition and execution monitoring. OWL-S consists of the following ontologies:

The topmost level consists of a Service ontology. The Service is described in terms of a ServiceProfile, ServiceModel and a ServiceGrounding ontology, which are as follows:

1. The service presents a ServiceProfile which has a subclass Profile. The Profile provides a vocabulary to characterize properties of the service provider, functional properties of the service like Inputs, Outputs, Effects and Preconditions (IOPEs) and non-functional properties of the service. The Profile is used for discovering the service. The service provider provides this description to the directory service.

2. The service is describedBy a ServiceModel which has a subclass called Process. The Process consists of all the functional properties of the service; the Profile on the other hand need not include all the functional properties. The service

could be a collection of atomic services, composite services or a combination of both. The Process lets the service provider describe services in terms of IOPEs and is used for composition.

3. The service supports a ServiceGrounding, which has a subclass called Grounding. The Grounding provides an interface to plug in WSDL descriptions. It also provides eXtensible Stylesheet Language Transformations (XSLT) stylesheets for transforming between XML and RDF/XML. Grounding indicates how each atomic service can be invoked using a WSDL *operation* (this is equivalent to an atomic service in WSDL) .

According to OWL-S 1.0 specs, the service descriptions are instances of the Profile and Process ontologies.

Consider the example of a currency conversion service as it would be described in the process model.

```
<process:AtomicProcess rdf:ID="CurrencyConverter">
        <process:hasInput rdf:resource="#SourceCurrency"/>
        <process:hasOutput rdf:resource="#TargetCurrency"/>
</process:AtomicProcess>

<process:Input rdf:ID="SourceCurrency">
        <process:parameterType rdf:resource="#Currency"/>
</process:Input>

<process:UnconditionalOutput rdf:ID="TargetCurrency">
        <process:coOutput rdf:resource="#Currency"/>
</process:Output>

<owl:Class rdf:Id="Currency">
</owl:Class>
```

```
<owl:DatatypeProperty rdf:ID="currencyType">
<rdfs:domain rdf:resource="#Currency"/>
        <rdfs:range rdf:resource="&xsd;#string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="amount">
<rdfs:domain rdf:resource="#Currency"/>
        <rdfs:range rdf:resource="&xsd;#float"/>
</owl:DatatypeProperty>
```

The above fragment of RDF/XML shows the currency conversion service with a single input and output. SourceCurrency and TargetCurrency are instances of the class Currency. Currency is defined as the class with two properties – currencyType and amount. The diagramatic representation of this service is shown in Figure 4.1. The service takes in a instance of the class SourceCurrency and produces an instance of class TargetCurrency of equivalent value. Instances of SourceCurrency and TargetCurrency contain both the type of the currency and the amount.

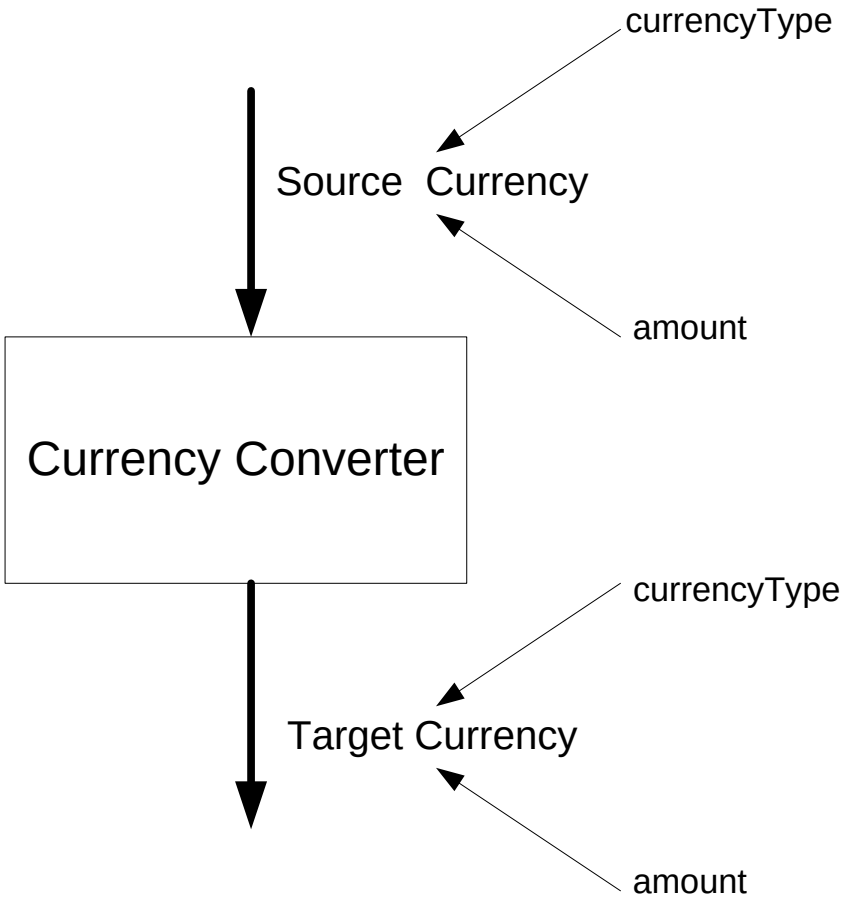

FIG. 4.1. **Currency Conversion Service**

Chapter 5

PLANNING FOR COMPOSITION

Web service composition involves ordering a set of atomic services in the correct order to satisfy a given goal. This problem can be viewed as a planning problem P described by the tuple $<G,A>$ where G is the goal and A is the set of actions. When mapping the planning problem to service composition, actions are replaced by services. Our planner uses a backward-chaining algorithm to build the plan. The output of this algorithm is a plan that achieves the given goal and an initial state, which indicates the knowledge that needs to be provided by the user to execute the plan.

Although the atomic process describes the effects of executing the service, we do not use it for composing the plans but relying solely on preconditions and outputs. Our definition of effects is different from the definition of effects as specified by OWL-S. According to the OWL-S definition, if a service that produces an effect is invoked, the physical state of the world changes. We claim that effects (in the OWL-S sense) and preconditions are both implicit in the output produced by the service invocation.

A service can be invoked if the precondition associated with it holds. Preconditions associated with services are sufficient to determine the sequencing constraints between services. These constraints are used to build the plan. We use effects to address the problem of automatic invocation of the service. In our system, effects are used to manipulate data structures within the agent.

5.1 Planner

The Planner is designed in Java. Jess [9] is used as the knowledge base (KB) and contains facts stored as predicate-subject-object (PSO) triples. Jess is a Java-based expert system supporting framing rules and queries. We also use Jess to extract planning operators.

A service operator is characterized as *Service(S,P)* where:

1. S is the service name
2. P is the list of inputs, ouputs and preconditions associated with the service

In our implementation, we represent service operators as a Java class with IOPEs as fields of this class. The logical equivalent of these operators can be represented as:

```
(service servicename
    (input i)
    (output o)
    (precondition p))
    (effect e))
```

We use this representation for all subsequent chapters.

5.2 Extraction of Operators from Service Descriptions

The service descriptions are in OWL-S and stored in RDF/XML. The extraction of planning operators involves three steps:

1. Converting RDF/XML into Predicate-Subject-Object (PSO) triples using Jena [10] libraries.

2. Extraction of planning operators from service description.

3. Associating additional parameters to operators.

The conversion of the service description to PSO triples is a simple process that makes use of the Jena libraries. After conversion, all the triples are asserted as facts into a Jess KB. Extraction of planning operators is done using a combination of rules and queries (*defrule* and *defqueries* in Jess). For each service, the entire KB is queried for its corresponding IOPEs and a service/planning operator is built using this information.

5.2.1 Additional Constraints

The third step is optional but can be very useful in certain cases. In the examples we present in our discussions, we assume that there are two entities (service provider and end user) involved during the interaction, but there could be scenarios that involve more than two entities. The service provider providing services to an end user could itself be using the services provided by a second service provider. For example, a travel

service providing a vacation package could be using a weather service or a currency exchange service from some other service provider. The travel service might choose to allow end users to use the weather or currency service. The weather and currency service providers might be charging for using their services; therefore, the travel service might impose additional constraints to permit only registered users to use these services. In the above scenarios the same weather service looks different depending upon who is using it. The travel service provider adds additional constraints to the weather or currency service and exposes these modified service to the end user.

The service provided by the weather service provider might look like this:

```
(service WeatherLookUp
    (input ZipCode)
    (ouput Weather))
```

The travel service displays this service to the end user as follows:

```
(service TravelWeatherLookUp
    (precondition RegisteredUser)
    (input ZipCode)
    (ouput Weather))
```

5.3 Planning Algorithm for Composition

The algorithm used by our planner is a simple backward-chaining algorithm. The user initializes the planner by specifying a goal. The planner looks for service(s) whose output(s) satisfy this goal and include these in the plan. The inputs and preconditions of the newly included service(s) are treated as unsatisfied goals. Now the planner tries

to satisfy the unsatisfied goals. This corresponds to a new iteration which involves executing the same steps described above. Planning terminates when the planner fails to find any operators that satisfy any of the unsatisfied goals. If all unsatisfied goals correspond to user inputs then the plan *could* be successful. This uncertainty is a result of the fact that services are non-deterministic. A service could fail due to infrastructure problems like lost messages or network failure. A service could also fail at a higher level, i.e., the process of service might have executed successfully without producing the intended result. For example, there might be a service in the plan that adds an item to the shopping cart. If the particular item is out of stock then the service did not produce the intended result. The composer provides a plan to execute the services in the right order and the invocation engine invokes the service in that order. The plan is only successful if all the services in the plan execute as intended: this can be ascertained only during execution.

Chapter 6

PRECONDITIONS AND EFFECTS

Modelling of effects and preconditions is an open issue in the OWL-S community. OWL-S 1.0 provides the definition of preconditions and effects. This is currently a subject of discussions and a concrete defintion has not been specified. Guidelines for modelling preconditions and effects have not been elaborated. We introduce our notion of preconditions and effects and discuss how these can be used in web service composition. Our approach for incorporating preconditions and effects is modular: our technique allows existing services to be described in OWL-S.

6.1 Disconnected Composition and Invocation

Our earlier work [17] relied on connecting effects with preconditions for building the plan. More recently, we have adopted an approach in which services are linked using outputs and preconditions. Effects are used for another purpose explained later in this section. We illustrate this approach with two atomic services from the process model of an imaginary shipping service:

```
(service CreditCardAuthorization
    (input CardNum)
    (input CardExpDate)
    (input CardType)
    (output CardStatus))

(service Ship
    (input DestA)
    (input DestB)
    (precondition CreditVerified)
    (effect PackageShipped)
    (output ShippingInvoice))
```

These services can be composed only at run time. The Ship service can be executed only when the precondition CreditVerified evaluates to true. The information needed to evaluate this precondition is available only after the CreditCardAuthorization service is executed. In the above instance, the output CardStatus of CreditCardAuthorization is used for precondition evaluation. The sequencing constraint between the two services is not evident from the service description before execution. Automatic compostion and invocation in our system is a two-step process: composition followed by invocation. We use the *realizedBy* construct to ascertain the list of outputs required to achieve a precondition. Whether they actually produce the desired effect of realizing the precondition can be determined only during service execution.

Use of the *realizedBy* construct enables us to build the plan without executing a single service, therefore achieving disconnected planning and execution. By

disconnecting composition and invocation one can ascertain from the plan and the unsatisfied goals whether the plan can achieve the user's objective. Once we know if the plan can achieve the user's objective, we start the invocation process.

Using the *realizedBy* construct, the shipping service description would be:

Shipping service using *realizedBy* construct:

```
(service CreditCardAuthorization
    (input CardNum)
    (input CardExpDate)
    (input CardType)
    (output CardStatus))

(service Ship
    (input DestA)
    (input DestB)
    (precondition pCreditVerified)
    (effect PackageShipped)
    (output ShippingInvoice))

(realizedBy pCreditVerified CardStatus)
```

From the use of *realizedBy* construct it is evident that execution of the Ship service needs to be preceded by the CreditCardAuthorization service. The *realizedBy* property is associated with a precondition. These could be provided by the service provider or could be generated by the agent as well. For example, if the agent is aware of a service that produces CardStatusResult as output and if this output is equivalent to the CardStatus services, the agent can do the following inferencing.

```
(realizedBy pCreditVerified CardStatus)
(sameAs CardStatus CardStatusResult)
=>
(realizedBy pCreditVerified CardStatusResult)
```

If more than one service realizes the same precondition, the planner could select one service using the following heuristics:

1. Pick a service randomly
2. Select the operator (service) with the least number of preconditions or inputs,
3. Select operators that have pre-conditions that are known to be easily satisfiable.

These are some of the issues we have identified but our current planner does not handle such cases.

6.2 Modelling Effects

We use effects as side effects of executing a service at the agent's end, resulting in changes to the agent's KB. This definition is different from effects as defined by OWL-S specifications. OWL-S defines effects as events that alter the state of the world. In our system, change in world state is conveyed using outputs. The motive of having an effect (in OWL-S) is to differentiate between information-gathering services (also known as sensing) and services that alter the physical state of the world. In other words, services that produce only outputs are idempotent whereas services with at least one effect are non-idempotent. This differentiation is not required. Most existing web services make use of WSDL descriptions and WSDL does not differentiate between idempotent and non-idempotent services. Typically, servers maintain some state information about the client which ensures that the client doesn't unintentionally execute the same operation twice. Differentiation between idempotent and non-idempotent services only helps planners which make use of simultaneous

planning and execution.

Just like preconditions, which are realized from outputs, effects are implicit in the output. For instance, the precondition pCreditCardVerified is realized from the CardStatus output. Similarly, the effect PackageShipped can ascertained from the information provided by the ShippingInvoice output. Having a special construct to convey this information does not serve any purpose other than what was conveyed by the output.

We use effects in our description to enable increased automation in our agent. We use effects to manage data in the KB of the agent, using reification. Take the case of a routine in Java that is used to add objects into a queue.

```
public String AddToQ(Queue Q, Object A)
{
   Q.add(ob);
   Return ob+"- added";
}


public String AddToQ1(Queue Q, Object A)
{
   Q.add(ob);
   Return Q;
}


public String DeleteFromQ(Queue Q, Object A)
{
   Q.add(ob);
   Return Q;
}
```

In this routine the inputs are object A and object Q. The output is a message indicating that an object has been added. The effect in this case would be that there is a new object A in the queue. Note that the equivalent service description of this routine would look like the signature of the routine *public String AddToQ(Queue Q, Object A)*. This signature does not convey information about the effect that Q now has A in it. One could argue that if the service returns the object Q (in method AddToQ1) as the output, then by inspecting the content of Q, one can determine this effect. The new routine will succeed in conveying the effect of the addition of an element. There are other cases where this technique of returning the object will fail. Consider a case where the agent decides to add an element to the queue (using service AddToQ1) and then deletes it using DeleteFromQ. On addition, the AddToQ1 will provide the object Q with A in it. To delete an object from the queue, the agent has to reset its state of the object Q and then generate the new state of Q, from the output. The output does not convey the required reset and generate operations to the agent. We propose a technique to achieve this by using two primitive operations – addition and deletion of facts. Knowledge in semantic web based systems is stored as facts which are represented as triples. By addition and deletion of facts we can convey effects to the agent. The addition of an object into the queue is brought about by adding statements into the KB and the deletion of the object is brought about by the deletion of facts from the KB that represent the object being removed.

We further explain our idea of effects by using an example that consists of the

following services: Login, LookUp, AddToCart and RemoveFromCart.

The LookUp service takes a keyword as input and provides a list of items that matched the keyword as the outptut. On receiving the list of items, the agent determines a match using arbitrary attributes of the item it intends to add to the Cart. Once the right item is determined, it is sent as input to the AddToCart service which brings about the *effect* of adding the item to the Cart. Similarly, the effect of removing an item from the Cart can be modelled by removing statements. The Login, LookUp, AddToCart and RemoveToCart services illustrated below have been used in the rest of the chapter. Inputs and outputs (in bold) correspond to actual items that can be described in an equivalent WSDL-based system; the remaining parameters (preconditions and effects) are part of the OWL-S description of the service. The values of these parameters are synthesized from the outputs.

```
(service Login
    (input username)
    (input password)
    (output SessionInfo)
    (effect InitializeClient))

(service LookUp
    (input item)
    (output itemList))

(service AddToCart
    (input Item)
    (precondition CartExists)
    (output MsgItemAdded)
    (effect ItemInCart))
```

```
(service RemoveFromCart
    (input Item)
    (precondition CartExists)
    (precondition ItemInCart)
    (output MsgItemRemoved)
    (effect ItemRemoved))
```

We integrate effects by extending the OWL-S ontology. We introduce a new class called Event which is pointed to by owls:ceEffect property of the class owls:ConditionalEffect. ceEffect and ConditionalEffect are parts of the existing OWL-S ontology. An Event can add statements or remove statements or both.

```
<!--Creates a cart at the client-->
 <ace:Event rdf:ID="CreateCartEffect">
   <ace:addStatements rdf:parseType="Collection">
      <bk:Cart rdf:ID="Your_Cart"/>
   </ace:addStatements>
  </ace:Event>
```

The above Event states that CreateCartEffect is an effect and the agent can bring about this effect by adding facts to its KB enclosed within the addStatements property. The fact within the addStatements is (PropertyValue (predicate type) (subject Your_Cart) (object Cart)) i.e., the instance Your_Cart is of type Cart. By adding these statements the agent has created a Cart data-structure in its KB.

```
 <!--Adds an item to the cart-->
 <ace:Event rdf:ID="AddToCartEffect">
    <ace:addStatements rdf:parseType="Collection">
      <ace:Item rdf:about="Item1">
       <ace:member rdf:resource="Your_Cart"/>
```

```
      </ace:Item>
    </ace:addStatements>
  </ace:Event>
```

The above event add an instance of Class Item to the cart instance using the member property.

```
<!--Removes the item from the cart-->
<ace:Event rdf:ID="RemoveFromCartEffect">
   <ace:removeStatements rdf:parseType="Collection">
     <ace:Item rdf:about="Item1">
      <ace:member rdf:resource="Your_Cart"/>
     </ace:Item>
   </ace:removeStatements>
  </ace:Event>
```

The above event removes an instance of Class item by removing the triple (PropertyValue (predicate member) (subject Item1) (object Your_Cart)).

CreateCartEffect is one of the many effects that correspond to InitializeClient effect of the service Login. CreateCartEffect adds statements (triples) to the KB. AddToCartEffect, by adding statements, has associated Item1 with the cart Your_Cart. Similarly, RemoveFromCartEffect has removed Item1 from the Cart.

6.3 Adding Semantics to Existing Services

As evident from the earlier example, adding semantics enabled a WSDL service to elaborate effects and preconditions without changing the existing WSDL-based framework. To enable the industry to adopt this technology, we must ensure that the

new system fits in well into their existing framwework (based only on WSDL). Figure 6.2 shows a simplified view of the current web service architecture. In a B2C scenario, contractual obligations are not brought about by negotiations; the service provider defines the contract and the consumer adheres to it. However, in a B2B scenario, terms and conditions of use of the service are fixed after a phase of negotiations. This paper primarily addresses the B2C scenario and ignores contractual obligations.

The WSDL file reveals the functionalities provided by the service provider using the operation construct of WSDL. The wsdl:operation is the equivalent of an atomic service. It also provides information about the structure of the messages to be exchanged and the location of the service. Figure 6.3 shows how semantics can be addded to an existing architecture. The new architecture not only incorporates semantics but also ensures backward compatibility; i.e., the client described in Figure 6.2 can be plugged into the architecture in Figure 6.3 without any changes to the services described by the WSDL file or the logic residing in the server. The client in Figure 6.3 has several additional components that enables it to utilize the additional information described in the semantic service descriptions. The client has a semantic web infrastructure (SWI) that enables it to read semantic descriptions and make inferences from them. All information coming into the client is stored in the form of PSO triples. The Discovery module uses the SWI to perform discovery and match-making. Discovery and match-making are broad topics themselves [12, **?**] and is beyond the scope of this paper. The composition module does the task of chaining together basic operations to

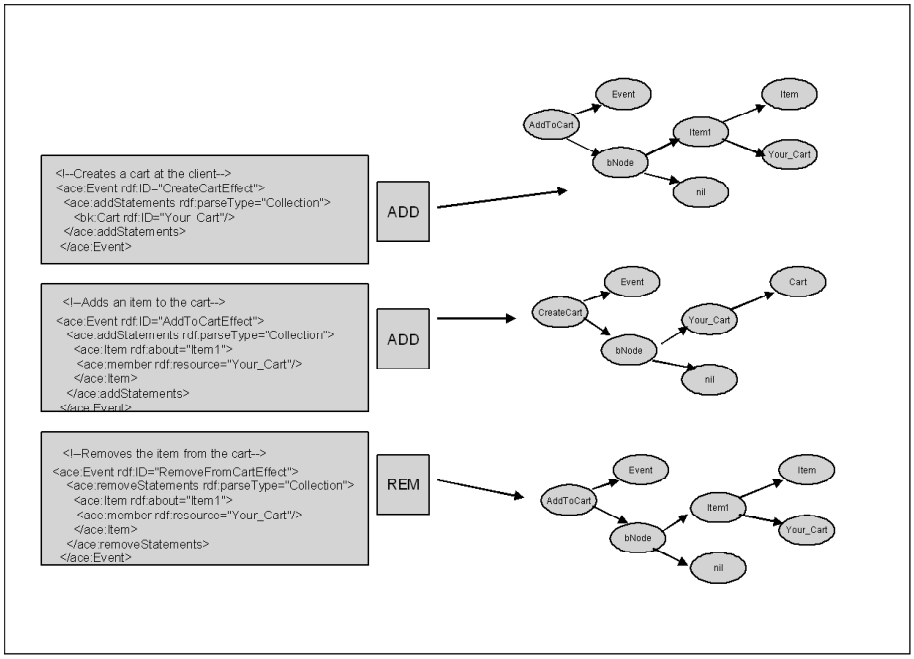

FIG. 6.1. **Modelling Effects by Reification**

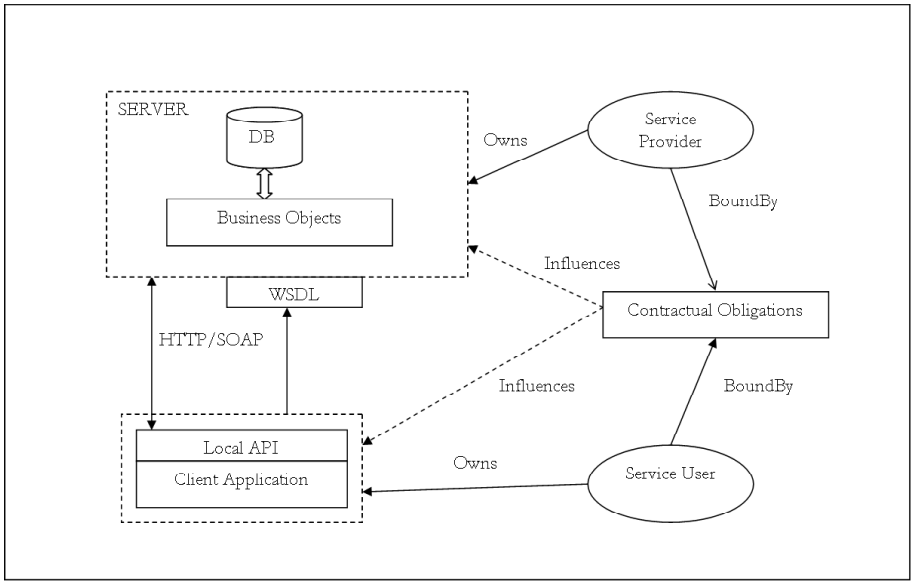

FIG. 6.2. **Current Web Service Architecture**

perform complex operations. The invocation engine then invokes services according to the plan. Each atomic service is grounded in an wsdl:operation that already exists as part of the old architecture.

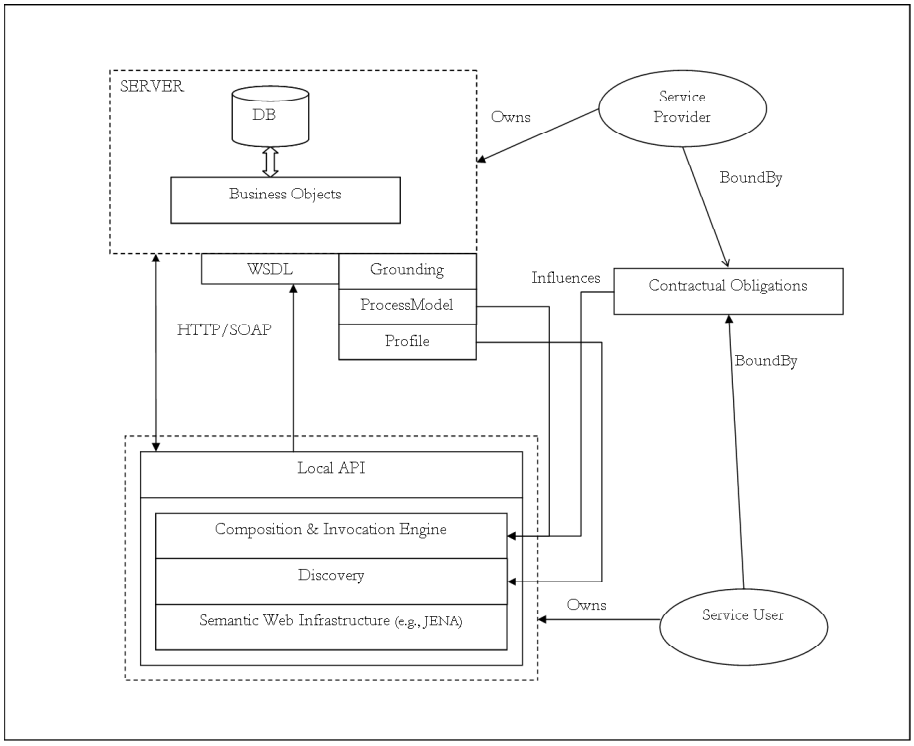

FIG. 6.3. **Semantic Web Service Architecture**

6.3.1 Mapping between RDF and XML

We present the design of our transformation logic. This logis is not part of our implemented system. We represent RDF as RDF/XML files in our implementation. Our discussion about mapping between RDF and XML deals with RDF/XML which is one of the ways of representing RDF.

All information stored in the agent is in the form of PSO triples which are obtained

from RDF represented in RDF/XML. Inputs and ouputs generated by WSDL services are in XML. We need a mechanism for transforming the XML messages into RDF/XML. Message1 (shown below) is an example of an output message from a WSDL-based search or look-up service. Outputs of WSDL based systems are in XML.

```
<!-- Message1-->
<itemList>
 <item partNum="872-AA">
   <productName>Lawnmower</productName>
   <USPrice>148.95</USPrice>
    </item>
    <item partNum="926-AA">
   <productName>Plasma Monitor</productName>
   <USPrice>39.98</USPrice>
  </item>
</itemList>
```

The above XML message provides information about a particular item of sale at the online shop. Each item has a part number, a product name and a price.

We transform this Message1 into the following RDF/XML message (Message2). The RDF/XML file is then converted into facts and asserted into the KB. The namespace information (attributes within the rdf:RDF tag) is introduced by the transformation logic. Transformation logic is provided by the service provider. It contains information to transform the XML message into an RDF/XML message.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
```

```
        xmlns:owl="http://www.w3.org/2002/07/owl"
        xmlns:bk="http://www.bookstore.com/bookStoreEffects/"
        xmlns="http://www.bookstore.com/workspace/">
  <bk:ItemList rdf:ID="itemlist">
  <bk:member>
      <bk:Item rdf:ID="gen872-AA">
        <productName>Lawnmower</productName>
        <bk:USPrice>148.95</bk:USPrice>
      </bk:Item>
  </bk:member>
  <bk:member>
      <bk:Item rdf:ID="gen926-AA">
        <bk:productName>Plasma Monitor</bk:productName>
        <bk:USPrice>39.98</bk:USPrice>
      </bk:Item>
   </bk:member>
  </bk:ItemList>
</rdf:RDF>
```

The following issues need to addressed when transforming from XML to RDF/XML.

1. Correct naming convention for instances of classes. For example, in the above case the value of partNum has been used to name instances and the name of these instances should be an XML Name. It is important to make sure that for a particular client the URIs of instances are unique and references to these should be correct.

2. In RDF/XML, the serialized information has a stripping pattern in which resources and properties alternate. However, XML doesn't require one to adhere to this pattern since there is no notion of resource or property in XML.

3. While transforming RDF/XML to XML, the transformation logic should make sure that the XML produced confirms to the XML schema in the WSDL doc-

ument. XML schema in the WSDL document specifies the structure of the messages (input and output).

It is also important to note that these transformations are not entirely syntactic. For example, in the AddToCart service, the output could just be a string which indicates whether the Item was added or not. Although such a message would make perfect sense to a user, the semantics of the message needs to be made explicit to the agent. It can also be the case that the string received as response from the AddToCart service does not have any reference to the Cart or the Item. In such cases, the transformation logic can be more complicated where it will have to maintain the input information of the AddToCart service which actually contains the references to the Cart and the Item. Figure 6.4 show the flow of messges in and out of the agent. Our design makes use of Extensible Stylesheet Language (XSL) for this transformation.

6.3.2 Realizing Preconditions and Effects

Preconditions and effects differ in the way they are represented and handled by the agent. An effect is brought about by either adding or removing statements from the KB; preconditions are conditions that evaluate to TRUE/FALSE. The evaluation of preconditions and the addition or removal of statements is during execution and not during planning. A precondition is evaluated to TRUE/FALSE by checking the information generated by applying the transformation logic to output messages. A service with a precondition(s) can be executed when the precondition(s) evaluates to

TRUE. Before executing each service, the preconditions associated with it must be checked. Effects, on the other hand, do not play a role in deciding if a service can be invoked, they merely manage the information at the client. There is no notion of preconditions and effects in WSDL. We need a way to incorporate preconditions and effects: without introducing new messages or changing the content of messages provided by the existing system.

The message corresponding to session information after successful login to a book store might look like this:

```
<?xml version="1.0"?>
<SessionInfo>
        <ShortName>Derick</ShortName>
        <SessionId>12456ARP</SessionId>
        <Recommendations>
                <book>Alchemist</book>
                <DVD>DOS for Dummies</DVD>
        </Recommendations>
        <WishList>
                <Misc>XXX</Misc>
                <Misc>YYY</Misc>
        </WishList>
        <Cart name="DerickCart"/>
        <Message>Login Successful</Message>
</SessionInfo>
```

The above example of an output message – SessionInfo is returned when a user by the name Derick logs into the book store using the Login Service. Based on his previous interactions, the book store might make some recommendations and provide a list of books the user intends to buy (WishList).

When the login operation fails, the message might look like this:

```
<?xml version="1.0"?>
<SessionInfo>
<Message>Faulty username or Password!</Message>
</SessionIno>
```

This message is used to synthesize the InitializeClient effect and the CartExists precondition. Among other things, initializing the client could involve creating and populating the user's WishList. We do this using the following effect:

```
<!--Creates and populates the wish list-->
 <ace:Event rdf:ID="InitializeClientEffect">
   <ace:addStatements rdf:parseType="Collection">
      <az:Cart rdf:ID="DerickCart"
      <az:WishList rdf:ID="DerickWishList"/>
      <az:Item rdf:ID="XXX">
      <az:member rdf:about=" DerickWishList ">
      </az:Item>
      <az:Item rdf:ID="YYY">
      <az:member rdf:about=" DerickWishList ">
      </az:Item>
   </ace:addStatements>
  </ace:Event>
```

When the login is successful, we generate the above piece of RDF from the contents of the SessionInfo message. When the facts in this message are asserted, the agent has created an instance of the Cart called DerickCart, created an instance of WishList called DerickWishList and added items XXX and YYY to the wish list. On login failure, the transformation logic does not produce the above message and no facts are asserted into the agent's KB.

We use a similar technique to generate preconditions. The session message also contains information about the cart and its contents. We generate the CartExistsPrecondition.

```
<process:Condition rdf:ID="CartExistsPrecondition">
    <ace:boundTo rdf:about="DerickCart">
    <ace:groundTo>
      <ace:TruthValue rdf:about="TRUE"/>
    </ace:groundTo>
</process:Condition>
```

The description states threat CartExistsPrecondition holds (is TRUE) for the instance DerickCart of the class Cart.

This description makes use of the Condition class of the extended OWL-S ontology. We have included two new properties: boundTo and groundTo. The boundTo property points to an instance and the precondition is only valid with respect to the specified instance or class. The groundTo property points to a class called TruthValue. The precondition is evaluated by checking the instance of TruthValue which can only be TRUE/FALSE. If the precondition evaluates to FALSE during invocation, this implies that the service did not produce the desired result. In this case, the invocation engine could request an alternative service that could realize the same precondition. In the worst-case scenario, the planner has to build a new plan excluding the service whose output failed to realize the precondition. The service description includes the InitializeClientEffect and CartExistsPrecondition but the client is responsible for binding these effects and preconditions to instances. For example, in

case of the CartExistsPrecondition, it is bound to Derick_Cart and is grounded to TRUE. Figure 6.5 illustrates realization of preconditions.

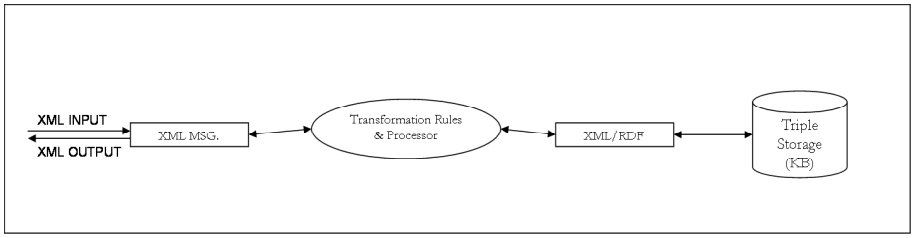

FIG. 6.4. **Transformation between RDF and XML**

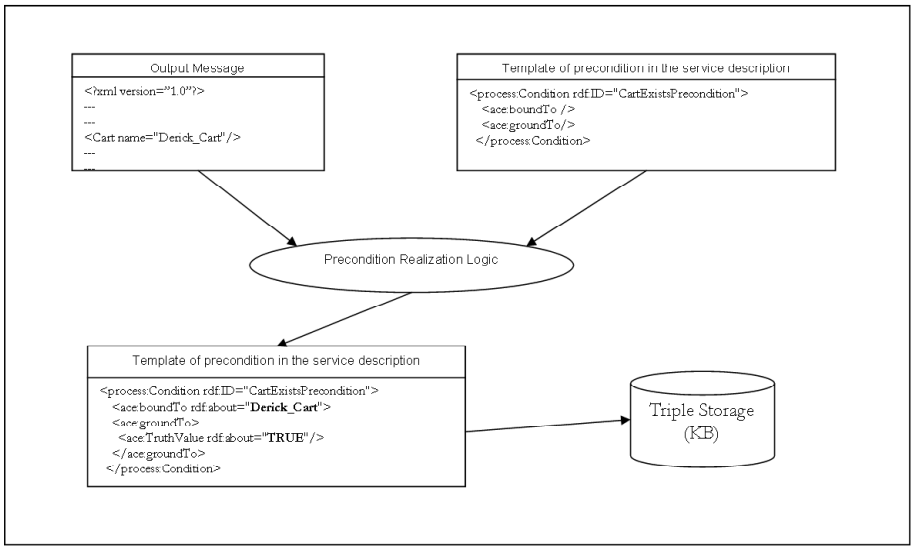

FIG. 6.5. **Realization of Preconditions**

Chapter 7

AUTOMATIC INVOCATION

This section explains the invocation algorithm for executing the plan. The plan provides the ordering constraints between services required to satisfy the user's goal. The agent now needs to determine what information needs to be sent as input, use the output to determine if the service was executed correctly and finally combine output information with the agent's existing knowledge to execute the next service. This set of tasks is repeated till all the services in the plan are executed. Note that although the plan is built using backward chaining, invocation (plan execution) is in the forward direction. Our algorithm for automatic invocation is shown below. We explain all the routines used in the algorithm is the next section.

```
Terminology:
input                          : class definition of the input
instanceTemplate(class)        : instances with unbounded rdf:ID and
                                 rdf:resource attributes
output                         : class definition of output\tains knowledge
                                 specified by the user
outputKB                       : KB that contains knowledge generated from
                                 outputs received by service executions
```

```
AutoInvoke(plan)
IF IsEmpty(plan)
      Return "Nothing to Execute!"
Get the topmost service(s) from the plan and store it in Services[]
FOR each service in Services[]
IF all precondtion of the service realize to 'TRUE'
      FOR each input
            QMResult=QueryAndMatch(input, userKB)
            IF (QMResult==NIL)
                  QMResult=QueryAndMatch(input, outputKB)
                  IF (QMResult==NIL)
                        pSol=Diagnose(input, outputs[])
                        IF (pSol==NIL)
                        Return "Execution Failed" -> input
For each service in Services[]
      output=execute(Service)
      Store output in outputKB
Return "Execution Successful"

QueryAndMatch(input, KB)
Results[]=ExQuery(instanceTemplateOf(input), KB)
IF (KB==userKB)
      IF (Results.size()==0)
            Return NIL
      IF (Results.size()==1)
            Return Results[0]
      IF (Results.size()>1)
            Return (PromptUser(input, "Make a choice", Results)
IF (KB==outputKB)
      IF (Results.size()==1)
            Return Results[0]
      IF (Results.size()>1)
            Return (PromptUser(input, "Make a choice", Results)
      IF (Results.size()==NIL)
            hints[]=PartQuery(instanceTemplateOf(input), userKB)
            IF (hints.size()==0)
                  Return NIL
            IF (hints[].size>1)
                  hint=PromptUser(input, "Choose a hint", hints[])
            IF (hints[].size==1)
                  hint=hints[0]
            MResult=ExQuery( hint, Results)
            IF MResult.length==1
                  Return MResult[0]
            IF (MResult==NIL)
                  Return NIL
            ELSE
                  Return PromptUser(input, "Make a choice", MResults)

ExQuery(qPattern, KB)
Return (Triples that match the query pattern)

PartQuery(input, KB)
qPatterns[0]=instanceTemplateOf(input);
level=0
WHILE (Result==NIL)
      FOR i=0 TO qPatterns.length
            Results=ExQuery(qPatterns[i],KB)
            IF Results!=NIL
                  Return Envelope(Results)
      qPatterns[]=GetInnerLevelInstance(input,++level)
END WHILE
```

```
Diagnose(input, outputs[])
Causes[]=GetServices(input, outputs[])
IF Causes[]==NIL
      pSol=PromptUser(input, "User input required")
ELSEIF Causes.length()==1
      pSol=PromptUser(input, "Execution Error", Causes[0])
ELSE
      pSol=PromptUser(input, "Execution Error", Causes)
RETURN pSol
```

7.1 Routines

This section describes the set of routines used for automatic invocation.

7.1.1 AutoInvoke

The AutoInvoke routine takes the plan as the input and has access to user information stored as facts in the userKB and the outputs received from earlier execution of services stored as facts in the outputKB: both these reside in Jess. The plan has n levels, with the head of the plan corresponding to level 0 and the tail of the plan corresponding to level n. We pick the operators (services) at level 0 and find check if all preconditions associated with them hold TRUE. We then query userKB and the ouputKB to find a match for their inputs. If a match is found for all the inputs, we execute the service and store the ouputs received in the outputKB. We then move on and pick operators from the next level. If all inputs are not found (QueryAndMatch returns NIL), we try to diagnose the cause of error using the Diagnose routine. If all the services in all the levels are executed successfully, then plan execution succeeds.

7.1.2 ExQuery and PartQuery

The ExQuery is used to retrieve exact matches from the KB. This routine returns all triples that corresponds to the given query pattern. It is typically used to find instances of a particular class. The method *instanceTemplateOf(A)* builds a query pattern that cab be used to retrieves all instances of class A.

PartQuery works differently. An instance of a class could contain other instances nested within it. It is possible that the user has specified some of the nested instances and not the outer instance. PartQuery has been designed to look for these nested instances and tries to determine the value of the outer instance from the outputKB. The PartQuery works by first trying to find a match that corresponds to the outermost instance; when this fails, it queries the KB for inner instances. If at any level there is a match, then this instance is enveloped and returned. The *Envelope* routine builds a query pattern for the outermost instance in which the inner instances have a value and outer instances are assigned variables.

7.1.3 QueryAndMatch

This routine has two purposes: (1) Query: find all instances of a particular class in a given KB and (2) Match: pick the correct instance if the query returns multiple instances. The input to this routine is a class definition that corresponds to the input of a service and the knowledge base to be queried. The KB can either be the userKB or the outputKB. The routine behaves differently based on the KB being queried.

If the input is the userKB and if the querying phase returns a single match, the routine returns the instance and terminates. In cases where multiple instances are found, the user is asked to pick the correct choice. The agent might not have enough infromation to pick the right choice. If querying doesn't return a single instance, the routine returns a NIL. The AutoInvoke routine responds to this by again invoking the QueryAndMatch routine, but now with the outputKB as the input.

When the KB being queried is the outputKB, the flow of the routine is somewhat different. The routine first queries the outputKB. If a single instance is returned, the instance is returned and the routine terminates. If more than one instance is returned, the user is asked to pick the right choice. If the query fails to return a value, the routine goes through an additional phase to determine the correct instance. This is where the behavior of QueryAndMatch with outputKB as the input KB differs from its behavior with userKB as the input KB.

The routine now tries to find a matching instance by using incomplete information that might be specified by the user in the userKB. A class definition that corresponds to an input of service could consist of other class definitions (nested classes). The user might have specified the instances of these nested classes in the userKB without instantiating the main outer class. This is an attempt to detemine the correct instance based on partial information specified by the user. We make use of the PartQuery to handle incomplete information. As explained in an earlier section, PartQuery looks for instances of nested classes. Even if a single nested instance

is found, then this instance is wrapped with all outer instances bound to variables. This *enveloped* query pattern is called a hint. The hint is now used to query the outputKB with the hope that an instance corresponding to the complete definition of the input class would be found. If a single result is found then this is the input to the service. If more than one instance is found, user is again prompted to make a choice. Finally, if querying with the hint doesn't return anything, a NIL is returned.

7.1.4 Diagnose

The missing input, and the outputs of all executed services are the input parameters for the Diagnose routine. In this subsection, all references to ouptut(s) mean output(s) of executed services only. The cause of the unavailable input is a result of either incomplete information provided by the user or due to the faulty execution of a service. The check for the first case has already been performed by the AutoInvoke routine. Diagnose routine now tries to determine the service that did not execute as intended. When invoking a sequence of services, the input to the first set of services is always user related information. The input to other services, is either user information or information from the output of a earlier executed service. If the input is derived from output, then we have two cases: (1) the ouput has the exact class defintiion as the input or (2) the class definition of the input is a part of the class definition the output. For example, the class Item (say an input to some service) is a part of the class definition of ItemList which is essentially a container for multiple instances of Item. The GetService checks if the class definition of the missing input is the same as one of the classes that correspond to the output of a service or if it is a nested class of one of the output classes. If the GetService does not find any match, we can conclude that the missing input is expected from the user. It GetService finds one or more matches then we can conclude that the corresponding services did not execute correctly. This list of service(s) is shown to the user. As an example, consider the following two atomic services:

```
(service AvailableFlights
      (input DeptAirport)
      (input DestAirport)
      (input DeptDate)
      (output ListOfFlights))

(service BookFlight
      (precondition ListOfFlightsAvail)
      (input FlightInfo)
      (input UserName)
      (input CreditCardInfo)
      (output BookingStatus)
```

The class descriptions are as follows (we defined only classes relevant to the subsequent discussion):

```
<owl:Class rdf:ID="ListOfFlights">
      <rdfs:subClassOf>
            <owl:Restriction>
                  <owl:onProperty rdf:resource="#member"/>
                  <owl:allValuesFrom rdf:resource="#FlightInfo"/>
            </owl:Restriction>
      </rdfs:subClassOf>
</owl:Class>
<owl:ObjectProperty rdf:ID="member"/>
<owl:Class rdf:ID="FlightInfo">
      <owl:intersectionOf rdf:parseType="Collection">
            <owl:Restriction owl:cardinality="1">
                  <owl:onProperty rdf:resource="#airline"/>
                  <owl:allValuesFrom rdf:resource="#Airline"/>
            </owl:Restriction>
            <owl:Restriction owl:cardinality="1">
                  <owl:onProperty rdf:resource="#deptTime"/>
                  <owl:allValuesFrom rdf:resource="&xsd;#string"/>
            </owl:Restriction>
      </owl:intersectionOf>
</owl:Class>
<rdf:Property rdf:ID="departure"/>
<rdf:Property rdf:ID="destination"/>
<rdf:Property rdf:ID="airlines"/>
<rdf:Property rdf:ID="deptTime"/>
```

A diagramatic representation of this Class would is shown in 7.1

Let us assume that the user provides the following information:

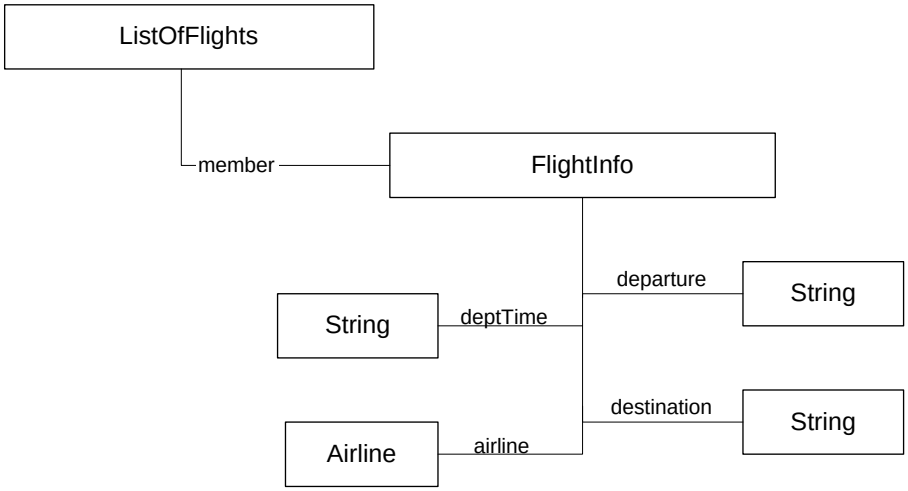

FIG. 7.1. **The ListOfFlights Class**

```
<fly:DeptAirport>
      <fly:name>London Heathrow</fly:name>
</fly:DeptAirport>
<fly:DestAirport>
      <fly:name>Venice Marco Polo</fly:name>
</fly:DestAirport>
<fly:DeptDate>
      <fly:Date>06/02/1979</fly:Date>
</fly:DeptDate>
<concept:UserName rdf:ID="gen534">
      <concept:firstName>Mithun</concept:firstName>
      <concept:lastName>Sheshagiri</concept:lastName>
</concept:UserName>
<concept:CreditCardInfo rdf:ID="gen453">
      <concept:number>1234 54634 6543 67373</concept:number>
      <concept:expiryDate>04/09</concept:expiryDate>
      <concept:type>AmEx Student</concept:type>
</concept:CreditCardInfo>
<concept:Preferences>
      <fly:Airline rdf:resource="easyjet"/>
</concept:Preferences>
```

The AvailableFlight service requires the departure airport, destination airport and the date. On querying the userKB, a match is found for all of these and therefore

the service is executed. The output produced by the service might look like this in OWL:

```
<fly:ListOfFlights rdf:ID="gen234">
<fly:member>
     <fly:FlightInfo rdf:ID="gen654">
          <fly:airline><fly:Airline rdf:ID="flybe"/></fly:airline>
           <fly:deptTime>1730 GMT</fly:deptTime>
     </fly:FlightInfo>
</fly:member>
<fly:member>
     <fly:FlightInfo rdf:ID="gen654">
          <fly:airline><fly:Airline rdf:ID="easyjet"/></fly:airline>
          <fly:deptTime>1822 GMT</fly:deptTime>
     </fly:FlightInfo>
</fly:member>
<fly:member>
     <fly:FlightInfo rdf:ID="gen654">
          <fly:airline><fly:Airline rdf:ID="BA"/></fly:airline>
          <fly:deptTime>1300 GMT</fly:deptTime>
     </fly:FlightInfo>
</fly:member>
<fly:member>
     <fly:FlightInfo rdf:ID="gen654">
          <fly:airline><fly:Airline rdf:ID="ryanair"/></fly:airline>
          <fly:deptTime>0613 GMT</fly:deptTime>
     </fly:FlightInfo>
</fly:member>
<fly:member>
     <fly:FlightInfo rdf:ID="gen654">
          <fly:airline><fly:Airline rdf:ID="quickfly"/></fly:airline>
          <fly:deptTime>0945 GMT</fly:deptTime>
     </fly:FlightInfo>
</fly:member>
</fly:ListOfFlights>
```

The simple representation of the above message would:

```
List of Flights
-----FlightInfo
--------airline---Flybe
--------deptTime--1730 GMT
-----FlightInfo
--------airline---BA
--------deptTime--1300 GMT
-----FlightInfo
--------airline---RyanAir
--------deptTime--0613 GMT
-----FlightInfo
--------airline---QuickFly
--------deptTime--0945 GMT
-----FlightInfo
--------airline---EasyJet
--------deptTime--1822 GMT
```

If only Flybe flew from London to Venice, the message would have been:

```
<fly:ListOfFlights rdf:ID="gen234">
<fly:member>
```

```
     <fly:FlightInfo rdf:ID="gen654">
          <fly:airline><fly:Airline rdf:ID="flybe"/></fly:airline>
           <fly:deptTime>1730 GMT</fly:deptTime>
     </fly:FlightInfo>
</fly:member>
</fly:ListOfFlights>
```

Consider the case where flights are available. The inputs UserName and CreditCardInfo are directly available from the user and values for these inputs are obtained by calling the QueryAndMatch routine for the userKB. However, FlightInfo is not part of userKB, so we check the outputKB. QueryAndMatch retrieves multiple instances of FlightInfo when it queries the outputKB and these correspond to various flights available for the date, departure and destination specified by the user. Picking the right flight involves the user and therefore we query the user to find a hint to narrow down the choice to just one. In this particular case, the user specifies a preference for a specific airline. However, this piece of information is a nested part of the FlightInfo. PartQuery retrieves the user's airline preference and envelopes it with FlightInfo. Now, the outputKB is queried again with a more constrained query and a single instance of FlightInfo is retrieved and returned to the AutoInvoke routine. There could be instances where in spite of hints, mulitple instances could be retrieved. In all such cases, the user is asked to make the right choice.

Now consider the case where flights that matches the user requirements are not found. AutoInvoke checks the userKB and outputKB to find a matching input and both these efforts fail; on both occasions, NIL is returned. AutoInvoke now invokes the Diagnose routine which retrieves the AvaiableFlights service as the possible cause

and informs the user. Knowing that AvailableFlights did not produce the intended ouput, the user can update the userKB to reflect his/her new options.

Chapter 8

CONCLUSIONS

In our thesis, we have presented a OWL-S-based Semantic Web framework that enables an agent to interact web services soley based on service descriptions. In doing so, we have proposed our techniques for service composition, modelling effects and preconditions and service invocation. We have laid emphasis on making our framework relevant to exisitng web services.

Semantic Web Services has great potential to contribute to the existing WSDL-based web service framework by enabling increased automation.

However, we encountered several obstacles and observations while designing and implementing our framework. We list some of these below:

1. OWL-S is by far the most mature initative to address the current limitations of WSDL-based web services.

2. The OWL-S/DAML-S are fairly recent initiatives with a low percentage of active contributors from the industry. As a result, tools for using OWL-S are few. For

example, we wrote all our services using a XML editor. The quality of OWL-S tools is closely tied to tools available for the Semantic Web.

3. The learning curve required to acquire competency to use OWL-S is steep. This is because OWL-S specifications are constantly changing and evolving (not neccessarily in a modular way).

4. Disagreements remain on key issues like preconditions and effects.

5. Convincing the industry to adopt OWL-S is only possible if the vision of Semantic Web is accepted and adopted by the industry.

6. Using and understanding OWL-S requires a good understanding of concepts like expert systems, ontologies and reasoning. It is not reasonable to expect software developers to acquires these skills. Tools which abstract the underlying complexity involved in using OWL-S will greatly help developers.

REFERENCES

[1] Amazon.com Web Services, `http://www.amazon.com/gp/aws/landing.html` , Amazon.com, 2003.

[2] alphaWorks: Web Services Toolkit, IBM, July, 2000. Availabe at `http://www.alphaworks.ibm.com/tech/webservicestoolkit` .

[3] Ankolenkar, A., M. Burstein, J. R. Hobbs, O. Lassila, D. L. Martin, D. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. R. Payne, and K. Sycara: 2002, *DAML-S: Web Service Description for the Semantic Web.* In *Proc. 1st Int'l Semantic Web Conf. (ISWC 02)*, 2002.

[4] Bacchus F. and Teh Y. W., *Making forward chaining relevant* In Proc. *4th Intl. Conf. AI Planning Systems*, pages 54–61, June 1998.

[5] Bussler, C., A. Maedche, and D. Fensel, *A Conceptual Architecture for Semantic Web Enabled Web Services.* In *ACM Special Interest Group on Management of Data, Volume 31(4)*, Dec, 2002.

[6] Cardoso J. and Sheth A., *Semantic e-WorkFlow Composition* In *Journal of Intelligent Information Systems, 21(3):191-225*, 2003.

[7] Christensen E., Curbera F., Greg Meredith, Sanjiva Weerawarana, *WSDL Web*

Service Description Language (WSDL). W3C Note 15, March 2001. Available at `http://www.w3.org/TR/wsdl` .

[8] DAML.org, March, 2001, `http://www.daml.org/` .

[9] Friedman-Hill, E. J., *Jess in Action: Java Rule-based Systems*, Manning Publications Co., 2003

[10] Jena, Version 1, *Jena Semantic Web Toolkit*, Hewlett Packard Company, 2002. Available at `http://jena.sourceforge.net/` .

[11] Kopena J., *DAMLJessKb*, November, 2002. Available at `http://plan.mcs.drexel.edu/projects/legorobots/design/software/DAMLJessKB/` .

[12] Li L. and Horrocks I., *A Software Framework For Matchmaking Based on Semantic Web Technology In Proceedings of the Twelfth International World Wide Web Conference (WWW 2003)*, 2003.

[13] McIlraith, S. and T. Son, *Adapting Golog for Composition of Semantic Web Services.* In: *Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning (KR2002)*, Toulouse, France, 2002.

[14] Paolucci M., Sycara K., and Kawamura T., *Delivering Semantic Web Services* In *Proceedings of the Twelfth World Wide Web Conference (WWW2003)*, Budapest, Hungary, May 2003, pp 111- 118

[15] Paolucci M., Kawamura T., Payne T. R., and Sycara K., *The First International Semantic Web Conference (ISWC) First International Semantic Web Conference (ISWC)*, Sardinia, June, 2002.

[16] Ponnekanti S. P.and Fox A., *SWORD: A Developer Toolkit for Web Service Composition.* In: *In Proc. of the Eleventh International World Wide Web Conference*, 2002.

[17] Sheshagiri M., desJardins M., Finin T., *A Planner for Composing Services in DAML-S Workshop on Planning for Web Services*, July, 2003, Trento, Italy.

[18] Srivastava, B., *Automatic Web Services Composition Using Planning.* In: *In Proc. of KBCS 2002*, Mumbai, India.

[19] WebServices-Axis, The Apache Software Foundation, 2003. Available at `http://ws.apache.org/axis/` .

[20] Weld, D., *An Introduction to Least Commitment Planning.* In: *AI Magazine, 15(4):27-61*, Winter 1994.

[21] Wu D., Parsia B., Sirin E., Hendler J. and Nau D., *Automatic DAML-S Web Services Composition Using SHOP2* In *Proceedings of 2nd International Semantic Web Conference (ISWC2003)*, Sanibel Islands, Florida, 2003

[22] XMethods, XMethods Inc, 2003. Available at `http://www.xmethods.com` .