DOI: 10.1007/s11432-006-2026-2

Dynamic web service composition based on OWL-S

Jing DONG[1], Yongtao SUN[2], Sheng YANG[1] & Kang ZHANG[1]

1. Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083, USA
2. American Airlines, 4333 Amon Carter Blvd, Fort Worth, TX 76155, USA
Correspondence should be addressed to Jing Dong & Kang Zhang (email: {jdong, kzhang}@utdallas.edu)
Received May 9, 2006; accepted September 4, 2006

Abstract Composing existing web services for enterprise applications may enable higher level of reuse. However the composition processes are mostly static and lack of support for runtime redesign. In this paper, we describe our approach to the extension of the OWL-S ontology framework for dynamic web service composition. We raise the level of abstraction and propose an abstract service layer so that web services can be composed at the abstract service level instead of the concrete level. Each abstract service is attached with an instance pool including all instances of the abstract service to facilitate fail-over and dynamic compositions.

Keywords: web service, instance pool, abstract service, ontology, dynamic composition.

1 Introduction

Recent advance on web services computing enables building business processes and systems through the discovery and integration of the existing services. The increasing number of web services available on the Internet not only facilitates such new technology, but also poses new challenges on how to support collaboration and orchestration of these services for business applications, especially real-time enterprises.

Currently, web services are typically described in terms of atomic and composite web services, using languages like BPEL[1,2] or OWL-S[3,4] which provide mechanisms for web service compositions. However, the processes of web service compositions tend to be static in the sense that these processes are normally generated off-line. Any changes to the part of a process may result in the reconfiguration of the whole process. It lacks the support of the capability of fail-over and dynamical redesign. This is especially critical for real-time enterprises since the systems cannot afford stopping, reconfiguring, and restarting. If some web services of a composition fail or the requirements change, the system needs to be able to change locally and reconfigure on-the-fly. For example, customers can have a stock trading service, which involves obtaining relevant quotes and

performing transactions. If one specific quoting service fails, the whole process can still proceed by switching to another quoting service.

Among the numerous available web services, many of them provide similar services. Even with different implementations, most of them present a similar interface to the end user. For example, there are several different on-line venders, such as Yahoo and MSN, which provide weather forecast service. Although they may have a different way to invoke the service, they all typically have an operation that accepts zip code and returns the weather information. However, there is currently little effort on abstracting these similar services into high-level common services. Although the OWL-S language provides a way to describe the hierarchical relationship between services, the recommended ontology framework is still limited to one root-level abstract service. Raising the level of abstraction and capturing similar services as a service pool are important, especially for dealing with fail-over and dynamic redesign in real-time enterprises and e-business.

In this paper, we propose an extension to the ontology framework based on OWL-S, which enables defining the composite services at the abstract service level. We provide new constructs to specify such higher level of abstraction. Our approach also includes a service instance pool that allows filtering and plugging in candidate services at runtime. In addition, we offer a planner prototype based on Java Theorem Prover (JTP) [5] that can automatically generate the composition processes on-the-fly. To illustrate our approach, we also provide a case study related to air travel itinerary.

The rest of this paper is structured as follows. Section 2 gives an overview of our approach. Section 3 defines our extension to the OWL-S Ontology Framework. We describe a case study to illustrate our approach in section 4. The last two sections present the related work and conclusions.

2 Overview

In this section, we first give a brief introduction to the OWL-S language and the recommended ontology framework. We then present a motivating example followed by the high-level architecture of our approach.

2.1 OWL-S language and ontology framework

OWL-S is an OWL-based Web Service Ontology that evolves from DARPA Agent Markup Language[6]. OWL-S defines a core set of markup language constructs for describing web services. Additionally OWL-S proposes a recommended ontology framework to facilitate the definition of web service. In this framework, each service has three sets of information: Profile, Process and Grounding. All these three pieces are connected together by the service ontology as shown in Fig. 1. Profile describes the semantic properties and capabilities of a service; Process represents the actual composition logic; and Grounding provides the physical binding information for runtime invocation, which also has a link to the WSDL file for that service.

The OWL-S Ontology Framework allows us to define a concrete web service. The Advertisement and Discovery information can be found in Profile. Process is used for

complex service composition logic. Grounding provides the detailed invocation access information.

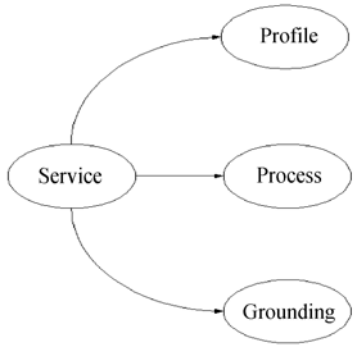

Fig. 1. OWL-S Upper Ontology.

2.2 A motivating example

Although OWL-S and its recommended ontology support the service hierarchy, they mainly focus on the definition of concrete web services. Each OWL-S service is mapped either to one physical concrete web service or to a fixed composition of a set of physical concrete web services, which is still one concrete (although not physical) web service looking from outside. All OWL-S classes, such as "Profile" (the child class of "ServiceProfile"), "Process" (the child class of "ServiceModel") and "Grounding", contain different parts of the information about the concrete service. In reality, an abstract service is as important as a concrete service, and can be even more valuable for a huge otology of services. Current OWL-S cannot represent a virtual abstract service. An abstract service may have multiple composition logics as the implementation, while one OWL-S web service normally has only one composition process. An abstract service usually represents a collection of similar concrete services, while an OWL-S service maps to only one concrete web service at runtime. Although the Grounding information can be bound to any WSDL allowing OWL-S to dynamically change the service instance, it does not have the dynamic customization information. Without such information, one has no clue whether the new service instance can fit for this OWL-S service.

For example, assume we build a hierarchy for a TV domain including the brand, screen type and size of a TV set. Fig. 2 presents an example of the TV concrete services, such as the Sony TV Service, Sony Flat TV Service, Sony 40″ Flat TV Service, and Sony 40″ Bravia TV Service[1)], which are defined as concrete services using the recommended OWL-S ontology framework. There are typically two levels: root level OWL-S ontology framework and concrete OWL-S services. The root level defines a recommended ontology framework which is discussed in the previous section. At the concrete service level,

1) Note that the Sony TV Service, Sony Flat TV Service, Sony 40″ Flat TV Service, and Sony 40″ Bravia TV Service refer to the corresponding services that produce the Sony TV Service, Sony Flat TV Service, Sony 40″ Flat TV Service, and Sony 40″ Bravia TV, respectively. For simplicity, we use the "TV service" and the "service that produces a TV" interchangeable in the rest of this paper, so do the "flight service" and the "service that checks flight information".

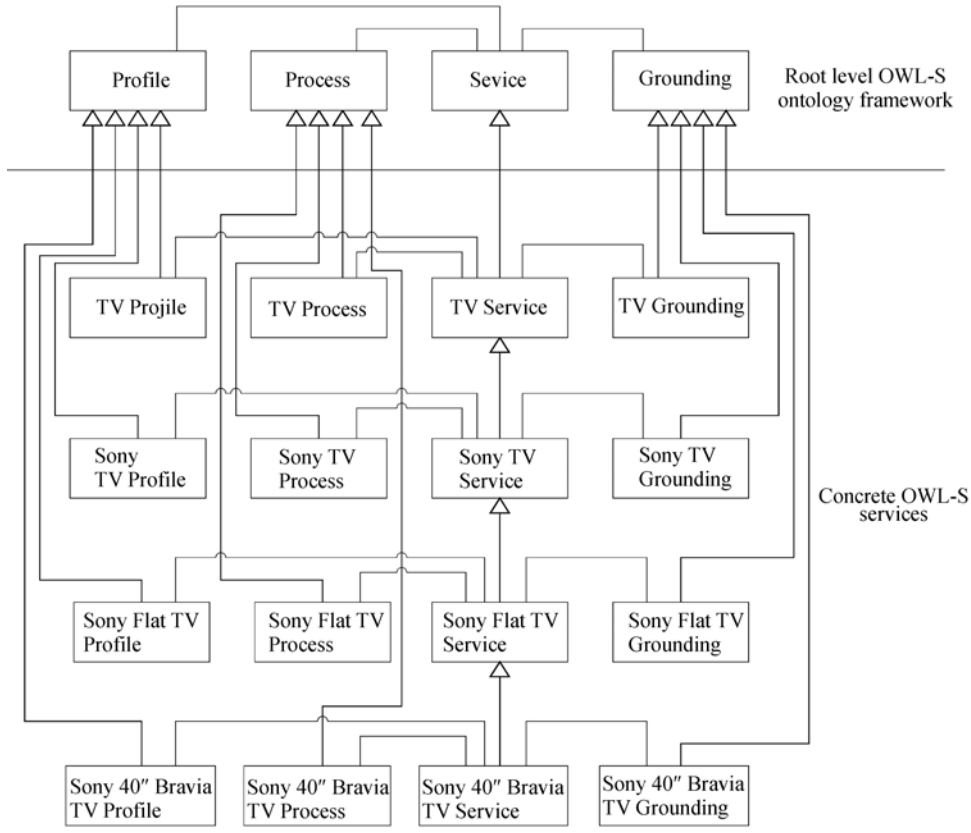

Fig. 2. An example of current OWL-S hierarchy.

only the Service allows a multi-level inheritance hierarchy where the children may indirectly reuse their ancestors (in addition to their direct parent). The concrete services can only directly inherit from the Profile, Process, and Grounding. It is normally impossible to have multi-level inheritance hierarchy besides the Service. This may cause some problems when we want to define a Sony Flat TV abstraction that can have multiple service instances to produce it. For instance, a Sony Flat TV abstraction may have instances whose brand is Sony with a flat screen, so do the Sony 40″ Flat TV Service and Sony 40″ Bravia TV Service. Currently, the profiles of all these three concrete services are the direct children of the Root Profile Class. In this case, the Sony 40″ Bravia TV Service Profile has no relation to the Sony 40″ Flat TV Service Profile (although they may share some common parameters), so do the corresponding Processes and Groundings. The fact that they are both the children of the Sony Flat TV abstract Profile is not captured. Similarly, all the three concrete Processes are the direct children of the Root Process Class. They are not related to each other and to the Sony Flat TV abstract Process. Thus, the knowledge that an abstract Sony Flat TV Process can have different kinds, i.e., it can be built in different sizes and ways, is lost. Furthermore, the Grounding can only map one instance to a concrete service. Using the current OWL-S, an abstract Sony Flat TV service can be defined such that it can only be grounded to one instance. In practice, an ab-

stract service may have multiple instances. For example, a weather forecast service may contain the instances such as Yahoo and MSN.

2.3 Architecture overview

Fig. 3 presents an architecture overview of our approach that focuses on the definition of abstract services based on both existing web service instances and the user's expected goals. This architecture contains a web-based composer, a backend standalone utility and an extended OWL-S services ontology. Users can specify their goals via the web-based composer that can generate the result service using the extended OWL-S Service ontology. The result service may be added back to ontology if it is brand new. The standalone utility is responsible for updating the instance pool information of service ontology if new concrete services detected.

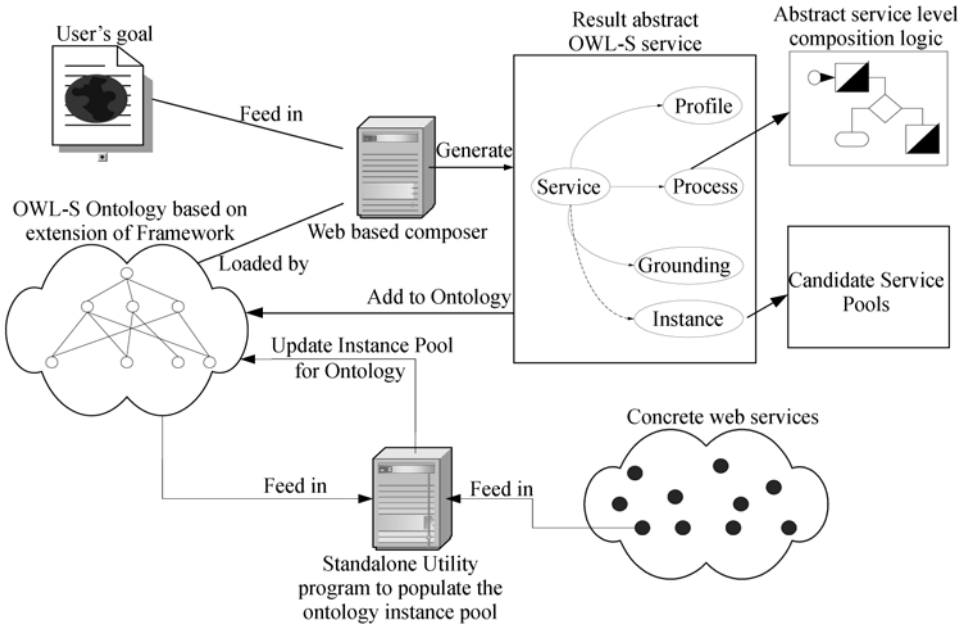

Fig. 3. Architecture overview.

We define an extension to the OWL-S recommended ontology framework for this purpose. We first define an abstract service hierarchy by grouping the available concrete services into different categories based on their functional characteristics, such as input and output, and specific nonfunctional service parameters. As shown in Fig. 4, for example, multiple-level inheritance hierarchy is enabled to represent the hierarchical relationships among these abstract services. A concrete service (existing OWL-S service) can be plugged into an appropriate level. For example, the Sony 40" Bravia service is a type of the abstract Sony Flat TV service, which is in turn a type of the abstract Sony TV service and the abstract TV service. Similarly, the concrete Sony TV service shown in Fig. 2 can be the instance of the abstract Sony TV service in Fig. 4. Since there are thousands of web services already deployed in the Internet and new services available every day which

may not be aware of the abstract service hierarchy, we develop a backend utility program to register a concrete service into a domain service hierarchy and dynamically update the service ontology when detecting a change. The newly introduced abstract service layer does not require the concrete service to completely conform to the definition of abstract service because our backend utility can automatically detect the equivalent relation between semantic concepts. If one specific TV abstract service has a "maximumWeight" serviceParameter and a concrete Sony TV service has a "weightLimit" serviceParameter, our utility program considers it a match if the "sameas" relationship has been defined for the "maximumWeight" and "weightLimit" serviceParameters in the ontology. In this paper, we assume that all concrete services of the same abstract service share the same interface information, i.e., they all have the same IOPE parameters, so that we can focus on the abstract service hierarchy. We plan to address different interface mapping in the future.

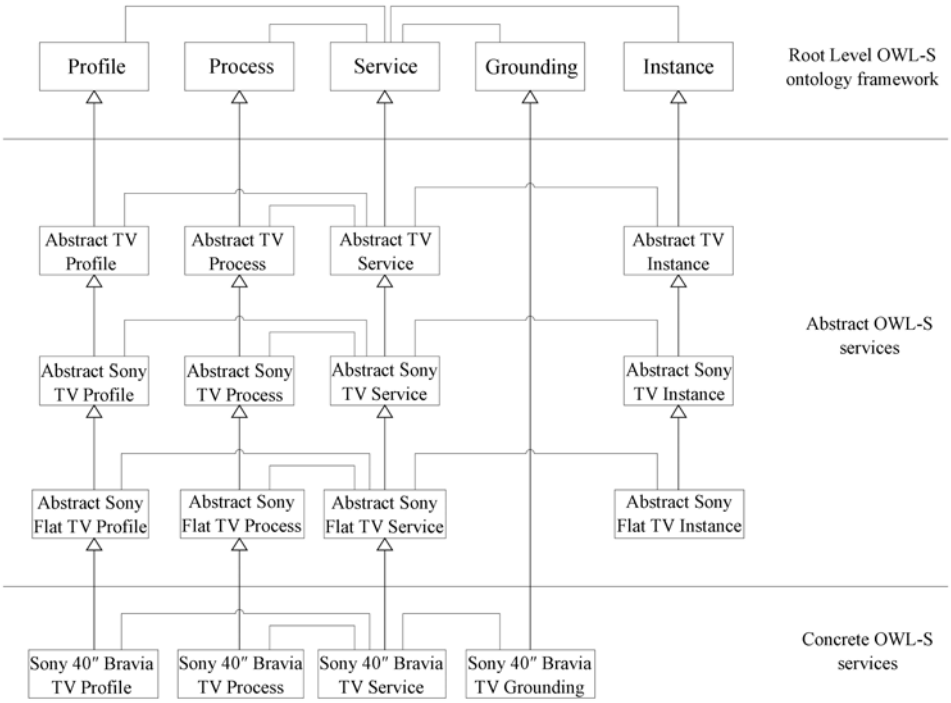

Fig. 4. An example for extended OWL-S hierarchy.

If a composition of services is requested for business or enterprise applications, either an existing abstract service can be matched or a new abstract service is defined with its Profile section containing the input/output and the semantic properties. The Process and new Instance sections can be generated by our composition planner in two steps. First, based on the ontology hierarchy of all available abstract services for existing services and the definition of the user's goals including both the functional requirement and nonfunctional service parameters, we can obtain the composition process (the Process section of the abstract service that satisfies the user's goals) using our composition planner based on

JTP [5]. This generated composite service matches all the input/output and flow-related semantic requirements. This can narrow down the candidate scope by filtering out all other unrelated web services. Second, other functional service parameters describing the user's goals are recorded in the Profile section of the abstract service. They are used to identify the actual candidate service pool for composite service and filled in the new Instance section if a direct concrete service candidate is found.

3 Extension to OWL-S ontology framework

In this section, we describe our proposed extension to the OWL-S recommended framework. This extension includes the information on the input/output, flow control, semantic property, and candidate instance pool of the abstract service in the ontology hierarchy. This new ontology framework contains the following features:

A new "Instance" section is added to the OWL-S recommended ontology framework. This new section provides the information about the candidate concrete service instances for this abstract service. These service instances can be a standard web service, an OWL-S or a BPEL service. This Instance section is different from the Grounding section since it does not provide a binding to a physical web service. Instead, it contains a collection of references to the available candidate service instances, and in turn to the candidate service's "Grounding". "Instance" and "Grounding" are mutually exclusive. Only abstract service has "Instance", which means it is an abstraction of a group of candidate services. "Grounding" is defined in current OWL-S to describe the access information of a concrete service. If the Instance section of an abstract service is not empty, there is at least one concrete service available for direct invocation. We call all these available concrete services the *instance pool* of the corresponding abstract service. At runtime, the system can pick up a candidate service from the pool and invoke it via its own binding information (like Grounding in OWL-S or WSDL in standard web service). If an abstract service does not have any candidate service instance, it may obtain its instance(s) from the Process section at runtime. For example, a young customer may want a service called a "Scooter TV" service that allows him to both ride a scooter and watch TV. Since this is not a typical service available, the Instance section of the "Scooter TV" abstract service is empty. The Process section of "Scooter TV" can specify that the flow of "Scooter TV" is just a "Scooter" service plus a "TV" service. Suppose both "Scooter" and "TV" have a number of candidate services in their Instance sections. In this way, a composite concrete service for the abstract service "Scooter TV" can be obtained by picking one candidate service instance from the instance pool of "Scooter" and another from that of "TV". The new Instance section and its relationships to other sections are illustrated in Fig. 4. More details about the Instance section is presented in section 3.2.

New constructs are added to the Process section of the recommended framework. In the Process section of the current OWL-S framework, each OWL-S service can only be an Atomic process, a Simple process, or a Composite process. No matter which type of service it is, it can only contain one work flow logic. We define a new "Abstract Process" type, which has an "abstractComposedOf" attribute to specify all possible composition

processes it can have. Only Abstract Service can have "Abstract Process" since it may have multiple implementation ways. It is purely a supplement to support the abstract service. For concrete services, the process type is still one of the atomic, simple and composite processes. In our approach, we can define a collection of atomic/simple/composite processes containing multiple work flow logics in the new abstract service. For example, the work flow logics of both Sony and GE TV services are included in the new abstract TV service. Either can be used to fulfill a TV service. More details are presented in section 3.3.

An abstract service does not contain the Grounding information since it is not mapped to any physical service. Instead, it gets the candidate instance from its Instance section which is like a resource pool.

In the existing OWL-S ontology framework, each service in the hierarchy is still a concrete service. With the above two framework extensions, we can define an abstract service layer, shown in Fig. 4, so that future service compositions can be made at the abstract level.

From software architecture and design perspective, our approach defines a service architecture that has more complex structure than the simple "subClassOf" relationship in the current OWL-S framework. With the support of abstract services, it is possible to define more complex service structures based on, e.g., inheritance, information sharing, and polymorphism. Therefore, the Profile, Process or Instance of an abstract service is the subclass of the Profile, Process or Instance of its parent service shown in Fig. 4. In the following sections, we present more details of our extension.

3.1 Extension to root level OWL-S

In order to connect the Instance section to its Service, a new predicate "implementedBy" is introduced to the root level OWL-S ontology framework. This links an Instance.owl to its Service.owl. The Instance.owl contains the instance pool information for the abstract Service. Additionally, some new constructs (see section 3.3) are introduced in ServiceModel which is the parent class of the Process. The Service.owl file with our extension is shown as follows. Fig. 5 shows the visual RDF schema of the Service.owl[1)], where our extensions are marked.

```
<!-- Service Implementation -->
<owl:Class rdf:ID="ServiceInstance">
    <rdfs:label>ServiceInstance</rdfs:label>
    <rdfs:comment> this class contains the candidate pool information for abstract service</rdfs:comment>
</owl:Class>

<!-- Being implemented by an instance  -->
<owl:ObjectProperty rdf:ID="implementedBy">
```

1) In the remainder of this paper, we only show the visual RDF schema without the corresponding RDF file to save space and for better visualization.

```
    <rdfs:domain rdf:resource="&service;#Service"/>
    <rdfs:range rdf:resource="&service;#ServiceInstance"/>
    <owl:inverseOf rdf:resource="&service;#implements"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="implements">
    <rdfs:domain rdf:resource="&service;#ServiceInstance"/>
    <rdfs:range rdf:resource="&service;#Service"/>
                <owl:inverseOf rdf:resource="&service;#implementedBy"/>
</owl:ObjectProperty>
```

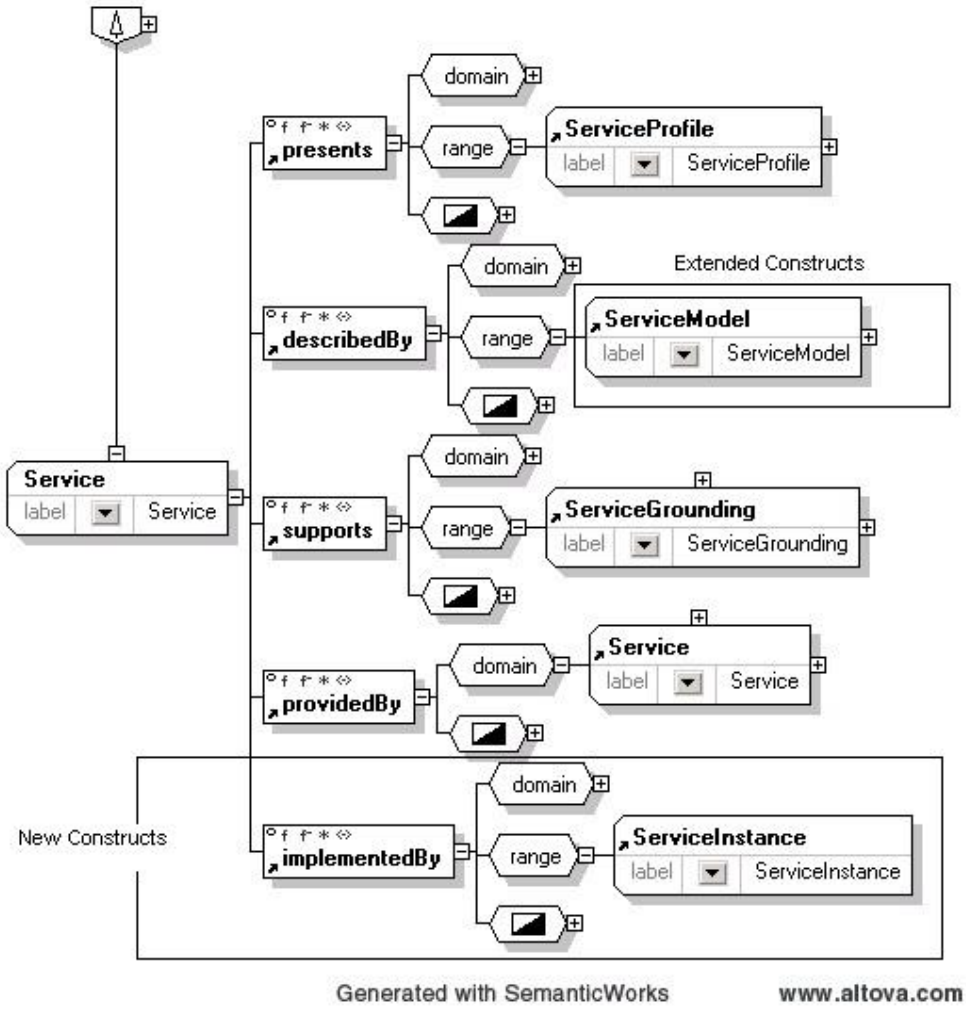

Fig. 5. Visual RDF Schema for extended OWL-S upper ontology.

● **ServiceInstance:** the root class that represents the instance pool of an abstract service shown in Fig. 5. Its subclass, like "Sony_TV_Instance" shown in Fig .6, contains all

reference information of the candidate instances for that particular abstract service, e.g., "Sony_TV_Service".

● **implementedBy:** An object property extended for a Service class as shown in Fig. 5. The resource of this property points to a ServiceInstance. Fig. 6 presents an example of a visual definition of the abstract service "Sony_TV_Service". implementedBy is used to connect the "Sony_TV_Service" and "Sony_TV_Instance".

● **implements:** An object property of a ServiceInstance class. It is an inverse property of implementedBy.

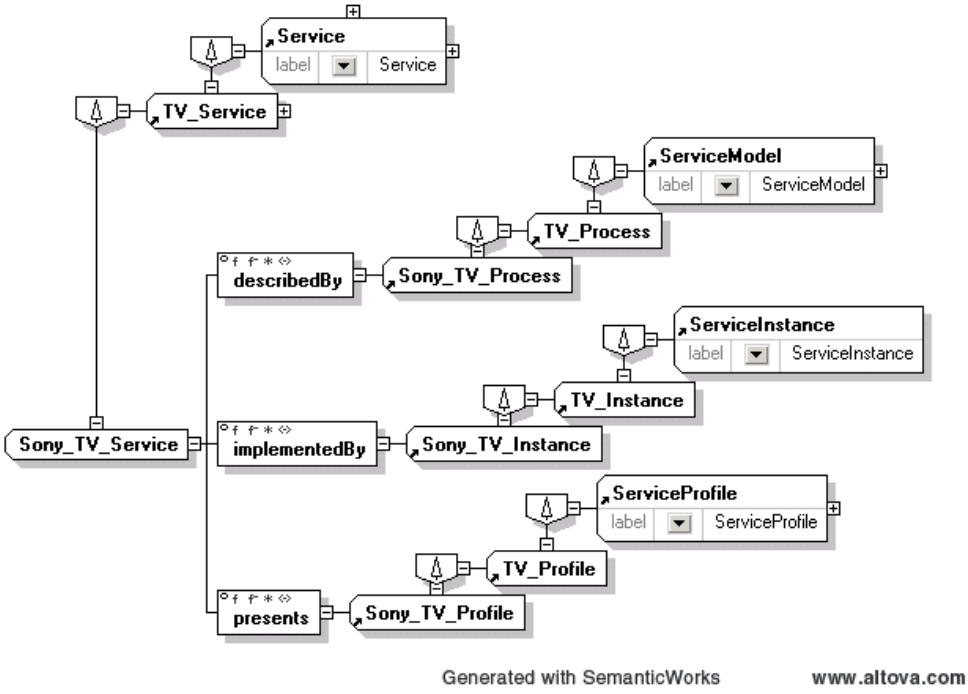

Fig. 6. Visual RDF Schema for Sony_TV_Service.

3.2 Instance.owl

As discussed previously, the Instance class has a brand new set of information introduced to describe the information of the instance pool of the available concrete service for the corresponding abstract service. It is a subclass of "ServiceInstance" shown in Fig. 5. Fig. 7 shows the visual RDF schema for the Instance class:

● **Instance:** A subclass of the ServiceInstance shown in Fig. 5. It is the root class for all abstract service "Instance". Fig. 8 presents an example of "Sony_TV_Instance" which is a subclass of Instance. The "Sony_TV_Instance" abstract service contains two concrete services in the instance pool. The overall relationship to other ontology framework objects can be found in Fig. 6.

● **preferenceOrder:** A data property of the Instance class. It specifies the user's preference when choosing the candidate service. It currently has four possible values: **Sequential, RoundRobin, Random** and **PriorityCode**. As an example in Fig. 8, the

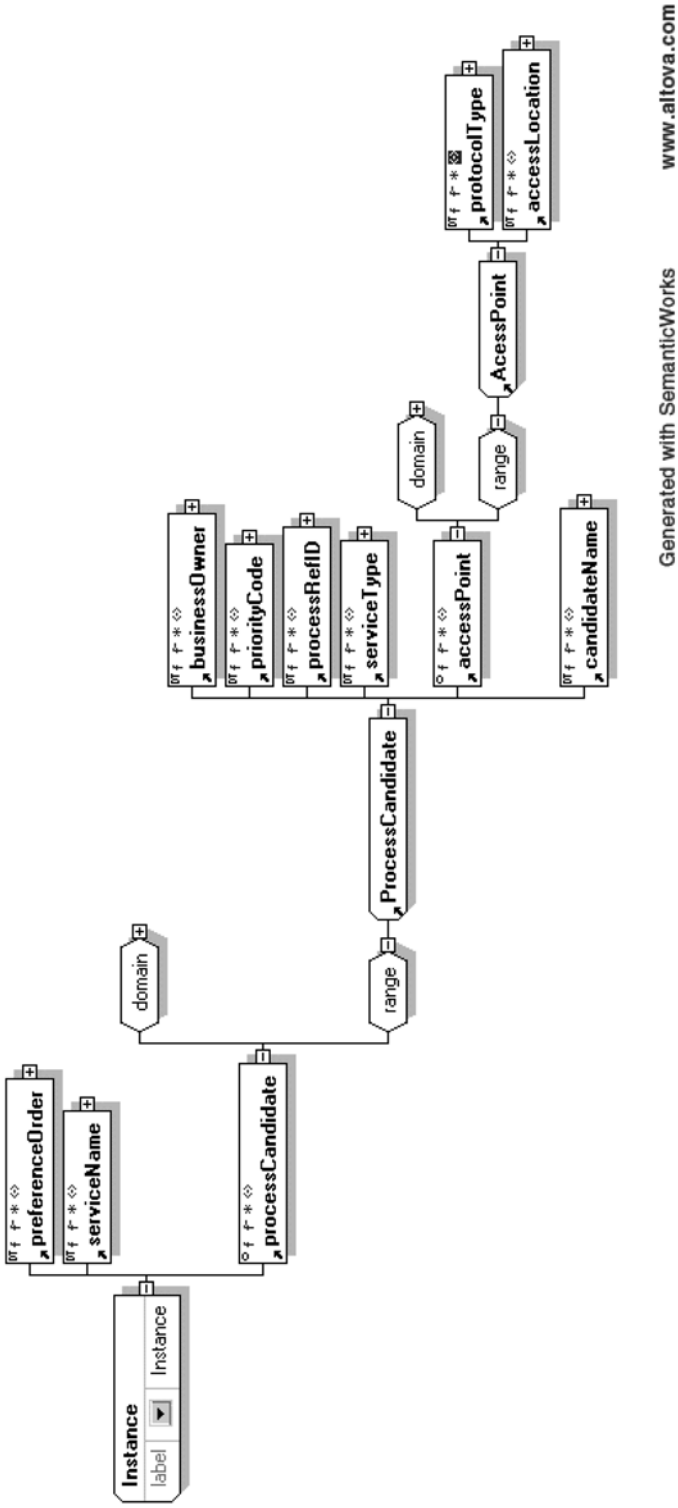

Fig. 7. Visual RDF schema for "Instance" ontology.

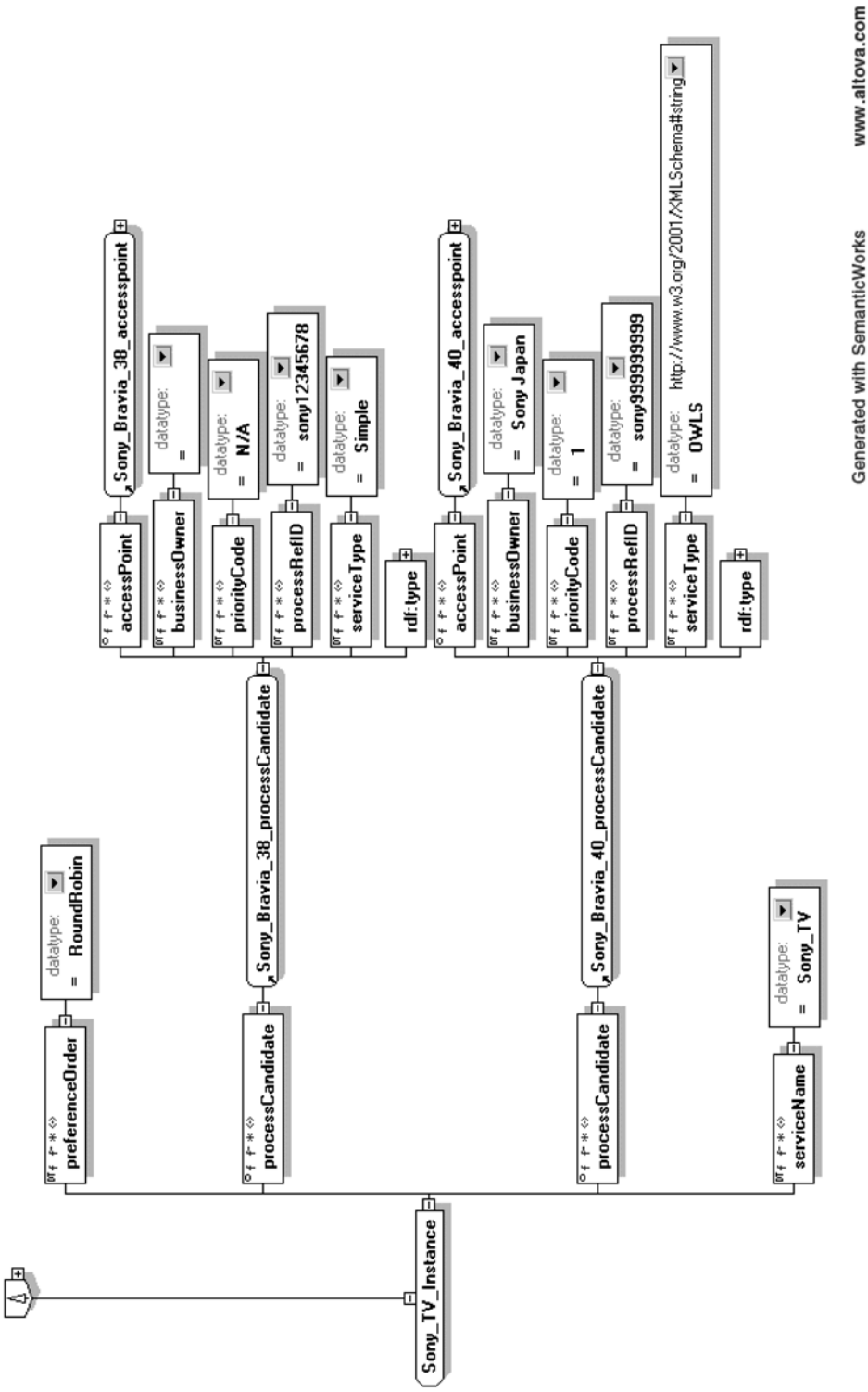

Fig. 8. Visual RDF schema for "Sony_TV_Instance".

"Sony_TV_Instance" has the "Random" as the value of preferenceOrder, which means the system randomly picks up a candidate service at runtime.

- **ProcessCandidate:** A class that represents an implementation instance. Each ProcessCandidate points to a real web service in the Internet, such as the "Sony_Bravia_38_processCandidate" or "Sony_Bravia_40_processCandidate" in the example in Fig. 8.
- **businessOwner:** a data property of the ProcessCandidate class that specifies the owner of concrete web service. Fig 8 shows "Sony_TV" as the business owner of this Instance class.
- **priorityCode:** a data property of the ProcessCandidate class that specifies the priorityCode assigned. The value of priorityCode is greater than or equal to 0 with 0 being the highest priority. If the value of preferenceOrder is set to PriorityCode, the system takes this property value to determine which candidate will be chosen at runtime.
- **processRefID:** a data property of the ProcessCandidate class that specifies the id of a concrete web service. It is an optional reference to the physical web service.
- **serviceType:** a data property of the ProcessCandidate class. Currently three types exist: Simple, BPEL and OWL-S. As defined in Fig. 8, for instance, "Sony_Bravia_40_processCandidate" is an OWL-S service.
- **AccessPoint:** A class that represents the access method of a concrete web service.

protocolType: a data property of the AccessPoint class. The possible value can be "HTTP", "FTP", and etc.

- **accessLocation:** a data property of the AccessPoint class that specifies the access address of a concrete web service. For example, the accessLocation for the "Sony_Bravia_38_accesspoint" is "http://www.sony.com/service/bravia38/". The system can use this location to retrieve the detailed information of a concrete service.
- **processCandidate:** An object property for an Instance Class. The resource of this property points to a ProcessCandidate class.
- **accessPoint:** An object property for a ProcessCandidate class. The resource of this property points to an AccessPoint class.

3.3 New Constructs in Process.owl

For each abstract service, the composition logic may not be unique. Suppose a traveler who wants to schedule a flight from New York to San Francisco. There can be a number of different itineraries, like a direct flight from New York to San Francisco, a connection flight of New York – Chicago – San Francisco, or a connection flight of New York – Denver – San Francisco. Thus, the abstract service needs to have the capability to represent multiple composition logics, which is not possible with the current "Process" class. Therefore, we introduce a new "AbstractProcess" class in the Process section as marked in Fig. 9. This class provides a collection of Process logics, like different itineraries from New York to San Francisco. It has an "abstractComposeOf" property which maps to a ControlConstruct. The ControlConstruct maintains a collection of available composition solutions, which can be one process (like an AtomicProcess, a SimpleProcess or a CompositeProcess) or a combination of processes and control constructs.

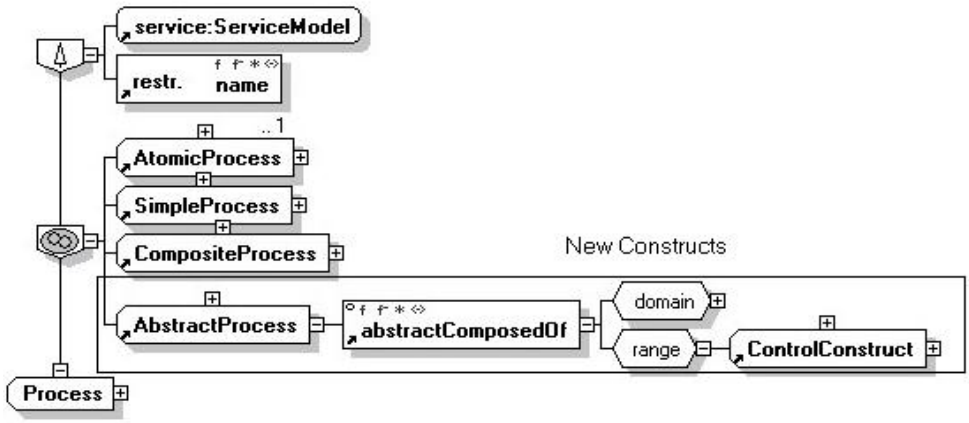

Fig. 9. Visual RDF schema for "Process" ontology.

● **AbstractProcess:** A class represents the possible composition logics. In current OWL-S framework, an actual Process is a subclass of the union of AtomicProcess, SimpleProcess and CompositeProcess. With the introduction of this new AbstractProcess, we are able to describe multiple composition solutions in the Process section of an abstract service.

● **abstractComposedOf:** an object property for AbstractProcess. A collection of ControlConstructs are linked to an AbstractProcess via this abstractComposedOf.

ControlConstruct: relocated from the CompositeProcess of the existing OWL-S Framework which can be used to describe a composition flow for one unique solution. Moving this construct from CompositeProcess to here allows us to describe a collection of possible solutions. In the "New York to San Francisco" Process, for example, it may use a sequence list (one subclass of ControlConstruct) to contain all possible itineraries.

3.4 Planner Prototype

In order to facilitate the generation of different composite processes, we develop a web-based composer. It utilizes the J2EE web architecture and the Embedded JTP server. Fig. 10 displays a screen shot of the system. This tool can be used to generate the resulting abstract OWL-S files. Users can load their special ontology while the default OWL-S ontology is automatically loaded in the planner. If an OWL-S service is specified by checking the "OWL-S Service" checkbox (see Fig. 10), the composer fetches the corresponding Profile, Process and Instance files if available. The customer's goals are converted into RDF entries and fed into the JTP inference engine. Consequently, the detailed inference steps are shown on the screen, and the resulting service files are generated when the user finishes the query and clicks the "Generate Result Service" button. Our prototype tool can be used for other service compositions, although we show flight service composition as an example in this paper.

One of the main differences between our extension and the original OWL-S is that we

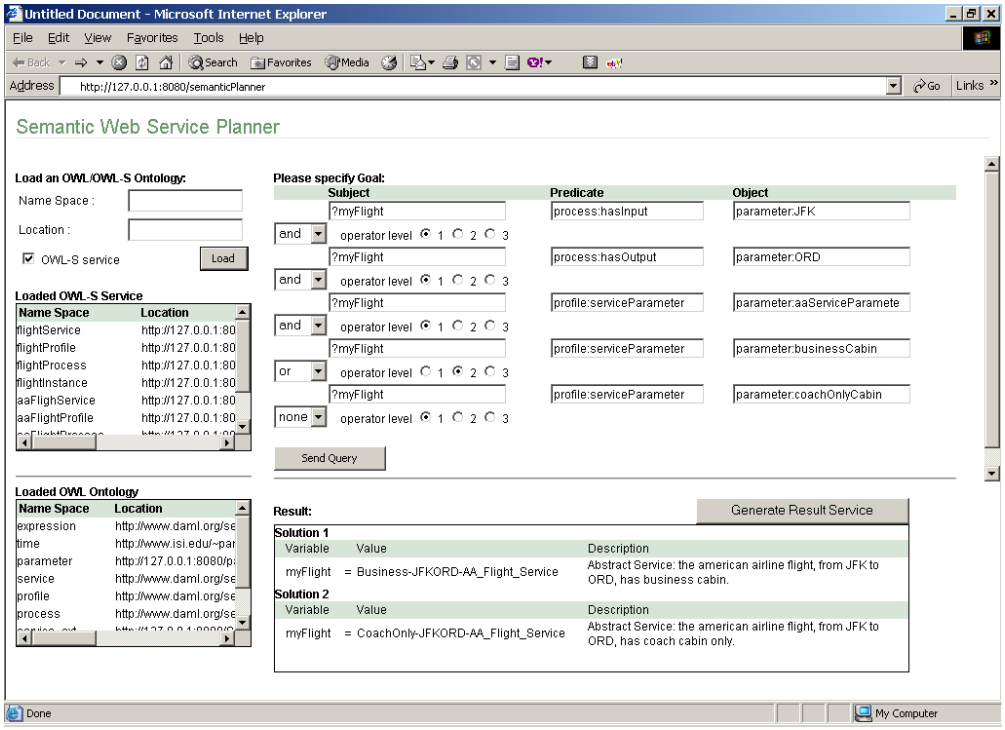

Fig. 10. Composer Prototype.

propose abstract service layers. Each specific abstract service needs to connect to its own profile, process and instance while maintaining the inheritance relationship with its parent class. To maintain the correct relationship for different parts of the same service, each abstract service uses the "someValuesFrom" "allValuesFrom" and "hasValue" restrictions. Like the following example, AA-Flight_Service is connected to AA-Flight_Profile, instead of the Root level Profile Class.

```
<owl:Class rdf:about="#AA-Flight_Service">
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:someValuesFrom>
                <owl:Class rdf:about="#AA-Flight_Profile"/>
            </owl:someValuesFrom>
            <owl:onProperty rdf:resource="http://www.daml
            .org/services/owl-s/1.1/Service.owl#presents"/>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="http://www.daml
           .org/services/owl-s/1.1/Service.owl#presents"/>
            <owl:allValuesFrom>
```

```
                    <owl:Class rdf:about="#AA-Flight_Profile"/>
                </owl:allValuesFrom>
            </owl:Restriction>
        </rdfs:subClassOf>
        ……..
```

Similarly, we use these restrictions to specify the IOPE for the profile and process of each abstract service. For instance, the profile of "AA-Flight_Service" in Section 4 can be defined with a "hasValue" restriction that sets the serviceParameter property to "aaServiceParameter", which means AA-Flight_Profile has a non-functional service parameter that is set to American Airlines. Likewise, "JFK-ORD-Flight_Profile" in Section 4 may have a "hasInput" property with the "hasValue" restriction set to airport "JFK" and a "hasOutput" property with the "hasValue" restriction set to "ORD".

Our planner prototype converts the user's goals into backend JTP query string and generates the Process based on the query result. The "and" and "or" operators are used on different levels to combine each JTP query string. Currently the prototype tool specifies three levels where level 1 is the top level. Fig. 10 shows an example of the functional requirement of the user's goal to get an abstract service which has the departure airport "JFK" (input) and the arrival airport "ORD" (output). The non-functional requirements of the user's goal are that the flight service belongs to American Airlines and has either BusinessCabin or CoachOnlyCabin. The following is the generated JTP queries:

```
(and
    (|http://www.daml.org/services/owl-s/1.1/Process.owl#|::|hasInput|
        ?myFlight |http://127.0.0.1:8080/Parameters.owl#|::|JFK|)
    (|http://www.daml.org/services/owl-s/1.1/Process.owl#|::|hasOutput|
        ?myFlight |http://127.0.0.1:8080/Parameters.owl#|::|ORD|)
    (|http://www.daml.org/services/owl-s/1.1/Profile.owl#|::|servicePara
        meter| ?myFlight
        |http://127.0.0.1:8080/Parameters.owl#|::|aaServiceParameter|)
    (or
        (|http://www.daml.org/services/owl-s/1.1/Process.owl#|::|servicePa
            rameter| ?myFlight |http://127.0.0.1:8080/Parameters.owl#|
            ::|businessCabin|)
        (|http://www.daml.org/services/owl-s/1.1/Process.owl#|::|servicePa
            rameter| ?myFlight |http://127.0.0.1:8080/Parameters.owl#|
            ::|coachOnlyCabin|)
    )
)
```

Users can keep refining their requirements by adding/modifying their goals via our prototype tool. The query results are shown on the screen for review. When users are satisfied with the result, the files containing the final result OWL-S service are generated and added back to the ontology.

Currently, our planner prototype can only handle sequential compositions. When the

JTP queries the result for the service components that satisfy the user's goals, the prototype will populate the Process abstractComposedOf property only with the "sequence" control construct. Suppose the user requests a flight service (JFK-LAX) from New York (JFK) to Los Angles (LAX), for instance, the "JFK-ORD" and "ORD-LAX" services are the query result. We will add support for the other control constructs in the future.

4 Case study

In this section, we describe a case study of a dynamic flight scheduling service. Customers' requirements can be specified in terms of the goals, such as departure/arrival city pair, airline preference, and desired cabin. Our approach and prototype tool can assist combining existing web services to satisfy the customer's requirements.

Consider a customer who wants to schedule a business trip from New York (JFK) to Los Angles (LAX). He prefers to take only one stop during the trip. Suppose the scheduled flight cannot depart as expected due to some reason. The airline company normally does not re-schedule the flight, which means the traveler is not automatically put on another flight if his/her flight is canceled or delayed. Table 1 shows all available flight services on a particular day.

Table 1 Flight information

Flight No.	Company	Has business cabin	City pair	Departure time	Arrival time	Cost (US$)
AA301	American	yes	JFK-ORD	8:00 am	9:20 am	200
AA303	American	yes	JFK-ORD	9:00 am	10:20 am	200
AA501	American	no	ORD-LAS	3:00 pm	4:30 pm	300
AA503	American	yes	ORD-LAS	4:00 pm	5:30 pm	300
AA505	American	no	ORD-LAS	4:50 pm	6:20 pm	300
AA701	American	yes	LAS-LAX	6:00 pm	7:00 pm	200
AA703	American	no	LAS-LAX	6:30 pm	7:30 pm	200
DAL1001	Delta	yes	JFK-ORD	9:00 am	10:10 am	180
DAL2001	Delta	no	ORD-LAX	4:00 pm	6:20 pm	650
DAL3001	Delta	yes	LAS-LAX	4:00 pm	5:30 pm	200
DAL4001	Delta	no	LAS-LAX	3:00 pm	4:30 pm	200

Consider a simple ontology for all flight services. This sample ontology follows "Flight -> Airline Company / City pair / Available Cabin -> detailed category abstract flight services -> bottom level abstract flight services" criteria to make the hierarchy (see Fig. 11). All concrete flight services can be mapped to one or more abstract services. For example, AA301 is an instance of the abstract services "Business-JFKORD-AA-Flight", and in turn the instance of "JFKORD-Flight", "AA-Flight", "Business-Flight" and their corresponding parent classes. With our extension on the abstract service level, web service compositions can be conducted at the abstract service level, instead of accessing the huge number of concrete service instances. Based on the user's goals, the following JTP query entries can be entered in the Goal section of our prototype tool shown in Fig. 10.

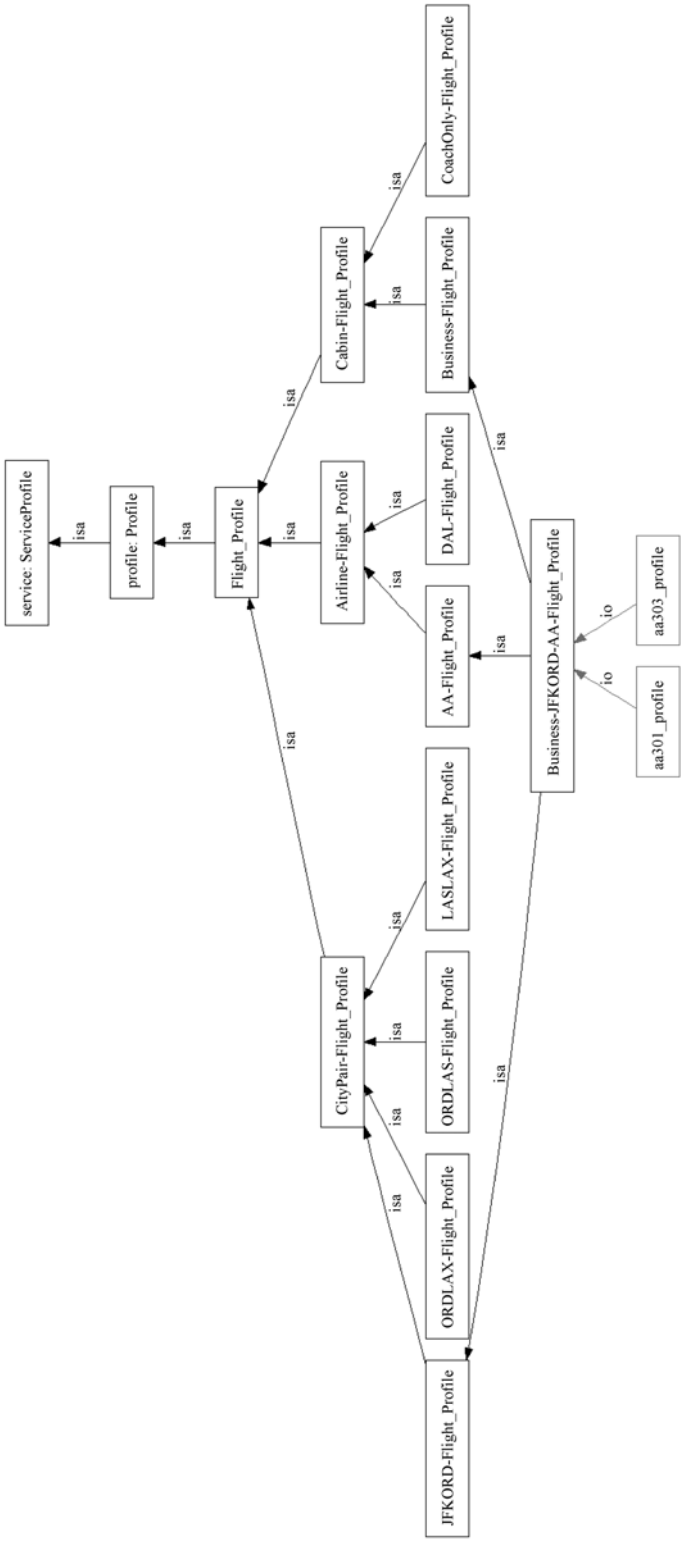

Fig. 11. Flight profile hierarchy.

Subject	Predicate	Object
- ?flight1	profile:hasInput	parameter:JFK
- ?flight1	profile:hasOutput	?tmpStopCity
- ?flight2	profile:hasInput	?tmpStopCity
- ?flight2	profile:hasOutput	parameter:LAX

The user's goals include both functional goals, such as the "hasInput" and "hasOutput" properties, and non-functional goals, such as choices of airline companies and cabin restrictions. The backend JTP inference engine responses only one solution at the abstract service level in this case. The "superInstance" in the result is a special instance for abstract service, which is used for inference purpose only. The following is the inference result.

```
    Bindings 1:
?tmpStopCity = |http://127.0.0.1:8080/Parameters.owl#|::|ORD|
?flight1 = |http://127.0.0.1:8080/Flight#|::|superInstance-jfkord-flight_profile|
?flight2 = |http://127.0.0.1:8080/Flight#|::|superInstance-ordlax-flight_profile|
```

Using our prototype tool, we get a solution containing two abstract flight services, "JFKORD-Flight_Service" and "ORDLAX-Flight_Service". Our prototype can generate the new composite service's Process class by adding those two abstract services in the "Sequence" control construct. "JFKORD-Flight_Service" has three concrete services in its Instance Pool (dal1001, aa301, aa303). "ORDLAX-Flight_Service" has "dal3001" and "dal4001" in its Instance Pool. In total, we get $3 \times 2 = 6$ concrete solutions.

Consider the customer chooses an itinerary of "aa301 --dal3001" initially. However, the flight "dal3001" is canceled when he/she arrives in Chicago. This is a typical real-time fail-over situation such that the customer cannot go back to New York to start a new trip. In our approach, the instance pool of the corresponding abstract service is searched first. Another possible substitution (dal4001) is found in this case to fail-over the problem.

Consider another case when the customer wants to switch to the business cabin in the ORD-LAX flight segment when he/she arrives in Chicago. The same airline company's service is also required to avoid checking in the baggage multiple times. This dynamic redesign requirement can be achieved by our planner with the following addition goals:

Subject	Predicate	Object
- ?flight1	profile:serviceParameter	?tmpAirline
- ?flight2	profile:serviceParameter	?tmpAirline
- typeOf	?tmpAirline	parameter:AirlineServiceParameter
-?flight2	profile:serviceParameter	parameter:businessCabin

Now the solution turns to be "Business-JFKORD-DAL-Flight_Service" followed by "Business-ORDLAX-DAL-Flight_Service". The corresponding concrete solution narrows to "dal1001" -> "dal3001".

5 Related work

A number of languages/frameworks have been developed based on the standard W3C web service language[7,8] to support web service composition. Among them, two major efforts are the BPEL4WS[1,2] and OWL-S[3,4] which define a standard for concrete composite web services. Our work is an extension of OWL-S at the abstract service level.

There are several different approaches in web service composition area. Rao *et al.*[9] proposed architecture for web service composition using the linear logic theorem proving. Both the service profile and customer goals are translated into the propositional linear logic and fed into the Jena planner[10]. The goal realization is based on the individual concrete web service. Similarly, Traverso *et al.* [11] used the EaGle language and the MBP Planner to generate composition result in π-calculus which was transformed to BPEL4WS. Mandell *et al.*[12] provided an automatic runtime discovery, composition and execution environment by integrating the BPEL4WS and OWL-S. These approaches focus on concrete web service composition; whereas our approach concentrates on the dynamic optimization and run-time fail over capability.

A transactional approach for web service composition is proposed in [13], where the accepted termination states are defined to allow the user to specify the required failure atomicity level. In contract, our approach focuses on fail-over and dynamic composition instead of failure atomicity.

WSMO (Web Service Modeling Ontology)[14] is another effort to address the semantic web services. WSMO relies on four core components: Ontology, Web Services, Goals and Mediators. There are several differences between the OWL-S and WSMO[15], e.g., separation of viewpoints of provider and requester in WSMO and explicit use of mediators to link the loose coupling core components. WSMO also describes similar concepts of OWL-S. For example, the OWL-S service profile can be expressed by the combination of the WSMO goal, the WSMO Web Service capability, and the Web Service non-functional properties[15]. Based on WSMO, several implementations like WSMX[16] and IRS-II[17] provide an execution environment for semantic web service, which enable the service registration, client discovery and invocation. The service compositions in WSMO, however, are still at the concrete service level. In contrast, our approach focuses on abstract level for fail-over and dynamic optimization.

6 Conclusions

This paper has proposed an extension to the OWL-S ontology framework for dynamic web service composition at abstract service level. Rooted from OWL-S, our approach inherits the capability of semantic clarity. New abstract service concept is extended to the OWL-S. Each abstract service has an instance pool. Our planner prototype can take the user's goals and the service ontology and feed them into the backend inference engine to generate the results that are abstract services instead of concrete services. Each of the resulting abstract services has an instance pool of all possible concrete service solutions. A case study illustrates the runtime fail-over and dynamic redesign using our approach. Our planner prototype is based on the embedded JTP, which has been developed for the

demonstration purpose.

In the future, we aim to continue to enhance the OWL-S ontology framework to support more complex optimization. We also plan to explore more case studies related to both the abstract level and instance level semantic constraints, and the user's goals with "must have" and "good to have" semantic constraints. Furthermore, we intend to integrate our composer prototype with an existing ontology server.

References

1 Andrews T, Curbera F, Oholakia H, et al. Business Process Execution Language for Web Services (BPEL4WS) 1.1. Online: http://www-106.ibm.com/developerworks/webservices/library/ws-bpel, May 2003

2 IBM. BPWS4J. http://www.alphaWorks.ibm.com/tech/bpws4j

3 DAML Services Coalition (alphabetically Ankolekar A, Burstein M, Hobbs J, et al.). DAML-S: Web service description for the semantic web. In: Proceedings of the International Semantic Web Conference (ISWC), Sardinia, Italy, Springer, 2002. 348－363

4 Dean M, Connolly D, Harmelen F, et al. OWL Web Ontology Language 1.0 Reference. http://www.w3.org/TR/2002/WD-owl-ref-20020729/

5 Stanford K S L. JTP. http://www.ksl.stanford.edu

6 Hendler J, McGuinness D. The DARPA Agent Markup Language. IEEE Intelligent Systems, Tends and Controversies, pp. 6-7, November/December 2000

7 Box D, et al. Simple Object Access Protocol (SOAP) 1.1. Online: http://www.w3.org/TR/SOAP/, 2001

8 Chinnici R, et al. Web Services Description Language (WSDL) 1.2. Online: http://www.w3.org/TR/wsdl/

9 Rao J, Kungas P, Matskin M. Application of linear logic to web service composition. The First International Conference on Web Services, Las Vegas, USA, June 2003. CSREA Press.

10 Jena - semantic web framework for java. Online: http://jena.sourceforge.net

11 Traverso P, Pistore M. Automated Composition of Semantic Web Services into Executable Processes. Technical report. # T04-06-08. Istituto Trentino di Cultura, 2004

12 Mandell D, McIlraith S. Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation. In: Proceedings of the Second International Semantic Web Conference (ISWC2003), Sanibel Island, Florida, Springer, 2003

13 Sami B, Claude G, Olivier P. Reliable Web services composition using a transactional approach. In: Proceedings of the IEEE International Conference on e-Technology, e-Commerce and e-Service, Hong Kong, IEEE, March 2005

14 Feier C, Roman D, Polleres A, et al. Towards Intelligent Web Service Modeling Ontology (WSMO). In: Proceedings of the International Conference on Intelligent Computing (ICIC) 2005, Hefei, China, August, 2005

15 Lara R, Roman D, Pollers A, et al. A conceptual comparison of WSMO and OWL-S. European Conference on Web Services (ECOWS 2004), Erfurt, Germany, September, 2004. 254－269.

16 Haller A, Cimpian E, Mocan A, et al. WSMX - a semantic service-oriented architecture. In: Proceedings of the International Conference on Web Services (ICWS 2005), Orlando, Florida (USA), July 2005

17 Motta E, Somingue J, Cabral L, et al. IRS-II: A framework and infrastructure for semantic web services. In: The Semantic Web – ISWC 2003, 2003. 306－318