Autonomic Service Adaptation in ICENI using Ontological Annotation

Jeffrey Hau, William Lee, Steven Newhouse
London e-Science Centre,
Imperial College London,
180 Queen's Gate, London, SW7 2AZ, United Kingdom
lesc@imperial.ac.uk

Abstract

With the advent of web services standards and a service-oriented Grid architecture, it is foreseeable that competing as well as complimenting computational services will proliferate. Current efforts in standardising service interface focuses on how one can execute these services in terms of their syntactic descriptions. Their capabilities and relations with other service types are only articulated through natural language in the form of documentation. In this paper, we seek to capture the capability of services by annotating their programmatic interface using the Web Ontology Language (OWL)[2] in relation to some domain concepts thereby allowing services to be semantically matched based on their ontological annotation. By inferences on this metadata, syntactically different but semantically equivalent service implementations may be autonomously adapted and substituted. We will conclude by applying this independent annotation to Java RMI and WSDL[8] service interface to show the autonomic adaptation process over multiple service oriented-architectures. Combining it with familiar high-level programming language, we demonstrate a practical service-oriented programming model.

1. Introduction

The World Wide Web has evolved from being a collection of hyperlinked documents to a platform delivering networked services. The e-Commerce community needs to integrate heterogeneous distributed systems within their supply-chain while the Grid Computing community must access federated and widely-distributed computing resources in a scalable, secure and coordinated manner. By representing these system units as well-defined services, we facilitate the reuse and substitution of compatible components into the workflow of a larger system.

Compatible services are often identified through subclass relationship or interface inheritance. Substitutable implementation are reused through polymorphism of superclass or abstract interface. Far from being an autonomous operation, the burden of using binding specific tools or APIs to generate stubs or perform remote invocation has added complexity in consuming remote services. This approach to identifying reusable services is insufficient in a service-oriented architecture. Multiple service providers might not agree on a single service interface hierarchy for competitive reason. These services are deemed incompatible although they potentially provide the same functionality. The intended capability of contemporary services are mostly expressed in natural languages, which hinders automated matching because of potential ambiguity and incompleteness. Research in formal methods has explored the precise description and intention of interfaces. Even though the task of developing formal specification is overwhelming and the need for program verification deemed unnecessary for many applications.

In this paper we present a framework for identifying compatible services based on their expressed capabilities in relation to some ontological concepts. By associating service operations to concepts and parameters to properties of the concepts, we demonstrate these semantic metadata, which links the syntactic elements of the service to the domain ontology, can be inferred to identify compatibility. We argue that the intelligence provided by the inference can identify the points of compatibility between the client requirement and the service. In addition, the middleware needs to possess a set of transformation rules for adapting and proxying the service consumable by the client.

1.1. Background

The Web Services Standards[9][10][8] collectively model the publish, find and bind operations of the service oriented architecture. The Web Services Description Language (WSDL) captures the programmatic inter-

face, network binding and message formats of the service irrespective of their implementing technologies. Its extensible nature is well-suited to act as a common language for describing service interface encoded in CORBA IDL, Java Interface or other proprietary interface description language. Nonetheless, it only expresses the syntactic specification of the service in terms of the name of the methods, as well as the expected types. The Open Grid Services Interface[11] (OGSI) builds on the relative maturity of WSDL with the added notion of transient service. It mandates a set of core interfaces and service data to reflect the need for a set of fundamental services in a Grid environment. It implants limited semantics to all Grid services in terms of the ability to query state and control life-cycle. However, custom services have to resort to the traditional methods to express their capability.

Research in Autonomic Computing[16] aims to transform computing tasks that require constant human interference and awareness, to be self-managing, self-healing and self-optimising. Efforts in the Semantic Web[4] has contributed standards and tools to autonomically extract structured information from the World Wide Web. As the Internet is evolving into a platform for service delivery, we would not only like to search and identify related service but to consume the service according to their service interface.

2. Related Work

2.1. Service Ontology

DAML-S[21] is an upper ontology for describing Web Services written in DAML+OIL. The three aspects of DAML-S - service profile, the process model and the service grounding, aim at making different aspects of Web Services computer-interpretable and hence enabling tasks such as invocation, discovery, verification etc. The framework presented in this paper annotates services using a toy service ontology written in OWL. Our framework has no assumption about the type of service ontology used. DAML-S could be easily adapted as our service ontology.

2.2. Semantic Matching

There are currently much research into the area of semantic service matching using ontologies. Projects such as [24], [26] and [25] all take advantage of ontological metadata to enhance current resource description. Reasoning systems are then used to provide inference capabilities to match resources semantically. However there is limited discussion on how to invoke syntactically incompatible but semantically equivalent services.

3. The Metadata Space

Expressing resource semantics on the World Wide Web has been the driving force behind the recent development of the Semantic Web. The Grid community is exploiting recent development in grid computing technologies and service oriented architecture to build the Semantic Grid[5], an grid infrastructure with resource and services semantics. In this paper we propose the concept of *metadata space* as realisation of the Semantic Grid. The separation of the metadata space from the physical grid in figure 1 clarifies the difference between service metadata and its implementation. In the metadata space, the usual manual extraction of service semantics becomes an autonomic process. Each grid resource is represented by its semantic annotation and thus resource interaction is focused purely on the semantic descriptions.

More specifically, the metadata space is an environment with a standard metadata publication and discovery protocol to facilitate the processing of metadata and semantic interaction between grid resources. The advantage of having the metadata space separated from the physical grid is to decouple grid resources from their implementation and hosting environment. Every participant in the metadata space is characterised as a metadata publisher. The metadata published by a publisher falls into one of the three categories - requirement, implementation or domain. We distinguish the publisher by their metadata category. This differs from the traditional service oriented architecture where participants can be characterised as either a client, a service, or a registry. The metadata published into the metadata space can then be processed by the Meta-Services to provide the autonomic semantic matching and interface adaptation. The following subsections discuss the different roles of publishers in more details.

3.1. Implementation Publisher

This role can be played by any service providers in the traditional service oriented architectures. The transformation form service to implementation publisher depends on the publication of its semantic annotation into the metadata space. An example of implementation publisher's semantic annotation is shown in figure 2[1]. The implementation provider behaves as a typical service provider within its hosting environment. It projects its semantic representation into the metadata space by the publication of its semantic annotation.

1 Following is a list of XML namespace prefix used in this paper rdf= http://www.w3.org/1999/02/22-rdf-syntac-ns#, owl= http://www.w3.org/2002/07/owl#, rdfs= http://www.w3.org/2000/01/rdf-schema#, art=http://lesc.ic.ac.uk/example/arithmetics

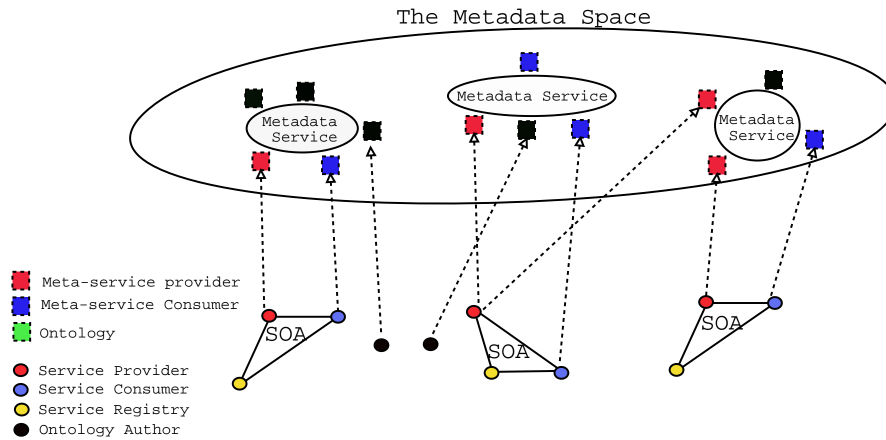

Figure 1. The metadata space in relation to different Grid SOAs

```
//Semantic Annotation
<rdf:Description rdf:about="..MathsService/add">
  <rdf:type rdf:resource="&art;Add"/>
  <art:hasOperand rdf:resource="../add/param/a"/>
  <art:hasOperand rdf:resource="../add/param/b"/>
  <art:hasResult rdf:resource="../add/return"/>
</rdf:Description>

//Java service implementation
public int sum(int[] ele) {
      //method body omitted
}
```

Figure 2. A Java service provider's semantic annotation in RDF

3.2. Requirement Publisher

A requirement publisher is any grid service consumer with the capability of publishing semantic annotation into the metadata space. The requirement publisher expresses its syntactic requirement in a programming language interface. Semantic requirement is then included into the source code through annotation. The syntactic service interface description has no importance in the metadata space. Requirement and implementation are matched by using the published semantic metadata. The interoperation between different service oriented architectures is achieved autonomically with the adaptation process. The requirement publisher (service consumer) does not need to intervene programmatically. The publishers can program with their interface without worrying about the plumbing code that obscures the application design.

3.3. Ontology Publisher

Anyone who writes and publishes ontology into the metadata space is characterised as an ontology publisher. This role is usually taken up by domain expert or standard bodies e.g the Dublin Core Metadata Initiative[17]. Ontology are used by meta-services as a knowledge base to inference operations. They can only then provide the essential semantic inference capability for the metadata space. The semantic matching and service adaptation process both rest crucially on the quality and quantity of the published ontologies in the metadata space. Trust and consistency of ontologies are important issues and are discussed in section 6.2.

3.4. Meta-service

The meta-services provide the metadata space with semantic matching and service adaptation capabilities. Semantic matching services find compatible implementation and requirement based on their published service semantics only. Adaptation services generate service adaptation by using the syntactic service detail such as type declaration, service binding etc. Adaptation services are characterised by services that reside in multiple grid architectures and/or different syntactic description. This enable them to act as the bridge across different services, thus providing a necessary adaptation between the requirement and the implementation publisher. Meta-services form the hotspots (see figure 1) of the metadata space, all three types of published information are gathered to enable the matching and adaptation process.

4. Semantic Matching and Service Adaptation

Semantic matching and interface adaptation are the two operations that enable the autonomic adaptation of grid ser-

vices. We envision a range of different metadata-service satisfying these purposes to emerge in the semantic grid. This section focuses on the high level requirement/specification rather than the implementation detail of these services.

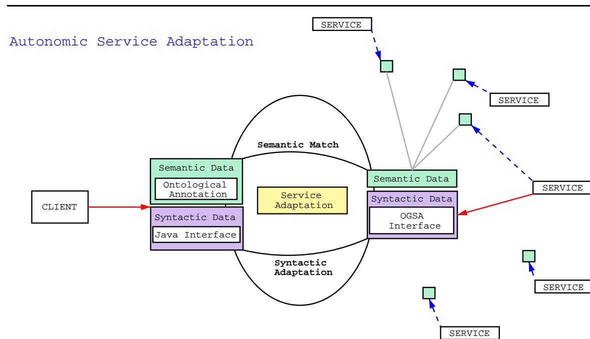

Figure 3. The autonomic service adaptation process

4.1. Semantic Annotation

The meaning of services is implicitly expressed by the implementation expressed in the form of the programming language source code. The purpose of the semantic annotation is to express this intrinsic meaning explicitly and in a machine processable way. The Resource Description Framework (RDF)[1] from W3C was designed to serve this purpose and the Web Ontology Language (OWL) builds on RDF to provide a way of adding domain specific vocabulary for resource description by using concepts taxonomy.

Semantic annotation of a service is developed in 2 stages. First, the user annotates a service with intended meaning. This expresses the service in terms of some domain concept and property a domain ontology. Annotation of the service semantics has to be written by the user. This step cannot be automated since it is only possible for the service programmer to know the intended meaning of the service. However once the original semantic annotation is established, new semantic information of the service could be inferred automatically from the original.Next, the different aspects of a service method signature [2] need to be described in a programming language independent way using RDF. Each aspect of the method signature is treated as a distinct RDF resource. This stage concentrates on expressing the syntactic meaning of the service by annotating the semantics of the definition of the service method. This metadata is essential for the adaptation process. Without it, the adaptation service will not be able to *understand* the meaning of the different service aspects. In figure 6 we introduce a toy object oriented programme ontology for annotating programme structure. This type of annotation can be generated automatically by a suitable implemented source code parser, thus moving some of the complexity of the annotation process away from the human writer.

2 It is important to note that semantic annotations can be written at different levels in the source code (statement, block, class,..) to give different levels of expressiveness. Due to the scope of the paper, we concentrate our efforts on the method level. Please see discussion for future works.

4.2. Semantic Matching

The first step in autonomic service adaptation is to find services that are conceptually equivalent to the client's requirements. These requirements are expressed through the semantic annotation of the interface by using OWL. This ties each of the interface method to a domain concept. Semantic annotations from implementation publishers are defined to be conceptually equivalent to the requirement when the requirement concept is inferable from the implementations' annotations. In practice, whether one concept is inferable from another depends on the search and inference capability of the semantic matching service and the presence of suitably defined ontology in the metadata space.

A semantic matching service will need to perform two main inference operations - class and property inferencing. Each interface annotation ties the concept of a method to a ontology class. The result of class inferencing is a set of services that are conceptually equivalent to the client interface method. Class property are represented by method signatures. Property inference narrows down the conceptually matched services by inspecting their properties and relations. This process refines the original list by taking into account ontology class properties. This is a crucial step in enabling service adaptation. It filters out conceptually incompatible services from the requirement. For example, a summation service with a parameter that refers to a precision property of a summation concept is incompatible with a requirement interface that is defined with two integers, both relate to the operand property of the summation concept. The final list of services are the ones that matched with the client annotation's class concept and properties. These are defined as conceptually compatible.

4.3. Interface and Invocation Adaptation

Adaptation service generates the adapted implementation code for the requirement publisher. Receiving a list of conceptually equivalent implementations from the matching service, this service dynamically generates the adaptor proxy on demand. In the adaptation stage, the focus shifts from the semantic metadata about the method to the method

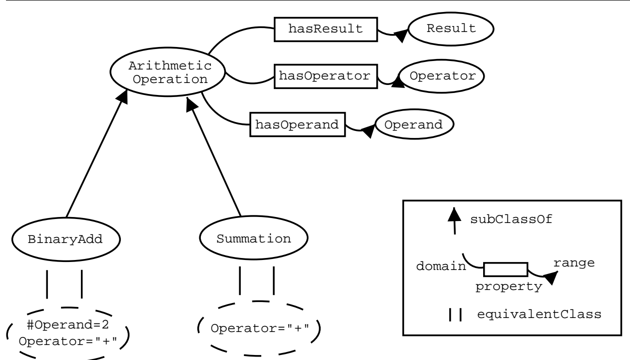

Figure 4. A toy ontology about addition

structure metadata. The aim of the adaptation service is to find a transformation function with the required method signature as the domain and the list of conceptually compatible service signatures as the range.

Invocation Architecture Selection First step in the generation of the adaptor is to identify the hosting architecture of the target service. If the target service's hosting architecture cannot be handled by any of the available invocation handler then the adaptation process for the particular service cannot continue. This identification process is achieved by inspecting the target service's syntactic description. We use WSDL as the syntactic service description language and the binding element in the WSDL document determines the hosting architecture.

Signature Transformation The structural information about method signature is described in the method structure metadata. This metadata alone is not enough for the transformation. Every adaptation service needs a set of basic transformation rules - axioms, to start a backward chaining transformation process. For example, a simple axiom that expresses the notion of an identity, is necessary in order to eliminate parameters that are conceptually equivalent, e.g. a+b = a+b+0. Type information can also be extracted from the requirement and implementation's program structure metadata. Inference from one to the other requires the presence of a suitable type ontology in the metadata space. In the example Java type ontology in figure 5, it is possible to infer that a pair of integers a and b are conceptually equivalent to an array of two integers. More specifically, the ontology allows an OWL inference engine to infer that an integer is an instance of element and therefore it can be treated as one of the element in an integer array. This is a very simple example of type inference, more sophisticated inferencing ability (such as cross programming language type inferencing) will come with more refined type ontology.

```
<rdf:RDF>
   <owl:class rdf:ID="Array"/>
      <rdfs:subClassOf>
         <owl:Restriction>
            <owl:onProperty
              rdf:resource="#elementType"/>
            <owl:Cardinality
              rdf:datatype="&xsd;Integer">
                1
            </owl:Cardinality>
         </owl:Restriction>
      </rdfs:subClassOf>
   </owl:class>
   <owl:class rdf:ID="IntArray">
      <rdfs:subClassOf rdf:ID="#Array"/>
      <rdfs:subClassOf>
         <owl:Restriction>
            <owl:onProperty
              rdf:resource="#elementType"/>
            <owl:allValuesFrom
              rdf:resource="#Integer"/>
         </owl:Restriction>
      </rdfs:subClassOf>
   </owl:class>
   <owl:ObjectProperty rdf:ID="elementType">
      <rdfs:domain rdf:resource="#Array"/>
      <rdfs:range rdf:resource="#Element"/>
   </owl:ObjectProperty>
   <owl:class rdf:ID="Integer">
     <rdfs:subClassOf rdf:ID="#Element"/>
   </owl:class>
</rdf:RDF>
```

Figure 5. A Simple Ontology of Java Types

4.4. A Short Example of Service Adaptation

In this section we will walk through a relatively simple service adaptation example. In the example we will illustrate the various important aspects of service adaptation which we went through in detail in the earlier sections.

Step 1 - User Requirement Annotation A user writes down the require service interface in his native programming language and annotate it with semantic metadata. In this example, the user writes a Java interface having a single add method.

```
public interface MathsService {
  public int add{int a, int b}
}
```

The user then annotates the interface using standard RDF/XML syntax and publishes it to the metadata space. Here the user describes that the method *add* has the type BinaryAdd (see figure 4 for the toy addition ontology), the parameter *a* and *b* are Operands and the return value of this method represent the concept Result. (OWL allows us to infer that *a* and *b* are Operands because the range of the hasOperand property is the Operand class).

```
<rdf:Description rdf:about="..MathsService/add">
  <rdf:type rdf:resource="&art;BinaryAdd"/>
  <art:hasOperand rdf:resource="../add/param/a"/>
  <art:hasOperand rdf:resource="../add/param/b"/>
```

```
  <art:hasResult rdf:resource="../add/return"/>
</rdf:Description>
```

Additional metadata is then automatically generated by a interface parser by using the Object ontology in figure 6. This and the above annotation are then published into the metadata space, ready for use for the matching and adaptation services.

```
<rdf:Description rdf:about=".../MathService/add">
 <rdf:type rdf:resource="&oo;Method"/>
 <oo:hasParameter rdf:resource="../add/param/a"/>
 <oo:hasParameter rdf:resource="../add/param/b"/>
 <oo:hasReturn rdf:resource="../add/return"/>
</rdf:Description>
```

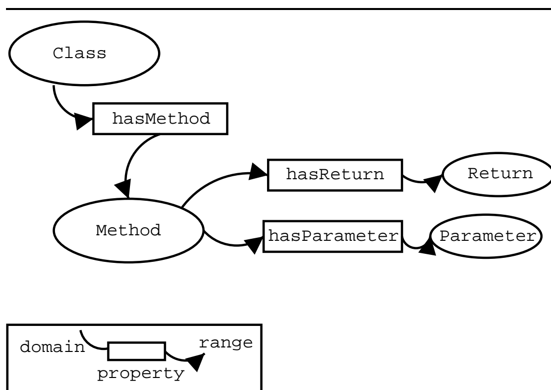

Figure 6. A toy ontology about objects

Step 2 - Semantic Matching Upon receiving the published requirement, the semantic matching service searches for compatible implementations. In this example, we assume two such implementations are found. The first implementation is a method that sums an array of integers.

```
<rdf:Description rdf:about="...SumService/sum">
  <rdf:type rdf:resource="&art;Summation"/>
  <art:hasOperand
    rdf:resource=".../sum/param/nums"/>
  <art:hasResult rdf:resource=".../sum/return"/>
</rdf:Description>

public int sum(int[] ele) {
  //method body omitted
}
```

The second interface takes three integers and return their sum.

```
<rdf:Description rdf:about="...Add3Service/add3">
 <rdf:type rdf:resource="&art;Summation"/>
 <art:hasOperand rdf:resource=".../add3/param/a"/>
 <art:hasOperand rdf:resource=".../add3/param/b"/>
 <art:hasOperand rdf:resource=".../add3/param/c"/>
 <art:hasResult rdf:resource=".../add3/return"/>
</rdf:Description>

public int add3(int a, int b, int c) {
      return a+b+c;
}
```

The ontology in figure 4 asserts that ”*X is a Summation iff X's hasOperator is a ”+”*” and also ”*Y is a BinaryAdd iff Y has exactly two operands and has operator ”+”*”. From these assertion, the matching service can infer that both of the above implementation are conceptually compatible with the requirement (since BinaryAdd is equivalent to Summation).

Step 3 - Service Adaptation For the first service implementation, the adaptation service can take advantage of the presence of a Java type ontology (figure 5) for adaptation generation. From the ontology, the adaptation service can infer that the two integers parameter a and b in the requirement can be treated as the integer array of 2 elements. The return value is an integer for both the requirement and implementation, thus the adaptation can be successfully generated.

The second implementation has one extra integer parameter, this cannot be inferred to be compatible with the requirement using our current Java type or addition ontology. To be able to adapt this implementation we need to find a way of *eliminating* the extra parameter. In this case the adaptation service needs to have an in-built understanding of the concept of an identity operand. Couple this fact with the following ontology, the desired adaptation can be generated by assigning zero to the third parameter.

```
<owl:Class rdf:ID="AdditionIdentityOperand">
  <rdfs:subClassOf rdf:resource="&art;Identity/>
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty
         rdf:resource="&art;hasValue"/>
      <owl:hasValue rdf:datatype="&xsd;Integer">
        0
      </owl:hasValue>
    </owl:Restriction>
  </rdfs:equivalentClass>
</owl:Class>
...
```

Step 4 - Invocation Architecture Selection The two semantically compatible services are passed to the adaptation service. The first step is to obtain the WSDL service descriptions. The first implementation has a SOAP over HTTP binding described by WSDL as follows

```
<binding name="SoapBinding"
         type="ns:SumPortType">
 <soap:binding style="document"
  transport="http://schemas.xmlsoap.org/soap/http"/>
  ......
</binding>
<service name="Sum">
  <port name="SumPort" binding="ns:SoapBinding">
    <soap:address location="example.org/Sum"/>
  </port>
</service>
```

the second implementation has a Java RMI binding,

```
...
<port name="RMIAddPort" binding="rmi:RMIBinding">
```

```
 <rmi:server nameserver="rmi://host"
             name="MathService"/>
</port>...
```

In this example, the adaptation service only has a Java RMI invocation handler. Therefore it picks the second service implementation, generates the adapted service and returns to the requirement publisher. Finally, the requirement publisher can use the adapted service by calling

```
MathsService aMathService =
  ICENIFramework.findService(MathsService.class);
aMathService.add(1, 2);
```

5. The ICENI Semantic Layer

The Imperial College e-Science Networked Infrastructure[18] - ICENI - is a service oriented/integrated Grid middleware that provides an augmented component programming model to aid the application developer in constructing Grid applications, and an execution infrastructure that exposes compute, storage and software resources as services with defined conditions of when and by whom these resources may be used. It utilises open and extensible XML schemas to encapsulate meta-data relating to resource capability, service availability and application behaviour.

We are currently developing a prototype implementation of the semantic matching and service adaptation framework presented in this paper. The goal is to integrate this framework with the meta-data facility in ICENI to bring about the semantic reasoning and adaptation ability. The prototype implementation is scheduled to be completed in three stages, the metadata space implementation lays the foundation as the knowledge store. Ontology processing is built on top of the metadata space and finally the service adaptation framework. In the following sections we go through a brief description of the implementation for each of the layers.

5.1. Metadata Space

The most important requirement of the metadata space is interoperability between different grid infrastructures. JXTA[6] satisfies this requirement by specifying its communication protocol in XML format and have multiple implementations of its core infrastructure in C, Java, .Net etc.

5.2. Metadata Processing

The metadata processing layer is handled by the JENA[7] semantic toolkit. One of the main reason for choosing JENA as the processing toolkit is its pluggable framework. JENA2 allows different inference engines[13][14] and ontology language to be used in our framework. It is unrealistic to assume that all participants in the metadata space will want to use OWL as the ontology language and a standard inference engine for all their needs.

5.3. Service Adaptation Framework

The service adaptation process will be built on top of the Apache Web Service Invocation Framework[15](WSIF). WSIF enables users to interact with abstract representation of web/grid services through their WSDL description. Our current work involve extending WSIF into the Grid Service Invocation Framework by adding the capability of processing WSDL extension elements defined in OGSI. This work forms the basis of our service adaptation framework.

6. Discussion

6.1. Workflow and Scheduling

Componentised software encapsulates reusable functionality to be deployed on-demand. Current research on Workflow languages[19][20] and Grid Scheduling focuses on the description of service interaction, as well as exploiting knowledge on performance characteristics of service implementation to make deployment decision. By raising the abstraction of service capability using ontology, schedulers can widen their scope of service selection. Ontological equivalent service implementation can be exploited if it can be adapted transparently to the abstract workflow. Effort in establishing ontological vocabulary for describing workflow and web services creates a foundation for further research in optimising workflow based on the conceptual intention of the composition.

6.2. Trust in Ontology and Adaptation

The metadata space is analogous to a public bulletin board. Concepts are being introduced by independent parties in a distributed and uncoordinated manner. This gives rise to the problem of ensuring logical consistency of ontology replicated at multiple autonomous peers. Moreover, the metadata space needs to guard against malicious attempts in introducing false concepts. Possible solutions might be to sign the OWL fragment using XML Digital Signature[23], and implement a distributed voting[22] in achieving common ontology consensus among peers. The quality and correctness of the adaptation would be difficult to verify without formal analysis of the adaptation code. By optionally exposing the adaptation service selection to the user-level, flexible selection schemes can be devised based on the level of trust.

6.3. Stateful Services

Service is considered as a collection of functionalities encapsulated in a set of methods. Our current prototype focuses on annotating and adapting at the method level. Multiple stateless services can be grouped and interchanged, so that they can be adapted to serve a single client interface requirement. However, invoking a method on a stateful service would impose state change, that causes side-effect when other methods are being called. As a result, stateful services can only be grouped to form one instance if their states are explicitly made shared and kept consistent.

7. Conclusion

In this paper we have presented an autonomic service adaptation framework by using semantic annotation. The three aspects of this framework were presented: the metadata space, semantic matching and service adaptation. The metadata space was proposed as an independent knowledge store for the semantic grid. Semantic matching allows us to search for services using their semantic rather than syntactic information. Finally, the service adaptation process exploits the intelligence gathered during the matching phase, and uses a set of transformation rules to generate an architecture independent binding on demand.

8. Acknowledgements

We gratefully acknowledge the support from the UK e-Science Core Programme sponsored by the Department of Trade and Industry; the Overseas Research Student Award administered by Universities UK on behalf of the Department of Education and Skills; and finally the European Data Grid Project funded by the European Union.

References

[1] Lassila O., Swick R.R., Resource Description Framework (RDF) Model and Syntax Specification W3C Recommendation 22 February 1999, http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/

[2] Patel-Schneider P.F., Hayes P., Horrocks I., OWL Web Ontology Language Semantics and Abstract Syntax W3C Working Draft 31 March 2003, http://www.w3.org/TR/owl-semantics/

[3] Heflin J, Web Ontology Language (OWL) Use Cases and Requirements W3C Working Draft 31 March 2003, http://www.w3.org/TR/webont-req/

[4] W3C Semantic Web Activity, http://www.w3.org/2001/sw/

[5] The Semantic Grid Community Portal, http://www.semanticgrid.org/

[6] JXTA, http://www.jxta.org/

[7] Jena Semantic Web toolkit, http://www.hpl.hp.com/semweb/jena.htm

[8] Christensen E., Curbera F., Meredith G., Weerawarana S., Web Service Description Language (WSDL), http://www.w3.org/TR/wsdl

[9] Box D., Ehnebuske D, Kakivaya G., Layman A., Mendelsohn N., Nielsen H. F., Thatte S., Winer D., Simple Object Access Protocol (SOAP)1.1 W3C Note 08 May 2000, http://www.w3.org/TR/SOAP/

[10] Universal Description, Discovery and Integration, http://www.uddi.org

[11] Tuecke S., Czajkowski K., Foster I., Frey J., Graham S., Kesselman C., Snelling D., Vanderbilt P., Open Grid Service Infrastructure (OGSI) draft, Feburary 2003

[12] Sintek M., Decker S., Kesselman C., Nick J. M., Tuecke S., The Physiology of the Grid (Draft)

[13] Sintek M., Decker S., TRIPLE - A Query, Inference, and Transformation Language for the Semantic Web, June 2002

[14] RACER, http://www.fh-wedel.de/ mo/racer/index.html

[15] Web Service Invocation Framework, http://ws.apache.org/wsif/

[16] Autonomic Computing, http://www.research.ibm.com/autonomic/

[17] Dublin Core Metadata Initiative, http://dublincore.org/

[18] Furmento N., Lee W., Mayer A., Newhouse S., Darlington J., ICENI: An Open Grid Service Architecture Implemented with JINI, SuperComputing 2002, November 2002

[19] Andrews T., Curbera F., et al, Business Process Execution Language for Web Service Specification Version 1.1, May 2003

[20] Arkin A., et al., Web Service Choreography Interface (WSCI) 1.0 W3C Note 8 August 2002

[21] Ankolekar A., et al., DAML-S: Web Service Description for the Semantic Web, The First International Semantic Web Conference (ISWC), June 2002

[22] Williams A., Krygowski T., Thomas G., Using Agents to Reach an Ontology Consensus, AAMAS 02, Bologna, Italy, July 2002

[23] XML Signature, http://www.w3.org/Signature/

[24] www.mygrid.org.uk

[25] Paolucci M, Kawamura T., et al,. Semantic Matching of Web Services Capabilities

[26] Tangmunarunkit H, Decker S., et al., Ontology-based Resource Matching in the Grid - The Grid meets the Semantic Web