**RC23808 (W0511-143) November 18, 2005**
**Computer Science**

# IBM Research Report

## Dynamic Discovery and Binding of Web Services to Abstract Web Process Flows

**Rama Akkiraju[1], John Colgrave[2], Kunal Verma[3], Prashant Doshi[3], Richard Goodwin[1]**

[1]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

[2]IBM United Kingdom Limited
Hursley Park
Winchester, United Kingdom

[3]Department of Computer Science
University of Georgia
415 Boyd GSRC
Athens, GA 30602

**Research Division**
**Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich**

# Dynamic Discovery and Binding of Web Services to Abstract Web Process Flows

Rama Akkiraju[1], John Colgrave[2], Kunal Verma[3], Prashant Doshi[3], Richard Goodwin[1]

[1]IBM T. J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532

[2]IBM United Kingdom Limited, Hursley Park, Winchester, United Kingdom

[3] Department of Computer Science, University of Georgia, 415 Boyd GSRC, Athens, Georgia, GA 30602

{akkiraju,rgoodwin}@us.ibm.com

colgrave@uk.ibm.com

verma@cs.uga.edu

pdoshi@cs.uga.edu

**Abstract**

Translating high-level business process flows--created by business analysts--to executable flows is often manual and time consuming. Service oriented architectures, enabled by Web services, show promise in enabling a more automatic translation process. While some work has been done to address this gap, the prior work does not provide flexible ways of discovering services and does not accommodate the inter-service dependencies that might exist while binding services to a high-level workflow process. In this paper, we present a novel approach to dynamically discover suitable services from a UDDI registry and to automatically bind them to abstract business process flows represented in BPEL. Our contributions are two fold: First, we present a way to dynamically bind services to abstract business process flows while considering inter-service dependencies and domain constraints. Second, we present a flexible mechanism to enhance UDDI's service discovery function. Using our approach, users can integrate multiple external matching services with a UDDI registry to support multiple external service description languages. The result is a system that enables business analysts to focus on creating appropriate high-level flows, while providing application developers with the tools required to translate these high-level flows into executable flows.

## 1. Introduction

Modeling and communicating business processes accurately and at appropriate levels of detail is critical to the success of integration projects. Current practice in the industry is to have business analysts model high level processes by interviewing relevant client personnel. The job of application developer team is to translate these high level processes into executable modules in preparation for deployment and execution. Typically, they do this by mapping the high level process models and integrating them with appropriate available services offered by service providers[1]. However, the translation of these high level business models to executable models is often manual and time consuming. This necessitates effective tools and techniques that can bridge the gap between the two models and assist application developers in their ability to implement and execute a business process.

Technologically, two factors have to be in place for this to happen. First, services should be able to express their capabilities and requirements semantically. Second, tools should be able to use this semantic information to match and compose suitable services to meet given service requirements. This dynamic discovery process forms the

foundation for automating the translation of high-level flows to executable flows. Below, we examine the technological foundations that are already in place and what additions are required to make this happen.

Industry efforts to standardize representation mechanisms for web service description, web process description, and service discovery have led to standards such as WSDL (Christenson et al., 2001), BPEL (BPEL T.C., 2002), and UDDI (UDDI T.C, 2002) respectively. However, they all lack semantic expressivity.

WSDL describes the interface of a service, and how to invoke it. BPEL provides a representation mechanism for specifying process execution flow. However, both WSDL and BPEL lack the expressivity required to represent the service capabilities, requirements and the context in which a service operates. UDDI provides directory services for Web services offered by the businesses. Unfortunately, the current UDDI specification (V2.0/3.0) suffers from the same semantic limitations as those of BPEL and WSDL. The current search functions in UDDI are limited to key-word based search for services and are insufficient for making automatic service selection decisions because it does not provide search based on service capability matching. By capturing the capabilities of services semantics can play a crucial role in enabling automation. The semantic Web community has already been working on this for sometime now.

Recently, the semantic Web community has developed an ontology markup language - OWL (OWL T.C, 2002). To address the lack of semantics in the industry-backed Web Services standards, this community developed an OWL ontology for Web Services known as OWL-S (OWL-S T.C, 2003). This OWL family of semantic markup languages together lays the foundation for automatic service discovery, and service composition (McIlraith et al., 2001). Some work has already been done to perform dynamic discovery and to bind services to processes automatically. However, these prior works do not provide flexible ways of discovering services and do not accommodate the inter-service dependencies that might exist while binding services to a high-level workflow process.

In this paper, we present a novel approach to discover services in a UDDI registry and to dynamically bind suitable services to abstract business process flows. We achieve this by providing semantic enhancements to a UDDI registry and use semantic modeling and matching techniques to compose services as part of service discovery. When integrated with a development environment, our technology can assist developers in transforming the high level process models created by business analysts to executable models thereby potentially saving valuable development

[1] We assume a service-oriented architecture (SOA) throughout this paper where software components are available as Web services.

time and reducing the overall business process integration time and implementation costs. Specifically, the contributions of our work are two fold.

First, we present a new design for enhancing the service discovery capabilities in a UDDI registry. Specifically, we provide an *external matching feature* in a UDDI registry that allows external matching services to play a role in the matching of UDDI entities against criteria supplied by the user. Using this new external matching feature:

- Service providers and requesters can publish the location of external descriptions of their service capabilities and service requirements respectively in a UDDI registry.
- Requesters can indicate that they would like external description matching to be performed for their requests by the UDDI registry.
- Third-party service providers can publish their matching engines as Web services in the UDDI registry.
- The UDDI registry can select suitable external matching services and dynamically invoke the selected matching service to carry out external description matching of compatible services against the requesters' requirements, which are also specified as external descriptions.

Our approach offers several advantages over the alternatives suggested in earlier works. First, it allows multiple external matching services developed by independent service providers to be integrated with a UDDI registry. For example, external-matching engines can be provided that can match descriptions written not only in OWL-S (OWL-S T.C, 2003) but also in other standard languages such as UML (UML T.C, 2003) and WSDL. Also, there can be more than one matching engine for each supported description language. This enables UDDI to offer best-of-the-breed search choice in finding services. Second, it allows for many types of matching. For example, requesters may use this approach to request not only semantic matching but also WSDL-based syntactic matching. By integrating multiple kinds of matching engines with UDDI, while allowing the service descriptions and the matching services to reside externally, we offer the much needed intelligent matching of Web services in UDDI.

Second, we argue that dynamic selection of individual Web services and binding them to process activities is often not a stand-alone operation. There may be many inter-service dependencies and domain constraints that need to be considered in selecting legal and appropriate services for realizing an abstract flow. For example, a process flow in which a document is encrypted using the services of a 512-bit encryption algorithm at one step might need to ensure that there exists a compatible service that can decrypt the document in a subsequent step.

Therefore, representing and accommodating inter-service constraints is crucial to the selection of a consistent set of service bindings when executing an abstract flow. To account for this, we provide a way of modeling and accommodating domain constraints and inter-service dependencies within a process flow by introducing the concept of 'scope' and binding all the services within a 'scope' at once.

Finally, we combine our two contributions in an overall architecture wherein abstract BPEL flows augmented with semantic annotations in OWL-S are automatically translated into executable flows via runtime discovery, composition, and service binding.

## 2. A Motivating Scenario

We consider a purchase order scenario. Say that an analyst captures the procurement process of a company in three high-level steps: (1) *lookup preferred suppliers* (2) *check item availability* with preferred suppliers and (3) *place purchase order*. The analyst then hands these models to application developers. The job of application developers is to find suitable services offered by preferred suppliers and to interface with their services in order to create executable models. In the case of large companies, the preferred supplier database could consist of thousands of suppliers. Compounding the problem is the fact that the interfaces provided by these suppliers are not standardized. One supplier calls their service *checkInventory()* while the other calls it *findItemAvailability()*. Moreover, the terminology used to describe the inputs and outputs could be different. While one service takes a *DueDate*, the other calls it a *DeliveryDate*. Also, sometimes, some services require specific information while others accept generic codes. For example, in retail industry, while one service requires a *UPCCode*, the other might require a *EANCode* (it turns out that a *UPCCode* can be passed in place of *EANCode* and that EAN scanners can parse *UPCCod*es since *UPCCode* is a subset of *EANCode*). If the goal is to create a single process to deal with procurement, in the case of a large company, that would mean creating a process that branches out to each preferred supplier's item availability interface. A better way to do this would be to have the system dynamically discover and rationalize these interface differences and have an adaptive process that can deal with these types of differences.

Also, high level processes may need further expansion before they can be executed. For example, *place purchase order* step could be expanded into the following sub-processes by the application developer if she discovers that the chosen service provider (say ABC Inc.) requires the parameters to be digitally signed and

encrypted : (a) sign parameters using the *signDocumentsDigitally()* service offered by the provider SecureDoc[2] (b) encrypt the parameters using the *encryptDocuments()* service offered by the provider SecureDoc (c) invoke *placePurchaseOrder()* service offered by the provider ABC Inc. by passing the signed and encrypted arguments. This process is shown in figure 1. In the absence of automatic service interface matching and binding technology, an application developer would have to manually sift through these differences in interfaces, select suitable services and bind them to the high-level processes to generate executable processes.

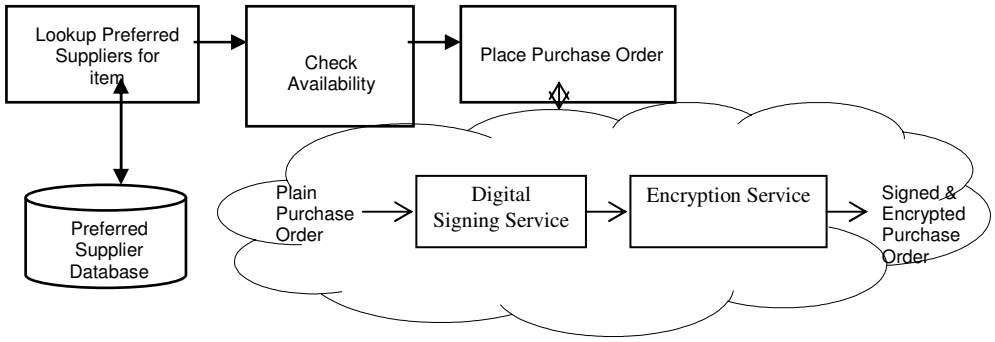


Figure 1: A Purchase order scenario process flow.

## 3. Overview of our Approach

In this section, we explain the details of our solution approach by referencing the purchase order scenario discussed in Section 2. In our design, first, an analyst creates a model of the domain in which a business process operates. For example, in the case of the purchase order scenario, analyst models concepts such as order item, delivery date, purchase order, digitally signed purchase order, encrypted purchase order, and customer profile etc. and also models the constraints on these concepts such as 'a document would have to be signed before it can be encrypted' etc. In our implementation, we use OWL to represent these domain models and constraints.

Second, an analyst models the high level process activities by describing their semantic requirements. For instance, the three high level activities namely *lookup preferred supplier*, *check item availability*, and *place purchase order* activities are described as OWL-S services with semantic descriptions of the specific requirements that are expected of services providing these functions. The semantic requirements of each activity are modeled using inputs, outputs, preconditions and effects (IOPEs in OWL-S terminology). These IOPEs reference the semantic concepts defined in the domain model represented as a (a set of) OWL document(s). It is to be noted that at this point only the expected behavior (effects) and conceptual inputs and outputs and constraints (preconditions) are

[2] SecureDoc, and ABC are fictional suppliers.

modeled by an analyst. These are not the low level interface descriptions of Web services as described by WSDL documents. Therefore, an analyst is not burdened to create low level Web services at this level of modeling.

Next, independent of the above steps, service providers (in our purchase order scenario these would be preferred suppliers such as ABC, and SecureDoc Inc.) prepare digital signing service, encryption service, check item availability and place purchase order services along with the semantic annotations (in OWL-S) and publish these services in a UDDI registry. These descriptions are later used in the selection of suitable services for a given set of requirements. The corresponding WSDL descriptions of these services are used for invoking the actual Web services. More details about the process of publishing and searching for suitable semantically annotated Web services is discussed in the 'semantic service discovery' section of the paper.

Finally, an analyst creates a process flow consisting of the three steps mentioned above by providing semantic descriptions of required behavior for each activity in the flow. We envision analysts interacting with a visual modeling tool to perform these three tasks.

The key components of our architecture are shown in Figure 2. They are: a Generic Web Service Proxy, A semantic UDDI module, a matching module, a dynamic binding and invocation module. In the following subsections we will elaborate on the workings of each of these components.

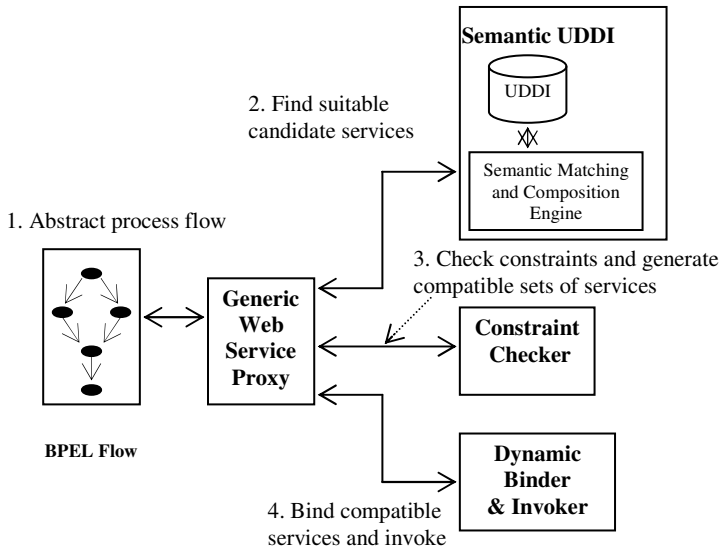


Figure 2: Interaction flow between abstract process flow and our dynamic service binder.

### 3.1 Accommodating Service Constraints and the Notion of Scope

The process starts with the creation of an abstract BPEL document by the business analyst. An abstract BPEL flow is divided into a set of unit scopes. This notion of a unit of scope indicates that activities within the

scope might have interdependencies and that service selection and binding should be done as an atomic operation. For instance, the technology of one service provider might be incompatible with that of another even though the capabilities of both of them match with those of requirements. In multi-item purchase order requests, it is possible to procure parts from multiple suppliers. In cases where there are technology constraints such as 'supplier A's items are incompatible with supplier B's items' it would be appropriate to bind all these services as a bundle, if such bindings can be found, to ensure that the service bindings generated are legal and compatible. It is to be noted that in this example, the individual service that offers a certain part by itself does not have any constraints. Therefore, it would be inappropriate to specify these dependencies as preconditions or effects of services. The inter-service dependencies become apparent because of the need for these services to work with other services in the context of a specific request. For example, in the electronics parts example, the fact that a distributor would like to purchase a compatible set of power cord, battery and network adapters is independent of the individual capabilities of each service. We represent information about the compatibility of various parts in this example in the domain model. An OWL excerpt of that domain ontology is given in the Constraint Checker section. To represent this kind of inter-service dependencies, we use the notion of scope in BPEL documents. In our approach, scoping is a way of defining a manageable search space for finding compatible services. Since humans possess the inherent capability to group related things in a given problem domain, we rely on business analysts to tell us the boundaries of scopes via abstract flow definitions. In essence, these abstract flows hide the details of activities within a scope. We bind a Generic Web service Proxy to each unit scope thus defined in the high-level BPEL process flow document. The Generic Web Service Proxy is a Web service defined via a WSDL document that can be statically bound to a node in the BPEL flow. We use this proxy to defer specifying the execution details of the activities within a unit of scope. We then deploy this high-level BPEL document in BPWS4J, IBM's BPEL execution engine.

A sample abstract BPEL4WS excerpt for electronics parts example is shown in Figure 3. The process consists of procuring three electronic parts – say batteries, power cords and network adapters. The flow is setup such that three supplier services have to be invoked to procure all the parts – one for procuring batteries, second for procuring power cords and the third for procuring network adapters. Therefore, at the time of setting up the BPEL process, we know the number of suppliers but we don't know who they are. This process of finding the suitable supplier that can supply a given part is framed as a semantic discovery problem wherein the request is modeled as an abstract web service with semantic descriptions. These semantic descriptions are used to discover a suitable service

which is then bound to its corresponding abstract service. However, since it is possible that the selection of a service of one service provider may interact with the others (example: supplier A's power cords don't work with supplier B's batteries), we use the notion of a scope information document to bind all the services that may have interdependencies among them at once to avoid these inconsistencies. Specific details are discussed below.

In the first step, the retailer's purchase order request is received by the distributor's order processing Web service. This is depicted as the *receive* activity in BPEL. The received purchase order data, which is stored in the variable *PurchaseOrderInputs*, is then sent to an order processing service of the distributor by invoking the operation *orderHandler*. This call to the order process service is crucial, as it decides the number of suppliers, part quantities and constraints among the services. The output of the distributor service *OrderDetails* contains the following three things:

1. Scope information document (discussed in detail later): This forms the basis for discovery of suppliers for the proxy.
2. Number of suppliers: This is used to loop over the number of suppliers and is used as a guard in the while condition.
3. Inputs for the suppliers: Used by the Generic Web Service Proxy for invoking the services offered by the suppliers. This includes part identification number, quantity requested, date of delivery, total price etc.

The *while* construct of BPEL is used to loop through each order item and to source the requested items from preferred suppliers via the proxy. Here, each order item can be procured from a different supplier. However, there might be domain constraints that make only certain service combinations valid. For example, as mentioned earlier, one supplier's items may be incompatible with other supplier's items because they use different standards. Therefore, it is important to apply those constraints at once and bind these services at once. To accomplish this, the proxy service takes the available inputs and the expected outputs from the scope information document and finds legal and suitable service sets (combinations) for binding. It must be noted that all the depedent services indicated via the scope information document are bound at once by the proxy service, Finally, a confirmation is sent back to the retailer via the *reply* activity..

```
<sequence>
        <receive partnerLink="Retailer">
        portType="order" operation="receiveOrder"
        Variable="PurchaseOrderInputs"  / >
```

```
<invoke partnerLink="Distributor">
portType="orderHandler "operation="orderHandler"
InputVariable="PurchaseOrderInputs"
OutputVariable="OrderDetails"  / >
<flow>
    <while condition= "bpws:getVariableData('supplierCounter') <
    bpws:getVariableData"(OrderDetails,numSuppliers)>
    <sequence>
        <invoke partnerLink="GenericWebServiceProxy">
portType="scopeHandler"operation="invokeProxy" inputVariable="OrderDetails"
outputVariable="OrderConfirmationOutput" / >
<assign>
    <copy>
        <from expression= "bpws:getVariableData('supplierCounter'+ '1' />
        <to variable="supplierCounter"/>
     </copy>
</assign>
  </sequence>
</while>
 </flow>
<reply partnerLink="Retailer">
portType="order" operation="receiveOrder"
Variable="OrderConfirmation"  / >
</sequence>
```

Figure 3: BPEL Process for Purchase Order Scenario

An example scope information document is shown in Figure 4. For each scope (one for this scenario), a scope information document must be created with the following information:

- the semantic descriptions of the service requirements represented in OWL-S ( SemSpecsURI tag in Figure 4). These are fed to the service discovery module for obtaining the partner services based on their semantic descriptions.
- domain constraints or service dependency constraints represented in OWL (ConstraintsURI tag in Figure 4). These are fed to constraint analysis module discussed in the next section.
- the location of public or private UDDI registries to find suitable matches (fed to the discovery module).

The scope information document can be extended to capture the requirements of any number of suppliers.

Details of how the Generic Web Service Proxy finds suitable services from a Web services registry are described in the Semantic Service Discovery section. Details of partner selection in the presence of domain constraints are given in the Constraint Checker subsection.

```
<Partners scope = 0>
<Partner id = 1>
<SemSpecsURI>
        http://localhost/owls/RequestElectronicParts.owl
</SemSpecsURI>
<ConstraintsURI>
        http://localhost/ontologies/electronic_parts.owl
</ConstraintsURI>
<UDDISpecs>
        <RequestTModel>"Specify UUID for the request TModel" </RequestTModel>
        <CategoryName> 'Electronic Components and Supplies' </CategoryName>
        <CategoryValue> 32.11.17.00.00 </CategoryValue>
</UDDISpecs>
</Partner>
</Partners>
```

Figure 4: A sample scope information document

As mentioned, the Generic Web Service Proxy module takes the following as inputs: the semantic descriptions of the service requirements represented in OWL-S, domain constraints or service dependency constraints represented in OWL, and the location of public or private UDDI registries to find suitable matches. At a high-level, the Generic Web Service Proxy discovers suitable services, automatically binds feasible sets and invokes them, and returns control to the upper BPEL flow. BPWS4J engine then proceeds with the execution of the remaining steps of the flow. In the following sections we describe the modules that the Generic Web Service Proxy interacts with.

### 3.2 Semantic Matching and Composition Engine

When the Generic Web Service Proxy is invoked, it sends a request to the UDDI registry to find services that match the requirements specified in its inputs. This is done first by sending a *find_TModel()* request to UDDI registry. Our modified *find_Tmodel()* method implementation in UDDI registry (explained in detail in Semantic Service Discovery section) retrieves a set of candidate services that are described in OWL-S and those that are advertised under related industry taxonomies. For example, in the electronic parts scenario, all the supplier service description tModels that are registered under a UNSPSC category known as 'Electronic Components and Supplies' are

retrieved. The UDDI registry then invokes a OWL-S semantic matching engine to perform matching between the capabilities of services that are retrieved in the previous step and requirements of those that are specified at a given node. Our semantic matching and composition engine is capable of finding simple services as well as compositions of sets of services that together match the given requirements[3]. These matching sets of services are then passed back to the Generic Web Service Proxy which invokes the Constraint Checker module to select compatible sets that meet the specified constraints.

### 3.3 Constraint Checker

The Constraint Checker module (see Fig. 2) takes the set of suitable services selected from the previous step for each node in the flow, the domain or service constraints and creates feasible/compatible sets of services ready for binding[4]. It uses SNoBASE- the semantic network based ontology management system (Lee et al 2003) to accommodate domain constraints and any interdependencies among services.

We illustrate *domain constraints* with an example. We consider an electronic parts supply chain, where a distributor has captured the relationships between electronic items such as network adapters, power cords, batteries, their corresponding technologies such as network type, voltage input/output specs, Lithium-Ion (Li-Ion) battery vs. Nickel Cadmium battery (Ni-Cad). In addition, the distributor's preferred suppliers and their technology constraints are also captured in the ontology. For example, the following OWL statements capture facts such as "Type1B is an instance of Battery", and constraints such as "Type1B works with Type2 Power Cords", "Type1B works with Type 3 Power cords" and "Type 1B works with Type 2 Network adapters".

```
<rdf:Description rdf:about="<ontologyPath>#Type1B">
<rdf:type>
        <owl:Class rdf:about="<ontologyPath>#Battery" />
</rdf:type>
<ns0:worksWith rdf:resource="<ontologyPath>#Type2PCh" />
<ns0:worksWith rdf:resource="<ontologyPath>#Type3PCh" />
<ns0:worksWith rdf:resource="<ontologyPath>#Type2NWA" />
</rdf:Description>
```

Figure 5: Domain Constraints in OWL ontology

[3] However, for now we only return simple service matches. Returning complex services from a UDDI registry results in service life cycle management issues that our design does not accommodate at the moment. This is discussed further in section 7.

[4] Depending on the efficiency requirements, one might consider first generating a set of compatible set of services (and their providers) that meet the constraints and then perform semantic matching on these. In our prototype system we have chosen to first match interfaces and then pass the matching services through constraint checker.

Based on the domain constraints, we consider an order procurement process for the distributor, where the distributor only wants to choose suppliers whose parts are compatible. We describe such *compatibility constraints* in DQL and are given below in Figure 6.

```
(Type1B rdf:Type <ontologyPath>/#Battery)
(Type1B ns0:worksWith ?X)
(?X rdf:type <ontologyPath>/#PowerCord)
(?X ns0:worksWith ?Y)
(?Y rdf:type <ontologyPath>/#NetworkAdapter)
(?P rdf:type <ontologyPath>/#PowerCordSupplier)
(?P ns0:supplies ?X)
(?N rdf:type <ontologyPath>/#NetworkAdapterSupplier)
(?N ns0:supplies ?Y)
```

Figure 6: Compatibility Constraints in DQL

Based on the constraints given above, if a suitable battery supplier is identified then the subsequent compatible power cord supplier and network adapter sets can be obtained (it is to be noted that the statements form conjunctions by default. <ontologyPath> refers to the location of the ontology file on host system. For example, in this case it refers to: http://localhost/ontologies/electronic_parts.owl file). We provide a generic interface to perform the constraint checking. Many heuristic approaches can be implemented for generating feasible sets of services. For the above example, we utilized a greedy algorithm, which will return the first compatible set. Various selection criteria such as cost, time, and quality can be employed for choosing a compatible set from the possibly many that get generated. The chosen compatible set of services is then passed to the dynamic binder and invoker module.

### 3.4 Dynamic Binder and Invoker

The dynamic binder module takes the generated flows and services and binds them to the corresponding abstract nodes in the BPEL flow and invokes them. Binding in this context is simply replacing the high level service interface specifications at each node in the BPEL document with a specific service instance that has been selected via semantic matching. Selection of appropriate services can be achieved by considering the cost, quality, and other ratings of service providers. As mentioned earlier, we have implemented a simple scheme where the first compatible set gets chosen for binding. In this system service invocation is not automated. To automatically invoke services from a client program, work is required to identify specific input parameters, their types, their sequence and to map

the data elements from client program with those required by the services to be invoked. In a more recent work, we have extended our system to accomplish this data mapping and matching. The system generates a mapping specification language (MSL) – a subset of XSLT as a result of the data mapping (Syeda-Mahmood et. al 2005). Using this xml, the system generates mediation glue code which when compiled and deployed can be used in automatically invoking the services.

### 3.5 Semantic Service Discovery

As mentioned earlier, service discovery is achieved via the semantic UDDI and OWL-S semantic matching and composition engine modules. In this section, we present a new design for providing an *external matching feature* in a UDDI registry that can better support any external matching service within UDDI, including semantic matching.

There are *four* role players in our approach: (I) service providers (II) service requesters (III) *matching service providers* and a (IV) UDDI registry. First, service providers, requesters and matching service providers annotate their service capabilities, requirements and matching service capabilities respectively with external descriptions. These descriptions can be in any format or markup language including OWL-S, UML or XML (The role players in this case would publish their capabilities and requirements as OWL-S external description tModels). Requesters would then formulate find_tModel() requests based on the new 'DescribedUsing' categorizations (explained later) to indicate to the UDDI registry that external matching is required. Figure 3 shows how external description matching in UDDI works. UDDI interprets these requests, and processes them. Processing is done in two stages. In the first stage, standard categorizations (such as industry classifications and external description format) specified in the request are applied to obtain all descriptions of services that are described in the requested format/style (eg: OWL-S/UML etc) under the given categories. This process serves as a filter and retrieves only relevant external description tModels for further processing.

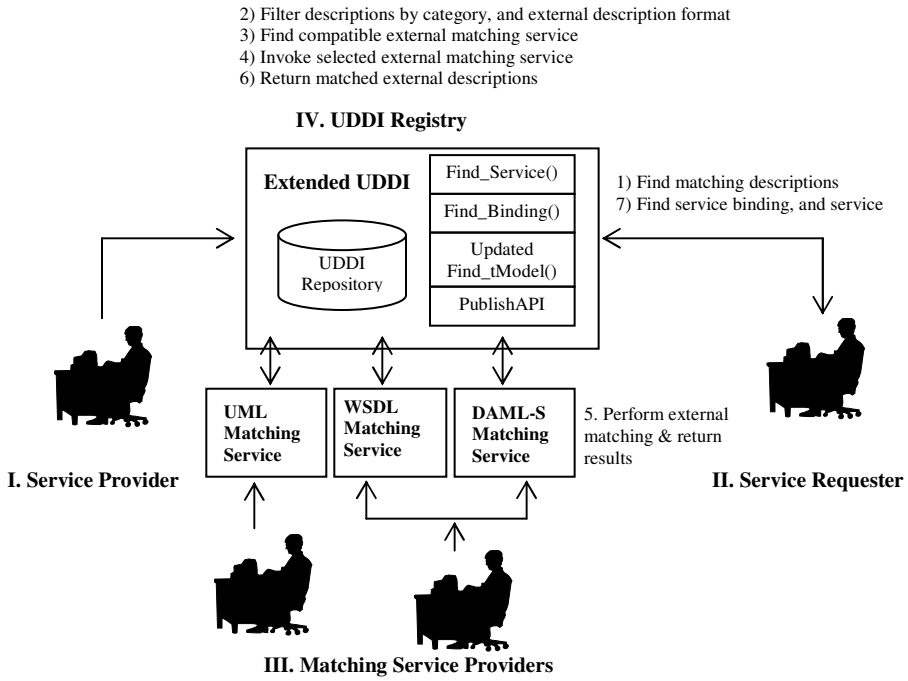


Figure 3. External description matching in UDDI

Then, the registry looks for available and compatible external matching services and chooses the appropriate external matching service. A discussion on the selection policies that can be applied to choose an external matching service are presented in 'selecting external matching service' section. The external descriptions obtained from the filtering stage are passed as inputs in a URL form to the external matching service along with the original descriptions of service requirements also as a URL. This is made possible via a standard matching service interface that we define. Provision to pass multiple service requirements and their corresponding external descriptions is also possible via a similar standardized matching service interface. Upon receiving a matching request, the chosen external matching service performs its matching and returns a list of matched URLs in a ranked order. The corresponding tModels are returned to the requester by the UDDI registry as a response to the *find_tModel()* invocation. Requesters can then proceed with *find_service()* and *find_binding()* invocations just as they would in regular UDDI find process to obtain specifics of binding templates to invoke the chosen services.

Our design consists of five elements. First, we provide a way to capture the external descriptions of service capabilities and requirements in UDDI. Second, to facilitate the dynamic invocation of external matching service(s) by a UDDI registry, we provide a standard interface for the matching engines. Third, we provide a mechanism to enable requesters to communicate to a UDDI registry that they would like external matching to be performed on

their requests. Next, we present a design for a UDDI registry to detect that an external match is necessary for a given request and to invoke compatible external matching services to obtain matching external descriptions. Finally, we present the steps that a requester must go through to find suitable services.

#### *3.5.1 Referencing external descriptions*

The provision in the UDDI data structures to refer to external information is limited to the overviewDoc element that appears in the following places in the UDDI v2 data structures:

1. the instanceDetails element that is part of the tModelInstanceInfo element
2. the tModel

The overviewDoc element consists of zero or more descriptions and an optional overviewURL. We make use of this overviewURL to refer to the external descriptions of services and requests. It is common for the overviewURL to be dereferenced using HTTP GET.

In the two data-structures that support capturing of external descriptions via overviewURLs, TModels are recommended for those instances of external descriptions that service providers/requesters would like to share with others (such as standardized representations). TModelInstanceInfos, on the other hand, are recommended for representing those instances of external descriptions that are specific to a given service/request. In this work, we have chosen the TModel approach (apart from keeping things simple, using Tmodels for external descriptions makes the inquiry API interfaces simple. It is also consistent with how WSDL definitions are published currently).

An external description tModel should be categorized with the appropriate tModel to indicate the type of external description, such as OWL-S. For doing this, we have introduced a new categorization, the DescribedUsing categorization (in this document, for ease of explanation, we use a convention such as XXXX…, YYYY…, CCCC…etc. for tModel Keys of tModels that we designed). It is as follows:

```
<tModel tModelKey="uuid:XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX">
 <name>urn:x-ibm:DescribedUsing</name>
 <description xml:lang="en">Used to categorise a tModel by/with a particular external description type/format.</description>
 <overviewDoc>
  <overviewURL>…</overviewURL>
 </overviewDoc>
 <categoryBag>
  <keyedReference tModelKey="uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4"
   keyName=”types” keyValue="categorization"/>
```

```
  </categoryBag>
</tModel>
```

The valid values of this categorisation are the keys of tModels that are categorized with keyName="externalDescription" and keyValue="true" using the standard UDDI general_keywords category system. Based on this design, all external descriptions of services and requests are published as tModels in UDDI with their overviewURLs pointing to the location of the corresponding external descriptions. A sample service capability tModel that uses this categorization is given below.

```
<tModel tModelKey="UUID:SSSSSSSS-SSSS-SSSS-SSSS-SSSSSSSSSSSS">
  <name>urn:x-ibm:servicename </name>
  <description xml:lang="en">desc.</description>
  <overviewDoc>
    <overviewURL> URL here</overviewURL>
  </overviewDoc>
  <!-- Categorize this tModel under DAML-S external description category.-->
<categoryBag>
    <keyedReference tModelKey="UUID: XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX"
                keyName="urn:x-ibm:DescribedUsing " keyValue="UUID: DDDDDDDD-DDDD-DDDD-DDDD-DDDD-
DDDDDDDDDDDD"/>
<!-- A categorization to indicate that this tModel describes service capabilities. →
<keyedReference tModelKey=" uuid:A035A07C-F362-44DD-8F95-E2B134BF43B4"
            keyName="urn:x-ibm:type"
            keyValue="capabilities"/>
  </categoryBag>
</tModel>
```

### *3.5.2 Registering external matching services*

A key objective of this design is to enable external service providers to publish their matching services in UDDI registries. For example, company X can publish its matching service that can match service descriptions that are represented in DAML-S and company Y can publish its matching service that match service descriptions that are represented in UML and so on. Also, there could be multiple matching services for the same description format e.g.: DAML-S. The result of this is that the UDDI registry will have a choice in selecting matching services in matching external descriptions. To facilitate dynamic invocation of external matching service(s) by a UDDI registry, we provide a standard interface for the matching engines. Each external matching service defined in a UDDI registry requires the following:

1. A businessEntity/businessService/bindingTemplate giving the endpoint of the external matching service.

2. A tModelInstanceInfo for each type of external description supported. A particular external matching service may support more than one format of description, and there may be more than one matching service that supports a particular format of description.
3. A tModelInstanceInfo for the tModel that represents the interface between the UDDI registry and the external matching service.

A WSDL port type shown below illustrates our standardized interface.

```
<message name="performMatchingRequest">
    <part name="requestDescriptionURL" type="xsd:string"/>
    <part name="candidateDescriptions" type="uddiext:OverviewURLBag"/>
  </message>
  <message name="performMatchingResponse">
    <part name="matchingDescriptions" type="uddiext:OverviewURLBag"/>
  </message>
  <portType name="ExternalMatchingService">
    <operation name="performMatching" parameterOrder=" requestDescriptionURL candidateDescriptions ">
      <input message="tns:performMatchingRequest" name="performMatchingRequest"/>
      <output message="tns:performMatchingResponse" name="performMatchingResponse"/>
    </operation>
  </portType>
```

To support this interface, we introduce a new datastructure called 'overviewURLBag' as shown below.

```
<xsd:element name="overviewURLBag" type="urn:x-ibm:overviewURLBag"/>
 <xsd:complexType name="overviewURLBag">
  <xsd:sequence>
   <xsd:element ref="uddi:overviewURL" maxOccurs="unbounded" />
  </xsd:sequence>
</xsd:complexType>
```

A tModel that represents the interface to an external matching service that refers to a complete version of this WSDL could be as follows.

```
<tModel tModelKey="uuid:ZZZZZZZZ-ZZZZ-ZZZZ-ZZZZ-ZZZZZZZZZZZZ">
 <name>urn:x-ibm:ExternalMatchingService</name>
 <description xml:lang="en">Represents the interface to an external matching service.</description>
 <overviewDoc>
```

```
  <overviewURL>…</overviewURL>
 </overviewDoc>
 <categoryBag>
  <keyedReference tModelKey="uuid:A035A07C-F362-44DD-8F95-E2B134BF43B4"
   keyName=”externalDescription” keyValue="true"/>
 </categoryBag>
</tModel>
```

A service provider that provides external matching services would follow the following steps to create a Web service for their matching services and publish them as external matching services in UDDI.

1. Create a Web Service for the matching service using the skeletal WSDL given above.
2. Publish the Web Service in UDDI registry as follows: Create a business entity, and a business service for publishing the external matching service.

#### *3.5.3 Communicating requester requirements*

Requesters can represent their requirements as external description tModels just as service providers represent their service capability descriptions as tModels. If a tModel is registered to correspond to a set of requester requirements, then the key of the tModel must be passed in a keyedReference within a categoryBag, which requires another categorization scheme which has a set of valid values equal to the set of keys of tModels that represent shared requirements. This categorization scheme is represented by a tModel. It is defined once and can be used with any particular external description approach such as DAML-S. It looks as shown below.

```
<tModel tModelKey="uuid:YYYYYYYY-YYYY-YYYY-YYYY-YYYYYYYYYYYY">
 <name>urn:x-ibm:ClientRequirementsCategorizationTModel</name>
 <description xml:lang="en">Used in a find_tModel call to pass a client tModel that refers to an external description that is to be matched.</description>
 <overviewDoc>
  <overviewURL>…</overviewURL>
 </overviewDoc>
 <categoryBag>
  <keyedReference tModelKey="uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4"
   keyName=”types” keyValue="categorization"/>
 </categoryBag>
</tModel>
```

To differentiate between descriptions of service capabilities and requester requirements, we use the general_keywords categorization. We use keyName=”urn:x-ibm:type” and keyValue=” capabilites" to categorize

service capability descriptions and keyName=”urn:x-ibm:type” and keyValue=”requirements" to categorize requester requirements.A sample service requirements tModel is given below.

```
<tModel  operator="..." tModelKey="CCCCCCCC-CCCC-CCCC-CCCC-CCCCCCCCCCCC">
        <name>urn:x-ibm:name</name>
        <description xml:lang="en"> A tModel to reference the external description of requester’s service request
</description>
        <overviewDoc>
          <overviewURL> URL here </overviewURL>
        </overviewDoc>
        <categoryBag>
   <keyedReference
        tModelKey="UUID: XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX "
                keyName="urn:x-ibm:DescribedUsing"
                keyValue="UUID: DDDDDDDD-DDDD-DDDD-DDDD-DDDD-DDDDDDDDDDDD "/>
<!-- A categorization to indicate that this tModel describes service requirements. →
<keyedReference tModelKey=" uuid:A035A07C-F362-44DD-8F95-E2B134BF43B4"
                keyName="urn:x-ibm:type"
                keyValue="requirements"/>
        </categoryBag>
</tModel>
```

***3.5.4 Detecting the need for external matching***

When a service requester invokes a find_tModel() API on the UDDI registry by passing their requirements tModelKey, the UDDI Registry must detect that the request is for external description matching, rather than a literal match on the tModel key, as is normally the case. The decision must be made during the processing of find_tModel. This is done in two steps. UDDI first looks at the categorybag in the find_tModel request to note the format (eg: DAML-S/UML etc) of external descriptions requested and then excludes these from the category bag. Then, standard categorizations (such as industry classifications) specified in the request are applied to obtain all descriptions of services that are described in the requested format/style under the given categories. This process serves as a filter and retrieves only relevant external description tModels for further processing. These external descriptions obtained from the filtering stage are passed as inputs in a URL form to the external matching service along with the original descriptions of service requirements. In the second step, the registry looks for available and compatible external matching services and invokes the appropriate external matching service by passing the requirements as well as the filtered descriptions. This dynamic invocation is made possible via standard matching

service interface that we define. The matching engine performs its matching (semantic or syntactic) and returns a subset of descriptions back to the registry which are then presented to the requester as a response to find_tModel() request. Next, we discuss the process by which a UDDI registry can select a suitable external matching service.

#### *3.5.5 Selecting external matching service*

The selection of external matching service by a UDDI registry dynamically is an interesting problem in itself. We perform this selection in two steps. First, UDDI registry obtains a set of all compatible matching services that can perform matching of external descriptions in a given format as specified by the requirements. For example, if the user request was to find web services whose external descriptions are described in DAML-S, then UDDI must find a set of external matching services that can match descriptions in DAML-S. This can be done as follows.

```
<find_service generic="2.0" xmlns="urn:uddi-org:api_v2">
        <tModelBag>
        <tModelKey uuid:DDDDDDDD-DDDD-DDDD-DDDD-DDDDDDDDDDDD </tModelKey>
        <tModelKey uuid:ZZZZZZZZ-ZZZZ-ZZZZ-ZZZZ-ZZZZZZZZZZZZ> </tModelKey>
        </tModelBag>
</find_service>
```

Once a set of compatible matching services are obtained, then in the second step UDDI registry chooses the most suitable matching service(s) that can perform the matching of external descriptions. Many policies can be applied in this selection. Below we list some that we have considered in our design.

*First Available*: In this simple policy, the first available matching service is chosen. A matching service is considered to be available if it can be invoked and a subset of inputs can be obtained as a result of matching. In our prototype, we have implemented this policy.

*Last Successful*: UDDI registry can keep a record of the previously successfully invoked matching services. In this policy, UDDI registry chooses a matching service that it has last invoked.

*Most Successful*: If a record of all successfully invoked matching services can be maintained, the registry can select the most frequently successful matching service.

*Union of All*: The registry can invoke all the matching services and obtain a union of matching results while eliminating duplicates. Additional algorithms (Syeda-Mahmood et al., 2005) can be applied to rank these merged results in the end. The performance implications of invoking multiple matching services should be weighed against the need to generate as many matching results as possible in selecting this policy.

*Intersection of All*: Multiple matching services can be invoked and an intersection of the results can be compiled as a result. This can be implemented as a progressive sequential filtering mechanism in which the outputs of one matching service are fed as inputs to the other or as a parallel invocation of multiple matching services where the intersections are computed at the end. The performance implications of each have to be taken into consideration. *Rating-based*: Assuming third-parties or requesters can rate the services offered by the service providers in a UDDI registry, these ratings can be used in the selection of matching services. Also, by standardizing the invocation interface for matching services we have simplified the matching problem. This standardization enables the registry to dynamically invoke the chosen matching services using a WSIF framework (WSIF T.C).

#### *3.5.6 Finding suitable Web Services*

Finding suitable Web services in a UDDI registry in our design consists of three steps. Below, we discuss how requesters can frame their queries in each step.

**Finding matching external descriptions:** Requesters formulate find_tModel() requests based on the new 'ClientRequirementsCategorizationTModel' categorization to indicate to the UDDI registry that external matching is required. As discussed earlier, UDDI performs external description matching and returns a tModelList containing all the tModels whose external descriptions match the requirements specified by the requester. The *find_tModel()* method will rank the tModels based on the similarity of a given tModel to the given requester requirement. Therefore, if a requester is interested in invoking only one service that best matches their needs, then they could consider invoking the service that describes their external descriptions via the first tModel given in the tModelList. This *find_tModel()* request can be framed as follows:

```
<find_tModel generic="2.0" xmlns="urn:uddi-org:api_v2">
   <categoryBag>
      <keyedReference tModelKey="uuid:YYYYYYYY-YYYY-YYYY-YYYY-YYYYYYYYYYYY"
keyName="describedUsing" keyValue="uuid:CCCCCCCC-CCCC-CCCC-CCCC-CCCCCCCCCCCC"/>
   </categoryBag>
</tModel>
```

**Finding services**: Once a set of tModels that match the specified requirements have been found, then a requester can find the corresponding services and their bindings. This can be done in two steps. First via *find_service()* and then via *find_binding()*. As mentioned earlier, requesters can either select the first tModel from the returned tModelList of the *find_tModel()* inquiry or choose a tModel in some other way and invoke *find_service()* inquiry with this

tModelKey. This is a normal *find_service()* invocation. This returns a list of all services that implement the external description captured in the chosen tModel. Again, a requester has a choice of services here. Since all the resulting services implement the same external description (described in a shared tModel) that matched their requirements, requesters can choose which services' bindings they would like to obtain in preparation for invocation (requesters might use many criteria such as: choosing those service providers with whom they have prior business relationships or that they trust the most, or that have minimum quality guarantee ratings etc.). A sample find_service() request is shown below.

```
<find_service generic="2.0" xmlns="urn:uddi-org:api_v2">
    <tModelBag>
  <tModelKey> "Specify first tModelKey returned from the tModelDetails of find_tModel() request above"</tModelKey>
    </tModelBag>
</find_service>
```

**Finding binding templates of services**: Once a service key is obtained, the next step is to obtain a corresponding binding template whose fingerprint matches with the tModel chosen from stage 1. This can be done by invoking a find_binding() inquiry with the service key chosen from find_service() stage and the tModelKey chosen from find_tModel() stage. In this design, we assume that there is only one binding template per (set of) fingerprint(s) for simplicity. In this design, we also assume that the resulting bindingTemplate consists of at least two tModelInstanceInfos: one for representing the fingerprint of external descriptions (DAML-S descriptions) and the other for WSDL specification of this service. Requesters can now retrieve WSDL bindings (or other bindings for runtime invocation) from the resulting binding template and prepare their application to invoke this service. A sample find_binding() request is shown below.

```
<find_binding generic="2.0" serviceKey="service Key of the service chosen from find_service above" xmlns="urn:uddi-org:api_v2">
    <tModelBag>
  <tModelKey> "Use the same tModelKey used in find_service() above."
</tModelKey>
    </tModelBag>
</find_binding>
```

## 4. Implementation Details

Our prototype is developed in Java and uses IBM's WebSphere Application Server deployment environment. The running time of the prototype includes the time taken to load the relevant ontologies (done once and retained in

memory) and to inference the relationships. Since the size of the ontologies in our sample domains is relatively small (on the order of dozens of concepts), SNoBASE keeps all the ontological concepts and instances in memory for fast access. We are currently enhancing the functionality of our ontology management system to scale better.

We used the following technologies in developing our prototype: (1) OWL-based semantic markup languages for ontology, and service capability representations (OWL-S) (2) A semantic UDDI server (Colgrave et. al 2003) as described previously for finding suitable services based on IBM's WebSphere UDDI registry (IBM, 2003) which implements the UDDI V2 specification (3) an OWL-S matching engine (Doshi et. al, 2003) for matching service semantics and augmented with the constraint checker module for reasoning with inter-service dependencies and constraints (4) a semantic network based ontology management system, SNoBASE (Lee et al, 2003) that offers a DQL-based (DQL T.C., 2003) Java API for querying ontologies represented in DAML+OIL/OWL (5) IBM's ABLE (Bigus et al 2001) engine for inferencing (6) BPEL for representing the process flows (7) BPWS4J, IBM's BPEL execution engine (BPWS4J 2002) and (8) WebSphere Application Server (IBM 2003): IBM's Java application server for deploying and executing Web Services and BPEL flows. Our system is available to the community for download, as part of the ETTK toolkit (ETTK, 2005).

## 5. Related Work

Sivashanmugam et. al. developed a template-based approach to capturing the semantic requirements of process services in the METEOR-S Web Service Composition Framework. (Sivashanmugam et al., 2004). The semantic information about services in the templates can be used to dynamically discover suitable services and generate executable BPEL documents. An approach is presented by Sirin et. al. for semi automatically generating process compositions using semantic capabilities of Web services (Sirin et. al., 2003). Mandel et. al. (Mandel & McIlraith, 2002) present an approach to combine DAML-S and BPEL for achieving dynamic binding. They also account for user defined constraints in service selection. A significant difference between this work and our approach is that we capture and reason with the inter-service dependencies. In order to do this, we introduced the notion of scope of related services within the BPEL flow and use it to bind all services that are related to accommodate their domain constraints and service dependencies. The result is a set of bindings that are legal and feasible in the operating domain. Moreover, their registry search mechanism does not support dynamically matching services that are described in multiple semantic modeling languages like we do in our enhanced UDDI approach.

Some efforts have also been made to accommodate service semantics in UDDI and to propose ways to enhance its search functions.

Paolucci et. al. (Paolucci et al, 2002-2) in their approach, propose to enhance search in UDDI by intercepting the search calls to a UDDI registry and performing semantic matching outside of the UDDI registry. The functionality of UDDI registry itself is untouched in this approach. While this approach is a good start, it has an inherent disadvantage. Every user of UDDI registry has to have the infrastructure developed by Paolucci et. al., for the semantic matching to take place. This is not only cumbersome but also limits the general availability of this function. To address this limitation in a follow up work, Akkiraju et. al. present a design mechanism for a tighter integration of semantic matching with UDDI registry by directly extending UDDI's inquiry Application Programming Interface (API) (*find_service()*) and its implementation. This approach incorporates semantic matching directly in UDDI registry by altering the *find_service()* API that users of UDDI registry are familiar with. While this is a workable solution, it proposes embedding matching capability into UDDI registry implementation. This makes UDDI matching specific to a particular service description language - in their case DAML-S.

In our approach, we present a natural and seamless integration of semantic matching engines with a UDDI registry by publishing a matchmaker as yet another Web Service in UDDI. This eliminates the need for installing the matchmaking infrastructure on clients' side and allows independent service providers to offer their matchmaking engines for matching via UDDI. Also, in our approach, we offer best-of-breed match solutions by offering the possibility to employ multiple match making services to fulfill a given request. Moreover, our approach is not restricted to DAML-S based semantic matching. We allow for any format or type of matching including WSDL, and UML.

## 6. Conclusions and Future work

In this paper, we presented a novel approach to dynamically discover suitable services from a UDDI registry and to automatically bind them to abstract business process flows represented in BPEL while considering inter-service and domain dependencies and constraints. The result is a system that allows workflow designers to focus on creating appropriate high-level flows, while providing a robust and adaptive runtime environment for developers to translate these high-level flows into executable flows. In this work, we assume that the services that are discovered from a UDDI registry are always available and the context in which a workflow process is constructed is stable. However, in real-world services that were once available may no longer be available and context might change based on

business situations. To account both for uncertainty and the dynamic nature of environment, we are currently exploring service execution monitoring and recovery of process flows that are dynamically composed using probabilistic models (Doshi et al., 2004).

Our work leverages the advances in semantic web technologies, to augment the flexibility of the current industry standards. Specifically, we have augmented the service discovery capabilities of UDDI. UDDI does not currently allow for external matching services to play a role in the matching of UDDI entities against criteria supplied by the user. In this work, we have presented a new design and implementation which allows multiple external matching services to be integrated with a UDDI registry. In our design, these external matching services can support multiple external description formats and styles of matching. The result is a UDDI registry with flexible and intelligent service search function that can be used for dynamic service selection. While our approach supports the invocation of semantic matching services that can perform service compositions, returning the compositions raises some issues. For example, creating ad-hoc new services via compositions in a UDDI registry would open up service life cycle and ownership issues. For the time being, we believe that they are better left outside the domain of the UDDI registry. An alternative approach is to use a UDDI proxy for presenting service compositions as described by Akkiraju et. al. in their work (Akkiraju et al, 2003).

## 7. References


Akkiraju R., Goodwin R., Doshi P., and Roeder S. 2003. A Method for Semantically Enhancing the Service Discovery Capabilities of UDDI In the *workshop proceedings of Eighteenth International Joint Conference on Artificial Intelligence. Information Integration on the Web*. WEB-1 pg: 87-92

Ankolekar A., Burstein M., Hobbs J. J., *et al*. 2001. DAML-S: Semantic Markup for Web Services. In *Proceedings of the International Semantic Web Working Symposium (SWWS).*

BPEL Technical Committee (TC). 2002. Business Process Execution Language: BPEL. IBM Developer Works Article. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/

Christenson E., Curbera F., Meredith G., and Weerawarana S. 2001. Web Services Description Language (WSDL). www.w3.org/TR/wsdl

DAML Technical Committee (TC). 2000. DARPA Agent Markup Language- DAML. http://www.daml.org

DAML+OIL Technical Committee (TC). 2001. DAML+OIL. http://www.daml.org/2001/03/daml+oil-index

DAML-S Technical Committee (T.C). 2002. DAML ontology for Web Services. http://www.daml.org/services/

Doshi P., Goodwin R., and Akkiraju R. 2003. Parameterized Semantic Matching for Workflow Composition. IBM Technical Report. RC23133. Available upon request.

Doshi P. Goodwin R., Akkiraju R., Verma K. 2004. Dynamic Workflow Composition using Markov Decision Processess. In the proceedings of International conference on Web Services ICWS 2004, San Diego, CA.

DQL Technical Committee 2003. DAML Query Language (DQL) http://www.daml.org/dql

ETTK: Emerging Technologies ToolKit – Semantic Tools for Web Services http://www.alphaworks.ibm.com/tech/wssem

Fox M., and Long, D., PDDL2.1: An Extension to PDDL for Expressing Temporal Domains, The AIPS-02 Planning Competition Committee, 2002. http://www.dur.ac.uk/d.p.long/competition.html.

IBM 2002. The IBM Business Process Execution Language for Web Services Java™ Run Time (BPWS4J). http://www.alphaworks.ibm.com/tech/bpws4j

IBM 2003. IBM Websphere Application Server http://www-3.ibm.com/software/info1/websphere/index.jsp?tab=products/appserv

Lee J., Goodwin R. T., Akkiraju R., Doshi P., Ye Y. SNoBASE: A Semantic Network-based Ontology Ontology Management. http://alphaWorks.ibm.com/tech/snobase. 2003.

McIlraith S., Son T., and Zeng H. 2001. Mobilizing the Semantic Web with DAML-Enabled Web Services. *Semantic Web Workshop*.

Mandel, D., McIlraith S., 2003 Adapting PBEL4WS for the semantic web: The bottom up approach to web service interoperation *Second International Semantic Web Conference (ISWC2003),* Sanibel Island, Florida, 2003.

OWL Technical Committee (T.C). 2002. Web Ontology Language (OWL). http://www.w3.org/TR/2002/WD-owl-ref-20021112/

OWL-S Technical Committee (T.C). 2002. Web Ontology Language for Web Services. http://www.daml.org/services/owl-s/

Paolucci M., Kawamura T., Payne T. R., and Sycara K. 2002. Semantic Matching of Web Services Capabilities. *The First International Semantic Web Conference (ISWC)*, Sardinia (Italy).

Paolucci M., Kawamura T., Payne T. R., and Sycara K. 2002. Importing the Semantic Web in UDDI. *In Web Services, E-Business and Semantic Web Workshop*.

Sirin E., Hendler J., and Parsia B. 2003. Semi-automatic composition of web services using semantic descriptions. Web Services: *Modeling, Architecture and Infrastructure workshop in conjunction with ICEIS2003*, April 2003.

Sivashanmugam K., Miller J., Sheth A., Verma K. International Journal of E-commerce, Winter 2004-5, Vol. 9(2) pp. 71-106

Tanveer Syeda-Mahmood, Gauri Shah, Rama Akkiraju, Anca-Andreea Ivan, and Richard Goodwin Searching Service Repositories by Combining Semantic and Ontological Matching. In the proceedings of the Third International Conference on Web Services (ICWS), July 2005.

UDDI Technical Committee. 2002. Universal Description, Discovery and Integration (UDDI). http://www.oasis-open.org/committees/uddi-spec/

UML Technical Committee (T.C) 2003. "Unified Modeling Language". http://www.omg.org/uml/

Weerawarana S., Curbera F. 2002. Business Process with BPEL4WS: Understanding BPEL4WS. http://www-106.ibm.com/developerworks/webservices/library/ws-bpelcol1/

Verma K., Akkiraju R., Goodwin R., Doshi P., Lee J., On Accommodating Inter Service Dependencies in Web Process Flow Composition, AAAI Spring Symposium, 2004, pp. 37-43.

WSIF Technical Committee. "Web Service Invocation Framework" 2003 http://ws.apache.org/wsif/