RI 07003 March 2007 Computer Science

IBM Research Report

Raising Programming Abstraction from Objects to Services

Arun Kumar, Anindya Neogi
IBM Research Division
IBM India Research Lab
Block I, I.I.T. Campus, Hauz Khas
New Delhi - 110016. India.

Sateesh Pragallapati, D. Janaki Ram
Dept. of Comp. Sc. & Engg.,
Indian Institute of Technology Madras,
Chennai-600036, INDIA

IBM Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T.J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted is accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T.J. Watson Research Center, Publications, P.O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home

Raising Programming Abstraction from Objects to Services

Arun Kumar, Anindya Neogi
IBM India Research Laboratory
New Delhi-110016, INDIA
{kkarun, nanindya}@in.ibm.com

Sateesh Pragallapati, D. Janaki Ram
Dept. of CS&E, IIT Madras,
Chennai-600036, INDIA
{sateeshp, d.janakiram}@cs.iitm.ernet.in

Abstract

Traditional software development involves writing complex programs by reusing and building upon off-the-shelf software libraries. The benefit of this paradigm is compile time availability of both, component interfaces as well as corresponding implementations. This enables software developers to write new programs that build on existing functionality without worrying much about runtime environment. Current shift towards service oriented computing presents a different paradigm that involves actively running components in the form of services. Here, it becomes necessary to discover the component(s) first and to determine whether some exist that satisfy the requirements specified. This requires a search operation to be performed in runtime service registries and prevents the software developer from creating new service oriented program without having to rely upon component availability in the runtime environment. Moreover, new web service instances may come up or existing ones may go down dynamically. This leads to frequent expiry of searched results making dependent programs highly brittle. In this paper, we present a novel approach for services based software development that proposes a paradigm shift from objects to services as first class entities.

Keywords *Semantic Web Services, Programming languages, Matching, Object orientation, Ontology*

1 Introduction

Software development today involves use of software libraries that are available as programming language API and undergo change infrequently. The developer writes a program using existing functions or classes from these libraries and compiles it into an executable. Runtime environment considerations play a role only for generation of an appropriate executable. The world of web services presents a different challenge, however. Unlike traditional software libraries that come in the shape of *off-the-shelf* components, web services are *actively running* components that need to be composed together. Also, being autonomous, Web services may come up or go down dynamically unless there are offline contracts in place. Furthermore, web services are designed to be accessible programmatically to enable automation. These characteristics have kept the focus of Web Services tools and technologies towards runtime interactions leading to techniques for automated Web Service discovery, selection and composition, as well as Web Services orchestration and choreography etc.

However, two different models of development are emerging. In one, developers need to program software agents that accept the requirements from the end user. The services needed to fulfill those requirements are then automatically discovered, selected, composed and invoked by these agents. The work being done by Semantic Web Services community plays an important role towards enabling this vision. While this model matures, the other model being employed by application developers is along the lines of traditional software development. Developers code enterprise systems by first developing new web services or by building wrappers for legacy systems or by using existing known services as components in their programs.

It is in the latter model that there exists a significant gap in terms of current programming language abstractions that are inadequate for programming Service Oriented Architectures (SOA). The dynamic nature of web service availability makes the developers dependent on runtime environments. Developers are also expected to translate and encode high level service contracts into programs written in OO languages [10], and find ways to fulfill functional requirements (through composition, for example) as well as enforce policies containing non-functional requirements. Much of the information they need belongs to the runtime environment and is dynamic.

In prior work [12], we proposed a model that helps in alleviating some of these and other problems. However, development tools and languages need to provide adequate support in order to derive full benefits of such rich modeling. In this paper, we take a leap forward and propose to raise the level of abstraction in current programming languages from objects to services. Specifically, the core contributions of this paper are as follows:

- We present enhanced service matching techniques.
- We introduce language level operations that involve services as operands.
- We elevate services to first class entities with the help of above operations.

We hope that this approach would fuel a shift of focus in research from runtime aspects to design time capabilities of Web Services tools and technologies leading to a Service Oriented Software Development (SOSD) methodology.

2 Motivation and Background

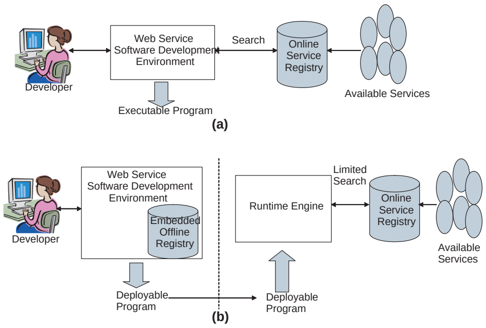
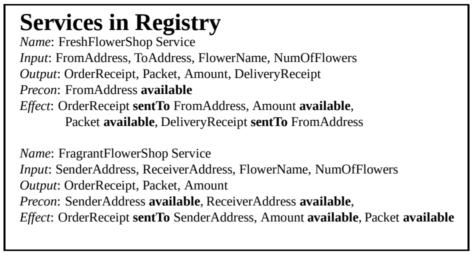


Figure 1. Web Services based Software Development Model (a) Current (b) Proposed

Figure 1 (a) depicts the current Web Services based software development model in which the developer makes use of web services that are currently available for invocation. However, some of these services might not be available for use when this program is actually executed making the program brittle. When software agents are used, the desired web service can be searched at runtime but it adds a significant delay into the service invocation path. There is a tradeoff between program correctness and performance here.



Services in Registry

Name: FreshFlowerShop Service
Input: FromAddress, ToAddress, FlowerName, NumOfFlowers
Output: OrderReceipt, Packet, Amount, DeliveryReceipt
Precon: FromAddress **available**
Effect: OrderReceipt **sentTo** FromAddress, Amount **available**,
Packet **available**, DeliveryReceipt **sentTo** FromAddress

Name: FragrantFlowerShop Service
Input: SenderAddress, ReceiverAddress, FlowerName, NumOfFlowers
Output: OrderReceipt, Packet, Amount
Precon: SenderAddress **available**, ReceiverAddress **available**,
Effect: OrderReceipt **sentTo** SenderAddress, Amount **available**, Packet **available**

Figure 2. *The FlowerShop Services*

In both cases, the service invocation may not succeed even though a suitable service might be available to service the request. Current approaches are too restrictive requiring strict matching of service capabilities with client requirements [7]. Even though approaches such as [20] use information retrieval techniques, and [18, 6] allow a softer notion of matching, each of these are limited in their effectiveness. We illustrate this with an example.

Consider the service descriptions shown in Figure 2, described in terms of a simplified representation of their Input, Output, Preconditions and Effects[1]. Here, two services - FreshFlowerShop service and FragrantFlowerShop service are offering similar functionality, i.e. that of a flower shop. While both accept a sender and receiver address, FreshFlowerShop service makes receiver address optional. It also provides a delivery receipt that is sent to the sender. Besides this, their descriptions use different terms for same concepts (e.g. FromAddress and SenderAddress).

Current matching tools are not likely to select FreshFlowerShop service as one of the matches if the user puts in a request for a service with same specification as that of FragrantFlowerShop service. The syntactic difference in their input specification can either be resolved through existing similarity based matching techniques or through the use of an ontology that defines equivalence relationship between FromAddress and ToAddress, for example. The difference in their outputs and effects, however, would result into a mismatch with current matching techniques even though FreshFlowerShop service can be used to serve a request for FragrantFlowerShop service due to the existence of a semantic relationship between them [12]. Specifically, FreshFlowerShop service is a *subtype* of FragrantFlowerShop service and can actually be used in its place.

To resolve these issues we propose a two-pronged

[1] www.daml.org/services/owl-s/

solution. First, we propose to follow the development model shown in figure 1(b). This helps in diluting the program correctness – performance tradeoff by isolating software development activities from the dynamics of the runtime environment. To achieve this, we propose to utilize the segregation of service description into *Service Type* description kept in an offline service registry and *Service Instance* description kept in runtime registry, as introduced in [2] .

Service Type is a semantic specification of the service consisting of one or more profileTypes (i.e interface descriptions), an optional description of the process model and a description of *internal* state maintained by the service [12]. This means that programs can commit only to service interfaces and not to their implementations which is good design practice [9]. For developer, this approach enables writing programs independent of runtime service characteristics. Service Instance, on the other hand, is an operational specification of the service consisting of a reference to the Service Type, one or more profiles (i.e. interfaces), its internal state and a grounding [12]. At deployment time, the search for an exact service instance is restricted to a small subset of services that conform to a Service Type. It can be guided by non-functional characteristics such as quality of service guarantees and partner selection techniques [17, 21] can be applied effectively there.

Second, to ensure success of the development model proposed above it is essential that support for functional matching of services needs to be made richer and stronger than what exists today. Towards that end we proposed a semantic model for building service oriented systems that captures relationships existing between different services [12]. Without compromising on any principles of SOA[2], we introduced the abstraction principle of Classification to define Service Types and Instances in the context of service oriented computing (SOC). To maintain the stateless service semantics we carefully chose not to commit to the *class* construct available in most OO languages. Instead, we adopted the notion of WS-Resource – a stateful resource and a web service acting upon it – as proposed in Web Services Resource Framework (WSRF) [3]. Similarly, we defined other abstraction principles such as aggregation (service composition), interface inheritance and polymorphism in the context of SOC.

In this paper, we define language level operators that can enable developers to harness such rich semantic modeling of services. The operators accept services as their operands where the services are described by their functional specification.

3 Services as First Class Entities

The concept of Service Type offers an equivalent of *data type* in programming languages. The range of a Service Type is defined by the set of all Service Instances conforming to that type. This allows us to treat services as first class language level entities since Service Types encapsulate the functional capability of the service needed at compile time. The semantics of associating a range of Service Instances with a Service Type is well captured by the *matching* primitive found in services literature [18, 6, 1].

The service matching process can be split into compile time matching and runtime matching to enable the model proposed in figure 1(b). Further, we enable functional matching of service at compile time. Since functional description of service deals with categorical concepts derived from an ontology [2, 12], it can be integrated into a service development environment independent of non-functional characteristics of the service instances such as data types of message parameters, QoS guarantees offered etc.

An IDE equipped with a registry of Service Types and compile time functional matching capability could be used for developing service oriented programs that treat services as another construct available at the language level. This would be similar to programming with Java where classes are used in programs and actual objects are created at runtime. In services case, the actual service instances could either be already existing ones or could be created through the use of service factories [8].

Selection of appropriate service instances, through matching of non-functional parameters, can take place at runtime and has been addressed elsewhere [17, 21]. In this paper, we focus on functional matching of services.

In next few subsections we build upon existing work and present an enriched view of matching to define meaningful service level operations.

3.1 Service Matching Refined

Functional matching of services involves functional parameters of the service description and enables service discovery, whereas non-functional matching of services enables service selection based upon parameters such as quality of service, security guarantees etc.

For functional matching, a softer notion of similarity is typically adopted than a purely ’exact’ match [18].

[2] http://en.wikipedia.org/wiki/Service-oriented_architecture

A similarity measure is defined that captures the semantic distance between the attributes associated with the requested service and advertised services. Authors in [18] define four degrees of match between two inputs or two outputs. These are determined based upon the relation between ontological concepts associated with those inputs and outputs.

An *exact* match is returned if both the concepts are equivalent, a *plugin* match is returned if the advertised output is a superclass of requested output, a *subsumes* match if advertised input is a subclass of requested input, else it is a *disjoint* match. The matching algorithm presented in [6] adds a *Container* and its complimentary *Part-of* match. Match from a service to another service is a Container match when first service's parameter contains the second service's parameter.

However, they [18, 15] use a set theoretic basis for defining these match levels. It does not conform to the object oriented principles that underlie the concepts of subsumption[3] [14]. For instance, apart from equivalence an exact match is also returned in [18] for outputs when requested output is a subclass of advertised output. The assumption made is that by advertising for an output O the provider commits to provide outputs consistent with every immediate subtype of O. This is not in agreement with the established concept of subtyping [22] which says that a subtype extends the definition of its supertype. In other words, a subtype can be used in place of a supertype but not the other way round [4].

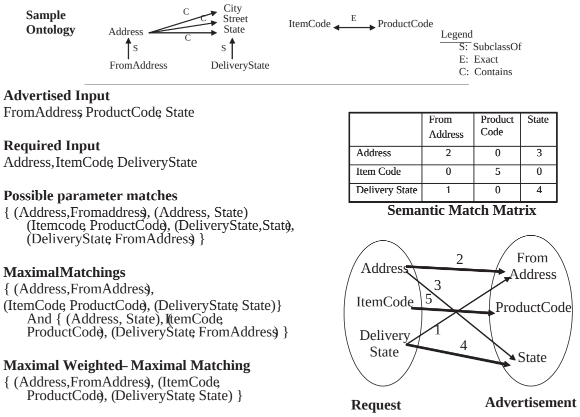
We adopt object oriented principles to redefine and refine these different levels of parameter matching, as shown in figure 3. The relation for Output parameters is defined from Advertisement to Request and for Inputs it is defined from Request to Advertisement. Therefore, for outputs (refer figure 3(b)), an *exact* match is returned if the advertised concept is equivalent to the requested concept, a *plugin* match is returned if the advertised concept is a subclass of requested concept, a *contains* match is returned if the advertised concept consists or is composed of the requested concept, a *subsumption* match is returned if the advertised concept is a superclass of the requested concept, a *part-of* match is returned if if the advertised concept is contained by the requested concept, otherwise *disjoint* match is returned.

Plugin match is the preferred one after *exact* as a service that accepts a more general input can be used for a service that expects a more specific input. It is followed by *contains* since a service that expects a component object can be used for a service that accepts a composite object as input. *Subsumption* followed by *part-of* come next because in both cases the requirements of the requester are partially met. *Disjoint* comes last. The horizontal dotted line in figure 3 indicates the threshold level above which the degree of match is expected to have practical applicability.

Relation***(Req,Adv)*** **(Inputs)**	Relation***(Advt,Req)*** **(Outputs)**	**Semantic Match Level**	Increasing Strength
Equal	Equal	**Exact**	5
Subclass	Sublass	**Plugin**	4
Contains	Contains	**Contains**	3
SuperClass	SuperClass	**Subsumption**	2
ContainedBy	ContainedBy	**Part-of**	1
Unrelated	Unrelated	**Disjoint**	0
(a)	(b)		

Figure 3. Semantic Match between parameters (a) Inputs (b) Outputs

The match levels above, specify the relationship that may exist between two individual parameters. For comparing the entire set of inputs (or outputs) of requested service with those of advertised service we first need to determine parameters correspond to each other. This is non trivial. Authors in [15] proposed the use of a maximum weighted–maximum cardinality matching algorithm [16] to determine the best match between request and advertisement parameters. As shown in Fig. 4, a semantic match matrix is computed that captures semantic distance between all pairs of attributes. Then using the above matching algorithm the best match across all parameters is selected. The resulting match is considered successful if semantic match value of each parameter considered in the match is equal to or above the specified match threshold.



Advertised Input
FromAddress, ProductCode, State

Required Input
Address,ItemCode, DeliveryState

Possible parameter matches
{ (Address,Fromaddress), (Address, State)
(Itemcode, ProductCode), (DeliveryState,State),
(DeliveryState, FromAddress) }

MaximalMatchings
{ (Address,FromAddress),
(ItemCode, ProductCode), (DeliveryState, State)}
And { (Address, State), (ItemCode,
ProductCode), (DeliveryState, FromAddress) }

Maximal Weighted– Maximal Matching
{ (Address,FromAddress), (ItemCode,
ProductCode), (DeliveryState, State) }

	From Address	Product Code	State
Address	2	0	3
Item Code	0	5	0
Delivery State	1	0	4

Semantic Match Matrix

Figure 4. Maximal Weighted Maximal Cardinality matching

[3] http://en.wikipedia.org/wiki/Subsumption

3.2 A Holistic View of Matching

Most of the existing approaches take a simplistic view of service matching. First, the semantic distance is computed for service attributes that are expressed merely as ontological concepts [18, 6] whereas actual descriptions could contain complex expressions as preconditions and effects of different operations. Second, the only service level operation available is *equivalence* that returns whether an exact or a lesser degree match exists between the services compared [17, 21]. Third, entire matching is performed at runtime introducing delays in the service discovery and composition processes.

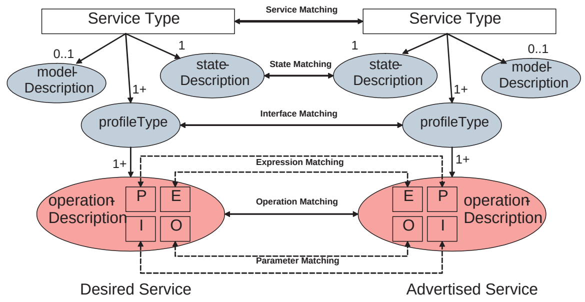


Figure 5. Layered Matching

An end-to-end view of functional service matching is presented in Fig. 5. To match a desired service with an advertised service, the internal components of the two service descriptions need to be matched first. This happens at different levels of abstraction.

Parameter Matching: Here, matching is done to compare individual attributes (such as input elements or output elements) involved in service descriptions. As mentioned above, the attributes are typically ontological concepts and a similarity measure is defined that represents the semantic distance between two attributes.

Expression Matching: When ontological concepts alone are used to represent all kinds of preconditions and effects, it can lead to ontology explosion and also result in a brittle ontology [13]. Expression matching defines similarity measure representing semantic distance between two (boolean) expressions defined over ontological concepts.

Operation Matching: The operation level matching process uses the parameter level matching results and expression level matching results to determine whether all <I,O,P,E> of the two operations being compared have a semantic match value above the specified threshold.

Interface, State and Model Matching: A service interface is a logical collection of related operations that the service offers. Therefore, interface level matching is simply a collection of operation matching results. State level matching determines the similarity of two services in terms of the internal state that they maintain. It is similar to parameter level matching since state is captured using simple ontological concepts. Matching based upon the internal process model of the services is called Model matching. It can be useful for temporal matching of services to ensure that services carry out certain steps in a particular order [1].

Service Matching: Similarity of two services is computed by aggregating the semantic distances between their corresponding operations and state descriptions.

Next, we present a refined equivalence operation defined over services as operands, based upon this end to end view.

3.3 A refined *equivalence* operator

While most existing approaches rely on parameter matching alone for matching service descriptions, we refine it with an algorithm for expression matching. Operation level matching could then use parameter matching for inputs, outputs and expression matching for preconditions, effects.

We represent preconditions and effects as boolean expressions involving unary or binary predicates involving concepts from service inputs and outputs as operands (refer Fig. 1). For the sake of simplicity, we present our discussion with conjunctions of binary predicates and ignore disjunctions and unary predicates for now.

Consider the following symbolic predicated belonging to an advertised and requested service description.

```
Advt.:  OpdA1  operatorA  OpdA2 ...(1)
Req  :  OpdR1  operatorR  OpdR2 ...(2)
```

Similar to Figure 3 for inputs, outputs, Figure 6 shows a table using which the semantic match level of two predicates in effects can be computed based upon the ontological relationship between the concepts represented by the operands. The left side column of the table lists the relation between first operand OpdA1 of the advertisement (1) and its corresponding operand OpdR1 in the request (2) shown above. The top header row of the table lists the relation between second operand of the advertised and the requested predicate. The values inside the table indicate the resulting semantic match relation between the effect predicates as defined from advertised service to requested service. For preconditions, the semantic match relationship is defined from requested service to advertised service. The table remains the same except with positions of OpdA1 and OpdA2 exchanged with positions of OpdR1 and OpdR2 respectively.

The semantic match levels for effects have similar interpretation as presented for outputs earlier. However, *exact*, *plug-in*, *contains*, *subsumption*, and *part-of* matches are returned either in the case of unary predicates or when *both* the operands of the predicates being compared have an equality, subclass, contains, superclass, or containedBy relation in the ontology respectively[4]. If the two operands of a predicate share a different relationship with the corresponding operands in the predicate being compared, the resulting match is one of *plug-in–contains*, *plug-in–subsumption*, *plug-in–part-of*, *contains-subsumption* or *subsumption–part-of*. Figure 7 shows, these semantic match levels sorted in terms of their match strength. These additional semantic match levels help in enabling predicate matches that otherwise result into a mismatch. For instance, consider the following predicates used as effects:

```
Advt.:  OrderReceipt sentTo SenderContact
Req  :  Receipt sentTo SenderAddress
```

The ontological relationships are as follows. OrderReceipt is a *subclass* of Receipt and SenderContact *contains* SenderAddress (in addition to SenderEmail, SenderMobile etc.). In this case, the advertisement indicates that an OrderReceipt would be sent to SenderContact (which included SenderAddress, SenderEmail and SenderMobile, etc.) as one of the effects. This service can very well be used for a request that requires a Receipt to be sent to SenderAddress. As per Figure 6, the match level here is *plug-in–contains* which is placed below *plug-in* and higher than *contains* in terms of match strength. For a client, that is fine with extra side-effects (such as receipt sent to SenderEmail and SenderMobile in addition to SenderAddress) this is valid a match.

Relation (Adv,Req)		OpdA2				
		Equal(OpdR2)	Subclass(OpdR2)	Contains(OpdR2)	SuperClass(OpdR2)	ContainedBy(OpdR2)
OpdA1	Equal(OpdR1)	Exact	Plug-in	Contains	Subsumption	Part-of
	Subclass(OpdR1)	Plug-in	Plug-in	*Plug-in-Contains*	Plug-in-Subsumption	Plug-in-Part of
	Contains(OpdR1)	Contains	*Contains-Plug-in*	Contains	Contains-Subsumption	Contains-Part-of
	SuperClass(OpdR1)	Subsumption	Subsumption-Plug-in	Subsumption-Contains	Subsumption	Subsumption-Part of
	ContainedBy(OpdR1)	Part-of	Part-of –Plug-in	Part-of-Contains	Part-of-Subsumption	Part-of
	Unrelated(OpdR1)	Disjoint	Disjoint	Disjoint	Disjoint	Disjoint

Figure 6. Semantic Matching for predicates in Effects

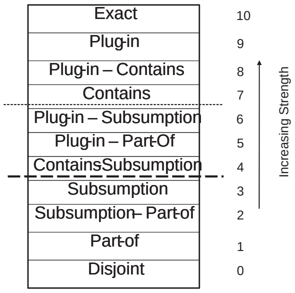


Figure 7. Semantic Match Level for Preconditions and Effects

Using expression matching for individual predicates, the match level of an entire effect (precondition) is computed as minimum of semantic match level of all predicates in the effect (precondition). Figure 8 presents the algorithm that we use for determining whether an entire advertised effect (precondition) matches with the requested one.

The first step finds the level of semantic match between each pair of predicates in the effect (precondition) of the advertised and requested service operation being compared. To do this, semantic match between each pair of operands of the predicate is considered. For the advertised and requested predicates given above, the semantic match level is computed as follows: Max(Min(match(OrderReceipt, Receipt), match(SenderContact, SenderAddress)), Min(match(OrderReceipt, SenderAddress), match(SenderContact, Receipt))) . match() function is computed using the table in figure 6.

Using the numerical values associated with the semantic match levels obtained, we get a semantic match matrix M similar to the one for inputs as shown in figure 4. In the second step, maximal weighted maximal matching algorithm is applied to determine the best matching pairs of predicates in the effect (precondition) being compared.

Third step verifies from the ontology that the advertised predicates in the best match indeed satisfy the corresponding requested predicate. If not, the pair's semantic match level is reduced to a very low value and the weighted matching algorithm is run again. As mentioned above, the match level of the entire effect (precondition) is computed as minimum of semantic match level of all predicates in the effect (precondition).

The operation level equivalence matching is computed as minimum of match levels of all inputs, output, precondition and effects of that operation. The service level equivalence matching is similarly obtained as minimum of semantic match levels of all operations and state of that service. Next, we introduce a few other operators

[4] Here, we assume that the operators in predicates being compared, i.e. operatorA and operatorR in this case, have an exact match. Situations where operators may not match exactly have not been addressed in this paper.

```
Step 1:  For all predicates i in advertised effect
             For all predicates j in requested effect
             Compute M(i,j) // semantic match level between predicate i and j
Step 2:  Compute WM - the maximal weighted maximal match.
Step 3:  For i = 1..n predicates of advertised effect (precondition)
              Assert ith predicate into the ontology
              Query for corresponding requested predicate
              If query fails then
                      Assign a very low semantic match level to this pair. Goto Step 2
              Else Continue
Step 4:  Find V = Min(Semantic Match Level of all predicates in WM)
         If ( V > Threshold) then return <match, match value>
         Else Return no match
```

Figure 8. Algorithm for matching of effects (preconditions)

thus progressing towards a library of service level operations.

3.4 A Library of Operations

Relationships other than equivalence are also important since many times a non-equivalent service can suffice (or may be necessary) for the task at hand. We introduce four other service level operators, namely *subtypeOf*, *supertypeOf*, *contains* and *componentOf*. All of these are binary operations.

SubtypeOf and superTypeOf operations determine whether two given services have a subsumption relationship. Essentially the operations try to establish whether one service is a subtype [14] of another. For that purpose, we utilized a modified form of the inheritance model defined for services, as presented in [12]. In brief, a service is a subtype of another if (1) it maintains zero or more additional state elements than the ones maintained by the other service, (2) it has zero or more additional operations in its interface compared to the other service, (3) maintains same inputs and adds zero or more outputs to the operations that are similar in both services, (4) has same or weaker preconditions and same or stronger effects for the operations that are similar in both services. Compared to [12], this follows a contravariant [4] approach and is necessary for enabling subtyping.

These two operators are important since using these a service based software developer could write programs that utilize related services if the exactly matching services are either not available or are not performing as per the desired non-functional parameter values. This opens up a number of possibilities for writing adaptable, fault-tolerant and robust service oriented programs. Further, administrators could write policies to switch to related services in case of failures or other situations. In addition, use of these operators coupled with late binding of a service invocation could enable services that provide differentiated quality of service to different customers.

Contains and componentOf operators determine whether a service is one of the components from which the other service has been composed. Essentially, the operators try to establish the Service composition relationship if any exists between the two services [12]. If the composition of the composite service is available then the operations have a trivial implementation else automated composition techniques such as those based upon AI planning [2] need to be applied.

These operators assume importance in the context of assetization of reusable services. With several assetized services in use, these could be used in programs that establish the composition hierarchy and make use of it for provisioning, load sharing and time sharing purposes.

3.5 Scalable Service Level Matching

The benefits of proposed matching techniques, in terms of performance gains would be realized as more and more Service Types get defined. The matching algorithms introduced would ideally be carried out offline since each service instance can be tagged with a Service Type name. Any new unlabeled service being advertised to a registry would go through the process of getting compared with existing ones. The new service would either be another instance of an existing Service Type, or be derivable from an existing Service Type or be a superclass of an existing Service Type. Otherwise, it results into creation of a new Service Type. This approach results into a simplified ontology of Service Types that consists of only Service Type names and relationships between them rather than entire functional specifications.

The clients looking for desired services can search the registry using appropriate Service Type names. This not only simplifies the information expected from the requester but also enables scalable searching. This is because searching is reduced to simple string comparison of Service Type labels rather than comparison of the entire functional specifications. Only when the requester has an unlabeled service specification that the entire matching would need to be performed to determine if the functional specification has a relationship with one of the existing service descriptions.

4 Prototype Implementation

Figure 9 describes our prototype implementation of the Service level matching engine that would be a core component of the offline registry embedded in a service developer IDE, as shown in figure 1(b). The Service

Matching Engine accepts a functional description of the requested service and uses matching techniques introduced in this paper to return the best matching advertised Service Type as well as the strength of the match from figure 7.

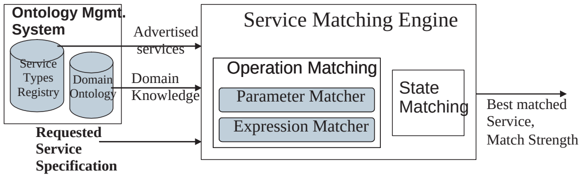


Figure 9. Service Level Matching Engine

The engine makes use of an ontology management system consisting of a domain ontology as well as services registry. We used IBM's Snobase system[5] for that purpose and modified it to support OWL-S v1.1. For our matching engine we implemented refined parameter matching and expression matching algorithms presented in the paper. On top of these we implemented the refined equivalence operation as well as subtypeOf, supertypeOf operations. Contains and componentOf operations rely on the previous work done for automatic composition using AI planning techniques [2].

For matching entire inputs (outputs) of two services as well as for matching preconditions and effects, we implemented the maximal weighted-maximal cardinality matching, using Munkres polynomial time (O(n^3)) assignment algorithm [16]. In the case of inputs and outputs, individual parameter matching coupled with the Munkres algorithm suffices for determining the match. For preconditions and effects, Munkres algorithm enables determining which predicates of a precondition (or effect) of request correspond to the precondition (effect) being compared from the advertisement. A further step, was performed by asserting the advertised predicates into the ontology and then determining whether the requested predicate is satisfied by the advertised one.

As mentioned before, entire this activity is offline one carried out at the time of admitting a new unlabeled service to the registry. The current prototype assumes that the number of predicates in precondition or effect of advertisement service is same as the number of predicates in precondition or effect of the requested service.

5 Related Work

Several researchers have contributed to matching technologies for services. [11] presents a survey and comparison of different matching approaches. However, the thrust of matching approaches has been towards automated matching rather than enabling it for a web services developer. [19] focus away from automatic composition in order to provide developers with a valuable utility to browse repositories based on already existing information. However, they move towards search as their goal rather than precise matching.

Similarly, clustering techniques have been used in the Woogle system [5] that enables similarity based searching for web services. However, they support WSDL based service specification and work with inputs, outputs. Enriched semantic specification including preconditions and effects are not considered. Moreover, the focus is again on search rather than precise matching.

Apart from heavy focus on automated matching, functional matching operations proposed have remained at the abstraction level of ontological concepts alone. Matching operations at the level of services have been restricted to attempts at defining equivalence operation [6, 18, 15]. Even there, the notion of an operation with services as operands does not find emphasis. For instance, the last step of service matching involves ranking of matched advertisements and requires computing an aggregate measure for similarity of the requested and advertised services. For that, [18] specifies sorting rules for individual input and output attributes alone and [6] does not consider ranking of matched services.

WS-Agreement based matching of providers and consumers is presented in [17]. The authors propose extensions to WS-Agreement schema that impose a structure while maintaining its original flexibility, as well as add semantics to enable reasoning. However, since WS-Agreement is meant for stating the assurances and requirements of Web services, the matching presented is primarily non-functional matching. In addition, they utilize Semantic Web technologies for enabling rich and accurate matches.

[20] have used information retrieval techniques to rank similar terms for dealing with ontology mismatch and related issues faced while matching for services. However, the focus is towards parameter matching rather than enabling a service level matching operation.

6 Conclusions and Future Work

In this paper, we proposed techniques for enabling services to be treated as first class language level entities. There are several benefits that can be derived from this approach. First of all it isolates service oriented devel-

[5] http://www.alphaworks.ibm.com/tech/snobase

opers from the dynamics of the runtime environments. Second, it enables fast and scalable discovery of services during invocation. Third, it enables developers to write programs and policies in terms of abstract service types rather than actual instances resulting in much more robust programs and widely applicable policies. Finally, it fuels the need and takes a first step towards a service oriented software development methodology.

We presented and prototyped different matching operations. In future, we intend to demonstrate the benefits of these in the context of a real life scenario and perform a study of productivity, performance and other gains enabled or losses induced by the proposed approach.

Acknowledgment

The authors would like to thank Dr. Biplav Srivastava for his comments and Rakesh Bangani, IIT Madras for adding support for OWLS v1.1 in Snobase.

References

[1] S. Agarwal and A. Ankolekar. Automatic matchmaking of web services. In *WWW*, 2006.

[2] V. Agarwal, K. Dasgupta, N. Karnik, A. Kumar, A. Kundu, S. Mittal, and B. Srivastava. A Service Creation Environment based on End to End Composition of Web Services. In *Proceedings of the 14th International World Wide Conference*, May 2005.

[3] T. Banks. Web Service Resource Framework (WSRF) - Primer v1.2. http://www.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf, May 2006.

[4] G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Trans. Program. Lang. Syst.*, 17(3):431–447, 1995.

[5] X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang. Similarity search for web services. In *Proceedings of the 30th VLDB Conference, Canada*, 2004.

[6] P. Doshi, R. Goodwin, R. Akkiraju, and S. Roeder. Parameterized Semantic Matchmaking for Workflow Composition. Technical Report RC23133. Available at http://dali.ai.uic.edu/pdoshi/ research/RC23133.html, March 2004.

[7] C. Facciorusso, S. Field, R. Hauser, Y. Hoffner, R. Humbel, R. Pawlitzek, W. Rjaibi, and C. Siminitz. A Web Services Matchmaking Engine for Web Services. In *Proceedings of 4th Intl. Conf. on e-Commerce and Web Technologies*, September 2003.

[8] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. http://www.globus.org/research/papers/ogsa.pdf, January 2002.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[10] P. Giambiagi, O. Owe, A. P. Ravn, and G. Schneider. Language-Based Support for Service Oriented Architectures: Future Directions. In *Proceedings of the 1st International Conference on Software and Data Technologies (ICSOFT 2006), Portugal*, Sept 2006.

[11] N. Kokash, W.-J. van den Heuvel, and V. D'Andrea. Leveraging web services discovery with customizable hybrid matching. In *IEEE International Conference on Service Oriented Computing (ICSOC)*, 2006.

[12] A. Kumar, A. Neogi, and D. J. Ram. An OO Based Semantic Model for Service Oriented Computing. In *Proceedings of IEEE International Conference on Services Computing (SCC), Chicago, USA*, Sept. 2006.

[13] A. Kumar, B. Srivastava, and S. Mittal. Information Modeling for End to End Composition of Semantic Web Services. In *Proceedings of ISWC, Ireland*, Nov 2005.

[14] B. Liskov and J. Wing. Family values: A behavioral notion of subtyping. Technical report, Cambridge, MA, USA, 1993.

[15] X. Luan. *Adaptive Middle Agent for Service Matching in the Semantic Web: A Quantitative Approach*. PhD thesis, Dept. of CS and EE, Univ. of Maryland Baltimore County, 2004.

[16] J. Munkres. Algorithms for the Assignment and Transportation Problems. *Journal of the Society of Industrial and Applied Mathematics*, 5(1), 1957.

[17] N. Oldham, K. Verma, A. Sheth, and F. Hakimpour. Semantic WS-agreement partner selection. In *Proceedings of the 15th International Conference on World Wide Web (WWW), Scotland*, May 2006.

[18] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara. Semantic matching of web services capabilities. In *Proceedings of the First International Semantic Web Conference, LNCS 2342*, pages 333–347, 2002.

[19] C. Platzer and S. Dustdar. A Vector Space Search Engine for Web Services. In *Proceedings of the Third European Conference on Web Services (ECOWS)*, 2005.

[20] T. Syeda-Mahmood, G. Shah, R. Akkiraju, A. Ivan, and R. Goodwin. Searching Service Repositories by Combining Semantic and Ontological Matching. In *IEEE International Conference on Web Services (ICWS)*, 2005.

[21] K. Verma, R. Akkiraju, and R. Goodwin. Semantic Matching of Web Service Policies. In *Proceedings of Second International Workshop on Semantic and Dynamic Web Processes (SDWP)*, 2005.

[22] A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, 1997.