Abstract

Having the notion of services brought into the domain of component based software design has enriched the domain of modular self-contained components with support for loose coupling, dynamic runtime discovery, and late binding. However, loose coupling, runtime discovery, and composition for a service-oriented component model in its current scale mostly applies to homogeneous services that admit to the specifications of the original service component model. This design is in conflict with the initial outlook of SOA in which reusability of existing services is considered a primary goal. In this paper, we discuss ReCoIn, a remote service composition and integration framework, which aims to bring functionalities of traditional Web services into the local repository of a service oriented component model. We discuss how ReCoIn enables remote Web services to be properly discovered, encapsulated, and used in a service oriented component model framework, offering better service reuse and fault tolerance. In this paper, we layout the overall architectural design for ReCoIn and present its initial prototype.

1. Introduction

Service-Oriented Component Model (SOCM) frameworks combine the benefits of service oriented architecture (SOA) and component based software engineering (CBSE) to bring together loose coupling from the SOA and coherent software architecture from the CBSE. The main objective of such systems is to support modularity at a high level of granularity by enabling dynamic replacement, update, and reuse of services. Adhering to the requirements of SOA, a SOCM introduces a service as a provided functionality characterized by its contract [6]. Service dependencies are resolved by using service-oriented Discovery and Binding; contracts describing inter-service relations present the application-level architecture (Composition); and runtime substitutability and replacement of services (Adaptation) is achieved through late binding. SOCM lends itself naturally to distributed processing. Reuse of distributed services in a SOCM starts with service provider modules registering their services with service registries. Client modules use these registries to look up their desired service functionalities. There have been various implementations of SOCM [6][7], yet the Open Service Gateway initiative (OSGi) [11] is arguably the most widely used. OSGi is a SOCM enabling development of Java applications from modular units called bundles [11]. The notion of bundles in the OSGi framework, however, supersedes the definition of components in that service interfaces defined in a bundle are completely separated from their implementations.

Typically, as we move from small scale service registries to large scale service repositories, we see increasing heterogeneity of services with respect to their service providers, their interface description languages (IDLs), their offered quality, etc. Ideally, this increasing heterogeneity should be handled gracefully by a distributed SOCM, capable of dealing with this diverse set of service repositories. Even though, the standard architecture for OSGi promotes a centralized model of SOCM, there have been several efforts to design a distributed model for OSGi [7][8][9]. Yet, most of these design architectures deal with integration and composition of homogenous services running on similar platforms but connected from distributed hosts. Increasing heterogeneity of services (and their associated descriptions) particularly in large scale service repositories remains a problem for current SOCMs.

In this paper, we propose ReCoIn as a framework to help with enhancing the discovery and integration of heterogeneous services into a SOCM framework. ReCoIn promotes the following contributions:

• Support for optimal discovery and integration of heterogeneous service types from diverse service repositories into one single SOCM framework.
• An open architecture for incorporating various discovery algorithms with different degrees of formalism – from signature based interface analysis to semantic based conceptual evaluation of services.
• Proper tooling for SOCM developers to look for, discover, and integrate services of heterogeneous types into a SOCM framework.

The rest of the paper is organized as follows. In Section 2, we discuss motivating scenarios. Section 3 provides the architectural design for ReCoIn followed by
discussing our prototype implementation in Section 4. We report on the some evaluation results in Section 5, while Section 6 compares our work to other approaches in the research community. Section 7 includes concluding remarks and future works.

2. Motivating Scenario

As described earlier, the need for integrating a remote services into a SOCM framework stems from the need to use a retrieval or update operation from an existing application, to use an existing or a new module or program, to request the execution of some business logic, etc.

Independent of the discovery protocol used, discovering a remote service to replace a local service, requires the interface specification for the remote service to match the specification for the local service. OSGi as one of the most prominent SOCM frameworks comes with a Java implementations in which service interfaces are defined in the form of Java interfaces. Java is chosen for its capability in separating a service interface from its implementation and facilitating late binding of interfaces to specifications, thanks to its powerful class loading architecture.

In order to manually find a remote service that matches the specifications of a local service, the developer needs to inspect the Java interface for a desired service in the local repository and then search for and find a remote service with matching interface. At the other end, for finding remote services, a study of the public service-finder service for the Seekda Web service search engine [16] states that discovering a service from a remote repository is subject to the inspection of the specifications for various services before proceeding with the selection and integration of services. Hence, the first hit for a discovered service is not necessarily the best hit. Subsequently, an optimal discovery and integration engine should enable the developer to inspect interfaces for multiple remote services to find the optimal alternative.

Having all the above arguments in mind, let us look at the following scenario for integration of a remote service into the local repository of a SOCM framework. Tavares and Valente in their introduction to OSGi [2] discuss an example OSGi service for a dictionary translator application that enables translation from one language to another. However, the number of languages that such a service supports is limited to the number of databases which store the translation information. If there is a need to extend the dictionary service to support more languages, one immediate solution would be to add another database with support for additional languages. A second solution would be to try and bring a remote service with a similar functionality into the context of the local OSGi framework. Not only does this second approach save on cost and effort to integrate the new service into the framework, but also it is faster than having to develop individual databases which need to be integrated with the framework and connected to the dictionary service.

3. Remote Service Composition and Integration Architecture (ReCoIn)

The Remote Service Composition and Integration framework (ReCoIn) enables integration of remote services into the local registry of a SOCM framework by offering discovery, matching, and binding of services. ReCoIn is based on top of OSGi and reconciles local OSGi services with remote services offering similar functionalities. ReCoIn follows a modular design which composes three major components, i) a Service Discovery component, ii) a Specification Manager component, and iii) a Service Binding component (see Figure 1).

The Service Discovery module is responsible for finding appropriate remote services and matching them to the existing local services by i) finding a local service in the local service registry of the SOCM, ii) extracting the syntactic or semantic signature for the local service, and iii) looking into remote service repositories (e.g., SLP or UDDI) to find a proper match for the local service.

Upon discovery of a proper remote service, the Specification Manager module i) inspects the internal specification of the local service, ii) compares it with the remote service contract specification, iii) extracts the degree of similarity between them, and iv) decides on how closely the remote service resembles the behaviour of the local service. The Specification Manager can also take non-functional parameters such as service availability and level of trust into remote service provider into account once dealing with inspection and analysis of remote and local service contracts. The Specification Manager can then actively include these parameters into how services are analyzed, selected, and sent to the binding engine.

Figure 1. The overall architecture for ReCoIn

Once a proper remote service is discovered by the discovery engine and approved by the Specification Manager (in collaboration with the developer), the Binding Engine takes the service contract specification for the remote service (e.g., the WSDL file for a Web service), generates a local stub or proxy for communication with the remote service, generates the hooks for the stub to connect to the SOCM framework, creates a SOCM-compliant component out of the generated stubs, and then installs the bundle onto the framework.
Dividing ReCoIn to these high-level modules facilitates expandability and generalizability of the whole system. The Discovery Engine is designed as an OSGi service with extensibility in mind. The discovery engine represents an abstract programming model which can provide a service lookup for different remote service registries by adding modules to support abstract to concrete mappings for communication protocols, search queries, and service specifications corresponding to a wide variety of service types (e.g., UDDI). The Discovery Engine supports various formalisms for querying service registries. By adding metadata to the programming model of an OSGi service, we can enable syntactic and semantic discovery of remote services that correspond to the interface description of the local service. The current implementation of ReCoIn augments the information in the MANIFEST file with another property named osgi.service.concepts which takes a list of concepts describing the behaviour of the local service. This set of concepts is then used for querying remote service repositories to find matches for the local service. We believe that the MANIFEST for OSGi services can further be augmented to include semantic information to describe the behaviour of OSGi services. The Specification Manager Service for ReCoIn, similar to the Discovery Engine, also follows an extensible design. Even though the current implementation of the Specification Manager only supports inspection of Web service descriptions (i.e., WSDL files), it can be extended to inspect the contracts for different types of remote services (e.g., REST [13] services) as well.

Finally, the Binding Engine can be extended to generate the required stubs for any type of service that the specification manager can accurately inspect and analyze (including Web services and REST services). The generated stub for our current implementation of ReCoIn only deals with Web services; however different binding plugins can be pushed into ReCoIn to enable generation of stubs for other types of services.

4. Prototype Implementation for ReCoIn

ReCoIn’s current implementation considers services in an OSGi service registry as local services and Web services as remote services. This means that, having Java service interfaces for the services registered with the service registry of an OSGi framework, ReCoIn looks for Web services that resemble the behaviour of local services and creates proper stubs and activators to bring the remote Web service into the OSGi service registry. ReCoIn then offers the functionalities for the newly integrated service to the services in the local registry of SOCM. Considering the behaviour of ReCoIn’s three main components (i.e., the discovery engine, the specification manager, and the binding engine), the process of binding a remote Web service to a local OSGi service contract can be divided into two phases of Local Service Inspection and Remote Service Integration.

4.1. Local Service Inspection

ReCoIn starts the process of integrating a remote Web service into the local repository of OSGi by first looking at the list of existing services in the OSGi service registry. For an OSGi service to be processable by ReCoIn, we have added an osgi.service.concepts property into the MANIFEST of the OSGi service. This descriptive property is used to reflect on the behaviour of the service to help with discovering similar services.

In order to search for remote Web services, we query the Seekda [16] service repository. “Seekda™ is the free search engine for discovering Web Services and their providers [16]” with almost 25000 registered services. A query to Seekda is rendered using the set of keywords added to the service MANIFEST file. The keywords can be considered as tags describing the behaviour of the service and are used to send a proper query to Seekda. The discovery strategy employed in ReCoIn is a mixture of spreading activation and relaxed discovery [15] where in the first phase of discovery, Seekda is queried and the set of potentially satisfactory services are retrieved. In the second phase, by performing the similarity evaluation on the interface descriptions for local and remote services and extracting the similarity measures, one can inspect extracted services and decide about which ones can potentially be integrated with the local SOCM framework.

Upon receiving Web services from Seekda, the Specification Manager component checks the WSDLs for the discovered services and tries to compare them with the specification for the local OSGi service. This is done by converting the Java interface for the OSGi service to a WSDL file using Apache’s Axis2 Java2WSDL tool [3]. Once the WSDL file for the Java interface is generated, it is compared against each of the WSDL files found by Seekda in order to obtain the degree of similarity between the remote Web service specification and the local WSDL reflecting on the local OSGi service.

Our current implementation of ReCoIn favours the use of Simpack in order to obtain a signature-based measure for the degree of similarity between the local service and the remote services. Simpack [4] is a Java API to measure similarities between entities (e.g., concepts in ontologies, classes in source codes, XML documents, and data streams). We have adopted tree-edit distance algorithm [1] from the Simpack API in order to measure the signature-based degree of similarity between the remote WSDL files and the generated WSDL file for the local service. It is important to note that by providing one round of service filtering in the discovery phase, prior to calculating the degree of similarity, we put the WSDL files for local service and remote services into the same conceptual context in terms of their functionalities. This significantly reduces the chance for dealing with false negatives when looking for remote services with a similar functionality. Upon calculating the degree of similarity between remote Web services and the local OSGi service, the degree of compliance for a service is calculated as a by-product.
of the degree of similarity (obtained from Simpack),
the availability rate (obtained from Seekda), and the
level of trust in the service provider (fed to the system
by service developers). Other non-functional properties
can also be taken into consideration to obtain a proper
degree of compliance for the service.

4.2. Remote Service Integration

ReCoIn integrates a remote Web service into the
OSGi service registry as a local OSGi service activated
via its corresponding bundle activator, called OSGi-
Axis2-Projectizer. Informally, the OSGiAxis2-
Projectizer service is a Java object instance published
in the OSGi service registry. It implements the inter-
face for the missing or overwritten local OSGi service
and integrates remote service functionalities by invok-
ing those methods from the remote service which im-
plement proper functionalities for the local service in-
terface. The OSGiAxis2-Projectizer also creates the re-
quired project layout for the OSGi framework by
launching the following system components: Proxy
Generator, Remote Service Stub Code Generation, Dy-
namic Code Compilation, and Specification Reflection
(see Figure 2). Below, we briefly look at the functional-
ities provided by each of the above components.

Proxy Generator receives the URI for services with
highest degrees of compliance from the Local Service
Inspection phase, which are then used for integration
with the local service registry. To implement a SOAP
dynamic proxy, Apache Axis2 is employed [3].

As for Remote Service Stub Generation, We have
opted for a top-down (contract first) approach to gen-
erate the proxy stub, deployed commonly for Web ser-
vice interface implementations. The top-down ap-
proach starts with the Web service description, which is
the WSDL document, and then goes on to expose the
Web service functionalities as methods for local invo-
cation.

To enable dynamic code compilation, the remote
service code generation creates classes that expose all
the methods and properties of the combined capabilities
at the remote endpoint, and convert the entire local re-
quest into remote calls. The stubs and skeletons for the
remote interfaces, as Java classes, are compiled dy-
namically for run time integration with the local OSGi
repository and undergo a remote service signature re-

Remote Service Wrapper

Remote Service Stub

Code Generation

Figure 2. Remote Service Integration in ReCoIn

4.2. Remote Service Integration

ReCoIn integrates a remote Web service into the
OSGi service registry as a local OSGi service activated
via its corresponding bundle activator, called OSGi-
Axis2-Projectizer. Informally, the OSGiAxis2-
Projectizer service is a Java object instance published
in the OSGi service registry. It implements the inter-
face for the missing or overwritten local OSGi service
and integrates remote service functionalities by invok-
ing those methods from the remote service which im-
plement proper functionalities for the local service in-
terface. The OSGiAxis2-Projectizer also creates the re-
quired project layout for the OSGi framework by
launching the following system components: Proxy
Generator, Remote Service Stub Code Generation, Dy-
namic Code Compilation, and Specification Reflection
(see Figure 2). Below, we briefly look at the functional-
ities provided by each of the above components.

Proxy Generator receives the URI for services with
highest degrees of compliance from the Local Service
Inspection phase, which are then used for integration
with the local service registry. To implement a SOAP
dynamic proxy, Apache Axis2 is employed [3].

As for Remote Service Stub Generation, We have
opted for a top-down (contract first) approach to gen-
erate the proxy stub, deployed commonly for Web ser-
vice interface implementations. The top-down ap-
proach starts with the Web service description, which is
the WSDL document, and then goes on to expose the
Web service functionalities as methods for local invo-
cation.

To enable dynamic code compilation, the remote
service code generation creates classes that expose all
the methods and properties of the combined capabilities
at the remote endpoint, and convert the entire local re-
quest into remote calls. The stubs and skeletons for the
remote interfaces, as Java classes, are compiled dy-
namically for run time integration with the local OSGi
repository and undergo a remote service signature re-

Remote Service Wrapper

Remote Service Stub

Code Generation

Figure 2. Remote Service Integration in ReCoIn

Remote Service Wrapper

Remote Service Stub

Code Generation

Figure 2. Remote Service Integration in ReCoIn

4.2. Remote Service Integration

ReCoIn integrates a remote Web service into the
OSGi service registry as a local OSGi service activated
via its corresponding bundle activator, called OSGi-
Axis2-Projectizer. Informally, the OSGiAxis2-
Projectizer service is a Java object instance published
in the OSGi service registry. It implements the inter-
face for the missing or overwritten local OSGi service
and integrates remote service functionalities by invok-
ing those methods from the remote service which im-
plement proper functionalities for the local service in-
terface. The OSGiAxis2-Projectizer also creates the re-
quired project layout for the OSGi framework by
launching the following system components: Proxy
Generator, Remote Service Stub Code Generation, Dy-
namic Code Compilation, and Specification Reflection
(see Figure 2). Below, we briefly look at the functional-
ities provided by each of the above components.

Proxy Generator receives the URI for services with
highest degrees of compliance from the Local Service
Inspection phase, which are then used for integration
with the local service registry. To implement a SOAP
dynamic proxy, Apache Axis2 is employed [3].

As for Remote Service Stub Generation, We have
opted for a top-down (contract first) approach to gen-
erate the proxy stub, deployed commonly for Web ser-
vice interface implementations. The top-down ap-
proach starts with the Web service description, which is
the WSDL document, and then goes on to expose the
Web service functionalities as methods for local invo-
cation.

To enable dynamic code compilation, the remote
service code generation creates classes that expose all
the methods and properties of the combined capabilities
at the remote endpoint, and convert the entire local re-
quest into remote calls. The stubs and skeletons for the
remote interfaces, as Java classes, are compiled dy-
namically for run time integration with the local OSGi
repository and undergo a remote service signature re-

Remote Service Wrapper

Remote Service Stub

Code Generation

Figure 2. Remote Service Integration in ReCoIn

Remote Service Wrapper

Remote Service Stub

Code Generation

Figure 2. Remote Service Integration in ReCoIn

4.2. Remote Service Integration

ReCoIn integrates a remote Web service into the
OSGi service registry as a local OSGi service activated
via its corresponding bundle activator, called OSGi-
Axis2-Projectizer. Informally, the OSGiAxis2-
Projectizer service is a Java object instance published
in the OSGi service registry. It implements the inter-
face for the missing or overwritten local OSGi service
and integrates remote service functionalities by invok-
ing those methods from the remote service which im-
plement proper functionalities for the local service in-
terface. The OSGiAxis2-Projectizer also creates the re-
quired project layout for the OSGi framework by
launching the following system components: Proxy
Generator, Remote Service Stub Code Generation, Dy-
namic Code Compilation, and Specification Reflection
(see Figure 2). Below, we briefly look at the functional-
ities provided by each of the above components.

Proxy Generator receives the URI for services with
highest degrees of compliance from the Local Service
Inspection phase, which are then used for integration
with the local service registry. To implement a SOAP
dynamic proxy, Apache Axis2 is employed [3].

As for Remote Service Stub Generation, We have
opted for a top-down (contract first) approach to gen-
erate the proxy stub, deployed commonly for Web ser-
vice interface implementations. The top-down ap-
proach starts with the Web service description, which is
the WSDL document, and then goes on to expose the
Web service functionalities as methods for local invo-
cation.

To enable dynamic code compilation, the remote
service code generation creates classes that expose all
the methods and properties of the combined capabilities
at the remote endpoint, and convert the entire local re-
quest into remote calls. The stubs and skeletons for the
remote interfaces, as Java classes, are compiled dy-
namically for run time integration with the local OSGi
repository and undergo a remote service signature re-

Remote Service Wrapper

Remote Service Stub

Code Generation

Figure 2. Remote Service Integration in ReCoIn

Remote Service Wrapper

Remote Service Stub

Code Generation

Figure 2. Remote Service Integration in ReCoIn

4.2. Remote Service Integration

ReCoIn integrates a remote Web service into the
OSGi service registry as a local OSGi service activated
via its corresponding bundle activator, called OSGi-
Axis2-Projectizer. Informally, the OSGiAxis2-
Projectizer service is a Java object instance published
in the OSGi service registry. It implements the inter-
face for the missing or overwritten local OSGi service
and integrates remote service functionalities by invok-
ing those methods from the remote service which im-
plement proper functionalities for the local service in-
terface. The OSGiAxis2-Projectizer also creates the re-
quired project layout for the OSGi framework by
launching the following system components: Proxy
Generator, Remote Service Stub Code Generation, Dy-
namic Code Compilation, and Specification Reflection
(see Figure 2). Below, we briefly look at the functional-
ities provided by each of the above components.

Proxy Generator receives the URI for services with
highest degrees of compliance from the Local Service
Inspection phase, which are then used for integration
with the local service registry. To implement a SOAP
dynamic proxy, Apache Axis2 is employed [3].

As for Remote Service Stub Generation, We have
opted for a top-down (contract first) approach to gen-
erate the proxy stub, deployed commonly for Web ser-
vice interface implementations. The top-down ap-
proach starts with the Web service description, which is
the WSDL document, and then goes on to expose the
Web service functionalities as methods for local invo-
cation.

To enable dynamic code compilation, the remote
service code generation creates classes that expose all
the methods and properties of the combined capabilities
at the remote endpoint, and convert the entire local re-
quest into remote calls. The stubs and skeletons for the
remote interfaces, as Java classes, are compiled dy-
namically for run time integration with the local OSGi
repository and undergo a remote service signature re-

Remote Service Wrapper

Remote Service Stub

Code Generation

Figure 2. Remote Service Integration in ReCoIn

Remote Service Wrapper

Remote Service Stub

Code Generation

Figure 2. Remote Service Integration in ReCoIn

5. Evaluation

We analyzed the performance and scalability of
ReCoIn modules which are in nature individual loosely
coupled OSGi services. For the performance and scal-
bility analysis, we report on i) the time needed for
each piece of the framework to execute and ii) the
measured probability for discovering stateless remote
services with simple WSDL interfaces. By simple, we
mean services that do not encompass complex data
types in their WSDL definitions, their set of offered
functionalities are not excessively large, and do not re-
fer to external resources and services that need to be re-
solved for the Web service to function properly (i.e., a
translation Web service and a currency Web service).

5.1. Performance Measures

We measured the amount of time required by each of
module implemented in the first prototype of ReCoIn
as described in Section 4. The performance measure-
ments were performed on a machine with a 2.5GHz
core duo Intel processor and 2 GB of RAM, connected
to the internet wirelessly with a maximum throughput
of 300 Mbps.
In Table 2, we present the amount of time ReCoIn spent on retrieving all translator services (145 services in total) available on Seekda corresponding to a local OSGi service.

<table>
<thead>
<tr>
<th>Process Name</th>
<th>Process Time Max (ms.)</th>
<th>Process Time Min (ms.)</th>
<th>Total (ms.)</th>
<th>Avg (ms.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieving translator service WSDL</td>
<td>2613</td>
<td>6</td>
<td>287</td>
<td>9.56</td>
</tr>
<tr>
<td>Service Wrapper</td>
<td>5404</td>
<td>1398</td>
<td>70321</td>
<td>2344.03</td>
</tr>
<tr>
<td>OSGiAxis2 Projectizer</td>
<td>2613</td>
<td>624</td>
<td>43510</td>
<td>1450.33</td>
</tr>
</tbody>
</table>

In another effort, we tried to measure the probability for a developer to successfully find an appropriate remote Web service for a local service in a SOCM framework. To do so, we developed a currency service with an interface similar to the one shown in Table 3.

```java
public interface Currency {
    public double getRate(
        String sourceRate,
        String targetRate);
    public String getCountryCode(
        String countryCode);
}
```

### Related Work

Remote OSGi (R-OSGi) is a distributed middleware platform capable of transparently distributing parts of an application by distributing their corresponding software modules to other accessible devices and platforms [9]. R-OSGi adds portability to OSGi and enables deployment of OSGi on resource constrained devices. R-OSGi however is only capable of distributing services that admit to the specifications and definitions of OSGi and fails in integrating more heterogeneous remote services into its framework. ReCoIn is different from R-OSGi in that ReCoIn is looking into how to bring heterogeneous services from third party service providers into the realm of SOCM frameworks.

RFC 119 in the R4.2 Specification of OSGi [11] is the most comprehensible effort in providing a design layout for incorporating remote and diverse services into the SOCM framework for OSGi. RFC 119 suggests inclusion of some reserved properties to the service programming model for OSGi services by introducing some reserved properties and some metadata for configuring distribution software. The changes to the service programming model for OSGi enables OSGi services to be exposed to the remote systems under different service specifications. Additionally, RFC 119 promotes inclusion of distribution service and discovery service to be added to the compendium document for the OSGi specification.

Diáz Redondo et al. [14] introduce the notion of Semantic OSGi as a mechanism for semantically enhanced discovery and invocation with the OSGi platform. They follow the specifications for OWL-S [17] and derive an ontology called OWL-OS to define conceptual models for OSGi service. Even though the approach of semantic OSGi greatly adds to the formalism of specifying services and enables conceptual inclusion...
of service intentions to the specification of services, it is heavily invasive to the original specification of OSGi services as the Java code for an OSGi service needs to be considerably modified in order to work in the context of semantic OSGi.

ReMMoC [12] is a reflective framework whose key operation is to dynamically adapt discovery and interaction protocols to match the current mobile service environments. ReMMoC is built on top of OpenCOM [10] by utilizing a minimal set of OpenCOM components. ReMMoC offers a two-tier architecture in which the top level component framework supports plugability for a variety of components. The top level OpenCOM-based component framework in ReMMoC holds the same role as of OSGi in the ReCoIn architecture. ReMMoC is the closest to our approach in designing ReCoIn, however, ReCoIn deals with integration of third party services into the SOCM platform by enabling support for various formalisms in service discovery and matching.

7. Conclusions and Future Work
In this paper, we discussed ReCoIn as an architecture reflecting on our efforts to enable optimal discovery and integration of remote services into a service-oriented component model framework, OSGi in particular. We described the motivations behind using such a system as invoking an existing business logic, or performing retrieval or update to an existing systems in order to improve on software reuse and reconfiguration. Our initial prototype shows considerable proofs for feasibility of the approach and opens various doors for future research to improve the behaviour of the system and bring it more to an enterprise level solution.

We solely rely on the degree of similarity between the local and remote interface signatures in our current approach for similarity checking procedure. Combining both signature and similarity comparison of services will also be examined as part of our future research plans. Furthermore, service interfaces declared syntactically using different service definition languages do not convey any semantic description of functionalities for their exposed methods. Hence, service specification matching of local services against remote services remains as a challenging problem in the absence of semantic specification of services. This issue can be resolved through leveraging the body of research in the area of Semantic Web services described through service ontology specifications shared across local and remote services.

8. References