



Data, Information and Process Integration
with Semantic Web Services

DIP

Data, Information and Process Integration with Semantic Web Services

FP6 - 507483

Deliverable

WP 4: Service Usage
D 4.6
Invocation module specification

Andreas Friesen
Joachim Quantz

December 13th, 2004

SUMMARY

This deliverable describes the invocation module specification. Invocation is the task of performing an actual call to an instance of a Web Service. The goal of this specification is to provide a standardized way of invocation for all DIP components.

This specification describes the invocation component at a level of detail allowing the users of the invocation module to understand how to use the invocation module using the specified API. It also describes the general architecture of the invocation module and provides implementation recommendations allowing the implementers to understand how to implement the API specified in this document.

This deliverable contributes to the Open Source Semantic Web Services Architecture and Exploitable Tools by providing a standardized way to perform the invocation task. It is thus relevant to all work packages in DIP developing components required to perform invocation of Web Services instances. The target audience comprises therefore the implementers of the invocation module and the developers of components using the invocation module (e.g. WSMX).

Disclaimer: The DIP Consortium is proprietary. There is no warranty for the accuracy or completeness of the information, text, graphics, links or other items contained within this material. This document represents the common view of the consortium and does not necessarily reflect the view of the individual partners.

Document Information

IST Project Number	FP6 – 507483	Acronym	DIP
Full title	Data, Information, and Process Integration with Semantic Web Services		
Project URL	http://dip.semanticweb.org		
Document URL			
EU Project officer	Brian Macklin		

Deliverable	Number	4.6	Title	Invocation module specification
Work package	Number	4	Title	Service Usage

Date of delivery	Contractual	M 12	Actual	13-Dec-04
Status	version. 1.0		final	<input checked="" type="checkbox"/>
Nature	Prototype <input type="checkbox"/> Report <input checked="" type="checkbox"/> Dissemination <input type="checkbox"/>			
Dissemination Level	Public <input type="checkbox"/> Consortium <input checked="" type="checkbox"/>			

Authors (Partner)	Andreas Friesen (SAP AG), Joachim Quantz (Inubit),			
Responsible Author	Andreas Friesen		Email	andreas.friesen@sap.com
	Partner	SAP AG	Phone	+49 721 69 02 86








Abstract (for dissemination)	This deliverable describes the invocation module specification. Invocation is the task of performing an actual call to an instance of a Web Service. The goal of this specification is to provide a standardized way of invocation for all DIP components.	
Keywords	Web Service, Invocation, Specification	

Version Log			
Issue Date	Rev No.	Author	Change
<dd-mmm-yy>	<nnn starting 001>	<author name>	<Description of the changes that were made to the preceding revision>
15-11-04	001	Andreas Friesen	Initial version for review
10-12-04	002	Andreas Friesen	Review comments incorporated






13-12-04	003	Joachim Quantz	Review comments on WSDL 2.0 incorporated. Proof-reading done.
-----------------	------------	-------------------	--

Project Consortium Information

Partner	Acronym	Contact
National University of Ireland Galway	NUIG  National University of Ireland, Galway <i>Ollscoil na hÉireann, Gaillimh</i>	Prof. Dr. Christoph Bussler Digital Enterprise Research Institute (DERI) National University of Ireland, Galway Galway Ireland Email: chris.bussler@deri.org Tel: +353 91 512460
Fundacion De La Innovacion.Bankinter	Bankinter 	Monica Martinez Montes Fundacion de la Innovation. BankInter Paseo Castellana, 29 28046 Madrid, Spain Email: mmtnez@bankinter.es Tel: 916234238
Berlecon Research GmbH	Berlecon 	Dr. Thorsten Wichmann Berlecon Research GmbH Oranienburger Str. 32 10117 Berlin, Germany Email: tw@berlecon.de Tel: +49 30 2852960
British Telecommunications Plc.	BT 	Dr John Davies BT Exact (Orion Floor 5 pp12) Aداstral Park Martlesham Ipswich IP5 3RE, United Kingdom Email: john.nj.davies@bt.com Tel: +44 1473 609583
Swiss Federal Institute of Technology, Lausanne	EPFL 	Prof. Karl Aberer Distributed Information Systems Laboratory École Polytechnique Fédérale de Lausanne Bât. PSE-A 1015 Lausanne, Switzerland Email : Karl.Aberer@epfl.ch Tel: +41 21 693 4679
Essex County Council	Essex 	Mary Rowllatt, Essex County Council PO Box 11, County Hall, Duke Street Chelmsford, Essex, CM1 1LX United Kingdom. Email: maryr@essexcc.gov.uk Tel: +44 (0)1245 436524
Forschungszentrum Informatik	FZI 	Andreas Abecker Forschungszentrum Informatik Haid-und-Neu Strasse 10-14 76131 Karlsruhe Germany Email: abecker@fzi.de Tel: +49 721 9654 0

Institut für Informatik, Leopold-Franzens Universität Innsbruck	UIBK  universität innsbruck	Prof. Dieter Fensel Institute of computer science University of Innsbruck Technikerstr. 25 A-6020 Innsbruck, Austria Email: dieter.fensel@deri.org Tel: +43 512 5076485
ILOG SA	ILOG  Changing the rules of business	Christian de Sainte Marie 9 Rue de Verdun, 94253 Gentilly, France Email: csm@ilog.fr Tel: +33 1 49082981
inubit AG	Inubit  the integration experts	Torsten Schmale inubit AG Lützowstraße 105-106 D-10785 Berlin Germany Email: ts@inubit.com Tel: +49 30726112 0
Intelligent Software Components, S.A.	iSOCO 	Dr. V. Richard Benjamins, Director R&D Intelligent Software Components, S.A. Pedro de Valdivia 10 28006 Madrid, Spain Email: rbenjamins@isoco.com Tel. +34 913 349 797
Net Dynamics Internet Technologies GmbH u. Co KG	Net Dynamics 	Peter Smolle Net Dynamics Internet Technologies GmbH & Co KG Prinz-Eugen-Strasse 68-70 A-1040 Wien, Austria Email: peter.smolle@netdynamics-tech.com Tel.: +43 1 503982615
The Open University	OU 	Dr. John Domingue Knowledge Media Institute The Open University, Walton Hall Milton Keynes, MK7 6AA United Kingdom Email: j.b.domingue@open.ac.uk Tel.: +44 1908 655014
SAP AG	SAP 	Dr. Elmar Dörner SAP Research, CEC Karlsruhe SAP AG Vincenz-Priessnitz-Str. 1 76131 Karlsruhe, Germany Email: elmar.dorner@sap.com Tel: +49 721 6902 31
Sirma AI Ltd.	Sirma  Ontotext Knowledge and Language Engineering Lab of Sirma	Atanas Kiryakov, Ontotext Lab, - Sirma AI EAD Office Express IT Centre, 3rd Floor 135 Tzarigradsko Chausse Sofia 1784, Bulgaria Email: atanas.kiryakov@sirma.bg Tel.: +359 2 9768 303

<p>Tiscali Österreich GmbH</p>	<p>Tiscali</p> 	<p>Dieter Haacker Tiscali Österreich GmbH. Diefenbachgasse 35 A-1150 Vienna Austria Email: Dieter.Haacker@at.tiscali.com Tel: +43 1 899 33 160</p>
<p>Unicorn Solution Ltd.</p>	<p>Unicorn</p> 	<p>Jeff Eisenberg Unicorn Solutions Ltd, Malcha Technology Park 1 Jerusalem 96951 Israel Email: Jeff.Eisenberg@unicorn.com Tel.: +972 2 6491111</p>
<p>Vrije Universiteit Brussel</p>	<p>VUB</p> 	<p>Carlo Wouters Starlab- VUB Vrije Universiteit Brussel Pleinlaan 2, G-10 1050 Brussel ,Belgium Email: carlo.wouters@vub.ac.be Tel.: +32 (0) 2 629 3719</p>

LIST OF KEY WORDS/ABBREVIATIONS

BPEL4WS – Business Process Execution Language for Web Services

EJB – Enterprise Java Bean

MEP – Message Exchange Pattern

RMI – Remote Method Invocation

UDDI – Universal Description Discovery and Integration

SOAP – Simple Object Access Protocol

SMI – Service Messaging Interface

XML – Extensible Markup Language

WSDL – Web Service Description Language

WSIF – Web Service Invocation Framework

WSMO – Web Service Modelling Ontology

WS-I – Web Services Interoperability Organization

WSML – Web Service Modelling Language

WSMX – Web Service Execution Environment

TABLE OF CONTENTS

SUMMARY	I
LIST OF KEY WORDS/ABBREVIATIONS	VII
TABLE OF CONTENTS	VIII
1 INTRODUCTION.....	1
2 CONTENT OF THE DELIVERABLE 4.6	2
3 REQUIREMENTS ON THE INVOCATION COMPONENT.....	3
3.1 Requirements specified in D4.1	3
3.2 Relationship to Semantic Web Services.....	4
4 FUNDAMENTAL APPROACHES	6
5 STATE-OF-THE-ART	8
5.1 Apache SOAP API.....	8
5.1.1 API for invoking SOAP RPC services	8
5.1.2 API for sending and receiving SOAP messages	10
5.2 Apache WSIF.....	11
6 WS INTEROPERABILITY REQUIREMENTS	15
6.1 WS-I Requirements	15
6.2 WSDL 2.0.....	16
7 INVOCATION MODULE ARCHITECTURE.....	18
8 INVOCATION MODULE API	20
9 PROPOSED IMPLEMENTATION OF THE DIP INVOCATION MODULE.....	25
10 CONCLUSION	26
REFERENCES	27

LIST OF FIGURES

Figure 1 Message exchange protocol dependent invocation.....	6
Figure 2 Message exchange protocol independent invocation.....	7
Figure 3 Service invocation using WSIF	13
Figure 4 Architecture of the invocation module.....	18

1 INTRODUCTION

Semantic Web Services rely on two core concepts:

- Semantic Web
- Service-oriented Architecture (SOA)

The goal of the Semantic Web is to extend the current Web with machine-processable semantics of data and information enabling computers for querying and managing semi-structured information.

Service-oriented Architectures allow publishing, finding and invoking of computational capabilities (resources) in the form of Web Services.

Semantic Web Services present an effort to lift the web to a new level of service, combining the machine-processable semantics based on ontologies with computational aspects based on SOAs. This significantly increases the potential of the Web by providing a way for automated discovery, composition, invocation of services, etc.

Invocation is the task of performing an actual call to an instance of a Web Service. Hence the invocation component is at the bottom of the technologies (to be) developed in DIP to realize the vision of Semantic Web Services. The invocation component belongs to the implementation level and does not explicitly deal with any semantic information. This specification relies on requirements and recommendations defined in “D4.1 Requirements and State-Of-The-Art Analysis for Service Usage” [11].

2 CONTENT OF THE DELIVERABLE 4.6

The deliverable comprises the following content:

- Requirements on the Invocation Component (Section 3)
- Fundamental approaches of Web Services invocation (Section 4)
- State-Of-The-Art Technologies (Section 5)
- WS Interoperability Requirements (Section 6)
- Invocation Module Architecture (Section 7)
- Invocation Module API (Section 8)
- Proposed Implementation of the DIP Invocation Module (Section 9)

Section 3 summarizes requirements on the invocation component specified in the requirements document, D4.1 and discusses the relationship between the invocation task the needs of Semantic Web Services.

Section 4 discusses two fundamental approaches for realization of the Web Services invocation.

In Section 5, State-Of-The-Art Technologies, two technologies representing the approaches discussed in Section 4 are described in detail.

Section 6 elaborates the requirements and restrictions on the usage of Web Services standards in order to guarantee interoperability. The architecture of the invocation module is introduced in Section 7 and is followed by the invocation module API specification in Section 8.

Finally, Section 9 proposes some recommendations for the implementation of the DIP Invocation Module.

3 REQUIREMENTS FOR THE INVOCATION COMPONENT

This section describes requirements for the invocation component as specified by “D4.1 Requirements and State-Of-The-Art Analysis for Service Usage” [11]. Furthermore, it discusses the relationship to the invocation of Semantic Web Services, since this issue raises some open questions that could not be completely solved during the work on this specification.

3.1 Requirements specified in D4.1

The following requirements specified in D4.1 have been taken as the starting point for the invocation module specification:

- The invocation component belongs to the implementation level. As a matter of fact, the invocation component does not deal with any semantic information at all. It simply performs the mundane task of placing a call to a remote Web Service according to a description and a payload that it receives from client components.
- A WSDL document describes which Web Service to call and how to call it and is part of the inputs of the invocation component. Other inputs include the actual payload of the request (i.e., the data passed as a parameter of the call to the remote Web Service).
- The invocation component needs a defined API to allow other components to use it. The functionalities of the invocation component cannot be expressed as a Web Service. A solution could be a small library in, e.g. Java, that client components can use to call the invocation component.

The analysis of the invocation task in D4.1 produced results that describe the perimeter of the responsibilities of the invocation component as follows:

- The task of the invocation component is to perform an actual call to exactly one external Web Service. This is done according to the WSDL description and the actual parameters to the call that the invocation component receives as input from its client components.
- This implies that the invocation component does not have to deal with any semantic information at all. In particular, a task such as the execution of a composition of services is outside the scope of the invocation component. Of course, a component that would perform the task of executing a complex composition of services would use the invocation component for invoking each individual service in the composition.
- Non-functional properties of the call such as, for example, the caching of outputs or error handling, are also not covered by the invocation component. If such functionalities are required, the Composition module will take care of

them. One possible way to do that would be for the Composition module to introduce in the composition extra *proxy* components that implement those non-functional properties such as security, validation, caching, monitoring, replication, error handling, etc.

- The invocation component must be able to invoke any service that is described using a well-formed WSDL document, with no *a priori* knowledge of the service required.

The invocation module specification described in this document fully covers the above requirements.

3.2 Relationship to Semantic Web Services

The initial reviews see a gap between the requirements covered by the current invocation module specification and invocation needs of Semantic Web Services. Thus, we decided to shortly discuss this issue in this document.

The execution of composed (orchestrated) services as well as compensations is in the responsibility of a business process execution engine (in DIP's case WSMX), as has been already stated in the previous section. Business process execution engines based on, e.g. BPEL4WS¹ provide an example for a non-semantic execution of orchestrated services.

From the viewpoint of the invocation task, we consider the following SWS requirements as relevant:

- Dynamic Invocation
- Asynchronous Interaction
- Semantic Interoperability

Dynamic invocation or binding means that the invocation component must be able to invoke any service with no a priori knowledge of the required service. This issue is covered by the specification.

Asynchronous interaction means that services will need to be able to send a message to DIP, i.e. a message receiver is required. In the DIP architecture document, D6.2, the invoker and receiver are combined in a component called Communication Manager. This was not foreseen in the requirements document, D4.1. A possible solution is to implement WSDL's "Request-Response" Message Exchange Pattern as asynchronous interaction. There is an experimental addition to WSIF providing support for asynchronous request-response. In that asynchronous request-response model, the response is handled in a different thread of execution from the originating request. To support this, the requestor registers a callback object or handler that is invoked when the response is received [13]. The current specification can be extended to support asynchronous request-response.

¹ See: <http://www-128.ibm.com/developerworks/library/ws-bpel/>

From DIP's perspective, a Semantic Web Service is either WSML-based or not WSML-based. A WSML-based service can natively handle WSML requests. In this case the invoker component will send data represented in WSML directly to the service. A service, which is not WSML-based, cannot handle WSML, so the invocation component needs to ask for some translation. In the architecture document, D6.2, this translation is the responsibility of an adaptor. In this case, the adaptor acts as a proxy for the service. The invocation component only has responsibility for sending the WSML data to the correct service or adaptor. At the time being it is not clear, how the invocation component will know when to use an adaptor and when it can send a WSML message directly to a service (Most likely the choreography part of the service description will indicate if a service can natively handle WSML. However, the choreography description is not finalized yet, so this issue remains open at this point of time.)

4 FUNDAMENTAL APPROACHES

The main task of the invocation component is to perform an actual call to an instance of a Web Service, i.e., to provide the Web Service instance with the required inputs (a call according to the description of the remote Web Service and a payload it receives from client components). The service will carry out its function and return any outputs to the invocation component. The invocation component will then return the received outputs, e.g., payload or exceptions, to the client component that placed the call.

The description of the Web Service interface and how to call it is described in a WSDL document [10]. The actual call is normally realized by an exchange of SOAP messages but also WSDL bindings to other technologies are possible, e.g., XML-RPC, Java-RMI, etc.

There are therefore two fundamental approaches to realize an invocation component. In the first approach the client component extracts the binding information from the WSDL document and calls the invocation component with the operation and payload already formatted in the protocol contained in the WSDL binding, e.g. SOAP (Figure 1)[8].

The disadvantage of this approach is that the client component has to support all possible bindings contained in WSDL. This approach adds unnecessary complexity to the client components and makes them difficult to maintain, since each time a new binding to be supported is introduced all client components have to be re-implemented. This approach is realized by, e.g., Apache SOAP and its successor Apache Axis [1],[2].

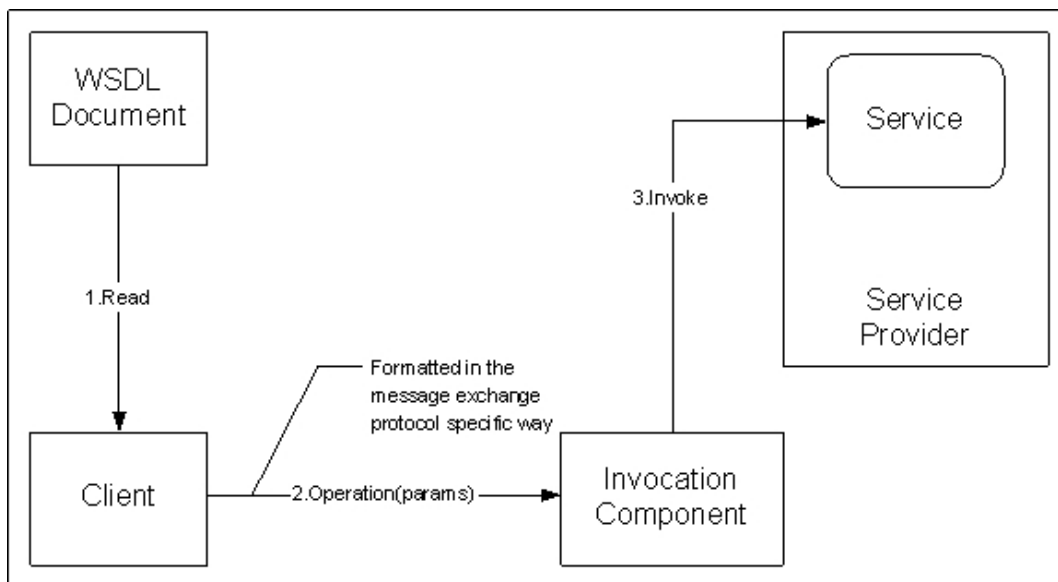


Figure 1 Message exchange protocol dependent invocation

In the second approach, the client component calls the invocation component with the WSDL document of the Web Service instance, the abstract operation to be called and the actual payload (i.e., the data² to be passed as a parameter of the call to the Web Service instance). In this approach it is the task of the invocation component to format the operation in a message exchange protocol specific way (Figure 2). The invocation component uses Service Messaging Interfaces (SMI) in order to communicate with Web Services over different messaging protocols (e.g., SOAP, Java-RMI, ...). This means, that the invocation component uses a SOAP Service Messaging Interface if the WSDL document specifies a SOAP binding, and it uses a Java-RMI SMI, if the WSDL document specifies a Java-RMI binding. However, for DIP at the time being only SOAP bindings are relevant.

This approach is more powerful than the approach shown in Figure 1, since it allows to program client components against an abstract service description, without any dependencies to the underlying “messaging protocol”. This makes the implementation of the client components independent of the WSDL bindings, i.e. if a new binding has to be supported only the invocation component has to be changed. For the client components these changes remain transparent. This approach is realized by, e.g., Apache Web Services Invocation Framework (WSIF) [18].

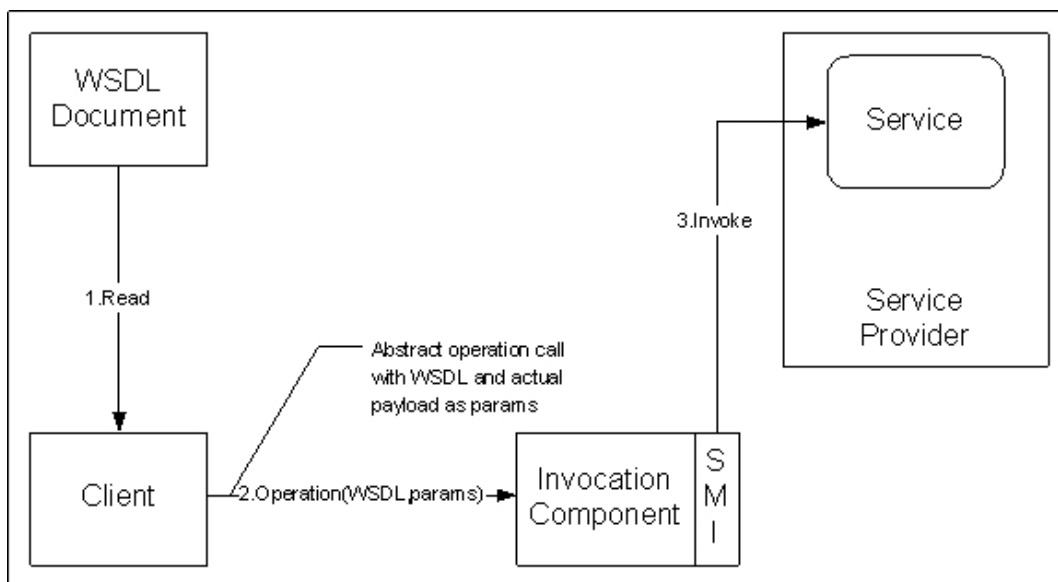


Figure 2 Message exchange protocol independent invocation

The analysis of the above approaches shows that the invocation should be based on the WSDL description and hide details of the messaging protocol (e.g. SOAP). The next section provides a detailed description of the state-of-the-art technologies representing solutions for the introduced approaches.

² In DIP the data is described in WSML/XML syntax, see <http://www.wsmo.org/2004/d16/d16.3/v0.1/20040909/>

5 STATE-OF-THE-ART

In order to explore the current state-of-the-art concerning Web Services invocation technologies, representatives of the two fundamental approaches described above are introduced. Apache SOAP API represents the first, SOAP-based approach. Apache WSIF represents the second, WSDL-based approach.

5.1 Apache SOAP API

Apache SOAP is an open-source implementation of the SOAP v1.1 specification in Java [2], [8].

Apache SOAP can be used as a client library to invoke SOAP services available elsewhere or as a server-side tool to implement services accessible via SOAP. As a client library it provides an API for invoking SOAP RPC services as well as an API for sending and receiving SOAP messages. As a mechanism to write new RPC accessible services or message-accessible services, it expects to be hosted by a servlet container (such as Apache Tomcat, for example). While the codebase can be extended to support non-HTTP transports, the provided code only has limited support for non-HTTP transports (specifically, only for SMTP). The invocation component is a client-side component, therefore only the client-side functionality of Apache SOAP is elaborated further.

A call of a Web Service consists of the following basic steps:

- Identify the service
- Identify the operation
- Identify the parameters
- Execute the operation
- Extract the result or the fault

The details of these steps are described in the following for the API for invoking SOAP RPC services and for the API for sending and receiving SOAP messages, respectively.

5.1.1 API for invoking SOAP RPC services

Writing clients to access SOAP RPC-based services is fairly straightforward. Apache SOAP provides a client-side API to assist in the construction of the SOAP request, and then to assist in interpreting the response. Conceptually, RPC-based services are relatively easy to understand, because they rely on the same concepts as any procedural language. To invoke a procedure, you need the name of the procedure and the parameters to pass to it. When the invocation completes, you need to extract any response information from the return value and/or output parameters.

The basic steps for creating a client which interacts with a SOAP RPC-based service are as follows:

Identify the service

- Obtain the interface description of the SOAP service, so that you know what the signatures of the methods that you wish to invoke are. (e.g. look at a WSDL file or directly at its implementation.)
- Make sure that there are serializers registered for all parameters which you will be sending, and deserializers for all information which you will be receiving back.
(Parameters must be serialized into/deserialized from XML before they can be transmitted/received, and so Apache SOAP provides a number of pre-defined serializers/deserializers which are available. If you need to transmit or receive a type which has not been registered, then you will need to write and register your own serializer/deserializer.)
- Create the `org.apache.soap.rpc.RPCMessage.Call` object. (The Apache SOAP Call object is the main interface to the underlying SOAP RPC code.)
- Set the target URI into the Call object using the `setTargetObjectURI(...)` method. (Pass in the URN that the service used to identify itself in its deployment descriptor.)

Identify the operation

- Set the method name that you wish to invoke into the Call object using the `setMethodName(...)` method. This must be one of the methods exposed by the service which is identified by the URN given in the previous step.

Identify the parameters

- Create any Parameter objects necessary for the RPC call and set them into the Call object using the `setParams(...)` method. (Make sure that you have the same number of parameters with the same types as the service is expecting. Also make sure that there are registered serializers/deserializers for the objects which you will be transmitting/receiving.)

Execute the operation

- Execute the Call object's `invoke(...)` method and capture the Response object which is returned from `invoke(...)`. (The `invoke(...)` method expects two parameters, the first is a URL which identifies the endpoint at which the service resides (i.e. `http://localhost/soap/servlet/rpcrouter`) and the second is the value to be placed into the SOAPAction header. Remember that the RPC call is synchronous, and so may take a while to complete.)

Extract the result or the fault

- Check the Response object to see if a fault was generated using the `generatedFault()` method.

- If a fault was returned, retrieve it using the `getFault(...)` method, otherwise extract any result or returned parameters using the `getReturnValue()` and `getParams()` methods respectively.

Because SOAP is supposed to be a standard, the clients created with the Apache SOAP API should be able to access services running on different implementations, and vice versa.

The above steps will be common in every SOAP implementation and in any invocation of a service that is accessible via an RPC-oriented protocol. However, the Apache SOAP API specific nature of the code makes it difficult to generalize the client code and make it more protocol-independent.

5.1.2 API for sending and receiving SOAP messages

Writing clients to access message-oriented SOAP services requires that you interact with a lower-level set of Apache SOAP APIs than you would otherwise have to if you were writing a SOAP RPC-based client. However, message-oriented services provide you with a finer grain of control over what is actually being transmitted over SOAP. (In fact, the RPC mechanism is built on top of this message-oriented layer.)

The basic steps for creating a client which interacts with a message-oriented SOAP service are as follows:

Identify the service

- Obtain the interface description of the SOAP service, so that you know what the format of the SOAP message should be (i.e. what headers it should have, what the body should look like, etc.) as well as the type of message exchange which will take place. (E.g., look at a WSDL file for the service, or directly at its implementation. Unlike SOAP RPC, there is no predefined message exchange pattern defined, so a message-oriented service may return a SOAP envelope, may return another type of data, or may return nothing at all.)

Identify the operation and parameters

- Construct an `org.apache.soap.Envelope` which contains the information that the SOAP service requires. At the very least, you will need to add an `org.apache.soap.Body` object to the envelope. You can optionally add headers as well. (Note: When the message is received on the server, it will be routed to the proper service by looking at the XML Namespace associated with the first child element in the body, and then to the correct method/function within that service via the name of the element itself.)
- Create an `org.apache.soap.messaging.Message` object. If you need to add MIME attachments to your message, then you can use the `addBodyPart(...)` method to do so.

Execute the operation

- Invoke the send(...) method on the Message object, providing the URL of the endpoint which is providing the service (e.g. http://localhost/soap/servlet/messengerouter), the actionURI, and your envelope.

Extract the result or the fault

- If your service is returning data, and assuming that the transport supports two-way interaction, then you need to retrieve the SOAPTransport object from the Message object (assuming that you don't already have a handle to it) by using the getSOAPTransport() method. You can then invoke the receive() method on the SOAPTransport object to retrieve the returned data.
- If the service is returning a SOAP Envelope, then you can parse the XML and pass the root element to org.apache.soap.Envelope's unmarshall(..) method to allow it to reconstruct a SOAP Envelope object for you.
- If an error has occurred on the server during the processing of the request, the server will automatically send back a SOAP Envelope with a SOAP Fault in the body describing what went wrong.

5.2 Apache WSIF

The Web Services Invocation Framework (WSIF) allows the application programmer to program against an abstract service description, in a protocol-independent manner [18]. WSIF supplies a simple API to invoke Web Services, no matter how or where the service is provided, so long as the service is described in WSDL. WSIF enables the user to interact with abstract representations of services instead of working directly with, e.g., SOAP APIs. The user works with the same programming model regardless of how the service is implemented or accessed. WSIF also supports dynamic adding of new bindings and dynamic replacement of current bindings [14].

In general, Web Services are not just SOAP. SOAP is a very important protocol providing a considerable contribution towards interoperability. However, also functionality not accessible via SOAP can be considered to be Web Services. In a broader view anything available over the network, with some associated functional description, can be considered to be a Web Service, e.g., EJB invocation using RMI/IIOP³. Since Web Services can be accessed in many different ways the functional descriptions of these services can be quite different. The Web Service Description Language (WSDL) provides a uniform way to describe heterogeneous services [10].

WSDL defines the functional characteristics of a Web Service in an abstract way, and allows binding of this abstract definition to a concrete access mechanism. The abstract portion of WSDL defines the interface of the Web Service, including the operations, the

³ <http://www.javaworld.com/javaworld/jw-12-1999/jw-12-iiop.html>

message parts, and their types. The concrete portion defines how the service can be accessed using some well-known communication protocol or mechanism; it includes the service endpoints and their protocol bindings. WSDL also allows and encourages the addition of new protocol bindings.

The usual Web Services programming model assumes SOAP as the underlying protocol, and this requires that the user knows the details of a particular SOAP client-side implementation. This becomes a problem when the user wants to use a different SOAP package; it will require re-integration of the SOAP code, because of a different API. In addition, if the user desires to access Web Services using a mechanism other than SOAP, a more generic set of APIs is required. The problem is how to operate on the level of the abstract service description, while still allowing multiple protocols to be used.

WSIF is a framework which allows the user to operate at a higher level than the usual protocol-specific programming entails. With WSIF, the user programs against the abstract service representation, instead of programming against a specific client-side implementation of a protocol such as SOAP. WSIF is a set of APIs which provide the user with a uniform means to consume Web Services, regardless of the way in which the service is implemented or provided. The only requirements are that the service has to be described in WSDL and that the appropriate protocol binding implementation is plugged into the framework.

WSIF supports two different approaches:

- The more traditional of the two approaches is to compile the WSDL document into a Java interface, an implementation of that interface (called the stub), and the necessary Java types, which reflect the specified WSDL document. The service can then be accessed using the generated Java stub.
- The other approach is to operate directly on arbitrary WSDL documents. The key difference is that no compilation cycle is required when using this approach. (The dynamic binding is inevitable in an open environment such as DIP.)

A WSDL document contains an abstract definition of a service. This definition is abstract because it only describes the operations and the types of the messages the operations receive and return. The types are described in terms of XML Schema types. However, the abstract definition alone is not enough to consume a service. Therefore, a binding to a concrete protocol, e.g., SOAP, is needed. It is necessary to translate the WSDL binding and port contained in the WSDL document into API calls using a particular protocol.

The steps in order to invoke a Web Service using WSIF APIs are similar to the steps discussed for the Apache SOAP API. However, they are protocol-independent.

Identify service

Load the WSDL document and use it with `WSIFServiceFactory` to create `WSIFService`. A `WSIFService` is a factory via which `WSIFPorts` are retrieved. Use the `WSIFService` to retrieve a `WSIFPort`. A `WSIFPort` represents a handle by which the operations of this `WSIFPort` can be executed.

Identify operation

Use the `WSIFPort` to create a `WSIFOperation`. There must be an operation in this port's `portType` with this operation name, input message name and output message name. The input message name distinguishes overloaded operations.

```
// Identify service
// Load definition from the WSDL document
javax.wsdl.Definition def =
org.apache.wsif.util.WSIFUtils.getDefinitionFromLocation("context
URL","location");
// Create WSIFService object using WSIFServiceFactory with WSDL
definition as parameter.
WSIFService serv = WSIFServiceFactory.getService(def);
// Create WSIFPort using WSIFService object with port name as
parameter.
WSIFPort port = serv.getPort("portName");
// Create WSIFOperation using WSIFPort with operation, input and
output names as parameters.
WSIFOperation op = WSIFPort.createOperation("operationName",
"inputName", "outputName");
// Create container for the input message
WSIFMessage input = op.createInputMessage();
// Create container for the output message
WSIFMessage output = op.createOutputMessage();
// Create container for the fault message
WSIFMessage fault = op.createFaultMessage();
// Set parameter values for the input message, e.g.,
input.setIntPart("paramName", paramValue);
...
// Execute operation
boolean success = op.executeRequestResponseOperation(input,
output, fault);
// Extract the result or the fault
if success then{
    double param = Output.getDoublePart("name");
    // Process output
}
else{
    String faultName = fault.getName();
    // Process fault
}
}
```

Figure 3 Service invocation using WSIF

Identify parameters

Use the `WSIFOperation` to create a container for the input message that will be sent via this port. Use the `WSIFOperation` to create a container for the output message that will be received via this port. Use the `WSIFOperation` to create a container for the fault message that will be received via this port.

A `WSIFMessage` is an interface representing a WSDL Message. In WSDL, a Message describes the abstract type of the input or output of an operation. `WSIFMessage` is the corresponding WSIF class which represents in memory the actual input or output of an operation.

Use set() methods of WSIFMessage to set parameter values for the input message (A WSIFMessage is a container for a set of named parts. The WSIFMessage interface separates the actual representation of the data from the abstract type defined by WSDL.)

Execute the operation

Use the execute() methods of WSIFOperation to execute the operation. If the method returns true then the output message contains useful information. Otherwise a fault message has been generated.

Extract the result or the fault

Examine the return value of the executed operation. If it is true, use get() methods of the WSIFMessage to extract the result. If it is false, use get() methods of the WSIFMessage to extract the fault.

Figure 3 contains a listing of Java source code illustrating the invocation steps using WSIF.

6 WS INTEROPERABILITY REQUIREMENTS

In order to guarantee interoperability of Web Services in DIP, the clarifications and amendments on Web Services specifications described in the **Basic Profiles** published by WS-I have to be taken into account [4],[5]. The Web Services Interoperability Organization (WS-I) is an open, industry organization promoting Web Services interoperability. WSI has been specifically founded to create, promote, or support generic protocols for interoperable message exchange between services [17].

6.1 WS-I Requirements

The following statements rely on the Basic Profile Version 1.0. This profile covers three important aspects of Web Services:

- Messaging
- Service Description
- Service Publication and Discovery

Regarding Messaging, the profile incorporates the following specifications by reference, and defines extensibility points within them:

- Simple Object Access Protocol (SOAP) 1.1 [8]
- Extensible Markup Language (XML) 1.0 (Second Edition) [9]
- RFC 2616: Hypertext Transfer Protocol (HTTP/1.1) [12]

Regarding Service Description, the profile incorporates the following specifications by reference, and defines extensibility points within them:

- Web Services Description Language (WSDL) 1.1 [10]
- XML Schema Part 1: Structures [15]
- XML Schema Part 2: Datatypes [7]

Regarding Service Publication and Discovery, the profile incorporates the following specifications by reference, and defines extensibility points within them:

- UDDI Version 2 API Specification [6]
- UDDI Version 2 Data Structure Reference [16]

In the scope of this document, the following clarifications on the usage of WSDL 1.1 have a particular impact on the functionality of the invocation module:

- Required Description
- Messages
- Port types
- Bindings

Required description: Either a service instance's WSDL 1.1 description, its UDDI binding template, or both **MUST** be available to an authorized consumer upon request.

This means that if an authorized consumer requests a service description of a conformant service instance, then the service instance provider must make the WSDL document, the UDDI binding template, or both available to that consumer.

Messages: In WSDL 1.1, `wsdl:message` elements are used to represent abstract definitions of the data being transmitted. It uses `wsdl:binding` elements to define how the abstract definitions are bound to a specific message serialization. The WS-I Basic Profile supports two types of message serialization: “document-literal” and “rpc-literal” bindings [4]. DIP should support exactly these two bindings as stated in D6.1 [3].

Port types: Solicit-Response and Notification operations are neither well-defined in WSDL 1.1 nor does it define bindings for them. Therefore, a Web Service Description must not use Solicit-Response and Notification type operations in a `wsdl:portType` definition. Furthermore, the WS-I Basic Profile disallows operation name overloading in `wsdl:portType`.

Bindings: The WS-I Basic Profile restricts the use of bindings to the WSDL/SOAP binding as defined in WSDL 1.1 Section 3. Furthermore, only HTTP as the underlying transport protocol and “literal”-encoding are allowed.

6.2 WSDL 2.0

This section briefly discusses the development of WSDL 2.0 and its impact on invocation in DIP.

Currently, the Web Services Description Working Group⁴ within the W3C Web Services Activity is working on a new version of WSDL – WSDL 2.0. In August 2004 it has released Last Call Working Drafts for WSDL Version 2.0. Public comments had to be made through October 4, 2004. However, it seems that comments were received and processed in November 2004.⁵

WSDL 2.0 is significantly different from WSDL 1.1, e.g. messages are dropped as components and interfaces are used instead of portTypes (interfaces support inheritance). Moreover, features and properties have been added to allow the modelling of service properties/features such as reliability, security, or routing (these examples are not part of the specification but are cited as potential usages for properties/features).

Three formal objections to portions of the draft have been submitted by Working Group participants as minority opinions. Interestingly, IBM, Microsoft and SAP object to the inclusion of features and properties, which they seem to prefer to be covered by WS-Policy or XML-based extensibility.⁶ On the other hand, IONA, Oracle, Sonic and SUN are proposing compositors for features and properties, allowing the expression of dependencies between features and properties.⁷ They object to not including these compositors in the draft and criticise reliance of a non-W3C-standard like WS-Policy.

⁴ <http://www.w3.org/2002/ws/desc/>.

⁵ <http://lists.w3.org/Archives/Public/public-ws-desc-comments/>.

⁶ <http://lists.w3.org/Archives/Public/www-ws-desc/2004Jul/0375.html>.

⁷ <http://lists.w3.org/Archives/Public/www-ws-desc/2004Jul/0371.html>.

WSDL 2.0 will likely still change before it becomes a W3C Recommendation, as comments and criticisms⁸ are processed by the Working Group. Additionally, it remains to be seen whether, when and how software vendors will support WSDL 2.0 in their Web Services products. It is therefore very unlikely that invocation in DIP will have to deal with WSDL 2.0 since WSDL 1.1 is most probably going to stay the main Web Services description language for the duration of the project. On the other hand, a short examination of the impact of WSDL 2.0 should be planned for the later stages, when WSDL 2.0 is accepted as a standard.

⁸ See, for example, Rich Salz, WSDL 2: Just Say No, XML.com, November 2004, <http://www.xml.com/lpt/a/2004/11/17/salz.html>.

7 INVOCATION MODULE ARCHITECTURE

The architecture of the invocation module relies on the following key abstractions from the WSDL specification (Figure 4):

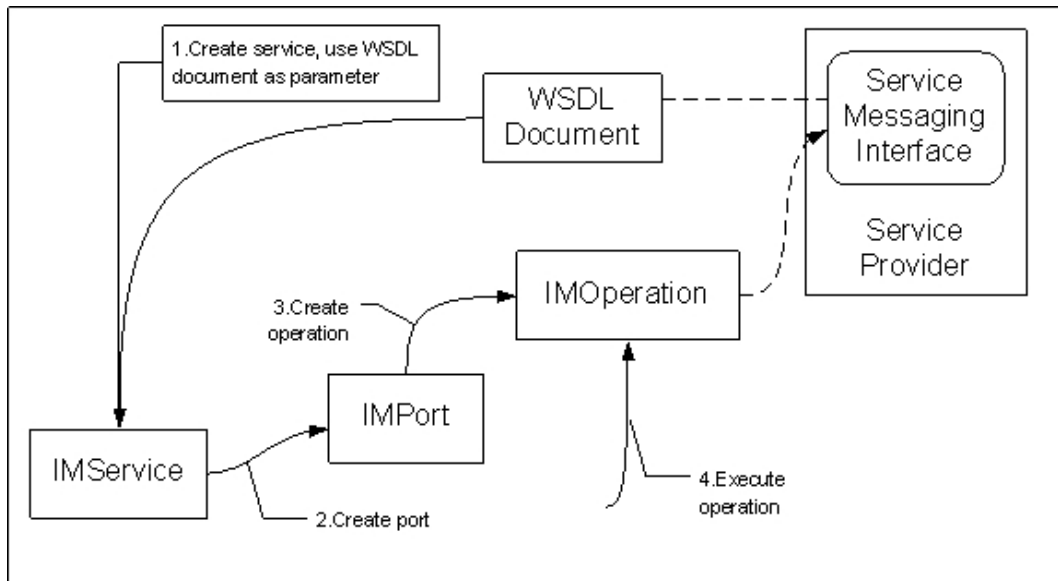


Figure 4 Architecture of the invocation module

- WSDL Document
- IService
- IMPort
- IMOperation

In the bindings part, a WSDL document contains the address of the Web Service. The bindings define also the protocol and formats for operations and messages for a specific port type.

IService is an interface responsible for the creation of specific port instances.

IMPort is an interface responsible for the creation of operations served by this specific port type.

IMOperation is a run-time representation of a Web Service operation. It is responsible for the actual call of a Web Service instance and relies on a binding to a specific Service Messaging Interface (e.g. SOAP). The implementation of a respective Service Messaging Interface is provided by a provider (e.g., for SOAP Apache SOAP API or Apache Axis).

In this architecture, only four steps (for request-response Message Exchange Pattern(MEP) five steps) are required in order to execute an actual call to a Web Service instance (Figure 4):

1. Create Service (with the WSDL document as parameter)
2. Create Port (with the port name as parameter to indicate the required port)
3. Create Operation (with the operation name as parameter to indicate the required operation)
4. Execute operation (with an input message as parameter)
5. If Request-Response MEP: Read output message or fault

8 INVOCATION MODULE API

The invocation module API, described in the following, has been designed to support WSDL 1.1 with a SOAP 1.1 (HTTP1.0/1.1) binding [8],[10]. Additionally, the API takes into account further restrictions on the WSDL 1.1 specification as specified by the Web Services Interoperability Organization in its Basic Profile [4].

The most important restrictions are:

- Support for “document-literal” and “rpc-literal” encoding styles only
- Support for consistent encoding style only (do not mix encoding styles, either use “rpc-literal” or use “document literal”)
- Support for “document-literal” style with only one message part
- Support for “One way” and “Request-Response” message exchange patterns only

The API specifies only functionality necessary to perform invocation of a Web Service according to the above restrictions. The justification for the above restrictions is described in detail in WS-I’s Basic Profile [4].

Note: There are no recommendations from WS-I for WSDL versions higher than 1.1. Therefore, for the time being support of higher versions would be in conflict with the WS Interoperability Requirements.

Class
<p><i>dip.invocationmodule.IMService</i></p> <p>Description: IMService is used to retrieve IMPorts. A WSDL description can contain descriptions for one or more services. A service description can contain descriptions for one or more port types. Therefore, appropriate constructor has to be chosen to make IMService unambiguous, because IMService is assumed to be for a specific port type</p>
Constructors
<p>dip.invocationmodule.IMService (java.lang.String wsdlUrl);</p> <p>Description: This constructor takes as argument a URL to a remote WSDL document and can be used if the WSDL document contains only one service and port type.</p>
<p>dip.invocationmodule.IMService (java.lang.String wsdlUrl, java.lang.String serviceNS, java.lang.String serviceName, java.lang.String portTypeNS, java.lang.String portTypeName);</p>

<p>Description: This constructor takes as argument a URL to a remote WSDL document. The service and port type have to be specified using arguments for service and port type names and name spaces.</p>
<p>dip.invocationmodule.IMService (javax.wsdl.Definition def);</p> <p>Description: This constructor takes as argument a javax.wsdl.Definition object containing WSDL document and can be used if the WSDL document contains only one service and port type.</p>
<p>dip.invocationmodule.IMService (javax.wsdl.Definition def, java.lang.String serviceNS, java.lang.String serviceName, java.lang.String portTypeNS, java.lang.String portTypeName);</p> <p>Description: This constructor takes as argument a javax.wsdl.Definition object containing WSDL document. The service and port type have to be specified using arguments for service and port type names and name spaces.</p>
<p>Methods</p>
<p>dip.invocationmodule.IMPort getPort(String portName)</p> <p>Description: Returns an IMPort for the specified port name.</p>

<p>Class</p>
<p><i>dip.invocationmodule.Class IMPort</i></p> <p>Description: IMPort is used to select operations to be executed</p>
<p>Constructors</p>
<p>Instantiate using getPort() method of the class IMService</p>
<p>Methods</p>
<p>void close()</p> <p>Description: Close this port if done using it.</p>
<p>dip.invocationmodule.IMOperation createOperation(String operationName)</p> <p>Description: Returns an IMOperation for the specified port name.</p>
<p>dip.invocationmodule.IMOperation createOperation(String operationName, String inputName, String outputName)</p> <p>Description: Returns an IMOperation for the specified port name, input message name, and output message name. The inputName and outputName parameters are optional and are required only for correct identification of overloaded operations. Web</p>

Services conformant to WS-I's Basic Profile do not use overloading anyway, so for those services it is safe to drop inputName and outputName parameters. Note: One-way operations require only input arguments.

Class
<p><i>dip.invocationmodule.IMOperation</i></p> <p>Description: IMOperation is used to create input, output, and fault IMMMessage, and to execute operation.</p> <p>Note: WSDL has four transmission primitives that an endpoint can support:</p> <ul style="list-style-type: none"> • One-way. The endpoint receives a message. • Request-response. The endpoint receives a message, and sends a correlated message. • Solicit-response. The endpoint sends a message, and receives a correlated message. • Notification. The endpoint sends a message. <p>Although the base WSDL structure supports bindings for these four transmission primitives, WSDL only defines bindings for the One-way and Request-response primitives. Furthermore, as stated in the previous section, WS-I limits the use of transmission primitives in Web Service Descriptions to One-Way and Request-Response.</p> <p><i>Note: The interface as described supports one-way and synchronous request-response. If a specification for asynchronous request-response is required one possible approach is to adapt the experimental solution from WSIF (There is an experimental addition to WSIF providing support for asynchronous request response. In their asynchronous request-response model, the response is handled in a different thread of execution from the originating request. To support this, the requestor registers a callback object or handler, that is invoked when the response is received [13]).</i></p>
Constructors
<p>Instantiate using createOperation() methods of the class IMPort.</p>
Methods
<p>dip.invocationmodule.IMMessage createInputMessage(String name)</p> <p>Description: Create input message for the indicated name.</p>
<p>dip.invocationmodule.IMMessage createOutputMessage(String name)</p> <p>Description: Create output message for the indicated name (Only required for request-response operation).</p>
<p>dip.invocationmodule.IMMessage createFaultMessage(String name)</p> <p>Description: Create fault message for the indicated name (Only required for request-</p>

response operation).
void executeInputOnlyOperation(IMMessage input) Description: Execute one-way operation providing IMMessage as input.
boolean executeRequestResponseOperation(IMMessage input, IMMessage output, IMMessage fault) Description: Execute request-response operation providing IMMessages as input, output and fault. The method returns true if output message has been received. The method returns false if fault message has been received.
java.lang.String getEncodingStyle() Description: Returns the encoding style: “document-literal” or “rpc-literal”.

Class
<p><i>dip.invocationmodule.IMMessage</i></p> <p>Description: An IMMessage represents a WSDL Message. In WSDL, a Message describes the abstract type of the input or output to an operation. IMMessage represents in memory the actual input or output message of an operation and separates the actual representation of the data from the abstract type defined by WSDL.</p> <p>An IMMessage has a message name, which can be used to distinguish between messages.</p>
Constructors
Instantiate using createInputMessage(), createOutputMessage(), or createFaultMessage methods of the class IMOperation.
Methods
java.lang.String getName() Description: Get the name of the IMMessage.
void setName(java.lang.String) Description: Set a name for the IMMessage.
void setDocLitPart(java.lang.String name, org.w3c.dom.Element part) Description: Set the document-literal argument to the input message. The method requires the name of the top-level part of the document-literal message and the document-literal message as org.w3c.dom.Element
org.w3c.dom.Element getDocLitPart(java.lang.String name) Description: Get the document-literal message for the indicated name of the top-level

part.
void setRpcLitPart(java.lang.String name, org.w3c.dom.Element part) Description: Set the rpc-literal argument to the input message. The method requires the name of the argument of the rpc-literal part and the rpc-literal part as org.w3c.dom.Element
org.w3c.dom.Element getRpcLitPart(java.lang.String name) Description: Get the rpc-literal part for the indicated argument name.

Class
<i>dip.invocationmodule.IMException</i> Description: An IMException is thrown in order to indicate something going wrong with the DIP invocation module and not something related to the service invocation itself.
Constructors
IMException(java.lang.String msg)
IMException(java.lang.String msg, java.lang.Throwable targetException)
Methods
java.lang.Throwable getTargetException()
setTargetException(java.lang.Throwable targetException)

9 PROPOSED IMPLEMENTATION OF THE DIP INVOCATION MODULE

Apache WSIF provides all functionality necessary for dynamic invocation of Web Services compliant with WSDL 1.1 and SOAP 1.1 specifications. In addition, it also supports bindings to Non-SOAP Web Services (e.g., EJB, JMS, JCA). However, in DIP, only SOAP bindings must be supported. Thus, the DIP Invocation Module is not required to support WSDL bindings to Non-SOAP messaging APIs.

We propose to implement the DIP Invocation Module as a light-weight Java component around Apache WSIF implementation. This component is required to:

- Support WSDL 1.1 with SOAP 1.1/HTTP 1.0/1.1 binding.
- Support “document-literal” and “rpc-literal” message encoding.
- Support One-Way and Request-Response transmission primitives.
- To guarantee interoperability, the restrictions specified in the WS-I Basic Profile should be taken into account.

10 CONCLUSION

In this document we have specified the DIP invocation module. The contribution of this deliverable is twofold:

- to provide an API specification of the DIP invocation module to be used by the DIP components (e.g., WSMX) required to perform a Web Services call and
- to provide the implementers of the invocation module with the specification of the API to be implemented and recommendations how to implement it.

We have described the fundamental architectural approaches solving the invocation problem and presented some State-of-The-Art technologies realizing them.

The restrictions described in the WS-I Basic Profile and/or decided in DIP allow reducing the complexity of the DIP Invocation Module Interface further in comparison to e.g., Apache WSIF. The result is a very simple API, which is easy to understand and use.

REFERENCES

- [1] Apache AXIS, <http://ws.apache.org/axis/>
- [2] Apache SOAP API, <http://ws.apache.org/soap/>
- [3] Altenhofen, M et all; D6.1 Report on requirements analysis and state-of-the-art, DIP Deliverable, November 2004
- [4] Ballinger, K et all; Basic Profile-Version 1.0 Final, WSI, <http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html>
- [5] Ballinger, K et all; Basic Profile-Version 1.1 Final, WSI, <http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html>
- [6] Bellwood, T. et all; UDDI Version 2.04 API Specification, July 2002, <http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm>
- [7] Biron, P.V. et all; XML Schema Part 2: Datatypes Second Edition, October 2004, <http://www.w3.org/TR/xmlschema-2/>
- [8] Box, D. et all; Simple Object Access Protocol (SOAP) 1.1, May 2000, <http://www.w3.org/TR/2000/NOTE-SOAP-20000508/>
- [9] Bray, T. et all; Extensible Markup Language (XML) 1.0 (Second Edition), October 2000, <http://www.w3.org/TR/2000/REC-xml-20001006>
- [10] Christensen, E. et all; Web Service Description Language (WSDL) 1.1, March 2001, <http://www.w3.org/TR/wsdl>
- [11] Duke, A. et all; D4.1 Requirements and State-Of-The-Art Analysis for Service Usage, DIP Deliverable, June 2004
- [12] Fielding, R. et all; RFC2616: Hypertext Transfer Protocol- HTTP/1.1, <http://www.ietf.org/rfc/rfc2616>
- [13] Fremantle, P; Applying the Web Services Invocation Framework, <http://www-106.ibm.com/developerworks/webservices/library/ws-appwsif.html?loc=dwmain>
- [14] Mukhi, N. K; How WSIF scores over the current client programming models for Web Services, <ftp://www6.software.ibm.com/software/developer/library/ws-wsif.pdf>
- [15] Thomson, H.S. et all; XML Schema Part 1: Structure Second Edition, October 2004, <http://www.w3.org/TR/xmlschema-1/>
- [16] von Riegen, C. et all; UDDI Version 2.03 Data Structure Reference, July 2002, <http://uddi.org/pubs/DataStructure-V2.03-Published-20020719.htm>
- [17] Web Services Interoperability organization, <http://www.ws-i.org>
- [18] Web Services Invocation Framework, <http://ws.apache.org/wsif/>