IBM Research Report

SOPI : An Object Oriented Semantic Web Programming API for Services Computing

Arun Kumar
IBM Research - India
ISID Campus, 4, Block - C,
Institutional Area, Vasant Kunj,
New Delhi - 110 070, India

Himanshu Chauhan
IBM Research - India
ISID Campus, 4, Block - C,
Institutional Area, Vasant Kunj,
New Delhi - 110 070, India

D Janakiram
Software Systems Lab
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.
SOPI: An Object Oriented Semantic Web Programming API for Services Computing

Arun Kumar
IBM Research India
4, Block C, Vasant Kunj Institutional Area
New Delhi 110070, India
kkarun@in.ibm.com

Himanshu Chauhan
IBM Research India
4, Block C, Vasant Kunj Institutional Area
New Delhi 110070, India
himchauh@in.ibm.com

D. Janakiram
Software Systems Lab
Dept. Of Comp.Sc. & Engg
IIT Madras
Chennai - 600036, India
djram@iitm.ac.in

ABSTRACT

Services Computing is fast turning into the mainstream programming paradigm for building enterprise systems that are distributed in nature. However, the programming power available to the developers of service oriented systems has been slow to catch up with the advances in technology. Object oriented APIs continue to be the prevalent mechanism for implementing web services based software systems. This creates a gap resulting from the absence of an abstraction that can model services at the language level and yet be able to meet the high requirements of the programming paradigm enabled by the concept of services.

We present SOPI, a semantic web based Service Oriented Programming API that bridges this abstraction gap by providing services as first class entities. It enables it in a fashion that makes automatic discoverability, invocation and composition possible at the language level. SOPI offers major operations for service manipulation including creation, discovery, invocation, composition, and inheritance. It relies on the use of semantics and object oriented design principles to achieve that. We present our design and implementation of the API.

1. INTRODUCTION

Creating manageable applications in the presence of heterogeneity, and maximizing software reuse to reduce time to market are some of the very important concerns for enterprise system designers of various organizations. Service oriented computing promises to ease out these problems by virtue of creating an infrastructure of loosely coupled business components residing in heterogeneous frameworks.

The success of this architectural paradigm has lead to several research efforts in the recent past that explore various ways of creating new services (especially from existing ones) and increasing the level of automation in service discovery, invocation, composition and interoperation. These fall along two prominent directions. The first one is about a distributed programming approach of specifying web services [4] through a well defined interface definition language in the form of Web Service Description Language (WSDL) [23] and the associated XML based protocol for information exchange called Simple Object Access Protocol (SOAP) [21]. This approach enables programmatic discovery, invocation and composition of services in a service oriented system. The second direction of research is aimed towards automating the steps of discovery, invocation, and composition of services through intelligent agents. These developments proposed under the umbrella Semantic Web technologies adopt a formal, logic based approach to specification of services including their interface, behavior, and process model [17].

Despite these advances in underlying technologies, there is a clear lack of abstraction available to developers building service oriented software systems. They are required to map high level service requirements coming from the business to programming constructs available in current object oriented languages such as Java and C# [7]. Further, they are forced to take into consideration dynamics of the runtime environment since services are actively running components rather than passive function/class libraries [11]. This highlights mismatch between the needs of services software developer and the programming models available today.

In this paper, we propose SOPI - a Service Oriented Programming API that is based upon semantic web technologies and incorporates object oriented design principles. Implicitly induced by the API is a programming model that proposes to stage the service oriented software development process into two stages. The first stage deals with specification of the service oriented program in terms service definitions of different kinds of services, available at compile time. The second stage deals with creation of an executable service oriented program that binds the specification of the program to available service instances ready to be used which match the specified (non-functional) requirements. This methodology segregates the compile time aspects of services software development from runtime aspects thus enabling improved automatic service discovery leading to robustness and efficient invocation and composition of services.

Specifically, the contributions of this paper are as follows:

- We present the architecture, design and implementation for enabling Service as a first class entity in a programming language leading to a new direction in service oriented programming. To the best of our knowledge, this is the first implemented proposal for realizing services as first class language level entities.
- We present the enhanced matchmaking of services based upon object oriented principles and using semantic web techniques. Specifically, we make use of OWL-S representation of services and utilize preconditions and effects expressed in SWRL rules to match requested
service behavior with available ones.

- We present an ontological two-level registry of services that utilizes the proposed matchmaking techniques to automatically build a hierarchical classification of services enabling faster and effective retrieval.

2. PROBLEM STATEMENT

The language level abstraction of an ‘object’ available to developers is not best suited for service oriented software development. There is a clear need to impart first class status to services, in prevalent object-oriented programming languages. In this section, we highlight the differences between the service oriented paradigm and the object oriented paradigms. We also discuss the challenges that need to be overcome to bridge this abstraction gap and overview some of the approaches that have attempted to do so.

2.1 Object Oriented Vs Service Oriented

We present a comparison of the two programming models based on following factors:

Level of Abstraction: In Object Oriented Software Development (OOSD), an object or a class instance is the basic unit available to software developers. It is a datastructure that captures the characteristics of a real world entity (rather than a business service) and is more closer to IT domain than to the business domain. For instance, a salary slip is more likely to be defined as an object rather than a payroll system. On the other hand, services are meant to capture the characteristics of and represent an entire business functionality without worrying much about how that functionality is realized.

In other words, services effectively capture the What of a business function whereas objects typically represent several components representing functionalities that come together to define How to realize that business function.

Runtime Environment: An object oriented program is typically meant to execute in a single runtime environment of the host language rather than span distributed hosting platforms. Services, however, are inherently distributed by definition and an end-to-end service invocation typically involves multiple, possibly heterogeneous, runtime platforms.

Level of Coupling: Due to their distributed nature, different services interacting with each other are loosely coupled. This means that a service oriented system may still continue to function, if one or more of the interacting services go down. Loose coupling allows a service client to switch to a new service instance with ease owing to well defined interfaces and the fact that services exist independently.

In contrast, the components in an object oriented system are very tightly coupled, executing within the same container and failure of one results in failure of its dependents.

Reuse: Object oriented programming allows different kinds of reuse. Reuse of code manifests itself in the form of reusable classes available as class libraries and also through implementation inheritance. This form of inheritance allows a derived class to be able to inherit the logic of the base class and also modify it, if needed. OO systems also support interface inheritance using the concept of behavioural subtyping [15] in which a derived class inherits (and possibly extends) the behaviour of the base class by conforming to the same interface definition. Component reuse and interface inheritance are also possible with services [12]. Implementation inheritance, however, is not available since services are not meant to expose their internal implementation.

Static Class Libraries Vs Dynamic Service Registries:

In OO systems, the developer has access to relatively static class libraries using which a client program is written. The components being shared are compiled class definitions available at development time, that do not change at runtime and are either built into the language api or made available as a distributable package.

Services, on the other hand, are active entities that get composed together at runtime. They are shared as components through service registries which maintain descriptions of services currently available for use. Service registries are dynamic in nature and change as often as new services come up or old ones go down.

Interface Definition Language (IDL) Vs Programming Language Constructs:

The current programming model for services is largely based upon the use of a standard IDL such as Web Services Description Language (WSDL) for discovering service interface and a standard XML based transport protocol to send invocation messages to the service. A developer needs to be familiar with the both of these to be able to effectively invoke a service.

Semantics: The semantics of a service are typically not available from the interface description as in WSDL, even though the service is meant to be discovered and invoked programmatically. In contrast, the semantics of an object being used is expected to be obtained from an API documentation. This is fair since objects are meant to be explicitly used and invoked through hand-coded programs as opposed to be discovered and invoked by software agents.

Due to the above mentioned differences, we need a language abstraction for representing services and their semantics to shield the service oriented architecture (SOA) developer from unnecessary details. At the same time it should expose service semantics to enable programmatic operations on services. The difference in the underlying principles and infrastructure make this a non-trivial issue.

2.2 Challenges

The primary hurdle against conceptualizing the notion of service as a programming language abstraction is the requirement of programmatic or automatic discovery [22], invocation, composition [1], orchestration and even recovery [3] of services. In OO languages, the responsibility of explicitly specifying an execution plan consisting of the exact objects to be instantiated, invoked and the sequence to follow, lies with the developer. Specifically, the following challenges arise:

Business Developer Vs IT Programmer: Given the difference in level of granularity between an ‘object’ and a ‘service’, a service oriented software development methodology should tend to be more declarative and closer to business user as compared to the object oriented approach which is more programmatic in nature. Enabling this with all the complexity involved is not straightforward.

Runtime: A service execution is split across the client runtime environment, the service registry and the service hosting environment. Unifying all the three through a programming construct is non-trivial.

Dealing with dynamicty: Services are typically hosted and offered autonomously and may come up or go down dynamically. This dynamic nature of services makes it extremely
hard to encapsulate them at the language level.

2.3 Existing Approaches

Some attempts have been implicitly made to address some of the challenges in a piecemeal manner. The ServiceJ [14] system proposes an extension of Java to enable support for Service-Oriented Computing in OO languages. To achieve that, it uses dynamic service selection and binding to handle volatility of services. It also deals with distributed nature of the service environment by offering a transparent fail-over mechanism that is configurable using declarative language constructs. In other words, the ServiceJ concept attempts to provide a language level representation of services. However, it does not fulfill the requirements identified in this paper, since it continues to offer an abstraction meant for the IT programmer rather than high level business process developer.

Zimmermann et al. [24] motivate the need for a Service Oriented Analysis and Design (SOAD) approach that leverages and builds upon existing approaches of Object Oriented Analysis and Design (OOAD), Enterprise Architecture frameworks and Business Process Modeling concepts. They recognize that the basic concepts of OOAD are applicable to SOA but on a higher level of abstraction than classes. Based upon this they identified the absence of support for cross platform inheritance and notion of first class service instance. Our proposed approach in this paper, fills in the gap identified by these practitioners.

3. OUR APPROACH

3.1 Services oriented development with object oriented methodology

Our premise is that for Service oriented computing (SOC) there is a lot to be leveraged from the research efforts and practical learnings that have gone into the object oriented paradigm. The key benefits of OO paradigm exemplified in the form of reuse, extensibility, reliability, maintainability and evolution are all needed for services as well. We believe that SOC can leverage the same architectural principles that have made object oriented software development as the most successful programming model till date.

In this paper, we provide a mechanism to achieve this amalgamation of the two paradigms and make it available in a programming language for direct use by developers. More specifically, we crystallize the object oriented abstraction principles of classification, composition and inheritance as introduced to SOC in [12] and make it available as an API to the service oriented software developer.

3.2 Semantic Web as the foundation

Capturing the semantics of a service in a programming construct and making it available to the developer is a key ingredient for making service oriented programs capable of performing automated operations. We make use of research accomplished in Semantic Web services 1 community to be able to do that. Further, we make use of Semantic Web Rules language (SWRL) 2 conditions which capture the semantics of behavioural interface of services. Finally, we utilize service matchmaking techniques based upon these semantic descriptions to enable a rich API for automated discovery of services in a scalable manner.

3.3 Jruby’s Meta-programming

To demonstrate the realization of concepts presented in this paper, we make use of Jruby’s 3 meta programming capabilities. Jruby is well suited for this task as it enables integration Java based software implementations while retaining the meta-programming capabilities of Ruby. Quick proto-typing and rapid testing of proposed constructs is the primary reason for us to choose a dynamic language instead of the traditional approach of creating a compiler for the new API in a programming language such as Java. This allowed us to focus more on the API design rather than the choice of platform to use for demonstration.

4. SOPI: AN API FOR SERVICE ORIENTED PROGRAMMING

In this section, we present the core elements of service oriented programming approach proposed in this paper. We introduce the representation of a Service in our framework, the programming model it induces and the API that the framework supports.

4.1 Service Representation

Based upon our previous work [10, 11, 12], we present our proposed representation of a Service that is split into two complimentary pieces rather than a single unit. The first half represents the core functional cum semantic description of a service that is independent of actual implementation. It is called Service Type and can be made available to a programmer in the software development phase. The second half of the representation, called Service Instance, captures non-functional cum operational description of the service and is bound to runtime characteristics of a particular instance of running service. This can be made available at runtime.

We use semantic web service technologies to realize the representation for a Service Type whereas the representation for Service Instances relies on web services description and protocols.

![Figure 1: ServiceType and ServiceInstance](http://www.w3.org/Submission/SWRL/)

As shown in Fig. 1, ServiceType is composed of two elements – a functional specification containing various interface descriptions, and constraints on non-functional capabilities of service instances. The functional specification prescribes the set of interfaces exported by any service that conforms to

---

1 http://www.daml.org/services/
2 http://www.w3.org/Submission/SWRL/
3 http://jruby.org/
this type. Each interface is essentially a semantic representation of a method and is described in terms of inputTypes to be supplied, the preconditions that must be satisfied before the method can be invoked, the outputTypes that can be expected as a result of the invocation and the results consisting of implicit effects that the method brings about in the environment where the service executes. Figure 3 gives an example of semantic description of a FlowerShop service. The entire functional specification in a ServiceType is based upon concepts defined in an ontology. This enables the software developers to write programs that can automatically reason upon the functionality of the service they intend to deal with. The non-functional requirements element is intended to capture the parameters and constraints on that instances of that ServiceType should satisfy.

Similarly, ServiceInstance is composed of three elements — a reference to its corresponding ServiceType, an operational specification of the service containing actual interfaces (corresponding to interface descriptions in ServiceType), and the set of non-functional capabilities of the instance. The operational description of the service consists of the exact interface that a client program needs to use to invoke the service. It specifies the inputs and outputs corresponding to inputTypes and outputTypes defined in the ServiceType. The ServiceInstance entity, in essence, acts as a proxy to the actual service that hides binding and protocol details from service clients and provides them a simple invocation interface. When used to invoke a method, a ServiceInstance implementation effectively collects the parameters supplied to it, validates them against the definition specified in the corresponding ServiceType and uses the binding information of the actual service instance to make a web service call to it.

For simplicity, we focus on a subset of this representation in this paper. Specifically, we restrict ourselves to a single interface both in ServiceType and ServiceInstance. Also, non-functional requirements/capabilities are not addressed in this paper.

4.2 Semantic Relations Between Services

Semantic representation of services, as crystallized in ServiceType, capture the entire behavior offered by the service and is amenable to be processed automatically. This property enables different services to be compared based upon their semantic descriptions.

Automatic service matchmaking has been an active problem of research in the recent past. Most of the existing literature so far has focused on comparing either only Inputs and Outputs parameters or simplistic preconditions and effects [13]. To the best of our belief, we present the first attempt towards matchmaking of services with rich preconditions and results expressed as SWRL rules also matched ontologically. Figure 3 presents the semantic match levels for input and output comparisons [13]. 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 Fig. 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 the advertised concept is contained by the requested concept, otherwise disjoint match is returned. [13] gives more details of match levels for expressions in preconditions and effects and how matchmaking across these different elements of a service description can lead to semantic matchmaking between services.

We now present our extended semantic match making approach which includes precondititon and result comparisons in addition to traditional input and output match techniques. The process of comparing service definitions begins by collecting the parameter sets of all the four categories inputs, outputs, pre-conditions and results from both the services. This process then passes lists of same category of these four scores. The pseudo-code below captures this approach in short:

```plaintext
MATCHSERVICES(service1, service2)
    int matchScore ← 10 > 10 stands for EXACT match > where as 0 represents a DISJOINT match
    for each type : [Inputs, Outputs, PreCons, Results]
        do params1 ← service1.getParamList(type)
           params2 ← service2.getParamList(type)
           typeScore ← MATCHPARAMLISTS(params1, params2)
           if typeScore < matchScore
               then matchScore ← typeScore
    return matchScore

The procedure called for computing comparison scores for parameter sets is MATCHPARAMLISTS. It iterates over both parameter lists and compares each parameter from first list against all from the second. Thus it generates an nxn matrix of scores. This score matrix is then passed on to another procedure FINDMAXIMALMATCH, which uses a maximal bipartite matching algorithm [18] to return the best match score for the matrix.
comparisons of precondition and result sets of services varies for Input and Output parameters. However, the algorithm described above is sufficient for performing first performs additional downward traversals to extract effect parameters from expression bodies, and then calls MATCH-PARAMLists with the extracted parameters.

MATCH-PARAMLists(paramList1, paramList2)

\[ \text{matrix} \leftarrow \text{int}[\text{paramList1.size}] \times [\text{paramList2.size}] \]

for \( i \leftarrow 0 \) to paramList1.size

do \( p1 \leftarrow \text{paramList1.get}(i) \)

for \( j \leftarrow 0 \) to paramList2.size

do \( p2 \leftarrow \text{paramList2.get}(j) \)

matchval \( \leftarrow \text{SEMANTIC-MATCH}(p1, p2) \)

\[ \text{matrix}[i][j] \leftarrow \text{matchval} \]

return FIND-MAXIMAL-MATCH(matrix)

The algorithm described above is sufficient for performing comparisons for Input and Output parameters. However, comparisons of precondition and result sets of services varies slightly, and uses an extended approach, which we call MATCH-PARAMsWithEffects. For such comparisons the algorithm first performs additional downward traversals to extract effect parameters from expression bodies, and then calls MATCH-PARAMLists with the extracted parameters.

MATCH-PARAMsWithEffects(paramList1, paramList2)

\[ \text{matrix} \leftarrow \text{int}[\text{paramList1.size}] \times [\text{paramList2.size}] \]

for \( i \leftarrow 0 \) to paramList1.size

do \( p1 \leftarrow \text{paramList1.get}(i) \)

for \( j \leftarrow 0 \) to paramList2.size

do \( p2 \leftarrow \text{paramList2.get}(j) \)

\( \text{effs1} \leftarrow \text{GET-EFFECTS}(p1) \)

\( \text{effs2} \leftarrow \text{GET-EFFECTS}(p2) \)

\[ \text{matrix}[i][j] \leftarrow \text{MATCH-PARAMLists}((\text{effs1}, \text{effs2})) \]

return FIND-MAXIMAL-MATCH(matrix)

Both the processes described above make use of procedure SEMANTIC-MATCH to perform semantic comparison on a pair of parameters. This procedure starts by getting semantic classes of parameters based on their semantic definitions in base ontology. After obtaining the class names/URLs, the procedure queries the ontology model to list all the statements having these classes as their subject and predicate values. Iterating over all such statements, the algorithm retrieves the ‘property’ from each of them, and determines if the relation represented by property value is stronger in as compared to the previous one. If a stronger relation is found, the algorithm assigns that as the result. At the end, after completing iterations over all the statements, the strongest relation found is returned.

SEMANTIC-MATCH(param1, param2)

\( \text{owlClass1} \leftarrow \text{param1.semanticClass} \)

\( \text{owlClass2} \leftarrow \text{param2.semanticClass} \)

\( \triangleright \) retrieve statements from ontology containing triplet:

\( \triangleright \text{owlClass1} \text{ relatedTo } \text{owlClass2} \)

\( \triangleright * \) is used as a wildcard to fetch all the statements

\( \text{iterator} \leftarrow \text{ontology.listStatements(owlClass1, *, owlClass2)} \)

\( \text{result} \leftarrow \text{DISJOINT} \)

while iterator.hasNext

\( \text{do} \)

\( \text{statement} \leftarrow \text{iterator.next} \)

\( \text{relation} \leftarrow \text{statement.getProperty} \)

if relation > result

then result \( \leftarrow \text{relation} \)

return result

4.3 A Programming Model for SOSD

In the current practice, a service oriented software developer binds the client program to the actual service to be invoked, at development time itself. This is because, the representation of an invocable service available to the developer today is the WSDL that is an interface description of a live service. Programming to an implementation (rather than just the interface) as is done today results in brittle programs that would break if the bound service goes down, changes its location or deteriorates in performance. Frameworks such as Web Service Invocation Framework (WSIF) [6] enable late binding of a service to its client program as long as the instance being bound conforms to the same interface as in WSDL description used for writing the client program. The port at which the service instance runs or the protocol used to access it may vary at deployment time. However, ideally the late binding mechanism should allow other details of a service to vary as long as the semantics of the actual service being used at runtime conforms to (and is compatible with) the semantics of the original service using which the program was created. This implies that not only the location of the service but also the datatype of the parameters passed to methods as well as the number of parameters etc. could vary. This increased flexibility can increase the robustness of an SOA manifold especially if the range of compatible alternate services could be discovered and invoked automatically thereby enabling self-healing. Figure 4 shows the programming model that gets induced by the representation of a service presented in the previous subsection and enables such late binding as discussed above. The model presented here decouples the client program from the actual service instances. A library of ServiceType descriptions is made available to the software developer (possibly incorporated into an integrated development environment). This library, which is similar to class libraries available in object oriented languages such as Java, is called ServiceType Registry. Description of service instances is stored in a complimentary part of service registry called Service-Instances Registry. A default ServiceInstance description corresponding to each ServiceType is also contained in the ServiceType registry. This is a dummy description as it does not refer to a real available service but it is used for initialization.

The SOPI API essentially makes use of this library of ServiceType descriptions as well as ServiceInstance descriptions and creates a wrapper around it to expose service as a first-class representation language construct. It exposes a MetaService construct that enables creation of new ServiceTypes. This allows the developer to define new kinds of
services. A ServiceType when bound to a ServiceInstance provides the full realization of a Service as an object. This Service object encapsulates the ServiceType as well as the ServiceInstance bound to the ServiceType. It exposes a semantic interface to query and possibly manipulate the semantic definition. It also exposes the operational interface of the ServiceInstance. In other words, Service acts as a proxy to the actual instance running elsewhere and provides the interface to invoke methods on the actual instance.

As shown in the figure, the developer creates client programs by initializing ServiceType with a dummy ServiceInstance description without worrying about the nuances of runtime environment and specific characteristics of actual running instances. This deployable program can then be executed in a services runtime environment. It is the responsibility of the runtime system to update the binding of ServiceTypes in the program to conforming ServiceInstances that best match the non-functional criteria specified in those ServiceTypes. For this purpose, the runtime system does a search on the ServiceInstance registry for ServiceInstance that either directly conforms to the ServiceType or can be used in its place by virtue of having been derived from that ServiceType. Once such a semantically matching ServiceInstance is found, the retrieved ServiceInstance is bound to the ServiceType in the program and the program executed else the execution fails in absence of a real running instance of the desired service.

For other situations, such a plugin match, developer/user input becomes essential.

In essence, the client program is tightly coupled with a ServiceType and is bound to a semantically compatible ServiceInstance satisfying non-functional criteria, only at runtime. It is this flexibility in binding that makes the system robust and shields it from dynamics of runtime environment since actual instance to be invoked is not bound at compile time.

4.4 The SOPI API

To support the programming model presented above, we propose a set of API calls for the developer that are convenient to use, are declarative in manner and yet enable important operations over services.

4.4.1 Service Creation

We assume that various services would be developed and hosted by various autonomous entities in different administrative domains. Service creation in a client program then, is essentially an operation that creates a language level representation of the desired service and at an appropriate time gets bound to a running ServiceInstance hosted elsewhere. The latter step is essentially instantiation of the ServiceType, in our terminology.

Service creation API is exposed by the MetaService construct available as part of the core language functionality. The creation is achieved either by instantiating existing ServiceTypes or by first creating a new ServiceType and then instantiating it.

MetaService.createServiceType(serviceName, serviceTypeURI):

4.4.2 Service Discovery

Automatically discovering instances of desired service is a key promise of service oriented computing. The ServiceType descriptions enable semantic search for desired services without having to look through the entire world of running instances. Having identified the exact ServiceType one needs, the search needs to be restricted to only those ServiceInstances that are labelled as instances of that ServiceType. This imparts scalability to the discovery process as it scopes down the search to a much small set by splitting it into two phases. In the first phase, the search is restricted to ServiceTypes registry whereas in the second phase it is restricted to only those entries of ServiceInstance Registry that are instances of the given ServiceType or those of semantically compatible ServiceType.

Apart from MetaService construct the developer has access to the ServiceTypes Registry for creating client programs. The discovery are defined over the Service Registry.

Registry.findServiceType(serviceTypeName, relation, doStrictMatch):

Finds ServiceTypes whose semantic match with serviceType produces a match of strength relation or better, based on the value of boolean variable doStrictMatch. For example, if doStrictMatch is set to false (which is default behaviour) and if the passed argument relation denotes a 'Sub-Class' relation, this method returns all service types whose semantic comparison with serviceType shows 'SubClassOf' or stronger 'Exact' match. However when it is set to true the method returns only those service types that have their semantic comparison result with serviceType as 'Sub-ClassOf'.

Registry.findAlternateService(serviceTypeName, relation, doStrictMatch):

This method internally invokes the findServiceType and findServiceInstance methods and return an alternate ServiceInstance matching the ServiceType functionally as well as non-functionally.

4.4.3 Service Invocation

Service.methodName(parameterList): This enables invocation on the actual service through the API. This is the real use of the API, as it presents an abstract interface for the actual service. If the service provides any management interfaces such as for starting pausing, shutting it down, those can be exposed as well.

4.4.4 Query APIs

ServiceType.getMatchLevel(serviceType): This operation compares two service types and indicates the semantic degree of match between those. The algorithm used to com-

---

4.4.1 Service Creation

MetaService.createServiceType(serviceName, serviceTypeURI):

---

4.4.2 Service Discovery

Registry.findServiceType(serviceTypeName, relation, doStrictMatch):

Finds ServiceTypes whose semantic match with serviceType produces a match of strength relation or better, based on the value of boolean variable doStrictMatch. For example, if doStrictMatch is set to false (which is default behaviour) and if the passed argument relation denotes a 'Sub-Class' relation, this method returns all service types whose semantic comparison with serviceType shows 'SubClassOf' or stronger 'Exact' match. However when it is set to true the method returns only those service types that have their semantic comparison result with serviceType as 'Sub-ClassOf'.

Registry.findAlternateService(serviceTypeName, relation, doStrictMatch):

This method internally invokes the findServiceType and findServiceInstance methods and return an alternate ServiceInstance matching the ServiceType functionally as well as non-functionally.

4.4.3 Service Invocation

Service.methodName(parameterList): This enables invocation on the actual service through the API. This is the real use of the API, as it presents an abstract interface for the actual service. If the service provides any management interfaces such as for starting pausing, shutting it down, those can be exposed as well.

4.4.4 Query APIs

ServiceType.getMatchLevel(serviceType): This operation compares two service types and indicates the semantic degree of match between those. The algorithm used to com-
pute semantic match between service types is described in
detail in section 6.3. \texttt{ServiceType.isEquivalentTo(a)}: Compares the service type instance \texttt{a} with the instance it is
called upon and returns whether both the instances are se-
manetically equivalent or not. This method uses \texttt{getMatch-
Level(a,b)} method described above to perform the compar-
isons. However, before calling this method, it also checks
if the invoking instance and the instance passed as argu-
ment are both same, in which case there is no need to per-
form further comparisons and this method reports them as
equivalent.
\texttt{ServiceType.isDerivedFrom(a)}: Determines if a given ser-
vice type instance is derived from (is subclass of) another
service type. The method queries for semantic match level
of the two instances by calling \texttt{getMatchLevel}, and if the
match level indicates that service type of \texttt{a} is superclass of
that of invoking instance this method returns \texttt{true}.
\texttt{ServiceType.isSuperServiceOf(a)}: This method acts as an
inverse of \texttt{isDerivedFrom}, and determines if service type of
invoking instance is superclass (based upon semantic defini-
tions) of service type of instance \texttt{a}.
\texttt{ServiceType.isDisjointWith(a)}: Checks if service type defi-
nitions of invoking instance and \texttt{a} are semantically disjoint.

5. SYSTEM ARCHITECTURE

5.1 System Overview

Development Environment: As shown in Figure 5, IDE is
the interface through which the services software developer
is exposed to SOPI API. It could be support for Services
in traditional programming languages such as Java or a Do-
main Specific Language or, as in our case, an API in dynamic
language (we provide SOPI as an api in Jruby as presented
later in this paper). The development environment inter-
faces with the Services Registry to retrieve ServiceTypes
and make them available to the developer.
Service Runtime Environment: This is what a service con-
sumer uses to execute a client program that invokes one or
more services. The client program created in the develop-
ment environment would have ServiceType variables bound
to dummy ServiceInstances. The runtime evaluates the best
available ServiceInstant from the underlying Service Reg-
istry that satisfy the non-functional criteria specified.
Service Registry is a key component of the SOPI Archite-
cture and we present its details in the next subsection. The
registry administrator and crawler programs are responsible
for keeping the registry updated based upon services avail-
able in the internet. Details of interfaces for these are not
covered in this paper.

5.2 Service Registry

We present a novel approach of capturing semantic as well
as operational interface description of services in a registry.
Essentially, a services registry in SOPI is a composition of
two separate but interlinked sub-registries. The first one,
called \texttt{ServiceTypes Registry (STR)}, consists of descriptions
of various ServiceTypes along with description of a conform-
ing dummy ServiceInstance. STR is available to program-
ners at development time. This is similar to availability
of class definitions in Java class libraries using which the
programmer creates objects in a program. Typically,
the description consists of semantic specification of the service
interface in terms of its inputs, outputs, preconditions and
results based on an ontology. ServiceType descriptions in
SOPI architecture are represented as OWL-S files.
The second sub-registry, called \texttt{ServiceInstance Registry}
consists of descriptions of ServiceInstances that capture details
of actual running instances including information required
to invoke those services. Specifically, this may include infor-
mation about the service invocation interface, the protocol
to be used to invoke it, the QoS parameters supported by
the service instance and the URL where it is available. It
is this second kind of registries that have been proposed so
far in the literature and are in use today in service oriented
runtime environments. ServiceInstance descriptions in SOPI
architecture are represented using Web Service Description
Language (WSDL). Since much work has been devoted to
such registries, we do not discuss them in detail in this paper.
As shown in Fig. 5 as well as Fig. 4, the resulting Services
Registry provides search on types of services as well as indi-
vidual instances. The STR part of the registry is made use of
by the developer at program development time whereas
SIR is used by the runtime to look for an appropriate Servi-
cInstance to bind. New entries in STR and SIR could either
be added automatically by a program or manually by a reg-
istry administrator. Next we describe details of ServiceType
Registry.

\texttt{ServiceType Registry}

As mentioned, STR contains semantic description of services
capturing their behavior interface. This description is suffi-
cient to determine the capabilities of a service and reason on
it. Since ServiceTypes are available at development time, it
implies that it is possible to compare and contrast capabil-
ties of different kinds of services in an offline environment.
This saves precious matchmaking time at runtime.
Furthermore, as a result of the comparison between differ-
ent ServiceTypes (typically, while adding a new type to the
registry) it computes and captures different relationships
among ServiceTypes to \texttt{automatically} build an ontology of
ServiceTypes. The matchmaking algorithms presented ear-
lier, let the registry determine whether newly added service
is related to an existing one through one of the semantic
relations introduced in semantic relations section including
equivalence, plugin, composite, supertype, subtype and dis-
joint.
Such an ontology of ServiceTypes enables an \texttt{ontological}
search for services to be performed by referring to their on-
tological names. This not only prevents having to match
services by comparing their interface descriptions each time but also enables richer service retrieval through queries over the relationships between those ServiceTypes. Not having to specify full descriptions for search makes service discovery more scalable than what is possible today.

STR consists following components:

**Service Loader** parses ServiceType descriptions and creates an object corresponding to each of those. The Service Loader performs primary validations on semantic descriptions as well as ensures that registry does not load multiple instances of same type description.

**Matcher** Given a pair of service objects, this module compares service components and calculates degree of match on component level as well as overall match scores between the two services.

**Discovery Manager** On invocation of query methods Discovery Manager traverses its internal graph representation of ServiceType ontology and repeatedly calls Matcher to discover ServiceTypes which match the discovery criteria.

6. IMPLEMENTATION

In this section, briefly describe different aspects of our prototype implementation of SOPI API. ServiceType descriptions are created using OWL-S and we use Jena’s OWL API and OWL-S API (both for Java) to parse and interpret these descriptions. Conditions in preconditions and results of OWL-S descriptions are encoded using Semantic Web Rules Language (SWRL). The language level constructs are implemented using Jruby’s metaprogramming capability which is integrated well with the Java code.

![Diagram of SOPI Implementation](http://example.com/diagram.png)

**Figure 6: SOPI Implementation**

6.1 JRuby Metaprogramming Layer

We define a MetaService Class in JRuby that has a create-ServiceType method with serviceTypeName and OWLS_URL as parameters. This method creates a subclass of MetaService Class which when instantiated has the entire semantic interface of the ServiceType available through methods. The methods corresponding to individual inputs, outputs, preconditions and results are prefixed I, O, P and R respectively. This subclass of the MetaService class with methods defined according to the semantic description specified, is the language level representation of that ServiceType. This can be used to create different instances of this ServiceType. Similar to ‘create’ the MetaService class has an instantiate method with WSDL_URL as a parameter. This method invokes instance_eval() which is meta-programming method in Jruby used for inserting class level methods dynamically sat runtime. So, a portType/message part should have an equivalent instance method to that is invokable.

This dynamically modified ServiceType object with new added interface corresponding to a ServiceInstance is the complete first class representation of a Service. It encapsulates both - the ServiceType interface as well as the ServiceInstance interface.

6.2 Service Representation and Matchmaking

We used OWL-S Jena API \(^5\) to parse ServiceType description available as OWL-S files and converted them to in-memory object instances. This object encapsulates the OWL-S representation and makes it available to the Jruby metaprogramming layer described above. We enhanced the service matchmaking algorithm implementation as presented in [11] to include the first ever preconditions and results based semantic matching with SWRL rules rather than simple ontological concepts, to the best of our knowledge.

Our initial implementation used SWRL API from CMU for parsing SWRL rules embedded inside OWL-S descriptions. However, the API turned out to be incomplete in functionality as it was unable to parse the SWRL rules in their entirety. Specifically, effect objects from the rule definitions could not be parsed successfully. For such requirements we have extended it for parsing the definitions and generating effect objects. Semantic reasoning on ontological models remains the most basic requirement for our approach, and for all such reasoning tasks we make use of Jena API, which allows interactions with ontology in an object oriented manner. All these layers put together form the base for our API layer implementation.

6.3 Ontological Discovery Of Services

We implement ServiceTypes Registry by using a map data structure called typeMap which holds type names (as keys) against a bucket (a list) of profile URLs bound to these names and an ontology of type names, called typeOnt containing their semantic relations. Using this setup, we implement the two primary operations addServiceType and findServiceTypes in following manner:

Adding a new service type to registry entails invoking addServiceType with a typeName and a serviceURL. On invocation, the method first validates parameter values. In the next step, the key set of typeMap is then looked up for typeName: (a) if this name is present in the set, the new service description (specified by serviceURL) is matched semantically against a description from the bucket in map, that is stored against the typeName. The result of this comparison is called relation. If it denotes an ‘Exact’ match, serviceURL is added to the existing bucket in the map. In case relation is not ‘Exact’, we replace typeName with an auto-generated unique type name uName and then follow steps same as (b).

(b) if the name is not present in key set then buckets against each key are iterated upon, and each profile from each bucket is compared semantically with profile from the argument. We use a variable bestMatch to retain the strongest matching profile found.

After each semantic comparison, its result is observed and compared with bestMatch. If this result denotes a stronger match, bestMatch is set to result. At any stage if it denotes an ‘Exact’ match, serviceURL is added to the existing

\(^5\)http://jena.sourceforge.net/
bucket in the map from which this match was found, and the iterations are terminated. If termination occurs by ‘Exact’ match, typeOnt is updated with the statement denoting the two profile types as equal, and the procedure finishes. Whereas if the relation of result is not ‘Exact’, we carry on iterating and updating bestMatch. After completion of iterations, and no ‘Exact’ match being found, we add an entry to typeMap having typeName as key having its value set to a new bucket containing serviceURL. Then typeOnt is updated with statement like : typeName has relation with bestMatch. In case serviceType is not provided, we use an auto-generated unique name and then execute the above steps.

Searching for a service type involves a similar procedure as described above. The difference is that in search operations, we do not perform add operations on typeMap but we do update typeOnt with all the relations found during the search. Also, search operations return the service type having best possible semantic match, and other service types which are equivalent to that service type (deduced from ontological reasoning).

7. EVALUATION

In this section we focus on evaluating the performance of our registry implementation under the combined effects of proposed service match-making approach and registry design. For our experiments we use OWLS-MX\textsuperscript{6} [8] dataset of service profiles. We carry out two set of experiments. First, we measure cost of add operations by adding services to registry, for which we add 200 service profiles to it. Second, we perform a set of experiments to compute the cost of query operations on this populated registry, for which we query a profile and to retrieve all matching profiles and their bindings. For both these sets, we also demonstrate the benefit of using the proposed concept of ServiceTypes by calling addService and findService operations first with manually assigned ServiceType names and then without ServiceType names provided. Same service is queried for in both cases. When a type name is not provided, the registry uses an auto-generated name and returns this name on completion of operation.

Figure 7 presents time variance of add operations on registry as its size and internal ServiceType ontology grows. It is evident from this figure that when a type name is known for a profile being added, the procedure takes significantly lower time. Figure 8 presents the times taken to retrieve same number of matches from the registry for a given query profile. As is intuitive from the map based implementation of registry, this figure also confirms that query operations finish in $O(1)$ time when the service type is known for desired service profiles.

8. RELATED WORK

Service matchmaking has been researched a lot in recent years both in the context of syntactic (WSDL) matching as well as for semantic matching. Kokash, et al. [9] present a survey and comparison of different approaches. OWLS-MX [8] provides a framework for hybrid semantic Web service matchmaking. It utilizes both reasoning techniques as well as traditional information retrieval techniques for service matchmaking. However, the thrust has been towards automated matching rather than enabling it for a web services developer.

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 [19, 16]. Even there, the notion of an operation with services as operands does not find emphasis.

Several researchers have either extended or felt the need of applying some of these OO concepts that are currently missing from SOC [2, 20]. The ServiceJ [14] system proposes an extension of Java to enable support for Service-Oriented Computing in OO languages. It uses dynamic service selection and binding to handle volatility of services. Our approach starts by first defining the right level of abstraction needed for service oriented computing and then goes on to provide language level constructs needed for services based programming. Papazoglou [20] provides a detailed comparison between services in SOA and objects in OOAD. However, they felt that concepts like polymorphism etc. are not applicable in SOA.

\textsuperscript{6}http://projects.semwebcentral.org/frs/download.php/255/owls- tc2.zip

Figure 7: Cost of addService() operation

Figure 8: Cost of findService() operation
manipulation and uses a messaging paradigm. Semantics of service descriptions and automating service operations is not considered.

9. CONCLUSION

There are two main contributions of this paper. First, it provides a mechanism to be able to offer a service abstraction to the developer by utilizing semantic web and web service technologies. We demonstrated it with an API that is made available to the developer by exploiting the metaprogramming capabilities of a dynamic language. This mechanism induces a new programming model as illustrated in the paper. Second, it makes use of this proposed programming model and enhanced matchmaking techniques to present a Service Registry, part of which is ontological in nature. The ontology of services is built automatically from service descriptions and is available to service programmers at development time. By virtue of its design it imparts scalability to the service search process which in the case of dynamicity of web services, often lies in the path of service invocation.

In future, we plan to incorporate support for service composition as proposed in [1] into SOPI API. While we chose to exploit JRuby’s metaprogramming facility for quick prototyping, adding support for the proposed model in Java is another direction we wish to pursue. Finally, we intend to build upon this work and add service instance matching functionality to our system to provide an end-to-end integrated development environment for service oriented software development. This would require, among other things, adding support for non-functional requirements matching.

10. REFERENCES


