# TripCom

*Triple Space Communication*

## FP6 – 027324

Deliverable

# D4.2

# TripCom Grounding for Semantic Web Services

Omair Shafiq
Dario Cerizza
Jacek Kopecky
Daniel Martin
Martin Murth
Brahmananda Sapkota
Germán Toro del Valle
Andrea Turati
Daniel Wutke

March 28, 2008

# Executive Summary

Deliverable D4.2 - *TripCom Grounding for Semantic Web Services* presents details about grounding required for Semantic Web Services from all possible aspects in order to use Triple Space Computing for communication and coordination. It includes defining the TripCom groundings for Web Service registry, i.e. representing the registry data model in RDF and mapping registry API with TS API. From description language perspective, it describes representing WSDL in RDF. From communication protocol perspective, it describes the TripCom bindings in WSDL, representing SOAP in RDF and describes Web Service invocation in Triple Space. It further explores the grounding issues in Semantic Web Services and details the grounding requirements, specification and implementation guidelines in Semantic Web Service Execution Environment (WSMX), i.e. for the end-point Web Service invoker, client-endpoint, resource and component management. For further useful illustration, it uses a usecase as an example to describe the viability of the solution.

# Document Information

| **IST Project Number** | FP6 – 027324 | **Acronym** | TripCom |
|---|---|---|---|
| **Full Title** | Triple Space Communication | | |
| **Project URL** | http://www.tripcom.org/ | | |
| **Document URL** | | | |
| **EU Project Officer** | Werner Janusch | | |

| **Deliverable** | **Number** | 4.2 | **Title** | TripCom Grounding for Semantic Web Services |
|---|---|---|---|---|
| **Work Package** | **Number** | 4 | **Title** | Triple Space and Semantic Web Services |

| **Date of Delivery** | **Contractual** | M24 | **Actual** | 31-Mar-08 |
|---|---|---|---|---|
| **Status** | version 1.0 | | final ☐ | |
| **Nature** | prototype ☐ report ☒ dissemination ☐ | | | |
| **Dissemination Level** | public ☒ consortium ☐ | | | |



| **Authors (Partner)** | Omair Shafiq (STI), Dario Cerizza (CEFRIEL), Jacek Kopecky (STI), Daniel Martin (USTUTT), Martin Murth (TUW), Brahmananda Sapkota (NUIG), German Toro del Valle (TID), Andrea Turati (CEFRIEL), Daniel Wutke (USTUTT) | | | |
|---|---|---|---|---|
| **Resp. Author** | Omair Shafiq | | **E-mail** | omair.shafiq@sti2.at |
| | **Partner** | STI | **Phone** | +43 (512) 507 - 6480 |





| **Abstract (for dissemination)** | This deliverable describes grounding aspects related to Semantic Web Services integration with Triple Space Computing. It explores the grounding aspects from the perspective of web service registries, description and communication. It further takes into account WSMX as being a Semantic Execution Environment and describes the grounding requirements and specifications. Furthermore, it uses a usecase to illustrate the grounding aspects. |
|---|---|
| **Keywords** | Semantics, Web Services, Semantic Web Services, Triple Space Computing, Grounding |

| **Version Log** | | | |
|---|---|---|---|
| **Issue Date** | **Rev No.** | **Author** | **Change** |
| 14/04/2007 | 1 | O. Shafiq | Initial outline created |
| 15/05/2007 | 2 | O. Shafiq | Incorporated initial comments from partners |
| 10/06/2007 | 3 | O. Shafiq | Incorporated initial outlines from the partners responsible for their sections |
| 27/06/2007 | 4 | B. Sapkota | Updated Sections 2.1 and 2.1.3 |
| 19/07/2007 | 5 | D. Wutke | Initial version of section 3.2, 4.2 |
| 24/07/2007 | 6 | M. Murth | First draft on Web service invocation |
| 01/08/2007 | 7 | D. Cerizza | Added use-case information |
| 31/08/2007 | 8 | D. Wutke | Revised section 3.2 and 3.2.1 |
| 26/09/2007 | 9 | O. Shafiq | Compiling next updated version and incorporating comments |
| 15/10/2007 | 10 | D. Wutke | Added input in section 1.4 |
| 30/10/2007 | 11 | O. Shafiq | Added input in section 3.1 and 4.4 |
| 03/11/2007 | 12 | D. Cerizza | Updated use-case information |
| 12/11/2007 | 13 | B. Sapkota | Extended Section 2 |
| 20/11/2007 | 14 | D. Wutke | Revised section 3.3, 3.3.1 and 3.3.2 |
| 12/12/2007 | 15 | B. Sapkota | Added input in section 2.1.2, 2.1.3 and 4.3 |
| 15/12/2007 | 16 | O. Shafiq | Finalized section 4.4 |
| 28/01/2008 | 17 | B. Sapkota | Added section 2.1.4 |
| 28/01/2008 | 18 | D. Wutke | Revised section 3.2, 3.2.1, 3.3, 3.3.1, 3.3.2, 3.3.3 |
| 28/01/2008 | 19 | B. Sapkota | Finalised section 2.1.4 and added sections 4.3.1, 4.3.2 |
| 31/01/2008 | 20 | O. Shafiq | Finalized section 3.1 TripCom Bindings for WSDL |
| 30/01/2008 | 21 | D. Wutke | Added section 4.2.3 |
| 19/02/2008 | 22 | A. Turati | Added conclusions and several comments |
| 20/02/2008 | 23 | O. Shafiq | Finalized section |
| 26/02/2008 | 24 | O. Shafiq | Overall text revision, editing and comments |
| 03/03/2008 | 25 | B. Sapkota | Updated Section 4 |
| 03/03/2008 | 26 | O. Shafiq | Updates in Chapter 1 |
| 03/03/2008 | 27 | A. Turati | Addressed several comments in Chapter 2 |
| 04/03/2008 | 28 | D. Cerizza | Detailed Step 4 and inserted the picture |
| 05/03/2008 | 29 | D. Wutke | Finalized section 3.3, revised sections 5.2 and 5.3 |
| 05/03/2008 | 30 | D. Cerizza | Enriched the descriptions of the inner steps and added some thoughts after the picture |
| 07/03/2008 | 31 | O. Shafiq | Added WSML, WSDL, SOAP-RDF codes in use-case chapter 5 |
| 10/03/2008 | 32 | O. Shafiq | Review and updates in Executive Summary, Introduction and Conclusions |
| 13/03/2008 | 33 | D. Cerizza | Revised Chapter 5 and made some corrections |
| 14/03/2008 | 34 | D. Cerizza | Revised Chapter 6 and made some corrections |
| 14/03/2008 | 35 | O. Shafiq | Final revisions, updates in list of abbreviations, and other editings |
| 14/03/2008 | 36 | D. Wutke | Final revisions and updates in listings and references |
| 15/03/2008 | 37 | J. Kopecký | Revised section 2.2.2, put WSDL2RDF in Tripcom SVN |
| 25/03/2008 | 38 | B. Sapkota | Updated Section 2 to address QA's Comment - 2 |
| 27/03/2009 | 39 | D. Wutke | Implemented QA's comments in sections 3 and 4.2. |
| 27/03/2009 | 40 | O. Shafiq | Implemented QA comments |

# Project Consortium Information

| Acronym | Partner | Contact |
|---|---|---|
| Semantic Technology Institute Innsbruck<br>`http://www.sti-innsbruck.at` | STI | Prof. Dr. Dieter Fensel<br>Semantic Technology Institute (STI)<br>Innsbruck, Austria<br>E-mail: dieter.fensel@sti-innsbruck.at |
| National University of Ireland, Galway<br>`http://www.deri.ie` | NUIG | Dr. Laurentiu Vasiliu<br>Digital Enterprise Research Institute (DERI)<br>Galway, Ireland<br>Email: laurentiu.vasiliu@deri.org |
| University of Stuttgart<br>`http://www.iaas.uni-stuttgart.de/` | USTUTT | Prof.Dr. Frank Leymann<br>Inst. für Architektur von Anwendungssystemen (IAAS)<br>Stuttgart, Germany<br>E-mail: frank.leymann@informatik.uni-stuttgart.de |
| Vienna university of Technology<br>`http://www.complang.tuwien.ac.at/` | TUW | Prof.Dr. eva Kühn<br>Institut für Computersprachen<br>Vienna, Austria<br>E-mail: eva@complang.tuwien.ac.at |
| Free University Berlin<br>`http://www.ag-nbi.de/` | FUB | Prof. Dr.-Ing. Robert Tolksdorf<br>AG Netzbasierte Informationssysteme<br>Berlin, Germany<br>E-mail : tolk@inf.fu-berlin.de |
| Ontotext Lab, Sirma Group Corp.<br>`http://www.ontotext.com/` | ONTO | Atanas Kiryakov, Vassil Momtchev,<br>Ontotext Lab, Sirma Group Corp.<br>Sofia, Bulgaria<br>E-mail: vassil.momtchev@ontotext.com |
| Profium OY<br>`http://www.profium.com/` | Profium | Dr. Janne Saarela<br>Profium OY<br>Espoo, Finland<br>E-mail: janne.saarela@profium.com |
| CEFRIEL SCRL.<br>`http://www.cefriel.it/` | CEFRIEL | Davide Cerri<br>CEFRIEL SCRL.<br>Milano, Italy<br>E-mail: cerri@cefriel.it |
| Telefonica I+D<br>`http://www.tid.es/` | TID | Noelia Pérez Crespo<br>Telefonica I+D<br>Madrid, España<br>E-mail: npc@tid.es |

# Table of Contents

1 Introduction 2
1.1 Grounding in general . . . 2
1.2 Scope and Objectives . . . 3
1.3 Target Audience . . . 4
1.4 Motivation . . . 4
1.5 Related Work . . . 5
1.5.1 WSMO Grounding . . . 5
1.5.2 OWL-S Grounding . . . 5

2 Triple Space Grounding for WS Registries 8
2.1 Grounding Registry Data Model to RDF . . . 8
2.1.1 Naming Conventions for Mapping . . . 10
2.1.2 Vocabulary for Mapping Registry Data Model . . . 10
2.1.3 Representing Registry Data Model in RDF . . . 10
2.1.4 Implementation Guidelines . . . 10
2.2 Representing WSDL in RDF . . . 12
2.2.1 RDF vocabulary for service description mapping . . . 13
2.2.2 WSDL RDF mapping implementation . . . 16
2.2.3 WSDL RDF mapping example . . . 16

3 Triple Space Grounding for Web Service Communication 19
3.1 TripCom bindings for WSDL . . . 19
3.1.1 WSDL bindings with SOAP . . . 20
3.1.2 WSDL bindings with other protocols . . . 22
3.1.3 WSDL bindings with Triple Space . . . 23
3.2 SOAP Mapping with RDF . . . 25
3.2.1 Mapping SOAP envelopes to RDF . . . 26
3.3 Enabling Web Service interactions over Triple Space . . . 30
3.3.1 WSDL 1.1 Message Exchange Patterns . . . 30
3.3.2 WSDL 2.0 Message Exchange Patterns . . . 31
3.3.3 Requirements for Triple Space-based Web Service interactions . 31
3.3.4 Mapping Web Service message exchange patterns to TS API operations . . . 32
3.3.5 Extended non-standard MEP . . . 34
3.3.6 Summary . . . 35

4 Triple Space Grounding in Semantic Web Service Execution Environment 37
4.1 Grounding component in Web Service Execution Environment (WSMX) 37
4.2 TripCom grounding for Web Service invoker . . . 38
4.2.1 State of the art . . . 38
4.2.2 Mapping the WSMX invoker to TS API operations . . . 39
4.2.3 Implementation guidelines for extendending the WSMX Invoker with support for Triplespace . . . 40
4.3 TripCom grounding for client-endpoint . . . 42

4.3.1 Mapping WSMX Communication Manager API with Triple Space API . . . . . 44
4.3.2 Implementation Guidelines . . . . . 44
4.4 TripCom grounding for Resource Manager . . . . . 45
4.4.1 Mapping Resource Manager API with TS API . . . . . 45
4.4.2 Using ORDI for mapping WSMO top level elements to RDF . . 46
4.4.3 Implementation guidelines . . . . . 48
4.5 TripCom grounding for Component Management . . . . . 49
4.5.1 Architectural details for integration and grounding TripCom with WSMX Component Manager . . . . . 50
4.5.2 Implementation guidelines . . . . . 51

5 Run Through Example 52
5.1 Step 1: The Toothache and the Urgency to visit a Dentist while Abroad 53
5.2 Step 2: The Laboratory Examination for Additional Control of Patient Status . . . . . 56
5.3 Step 3: The Surgical Operation at a Regional Hospital . . . . . 58
5.4 Step 4: The Notification of the General Practitioner at Home . . . . . 60

6 Conclusions 62
6.1 Grounding Registry Data Model to RDF . . . . . 66
6.2 Representing Registry Data Model in RDF . . . . . 68
6.3 WSDL RDF Mapping Example . . . . . 69
6.4 Mapping Resource Manager API with TS API . . . . . 73

# List of Abbreviations

| | |
|---|---|
| **API** | Application programming interface |
| **ANSI** | American National Standards Institute |
| **DBMS** | Database Management Systems |
| **ER** | Entity Relationship |
| **FOAF** | Friend Of a Friend |
| **HTTP** | Hyper Text Transfer Protocol |
| **IRI** | Internationalized Resource Identifiers |
| **JMS** | Java Message Service |
| **JRDF** | Java RDF |
| **LAN** | Local Area Network |
| **LGPL** | GNU Lesser General Public Licence |
| **MEP** | Message Exchange Pattern |
| **MIME** | Multipurpose Internet Mail Extensions |
| **N3** | Notation 3 |
| **N3QL** | N3 Query Language |
| **OASIS** | Organization for the Advancement of Structured Information Standards |
| **ORDI** | Ontology Representation and Data Integration |
| **OWL** | Web Ontology Language |
| **OWLIM** | OWL In Memory |
| **RDBMS** | Relational DBMS |
| **RDF** | Resource Description Framework |
| **RDFS** | RDF Schema |
| **RDQL** | RDF Data Query Language |
| **ROI** | RDF Input/Output |
| **RPC** | Remote Procedure Call |
| **SAIL** | Storage And Inference Layer |
| **SAWSDL** | Semantic Annotations for Web Service Description Language |
| **SEE** | Semantic Execution Environment |
| **SOFA** | Simple Ontology Framework API |
| **SPARQL** | SPARQL Protocol And RDF Query Language |
| **TS** | Triple Space |
| **UDDI** | Universal Description Discovery and Integration |
| **URI** | Uniform Resource Identifier |
| **URL** | Uniform Resource Locator |
| **W3C** | World Wide Web Consortium |
| **WSDL** | Web Service Description Language |
| **WSMO** | Web Service Modeling Ontology |
| **WSML** | Web Service Modeling Language |
| **WSMX** | Web Service Modeling Execution Environment |
| **XML** | Extensible Markup Language |
| **YARS** | Yet Another RDF Store |
| **YARSQL** | YARS Query Language |

# 1 Introduction

One of the major goals in this work package is to align Semantic Web Services with Triple Space Computing, i.e. to allow Triple Space-based service execution as well as communication and coordination for Semantic Web Services which requires interfaces to allow Semantic Web Services to communicate over a Triple Space. The goal of this deliverable is to enable both Semantic Web Services as well as traditional non-semantic Web services to use Triple Space as underlying communication layer through the definiton of a Triple Space grounding for (Semantic) Web services.

The term grounding means to transfer something to a reference point. In this case, Triple Space Computing is a reference point in terms of communication and coordination of Semantic Web Services. Therefore, we need to provide a grounding support to Semantic Web Services to enable them communicate and coordinate using Triple Space Computing. This linking of Semantic Web Services with Triple Space as being a communication layer is referred as grounding.

This deliverable discusses the grounding aspects for Web Services and Semantic Web Services in detail. As a first step, groundings required for Web Services in general have been proposed. This includes grounding required for Web Services Registry, service description and communication protocol. Secondly, grounding for semantic descriptions of Web Services (i.e. Web Service Modeling Language - WSML) have been investigated. Triple Space groundings for Web Service Execution Environment (WSMX) have also been proposed in order to enable WSMX manage its communication and coordination as well as storage issues over Triple Space. Finally, the proposed Triple Space bindings have been realized and evaluated based on an eHealth use-case.

The remainder of this chapter introduces the term grounding and its relevance for Semantic Web Services and Triple Space Computing. It further defines scope and objectives of the deliverable. Target or expected audience and the motivation for the Triple Space grounding have been identified. In the end, related work on grounding for Semantic Web Services for other protocols have been described and analyzed.

## 1.1 Grounding in general

Semantic Web Services model real world Web Services and therefore Grounding is an important aspect in order to relate both together and to enable the Semantic Web Services communicate with existing Web Services. Semantic Web Services provide the semantic level, i.e. WSMO based service and ontology model. Web Services provide the communication protocols to actually invoke real world Web Services.

In [14], authors distinguish between the two modeling levels of the service description, namely semantic level and invocation level.

**Semantic Level** represents the semantic model for services used in various stages of the execution process run on middleware. For this purpose we use the WSMO service and ontology model. WSMO defines service semantics including non-functional properties, functional properties (capability description) and interfaces (behavioral definition) as well as ontologies that define the information models on which the services operate.

**Invocation Level** describes the physical environment used for service invocation. In our architecture, we use the Web Services Description Language WSDL.

In order to be able to invoke the Web Services, a grounding must be defined from the semantic descriptions to the underlying WSDL and XML Schema definitions. Triple Space bindings have to be specified for WSDL so that Web Services can be invoked over Triple Space. Grounding of SOAP to Triple Space is also required to be provided in order to make the invocation requests compliant. Definitions of such grounding have also to be provided in the WSMO descriptions at the WSMO service interface level in order to store and exchange the semantic descriptions of Web Services over Triple Space.

## 1.2 Scope and Objectives

The scope of this deliverable is to address all possible grounding issues that may arises in enabling the Semantic Web Service use Triple Space Computing for communication and coordination. It ranges from standard Web Service protocols to that of Semantic Web Services.

For standard Web Services, the deliverable will address the grounding issues from the three major fundamental elements of Service Oriented Architecture, i.e. service registry, service description and service invocation/communication protocol.

The deliverable will explore the grounding required for the web service registry data model to enable clients using Triple Space for storage of registry data.

Mappings between the Web Service Description Language (WSDL) and RDF will be explored in order to represent the WSDL description of Web Services as RDF triples in Triple Space. The W3C Recommendation for WSDL Version 2.0 has already been provided with the support of RDF mapping. This deliverable will focus on evaluating the mapping for the purpose of Triple Space and to provide any possible extensions to fulfill the Triple Space requirements.

Grounding from the aspect of Web Service communication will also be explored. The deliverable will focus on replacing or providing an additional Triple Space communication protocol bindings in the Web Service Description Language (WSDL) description. Different possibilities will be explored to make the Web Service use Triple Space communication protocol for invocation, i.e. (1) if the Triple Space transport protocol will provide a support to current SOAP as underline communication infrastructure, or (2) Triple Space transport protocol will be built using SOAP, or (3) both the SOAP and the Triple Space transport protocols will be kept as alternatives. Furthermore, mapping SOAP data model with RDF will also be explored.

After exploring the grounding issues for Web Services communication over Triple Space from above mentioned aspects, the Semantic Web Service execution environment will be explored for the same purpose. The Triple Space grounding required in the Web Service Execution Environment (WSMX) which is one of the implementations of Semantic Execution Environment (SEE) specifications. There are different components in WSMX that will be investigated for any possible requirements of grounding, i.e. end-point Web Service invoker, client end-point interface, Resource Manager (for persistent storage of semantic data) and Execution Manager for coordination of all the components together during the system execution.

Finally, a detailed description of a realization of an eHealth use-case scenario is presented that employs the techniques and extensions to existing (Semantic) Web service technologies proposed in the deliverable to demonstrate their applicability.

## 1.3 Target Audience

This document describes the main building blocks for the integration of Triple Space with Web services and Semantic Web services. The target audience of this document includes researchers as well as practitioners that work in the areas of Web services, Semantic Web services and Web service registries such as UDDI and ebXML. Systems analysts and systems architects needing a thorough knowledge of Triple Space grounding to Web service technologies may also benefit from this document. Although no specific pre-knowledge is required to follow this document, basic knowledge in Web services, UDDI, ontologies, and RDF/S may allow better following the document and for gaining more benefits from it. The work should be of interest to anyone involved with Semantic Web Services and more generally also in Service Oriented Architecture.

This document contains examples of Web service descriptions and URIs used in interacting with Triple Space. To illustrate them as completely as possible, the examples include the names of individuals, products, and companies. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

## 1.4 Motivation

The goal of this deliverable is to enhance state of the art (Semantic) Web service technology in the field of *Web service registries*, *Web service descriptions*, and *Web service interaction* with Triple Space technology.

By grounding UDDI Web service registry information model to Triple Space, we provide the basis for the development of a Web service registry based on Triple Space, that is externally accessible through a UDDI interface and can thus be used as a drop-in replacement for existing Web service registries, while offering the same quality of services (e.g. scalability) as Triple Space itself. A further, more detailed description of various aspects involved in building a UDDI registry based on Triple Space is presented in deliverable document D4.3.

Following the proposed mappings between the *Web Service Description Language (WSDL)* and RDF, Web service descriptions can be stored in Triple Space and retrieved using the Triple Space query language.

Furthermore, using Triple Space as a platform for Web service interactions (i.e. message exchanges between Web service clients and services) exhibits a number of benefits when compared to existing Web service transports such as e.g. HTTP. WS communication based on Triple Space is inherently asynchronous in nature (in contrast to the currently widely used HTTP transport) and will exhibit the same quality of services as Triple Space (e.g. scalability, reliability). Furthermore, the space-based communication paradigm promises to be a suitable platform for realization of message exchange patterns that go beyond simple request-response interactions. In addition, Triple Space enables – through its mechanism of associative addressing which is common to space-based middleware infrastructures – communication between partners based on description of message payload, rather than message meta-information.

## 1.5 Related Work

The term grounding is not new in the area of Semantic Web Services. There already exist on-going efforts for providing grounding support to Semantic Web Services in order to connect the higher level semantic descriptions to the real-world or working standards. The sections below briefly discuss related activities.

### 1.5.1 WSMO Grounding

Web services are, in general, systems that provide certain functionality to their clients and communicate with them by exchanging XML data over computer networks. In order to interoperate successfully, the client and the service must agree on what kinds of messages can be exchanged and when, and what exactly the message exchanges will mean. In other words, the client and the service must agree on a common interface.

WSMO grounding [14] has been defined in two independent areas of relationship between WSMO and the syntactical descriptions of a Web service: data in WSMO ontologies has to be mapped to XML data, usually described using XML Schema, and the functional and behavioral service descriptions in WSMO have to be related to the description construct present in WSDL (using built-in WSMO grounding properties or using the SAWSDL specification). Using the grounding, WSMO Web services can interoperate with currently deployed SOAP based Web services and client frameworks.

WSMO service descriptions can be grounded to WSDL on the basis that WSDL provides the current industry standard for defining how messages can be exchanged between services over the Internet. There are two aspects to it:

**Data Grounding** While the data model of the input and output messages for WSDL services is defined using one or more XML Schemas, the data model for a WSMO service is defined using the conceptual model provided by one or more WSMO ontologies. This leads to the requirement to map between the ontological data in the state machine and its representation as XML messages (lowering from ontology instances to XML, lifting from XML to ontology instances).

**Choreography Grounding** WSMO choreography specifies that the client can access some choreography state data, but in WSDL the client and the service send the data to each other in the form of messages. The grounding must describe what messages are supposed to be sent by the client, and when the client should expect messages from the service. Above that, the grounding must also provide the necessary serialization and networking details, i.e. what underlying protocol (e.g. SOAP, HTTP) should be used for passing the messages, how the XML data is encapsulated in the underlying protocol, and where exactly the data should be sent. Such grounding can be specified with links from the WSMO description.

### 1.5.2 OWL-S Grounding

OWL-S services are characterized by a *Service Profile* that describes the functions provided by a service to allow a service requester to decide whether the service fulfills its need. Furthermore, an OWL-S service description includes a *Service Model* that describes how a service requester has to interact with the service to achieve a

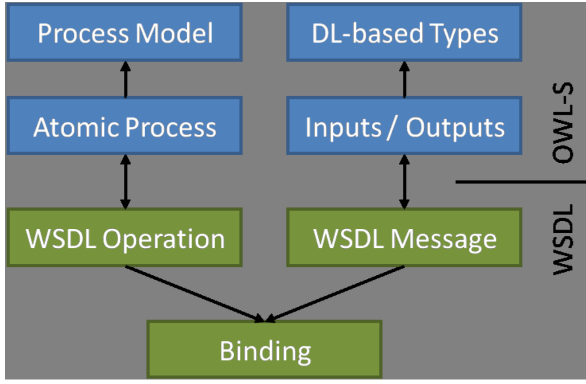


Figure 1.1: OWL-S grounding: Mapping abstract OWL-S information to concrete service access information.

particular outcome through a description of the request content semantics and its result. Both Service Profile and Service Model are part of an OWL-S service's abstract description. How the abstract description of a service is mapped to concrete service implementations is defined by a *Service Grounding*.

A service grounding in OWL-S refers to the process of "mapping from an abstract to a concrete specification of those service description elements that are required for interacting with the service" [17]. This mapping comprises interaction protocols, message format and serialization, the (network) transport used to convey messages to and from the service provider and information about service addressing. For describing a service's abstract interface, OWL-S uses its own process model and typing system. To ground this information to a concrete service implementation, this information is complemented with service binding information which is part of the WSDL (Web Service Description Language) descriptions of the service; for the purpose of integration with existing Web service standards such as WSDL and SOAP, OWL-S does not define a mechanism for describing concrete service access information itself. A high-level view of grounding in OWL-S is depicted in Figure 1.1.

Grounding OWL-S services to WSDL comprises three aspects: (i) grounding service's supported processes (i.e. its exposed operations), (ii) its input and output messages and (ii) the data types used in messages exchange during interaction with the service.

**Processes** In OWL-S a service's interface is represented as a *(composite) process*. Its compound elements (ordered through control constructs such as sequence, split, choice, ...) are *atomic processes* which are "a description of a service that expects one (possibly complex) message and returns one (possibly complex) message in response" [17]. When grounding an OWL-S service to WSDL, its atomic processes are grounded to WSDL (1.1) operations. Atomic processes that consist of both input and output messages are mapped to *request-response* operations; processes with only input messages to *one-way* operations; processes with only output messages to *notification* operations; and processes with input and output messages in reversed order are mapped to *solicit-response* operations.

**Input/output messages** Similar to OWL-S' concept of input received and output sent by a service, WSDL defines the concept of input and output messages of an operation supported by a Web service described in WSDL. Consequently, service

input and output is mapped to input and output messages of WSDL operations respectively.

**Data types** The corresponding concept to data types used in an OWL-S service's input and output, which are represented by OWL classes, are WSDL types. WSDL 1.1 allows for arbitrary type systems [5], however its default type system is XML Schema[1] (XSD). In case XSD types are used by the service implementation, OWL-S e.g. facilitates generating a service's input message by mapping XML serialized representations of OWL concepts to the XML schema data types supported by the particular service implementation to through XSLT (XSL Transformations).

To reflect the OWL-S WSDL grounding as described above in both the OWL-S and WSDL descriptions of a service provider, OWL-S defines (i) a number of WSDL extensions (e.g. the *owl-s-process* attribute of a WSDL operation elements that indicates the atomic process the operation corresponds to) to reference OWL-S information from within WSDL service descriptions and (ii) the OWL-S Grounding Class – and in particular its WSDLGrounding sub-class – to allow referencing WSDL elements from OWL-S descriptions (e.g. the URI of the WSDL document a grounding references to through the *wsdlDocument* property or the *wsdlOperation* property to indicate the WSDL operation for a particular atomic process).

[1] http://www.w3.org/XML/Schema

# 2 Triple Space Grounding for WS Registries

In traditional approaches Web services are published in service registries such as UDDI and ebXML. Service registries are part of the complete lifecycle of a service as they serve as the system of record of the service descriptions thereby making them searchable and encouraging reuse of existing common services.

In order to allow traditional Web service infrastructure to use Triple Space, it is deemed necessary to define mappings or bridges between Triple Space and traditional registries. Having such mappings at hand, existing systems can use Triple Space without hindering their internal implementations. In the existing systems, UDDI and ebXML are the commonly adopted industrial standards of service registries.

The UDDI data model [6] is a hierarchically-structured data model. It provides a top down approach, where a Web service description is divided into several categories with each category offering more detailed information about the registered Web services [20]. Each entity in the data model is identified by a unique universal identifier (UUID). The data model consists of structures that encapsulate information about Web services. Web service descriptions are not part of the UDDI specification, however, can be referenced by UDDI registry entries using *tModels*.

The ebXML data model provides a class hierarchy that allows the modeling of registry entries. The ebXML registry data entries provide metadata about registry objects. These entries are not limited to Web service descriptions. The data model itself is organized into 17 classes that enable, for example, the creation of taxonomies of registry entries or the grouping of related registry objects [20]. The main difference between these two data models lies in the way registered objects are categorized. ebXML offers a built-in extensible category schema, while UDDI relies on tModel links to an external classification schema.

To access these registries, either a custom-made API or a Java API for XML registries (JAXR) can be used. Grounding between Triple Space and commonly used registry standards is the focus of this section. Therefore, the common entities from UDDI registry that also capture corresponding entities from ebXML registry are grounded. The reason for taking this approach is to provide a generic grounding mechanism and to increase the scope of adoption.

## 2.1 Grounding Registry Data Model to RDF

Instead of focusing on one specific technology and *infomodel* (e.g., either UDDI or ebXML) it appears to be more appealing to focus on a general registry, i.e., to the commonly used parts of the registry infomodel. The term registry specifically refers to a technical component that conforms to a certain registry standard such as UDDI and ebXML. In the context of this deliverable, a registry refers to a collection of metadata describing mostly Web services. This document focuses on grounding commonly used parts of the registry data model which is service type definitions. A service type definition is modeled as a tModel in UDDI whereas ebXML models it as either a ClassificationNode or Concept. In the context of this document service type definitions are the main interest.

Though UDDI registry infomodels (especially tModels) are discussed and illustrated in this document, the proposed technique is equally applicable to other registry infomodels in order to design the corresponding solutions.

tModels are often referred as service type definitions and represent unique concepts or constructs. They are technical "finger prints" for a given service which may also function as namespaces to identify other entities, including other tModels. tModels are used to represent technical specifications such as service types, bindings and wire protocols. They are also used to implement category systems that are used to categorize technical specifications and services. tModels are keyed entities in UDDI and define attributes that can be used to specify information about the services. The XML snippet in Listing 2.1 represents an excerpt of the structure of a tModel; the complete tModel representation is available in the Annex, Listing 6.1.

```
1  <rdf:RDF
2      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3      xmlns:t="http://www.w3.org/2001/XMLSchema#">
4      <rdf:Description rdf:about="http://www.tripcom.org/tmodel.xsd#uddi">
5      <t:element rdf:parseType="Resource">
6        <t:name>tModel</t:name>
7        <t:type>uddi:tModel</t:type>
8      </t:element>
9      <t:complexType rdf:parseType="Resource">
10       <t:sequence rdf:parseType="Resource">
11         <t:element rdf:parseType="Resource">
12           <t:ref>uddi:description</t:ref>
13           <t:minOccurs>0</t:minOccurs>
14           <t:maxOccurs>unbounded</t:maxOccurs>
15         </t:element>
16         <t:element rdf:parseType="Resource">
17           <t:minOccurs>0</t:minOccurs>
18           <t:ref>uddi:overviewURL</t:ref>
19         </t:element>
20       </t:sequence>
21       <t:name>overviewDoc</t:name>
22     </t:complexType>
23     <t:targetNamespace>urn:uddi-org:api_v2</t:targetNamespace>
24     <t:element rdf:parseType="Resource">
25       <t:type>uddi:description</t:type>
26       <t:name>description</t:name>
27     </t:element>
28     <t:complexType rdf:parseType="Resource">
29       <t:simpleContent rdf:parseType="Resource">
30         <t:extension rdf:parseType="Resource">
31           <t:attribute rdf:parseType="Resource">
32             <t:ref>xml:lang</t:ref>
33           </t:attribute>
34           <t:base>string</t:base>
35         </t:extension>
36       </t:simpleContent>
37       <t:name>description</t:name>
38     </t:complexType>
39     <t:element rdf:parseType="Resource">
40       <t:type>uddi:overviewDoc</t:type>
41       <t:name>overviewDoc</t:name>
42     </t:element>
43     ...
44   </rdf:Description>
45 </rdf:RDF>
```

Listing 2.1: An XML snippet of tModel structure

The *name* is an identifier of the tModel. The *tModelKey* is a unique key assigned to the tModel. This key is used by other UDDI entities to reference the tModel. The *description* is a plain text description of a tModel. The *overviewDoc* points to an URL where a more detailed text description of the tModel can be found. The *identifierBag* groups keyed reference structures each with a single identifier. The *categoryBag* groups categories that a particular service is falls into. These categories and identifiers are referenced by *keyedReferences*.

A *keyedReference* contains tModelKey referring to the tModel that represents either an identifier or a categorisation system, keyValue contains the actual identifier or categorisation within this system, and a keyName representing the descriptive the name of the identifier or the category.

### 2.1.1 Naming Conventions for Mapping

In this document following naming conventions are used. XML snippets are shown as a way to concisely describe the examples. They are not intended to convey any specific requirements. RDF Schema namespace is identified by the URI-Reference `http://www.w3.org/2000/01/rdf-schema#` and is associated with the prefix “rdfs”. The RDF namespace is identified by the URI-Reference `http://www.w3.org/1999/02/22-rdf-syntax-ns#` and is associated with the prefix ‘rdf’.

### 2.1.2 Vocabulary for Mapping Registry Data Model

The vocabulary for mapping registry data model to RDF is defined based on RDF Schema. These vocabularies are the main constructs for describing and storing the registry data models in Triple Space. RDF Schema provides mechanisms for describing classes and properties of resources. The rdfs:Class can be used for defining each principle entity in tModel and rdf:Property is used for defining each attribute and relation. Thus by using these concepts from RDF Schema for mapping tModel structure makes it possible for storing UDDI description in Triple Space. However, instead of defining new vocabularies, existing ones from RDS Schema are used. This is sufficient for representing the UDDI datamodel in the form of OWL ontologies. Given the schema definition of a UDDI datamodel, an OWL ontology can be generated with the help of existing tools (e.g. Protégé [1]) as shown in Section 2.1.3.

### 2.1.3 Representing Registry Data Model in RDF

In the following mappings, a categoryBag is used to describe relationships between tModels as UDDI does not provide any built-in mechanism to do so. Using the rdfs:Class and rdf:Property concepts, the tModel excerpt shown in 2.1 can be represented in RDF as shown in 2.2; the RDF representation of the complete tModel structure is available in the Annex, Listing 6.2.

### 2.1.4 Implementation Guidelines

In this section we introduce the implementation approach for registry grounding. The main grounding task is performed by the *transformation service* provided by the registry component as shown in Figure 2.1. The transformation service itself is divided into *query transformation* and *data transformation*. The former is concerned with transformation between the queries supported by the UDDI interface and the TS API (i.e. SPARQL) queries whereas the latter is concerned with the transformation between instances of the UDDI infomodel and its RDF representation.

[1] http://protege.stanford.edu/

```
1  <?xml version="1.0"?>
2  <rdf:RDF
3    xmlns:p1="http://www.owl-ontologies.com/Ontology1194353329.owl#"
4    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
6    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
7    xmlns:owl="http://www.w3.org/2002/07/owl#"
8    xml:base="http://www.owl-ontologies.com/Ontology1194353329.owl">
9
10   <owl:Ontology rdf:about=""/>
11   <owl:Class rdf:ID="uddi_overviewDoc"/>
12   <owl:Class rdf:ID="uddi_tModel"/>
13
14   <owl:ObjectProperty rdf:ID="overviewDoc">
15     <rdfs:range rdf:resource="#uddi_overviewDoc"/>
16     <rdfs:domain rdf:resource="#uddi_tModel"/>
17   </owl:ObjectProperty>
18
19   <owl:DatatypeProperty rdf:ID="description">
20     <rdfs:domain>
21       <owl:Class>
22         <owl:unionOf rdf:parseType="Collection">
23           <owl:Class rdf:about="#uddi_tModel"/>
24           <owl:Class rdf:about="#uddi_overviewDoc"/>
25         </owl:unionOf>
26       </owl:Class>
27     </rdfs:domain>
28     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
29   </owl:DatatypeProperty>
30 </rdf:RDF>
```

Listing 2.2: XML snippet of UDDI structure

The implementation approach taken should follow the component based implementation approach and separate the concerns of external developers. To assist in implementation of the aforementioned services, the following guidelines are provided.

### Implementing Data Transformation

- Provide an interface for accessing and invoking this service.
- Use and extend vocabularies (when needed) provided in Section 2.1.2 for transformation purposes
- Generate reusable rules for transformation if needed or if possible
- Use existing open-source RDF library for handling RDF data model.
- Use existing open-source UDDI library for handling UDDI datamodel
- Use TS API to use Triple Space

### Implementing Query Transformation

- Provide an interface for accessing and invoking this service.
- Use and extend vocabularies (when needed) provided in Section 2.1.2 for transformation purposes
- Generate reusable rules for transformation if needed or if possible

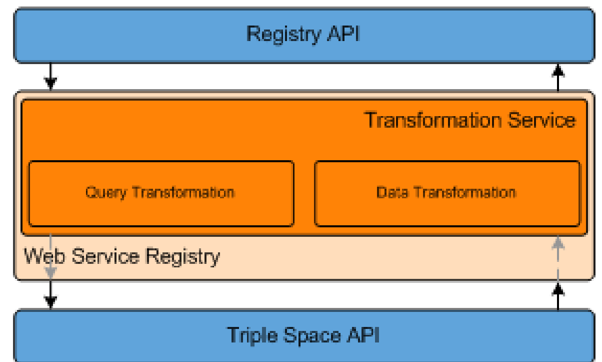


Figure 2.1: Transformation Service for Registry Grounding.

- Provide support for SPARQL query
- Use existing open-source RDF library for handling RDF queries.
- Use existing open-source UDDI library for handling UDDI queries
- Use TS API to use Triple Space

## 2.2 Representing WSDL in RDF

Web services are generally described in the Web Services Description Language (WSDL) which is an XML language that provides components for modeling abstract Web service interfaces, network bindings and concrete endpoints. While WSDL is written in XML, Triple Space stores data as RDF, W3C's official format for the Semantic Web data.

In TripCom Work Package 4, Web services are integrated with Triple Space. The following two scenarios require that WSDL descriptions should be stored in a triplespace:

- When new data in the triplespace triggers the need to invoke a Web service, the Web service bridges (described in D4.1[19]) create the request message with the appropriate input data, make the request to the Web service, and put the data from the response message back in the triple space.
- The triplespace can serve as a Web service description repository, so that Web service deployments can use the various space querying methods to discover Web services suitable for a given goal.

For the purpose of storing WSDL descriptions in triplespaces, and for integrating them with other data, the Web service descriptions encoded in WSDL need to be converted to RDF. The W3C Recommendation for WSDL Version 2.0 [3] is accompanied with an RDF mapping specification[2]. In this section, we briefly describe how the W3C WSDL RDF mapping works and how we use it in Triple Space.

[2] `http://www.w3.org/TR/wsdl20-rdf`

While WSDL 2.0 is the W3C standard, WSDL 1.1 is still the widely preferred version, and it does not provide an RDF mapping. However, since most WSDL descriptions conform to WS-I Basic Profile[3], they can be automatically transformed[4] into WSDL 2.0, and the result can be mapped to RDF.

Using an RDF form of WSDL may greatly simplify the discovery of Web service descriptions. Without RDF, a WSDL registry would generally expose a specific API tailored for the structure of WSDL components. With RDF, standardized RDF query languages, such as SPARQL, are fully sufficient for finding WSDL descriptions based on any number of criteria, such as endpoint address, implemented interfaces, use of a given WSDL binding, use of a predefined XML Schema element, presence of a given operation etc.

In order to integrate Web services with Triple Space, it is necessary that the RDF form of WSDL fulfills several requirements, which are listed below.

1. For discovery, the WSDL RDF form must allow finding WSDL services, interfaces and operations by their namespace-qualified names.

2. Also for discovery, the WSDL RDF form must allow finding WSDL services through their endpoint addresses.

3. The WSDL RDF form must be extensible to the same extent that WSDL 2.0 is extensible; for instance to allow annotations such as SAWSDL [21].

4. For grounding configuration, the WSDL RDF form must preserve SAWSDL `liftingSchemaMapping` and `loweringSchemaMapping` data grounding annotations on operation inputs and outputs.

5. For use by existing WSDL-aware tooling, the WSDL RDF form must either point to the original XML WSDL document, or it must enable reconstruction of a WSDL document fully semantically equivalent to the original one.

Notably, we do not require that the RDF form be *mutable*, i.e., we do not expect manipulation of the WSDL data in its RDF form, other than reading and querying.

As is, the W3C WSDL 2.0 RDF mapping document fulfills the first three mentioned requirements, and it has to be extended slightly to fulfill the latter two. The W3C mapping, together with our extensions, is described below in Section 2.2.1, and Section 2.2.2 describes the currently available implementation for the WSDL RDF mapping. Finally, Section 2.2.3 contains an example WSDL document and shows its RDF form.

### 2.2.1 RDF vocabulary for service description mapping

The W3C WSDL 2.0 RDF mapping defines an OWL ontology for representing WSDL data, and it also defines the mappings between the WSDL 2.0 component model and the corresponding RDF form. The mapping is only defined in the direction from a valid WSDL document into an RDF graph, and it is not intended for lossless round-trip transitions between the XML form and RDF.

[3] `http://ws-i.org/Profiles/BasicProfile-1.1.html`

[4] The WSDL 1.1→2.0 transformation tool is at `http://w3.org/2006/02/WSDLConvert`

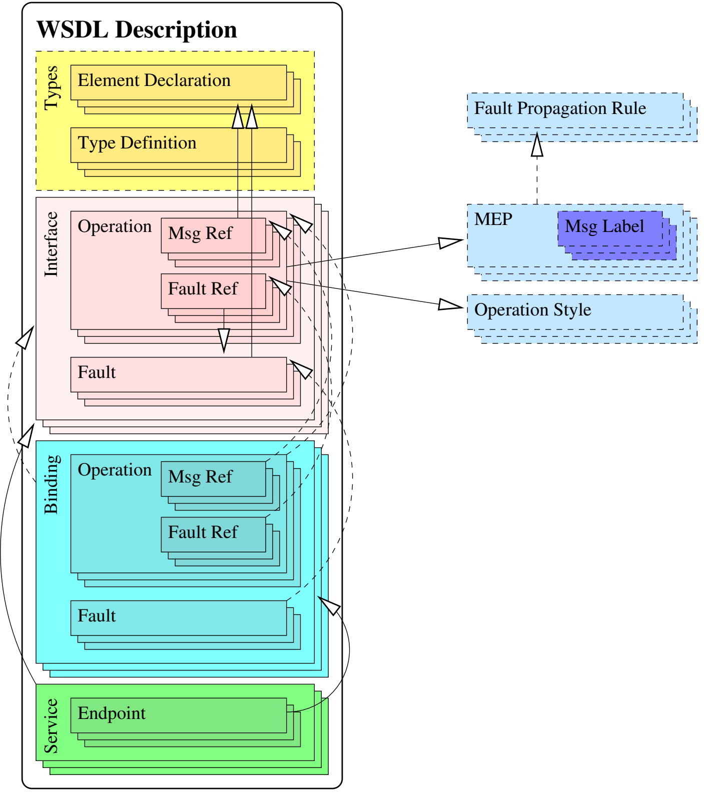


Figure 2.2: Structure of the WSDL 2.0 component model.

The core of the WSDL OWL ontology mirrors the WSDL 2.0 component model which, in turn, follows the tree of WSDL XML elements. A full description of the WSDL 2.0 component model is out of scope of this deliverable, but we will briefly sketch the structure (illustrated in Figure 2.2 with concrete components in solid boxes, and other entities dashed) and then discuss the significant details of the WSDL ontology.

A valid WSDL document is represented by a single top-level *Description* component that acts as a container for the *Interface*, *Binding*, *Service* components defined in this document or imported/included from other documents, plus any *Element declaration* and *Type definition* components coming from XML Schemas. Each of these components is identified by a namespace-qualified name (*QName*), and it is permissible that two components of different type (e.g. an Interface and a Binding) have the same QName.

An Interface component groups together related operations, represented as *Interface Operation* components, each of which follows some message exchange pattern and assigns schema element declarations to each of its messages. An interface can also extend other interfaces, assuming their operations as its own. Binding components follow the structure of the Interface component, specifying how the various messages are to be communicated over the wire.

Finally, the Service component represents a Web service which *implements* a single WSDL interface, and specifies any number of *Endpoint* components that provide a concrete network address and point to a corresponding binding.

In the component model of WSDL, most components are identified with their QNames. The WSDL specification provides a mechanism of identifying the compo-

nents with URI references, which are used in the RDF form. For instance, an operation called "opCheckAvailability" in an interface with the name "reservationInterface" and namespace `http://greath.example.com/2004/wsdl/resSvc` would be identified in the RDF form with the following URI (without the line break):

```
http://greath.example.com/2004/wsdl/resSvc#
          wsdl.interfaceOperation(reservationInterface/opCheckAvailability)
```

The XML form of WSDL is document-oriented, which means that separate WSDL documents are independent, unless one includes or imports the other. Apart from importing or including, there is no standard way of putting multiple WSDL documents together. In RDF, however, separate graphs can easily be merged. Therefore, while a WSDL document always contains a single Description component (independent of any imports or includes that this document contains), an RDF graph may contain any number of merged WSDL documents, and the corresponding number of Description components. This makes it possible to query all the available WSDL data with a single RDF store query.

Since the W3C WSDL RDF Mapping is not intended for lossless round-trip transformations, and since the RDF data does not correspond 1:1 with a WSDL document (indeed, as said above, the merged RDF data can represent many WSDL documents), in order to satisfy our requirement no. 5 we need to extend the mapping with a pointer to the original WSDL document. We do this by using the property `rdfs:isDefinedBy` on the Description components. This way the RDF form points to the source file of every WSDL Description component available in the triple store.

The WSDL 2.0 RDF mapping only deals with WSDL components (interface, binding etc.) and it does not provide an RDF form for XML Schema components, i.e. element declarations and type definitions. At the moment there is no RDF mapping for XML Schema, therefore the XML Schema components are referenced using their QNames. One of our requirements is that the WSDL RDF form must preserve SAWSDL `liftingSchemaMapping` and `loweringSchemaMapping` annotations on operation inputs and outputs. Neither the WSDL RDF mapping or the SAWSDL RDF mapping provides such functionality, therefore we need to provide the following Triple Space-specific extension, illustrated in Figure 2.3.

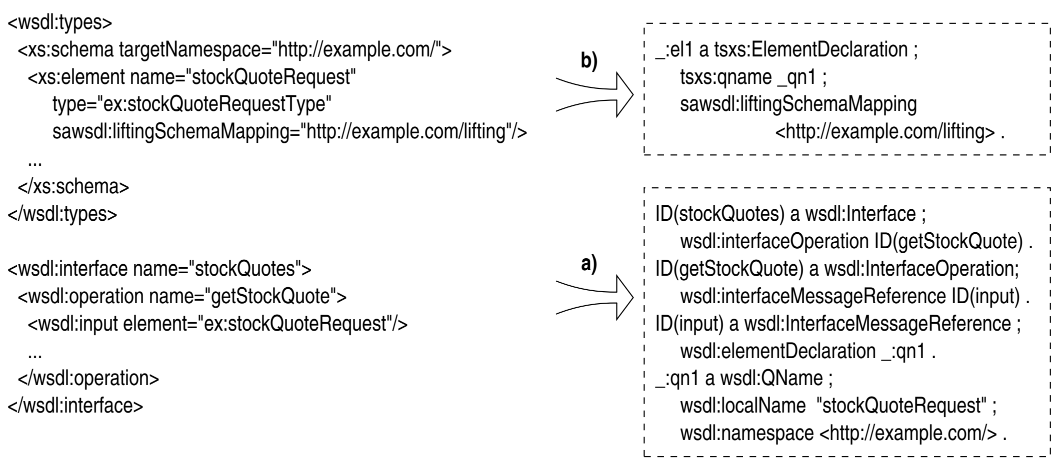


Figure 2.3: Extending WSDL RDF mapping to preserve grounding annotations.

In the figure, we can see on the left side an excerpt of a WSDL description of a stock quote Web service. The first part has an XML Schema that defines the

element "stockQuoteRequest", and the second part has a WSDL interface with a single operation, "getStockQuote", which uses that element for its input message. The lower block on the right side of the figure, marked as a), shows the result of the stock W3C WSDL RDF mapping — it represents the interface and operation information, and it refers to the element using its QName. Our extension[5] is shown in the upper block, marked as b). It represents the element declaration using a bnode (written `_:el1`) that has the same QName as the one referenced in the WSDL part a), and it carries the SAWSDL data grounding annotation.

To summarize, we extend the W3C WSDL 2.0 RDF mapping in two places: every Description component gets an `rdfs:isDefinedBy` pointer to the original WSDL document, and every XML Schema Element Declaration component used as an input or output message of a WSDL operation gets a `tsxs:ElementDeclaration` instance that represents the component and carries its SAWSDL annotations.

### 2.2.2 WSDL RDF mapping implementation

We have an implementation for the WSDL 2.0 and SAWSDL RDF mapping (as specified, at this time without our extensions as described above). The implementation sources are available in the Tripple Space implementation SVN repository in the `ws` component directory as a `wsdl2rdf` subcomponents. The implementation is built on top of Woden, an API for accessing WSDL 2.0 documents, and its extension Woden4SAWSDL[6] that supports SAWSDL annotations.

The main class is `org.tripcom.ws.wsdl2rdf.Wsdl2Rdf` whose `main()` method takes a single parameter, the location of the source (SA)WSDL document, and prints out the RDF form in N-Triples.

This implementation does not support the extensions described in the previous subsection. A version with that support will be provided within the Triple Space prototype.

### 2.2.3 WSDL RDF mapping example

Listing 2.3 shows an excerpt of a sample WSDL document (a complete version of the WSDL document is available in the Annex, Listing 6.3), taken from the WSDL 2.0 Primer[7] and enhanced with SAWSDL semantic annotation on lines 16 and 21.

Listing 2.4 shows an excerpt of the the RDF form of this WSDL document (a complete version of the WSDL document is available in the Annex, Listing 6.4); line 14 contains the first Triple Space extension, pointing to the source of this document, and lines 43-46 contain the second extension, representing an XML Schema element with its SAWSDL annotation.

[5] The namespace prefix `tsxs` is meant to signify the *Triple Space XML Schema* RDF mapping.
[6] Woden4SAWSDL is available at `http://lsdis.cs.uga.edu/projects/meteor-s/opensource/woden4sawsdl`
[7] `http://www.w3.org/TR/wsdl20-primer#example-initial`

```
1  <description xmlns="http://www.w3.org/ns/wsdl"
2      targetNamespace="http://greath.example.com/2004/wsdl/resSvc"
3      xmlns:tns="http://greath.example.com/2004/wsdl/resSvc"
4      xmlns:ghns="http://greath.example.com/2004/schemas/resSvc"
5      xmlns:wsoap="http://www.w3.org/ns/wsdl/soap"
6      xmlns:sawsdl="http://www.w3.org/ns/sawsdl"
7      xmlns:wsdlx="http://www.w3.org/ns/wsdl-extensions">
8
9    ...
10
11   <types>
12     <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
13         targetNamespace="http://greath.example.com/2004/schemas/resSvc"
14         xmlns="http://greath.example.com/2004/schemas/resSvc">
15       <xs:element name="checkAvailability" type="tCheckAvailability"
16             sawsdl:loweringSchemaMapping="http://greath.example.com/lowering.xslt" />
17     ...
18     </xs:schema>
19   </types>
20   <interface name="reservationInterface"
21       sawsdl:modelReference="http://greath.example.com/ontology#reservation" >
22     <operation name="opCheckAvailability" wsdlx:safe="true"
23       pattern="http://www.w3.org/ns/wsdl/in-out"
24     style="http://www.w3.org/ns/wsdl/style/iri">
25         <input messageLabel="In" element="ghns:checkAvailability" />
26         <output messageLabel="Out" element="ghns:checkAvailabilityResponse" />
27     </operation>
28   </interface>
29   <binding name="reservationSOAPBinding" interface="tns:reservationInterface"
30     type="http://www.w3.org/ns/wsdl/soap"
31     wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/">
32      <operation ref="tns:opCheckAvailability"
33       wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response"/>
34   </binding>
35   <service name="reservationService" interface="tns:reservationInterface">
36      <endpoint name="reservationEndpoint" binding="tns:reservationSOAPBinding"
37     address="http://greath.example.com/2004/reservation"/>
38   </service>
39 </description>
```

Listing 2.3: WSDL 2.0 primer example, with added SAWSDL annotations

```
1  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
2  @prefix sawsdl: <http://www.w3.org/ns/sawsdl#> .
3  @prefix whttp: <http://www.w3.org/ns/wsdl/http#> .
4  @prefix wsdl: <http://www.w3.org/ns/wsdl-rdf#> .
5  @prefix wsdlx: <http://www.w3.org/ns/wsdl-extensions#> .
6  @prefix wsoap: <http://www.w3.org/ns/wsdl/soap#> .
7  @prefix tsxs: <http://tripcom.org/ns/tsxs#> .
8
9  <http://greath.example.com/2004/wsdl/resSvc#wsdl.description()>
10   a wsdl:Description ;
11   wsdl:interface <http://greath.example.com/2004/wsdl/resSvc#wsdl.interface(reservationInterface)> ;
12   wsdl:binding <http://greath.example.com/2004/wsdl/resSvc#wsdl.binding(reservationSOAPBinding)> ;
13   wsdl:service <http://greath.example.com/2004/wsdl/resSvc#wsdl.service(reservationService)> ;
14   rdfs:isDefinedBy <http://greath.example.com/2004/wsdl/resSvc.wsdl> .
15
16 <http://greath.example.com/2004/wsdl/resSvc#wsdl.interface(reservationInterface)>
17   a wsdl:Interface ;
18   rdfs:label "reservationInterface" ;
19   wsdl:interfaceOperation <http://greath.example.com/2004/wsdl/resSvc#wsdl.interfaceOperation(
         reservationInterface/opCheckAvailability)> ;
20   sawsdl:modelReference <http://greath.example.com/ontology#reservation> .
21
22 <http://greath.example.com/2004/wsdl/resSvc#wsdl.interfaceOperation(reservationInterface/opCheckAvailability)
       >
23   a wsdl:InterfaceOperation ;
24   rdfs:label "opCheckAvailability" ;
25   sawsdl:modelReference wsdlx:SafeInteraction ;
26   wsdl:interfaceMessageReference
27     <http://greath.example.com/2004/wsdl/resSvc#wsdl.interfaceMessageReference(reservationInterface/
           opCheckAvailability/In)> ,
```

```
28      <http://greath.example.com/2004/wsdl/resSvc#wsdl.interfaceMessageReference(reservationInterface/
            opCheckAvailability/Out)> ;
29    wsdl:messageExchangePattern <http://www.w3.org/ns/wsdl/in-out> ;
30    wsdl:operationStyle <http://www.w3.org/ns/wsdl/style/iri> .
31
32  <http://greath.example.com/2004/wsdl/resSvc#wsdl.interfaceMessageReference(reservationInterface/
        opCheckAvailability/In)>
33    a wsdl:InterfaceMessageReference , wsdl:InputMessage ;
34    wsdl:elementDeclaration <urn:uuid:ead76f4f-f5bc-4ca7-af8c-460a9c99fa53> ;
35    wsdl:messageContentModel wsdl:ElementContent ;
36    wsdl:messageLabel <http://www.w3.org/ns/wsdl/in-out#In> .
37
38  <urn:uuid:ead76f4f-f5bc-4ca7-af8c-460a9c99fa53>
39    a wsdl:QName ;
40    wsdl:localName "checkAvailability" ;
41    wsdl:namespace <http://greath.example.com/2004/schemas/resSvc> .
42
43  <urn:uuid:c3220a0a-3a1b-4f51-8635-cfe6273024b4>
44    a tsxs:ElementDeclaration ;
45    tsxs:qname <urn:uuid:ead76f4f-f5bc-4ca7-af8c-460a9c99fa53> ;
46    sawsdl:loweringSchemaMapping <http://greath.example.com/lowering.xslt> .
47
48  <http://greath.example.com/2004/wsdl/resSvc#wsdl.binding(reservationSOAPBinding)>
49    a wsdl:Binding , <http://www.w3.org/ns/wsdl/soap> ;
50    rdfs:label "reservationSOAPBinding" ;
51    wsdl:binds <http://greath.example.com/2004/wsdl/resSvc#wsdl.interface(reservationInterface)> ;
52    wsdl:bindingOperation <http://greath.example.com/2004/wsdl/resSvc#wsdl.bindingOperation(
          reservationSOAPBinding/opCheckAvailability)> ;
53    wsdl:bindingFault <http://greath.example.com/2004/wsdl/resSvc#wsdl.bindingFault(reservationSOAPBinding/
          invalidDataFault)> ;
54    whttp:defaultQueryParameterSeparator "&" ;
55    wsoap:protocol <http://www.w3.org/2003/05/soap/bindings/HTTP/> ;
56    wsoap:version "1.2" .
57
58  <http://greath.example.com/2004/wsdl/resSvc#wsdl.bindingOperation(reservationSOAPBinding/
        opCheckAvailability)>
59    a wsdl:BindingOperation ;
60    wsdl:binds <http://greath.example.com/2004/wsdl/resSvc#wsdl.interfaceOperation(reservationInterface/
          opCheckAvailability)> ;
61    wsoap:soapMEP <http://www.w3.org/2003/05/soap/mep/soap-response> .
62
63  <http://greath.example.com/2004/wsdl/resSvc#wsdl.service(reservationService)>
64    a wsdl:Service ;
65    rdfs:label "reservationService" ;
66    wsdl:endpoint <http://greath.example.com/2004/wsdl/resSvc#wsdl.endpoint(reservationService/
          reservationEndpoint)> ;
67    wsdl:implements <http://greath.example.com/2004/wsdl/resSvc#wsdl.interface(reservationInterface)> .
68
69  <http://greath.example.com/2004/wsdl/resSvc#wsdl.endpoint(reservationService/reservationEndpoint)>
70    a wsdl:Endpoint ;
71    rdfs:label "reservationEndpoint" ;
72    wsdl:address <http://greath.example.com/2004/reservation> ;
73    wsdl:usesBinding <http://greath.example.com/2004/wsdl/resSvc#wsdl.binding(reservationSOAPBinding)> .
```

Listing 2.4: WSDL 2.0 primer example in RDF form

# 3 Triple Space Grounding for Web Service Communication

In this section a description of a Web Service binding for Triple Space is presented. This binding comprises three different aspects that are necessary to enable Web Service requester and provider to interact (i.e. exchange messages) over Triple Space.

In Section 3.1 a WSDL binding for Triple Space is proposed which describes how the information that is necessary for service requesters to access a Web Service over Triple Space can be encoded in the WSDL description of that service. As a basis for this work, a brief overview of existing WSDL bindings is presented first. Subsequently, the proposed WSDL binding for Triple Space is described in detail and corresponding examples are presented. In Section 3.2 an extensive description of how SOAP envelopes can be transformed to RDF graphs in such a way that they (i) can be transmitted between Triple Space enabled SOAP processing nodes and (ii) they provide all information necessary for message delivery and retrieval (such as endpoint addressing or message correlation) in a format accessible to Triple Space clients is given. In Section 3.3, following the description of the RDF graph representation of SOAP envelopes, a description of how the existing Web service message exchange patterns defined as part of the WSDL 2.0 specification can be realized using TS API operations is provided. In addition to the standard message exchange patterns, three custom extended message exchange patterns are introduced that make use of Triple Space's special communication characteristics such as publishing data once and have it consumed non-destructively by multiple communication partners.

## 3.1 TripCom bindings for WSDL

Web Service Description Language (WSDL) contains communication protocol bindings for the service description. Currently, it is SOAP. Our goal is to define the bindings of Triple Space communication protocol in the WSDL that should enable Triple Space clients to communicate and invoke Web Services. In other words. It will enable the Web Services based on current standards to be invoked with Triple Space based communication protocol.

It concerns with finding a way to have Triple Space bindings in the service description, secondly it requires binding the communication protocol. There are two major steps involved in binding the Triple Space communication protocol with Web Services. First is to provide protocol binding information in the description language of Web Services (i.e. WSDL) which refers to provide TripCom binding in WSDL. Second is to provide communication protocol (i.e. SOAP) bindings itself with the Triple Space communication which refers RDF representation of SOAP.

There are different possibilities that have been figured out based on how existing communication protocol of Web Services can fit-in with new Triple Space based communication protocol, as described below:

- The first possibility is the Triple Space transport protocol will provide a support to current SOAP as underline communication infrastructure. In that case, SOAP envelope will be represented in RDF and will be sent over Triple Space in asynchronous manner.

- The Triple Space transport protocol will be built over SOAP. It will include extension of current SOAP protocol to make it RDF compatible and asynchronous, based on the Triple Space API [18].
- Both the SOAP and Triple Space transport protocol will be kept as alternative communication channels. This approach seems to be most appropriate at the moment as it introduces a new Triple Space based communication mechanism with Web Services without disturbing existing W3C based SOAP standards. However, this approach has certain limitations as it has to keep the current standards valid while trying to have new features.

Due to the nature of the Triple Space communication protocol and in order to enable Web Services with asynchronous communication, the first option presented above have been found reasonable to adapt where the idea is to alter the current SOAP communication protocol and its bindings with WSDL to let the Triple Space communication protocol support the communication of Web Services. With this, Web Services will be able to be invoked using the underline Triple Space middleware.

In the subsections below, a review of current state of WSDL bindings with SOAP has been carried out. WSDL bindings with other related communication protocols like JMS have also been taken into consideration before defining the WSDL bindings for Triple Space.

### 3.1.1 WSDL bindings with SOAP

The current status of WSDL specifications [5] includes a binding for SOAP endpoints. It provides an indication that a binding is bound to the SOAP protocol. It provides a way of specifying an address for a SOAP endpoint. A list of definitions for Headers that are transmitted as part of the SOAP Envelope. WSDL bindings for SOAP defines the message format and protocol details for accessing a Web Service.

The bindings supports the WSDL for having following information to access a service:

- An indication that a binding is bound to the SOAP protocol
- A way of specifying an address for a SOAP endpoint
- The URI for the SOAPAction HTTP header for the HTTP binding of SOAP
- A list of definitions for headers that are transmitted as part of the SOAP Envelope

An example of request-response operation is given below:

```
1  <wsdl:binding name="mathSoapBinding" type="impl:math">
2    <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
3
4    <wsdl:operation name="add">
5    <wsdlsoap:operation soapAction="" />
6    <wsdl:input name="addRequest">
7    <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="http://
         DefaultNamespace" use="encoded" />
8    </wsdl:input>
9
10   <wsdl:output name="addResponse">
```

```
11    <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="http://localhost
          :8080/axis/math.jws" use="encoded" />
12    </wsdl:output>
13    </wsdl:operation>
14
15    </wsdl:operation>
16    </wsdl:binding>
17    <wsdl:service name="mathService">
18
19    <wsdl:port binding="impl:mathSoapBinding" name="math">
20    <wsdlsoap:address location="http://localhost:8080/axis/math.jws" />
21    </wsdl:port>
22    </wsdl:service>
```

Listing 3.1: Sample WSDL code snippet showing SOAP binding

The **binding** element has two attributes - the name attribute and the type attribute.

The name attribute (you can use any name you want) defines the name of the binding, and the type attribute points to the port for the binding, in this case the "glossaryTerms" port.

The **soap:binding** element has two attributes - the style attribute and the transport attribute.

The style attribute can be "rpc" or "document". In this case we use document. The transport attribute defines the SOAP protocol to use. In this case we use HTTP.

The **operation** element defines each operation that the port exposes.

For each operation the corresponding SOAP action has to be defined. You must also specify how the input and output are encoded. In this case we use "literal".

An example is given below where WSDL defines message format and protocol details for accessing the Web Service.

```
1   <wsdl:binding name="mathSoapBinding" type="impl:math">
2     <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
3
4     <wsdl:operation name="add">
5     <wsdlsoap:operation soapAction="" />
6     <wsdl:input name="addRequest">
7     <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="http://
          DefaultNamespace" use="encoded" />
8     </wsdl:input>
9
10    <wsdl:output name="addResponse">
11    <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="http://localhost
          :8080/axis/math.jws" use="encoded" />
12    </wsdl:output>
13    </wsdl:operation>
14
15    </wsdl:operation>
16    </wsdl:binding>
17    <wsdl:service name="mathService">
18
19    <wsdl:port binding="impl:mathSoapBinding" name="math">
20    <wsdlsoap:address location="http://localhost:8080/axis/math.jws" />
21    </wsdl:port>
22    </wsdl:service>
```

Listing 3.2: WSDL bindings with SOAP

Listing 3.2 shows binding part of WSDL code. The binding information in WSDL is about message transport protocol, encoding type and default namespace for the messages to be sent for each of the operation defined in portTypes. It further URL to access the service.

### 3.1.2 WSDL bindings with other protocols

The Web Services community although have convinced that Web Services are a great way of integrating two systems that are built using different technologies. However, the problem that arises is that some applications require very high reliability for individual transactions. SOAP over HTTP is limited in this type of application. The basic problem is that HTTP itself just does not provide guaranteed delivery.

One way is to get over this challenge is by running SOAP over JMS. It is by using JMS as a replacement for HTTP as the underlying transport for SOAP communications. Using Apache Axis, this means sending messages using the Axis APIs, but having the actual communications to the SOAP server be processed using JMS instead of being sent over HTTP. It brings improvement in reliability for mission critical applications.

The way is using HTTP for communications between the SOAP client and server, but having the SOAP messages be persisted in JMS inside the Web Service client before they are sent; this way they are persisted until the HTTP communications return successfully. In a different situation, it could also mean persisting the SOAP Messages using JMS inside the Web Service server application once they are received.

#### Example

In this section, an example has been presented that shows how to use the SOAP over JMS. For using the SOAP over JMS, the transport attribute <soap:binding> tag have to indicate that JMS is used, i.e. as mentioned below:

```
1  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/jms"/>
```

For SOAP over JMS, the <wsdl:port> tag must contain a <jms:address> element. This element provides the information required for a client to connect correctly to the Web Service using the JMS programming model. Typically, it is the stubs generated to support the SOAP over JMS binding that act as the JMS client. Alternatively, the Web Service client can use the JMS programming model directly.

The <jms:address> element takes this form:

```
1  <jms:address
2
3       destinationStyle="queue"
4       jmsVendorURI="http://ibm.com/ns/mqseries"?
5       initialContextFactory="com.ibm.NamingFactory"?
6       jndiProviderURL="iiop://something:900/wherever"?
7       jndiConnectionFactoryName="orange"
8       jndiDestinationName="fred"
9  />
```

Listing 3.3: JMS address

Here is an example of a WSDL that defines a SOAP over JMS binding:

```
1  <?xml version="1.0" encoding="UTF-8"?> <wsdl:definitions
2             name="StockQuoteInterfaceDefinitions"
3           targetNamespace="urn:StockQuoteInterface"
4           xmlns:tns="urn:StockQuoteInterface"
5           xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
6           xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
7           xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
8       <wsdl:message name="GetQuoteInput">
9           <part name="symbol" type="xsd:string"/>
10      </wsdl:message>
11      <wsdl:message name="GetQuoteOutput">
```

```
12            <part name="value" type="xsd:float"/>
13        </wsdl:message>
14
15        <wsdl:portType name="StockQuoteInterface">
16            <wsdl:operation name="GetQuote">
17                <wsdl:input message="tns:GetQuoteInput"/>
18                <wsdl:output message="tns:GetQuoteOutput"/>
19            </wsdl:operation>
20        </wsdl:portType>
21
22        <wsdl:binding name="StockQuoteSoapJMSBinding" type="tns:StockQuoteInterface">
23            <soap:binding style="rpc"
24                      transport="http://schemas.xmlsoap.org/soap/jms"/>
25            <wsdl:operation name="GetQuote">
26                <soap:operation soapAction="urn:StockQuoteInterface#GetQuote"/>
27                <wsdl:input>
28                    <soap:body use="encoded" namespace="urn:StockQuoteService"
29                            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
30                </wsdl:input>
31                <wsdl:output>
32                    <soap:body use="encoded" namespace="urn:StockQuoteService"
33                        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
34                </wsdl:output>
35            </wsdl:operation>
36        </wsdl:binding>
37        <wsdl:service name="StockQuoteService">
38            <wsdl:port name="StockQuoteServicePort"
39                         binding="sqi:StockQuoteSoapJMSBinding">
40                <jms:address destinationStyle="queue"
41                        jndiConnectionFactoryName="myQCF"
42                        jndiDestinationName="myQ"
43                        initialContextFactory="com.ibm.NamingFactory"
44                        jndiProviderURL="iiop://something:900/" />
45            </wsdl:port>
46        </wsdl:service>
47    </wsdl:definitions>
```

Listing 3.4: WSDL SOAP over JMS binding

### 3.1.3 WSDL bindings with Triple Space

For providing binding information with Triple Space, information about accessing the Web Service through Triple Space has to be provided. Web Service client and Web Service act as Triple Space clients while communicating over Triple Space. The communication for invoking a Web Service over Triple Space will be same as one Triple Space client is sending a message to another Triple Space client. Therefore, the Web Service will be a Triple Space client that is subscribed to listen for incoming triples in a logical sub-space. Whereas, the Web Service client will be another Triple Space client that will be publishing data on the Triple Space. The information required for a Triple Space client to send a message to another client consists of identification (i.e. URI) of the target client, and sub-space ID to which the target client is subscribed to.

The binding information also contains the encoding type of the protocol which will be referred to RDF representation of SOAP in 3.2. The figure 3.1 shows interaction scenario over Triple Space for invoking a Web Service.

As a first step, client publishes SOAP invocation message represented in RDF over Triple Space, along with Triple Space based UDI identification of the Web Service. The message is published in the sub-space where the Web Services is subscribed to. Web Service as being a Triple Space client is notified for the message received. The service is invoked in this way and result is published back on the sub-space in the name of client ID (i.e. URI) that invoked the Web Service over Triple Space. The client is notified for the message that contains response from the Web Service.

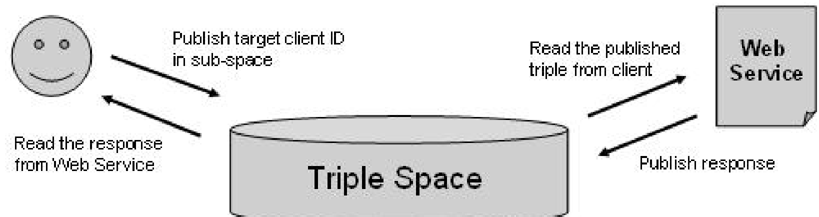


Figure 3.1: Web Service invocation scenario over Triple Space

There are following required updates identified to be done in the WSDL:

- **transport mechanism** provides schema information about the transport protocol that is being used
- **encoding style** mentions the encoding information and namespace information
- **address location** provides information about exact location that is used for directly accessing the Web Service

The transport information has to be changed that should reflect URI for schema information for RDF representation of SOAP that has been proposed in 3.2. Secondly, encoding information for each of the message (in or out or both) has to be provided with the updated information of new RDF encoding of message. Lastly, the address location information has to be updated to provide URI to access the destination client and the ID of sub-space where the message has to be published.

Based on the information above, the proposed updates in the information are to be implemented as follows:

- the transport property has to reflect the underline Triple Space transport protocol
- Triple Space based RDF encoding for the SOAP message has to be mentioned to invoke the Web Service
- Address information should contain the client URI and sub-space ID of the triplespace where the target Web Service is subscribed

Based on the findings, the listing 3.5 presents an example to show the bindings part of WSDL with Triple Space bindings. The transport protocol information points to the Triple Space communication protocol schema, i.e. `http://www.tripcom.org/protocol/tripcomcommunication`. The encoding points to the RDF representation of SOAP, i.e. `http://www.tripcom.org/protocol/tripcomcommunication/encoding`. The address information mentions URIs of target client (i.e. Web Service), i.e. `tripcom://www.example.org/clientabc` and the sub-space to which the Web Service is subscribed to `tripcom://www.example.org/subspace123`.

```
1  <wsdl:binding name="mathSoapBinding" type="impl:math">
2    <wsdlsoap:binding style="rpc" transport="http://www.tripcom.org/protocol/tripcomcommunication" />
3
4    <wsdl:operation name="add">
5    <wsdlsoap:operation soapAction="" />
6    <wsdl:input name="addRequest">
7    <wsdlsoap:body encodingStyle="http://www.tripcom.org/protocol/tripcomcommunication/encoding" namespace
         ="http://DefaultNamespace" use="encoded" />
```

```
8    </wsdl:input>
9
10   <wsdl:output name="addResponse">
11   <wsdlsoap:body encodingStyle="http://www.tripcom.org/protocol/tripcomcommunication/encoding" namespace
         ="http://DefaultNamespace" use="encoded" />
12   </wsdl:output>
13   </wsdl:operation>
14
15   </wsdl:operation>
16   </wsdl:binding>
17   <wsdl:service name="mathService">
18
19   <wsdl:port binding="impl:mathSoapBinding" name="math">
20   <wsdlsoap:address targetclient="tripcom://www.example.org/clientabc" subspace="tripcom://www.example.org
         /subspace123"/>
21   </wsdl:port>
22   </wsdl:service>
```

Listing 3.5: Sample WSDL description with Triple Space bindings

#### Implementation guidelines for Triple Space bindings for WSDL

This section provides implementation guidelines in order to enable the Web Services to be invoked using Triple Space. Assuming that the Apache Axis (`http://ws.apache.org/axis2`) will be used for the implementation purposes, guidelines for making the updates and additions have been proposed.

Apache Axis has two major subsystems which was called as Java2WSDL and WSDL2Java [10]. As name suggests, Java2WSDL allows generating a WSDL description of a Web Services to be exposed from a Java Interface. Similarly, WSDL2Java converts WSDL description exposed over the Web and generates Java artifacts to be usedfor client to access the Web Service. Based on the proposal in section 3.1.3 for providing Triple Space bindings for WSDL, the two tools (Java2WSDL and WSDL2Java) are required to be updated.

Java2WSDL API [10] implementation has to be extended to make it able to generate the WSDL description of Web Services from Java interface, based on the propositions made in the above section. The WSDL description generated by Java2WSDL should reflect all the updates proposed for providing Triple Space bindings for WSDL, i.e. mentioning the Triple Space as transport protocol, RDF representation of SOAP as encoding style and providing address information as sub-space and client IDs.

WSDL2Java API [10] implementation has also to be enhanced in the similar way. It should be able to process the WSDL description with the proposed extensions for Triple Space bindings and generate the Java artifacts accordingly.

## 3.2 SOAP Mapping with RDF

The SOAP messaging framework [12] defines an XML-based message format and a processing model for Web Service interactions. While in D4.1, the general ideas behind SOAP and its relation to TripCom were introduced, the work in the following paragraphs and Section 3.3 is focused on providing a description of a *SOAP Web Service binding for Triple Space*. The SOAP messaging framework defines the mechanism of *SOAP bindings* [13, 22] for enabling transmission of SOAP messages between SOAP processing nodes over a network, using potentially different network transport protocols. For this purpose, they define (i) a serialization of the SOAP infoset in such a way that it can be transmitted by a sender over the chosen network transport protocol

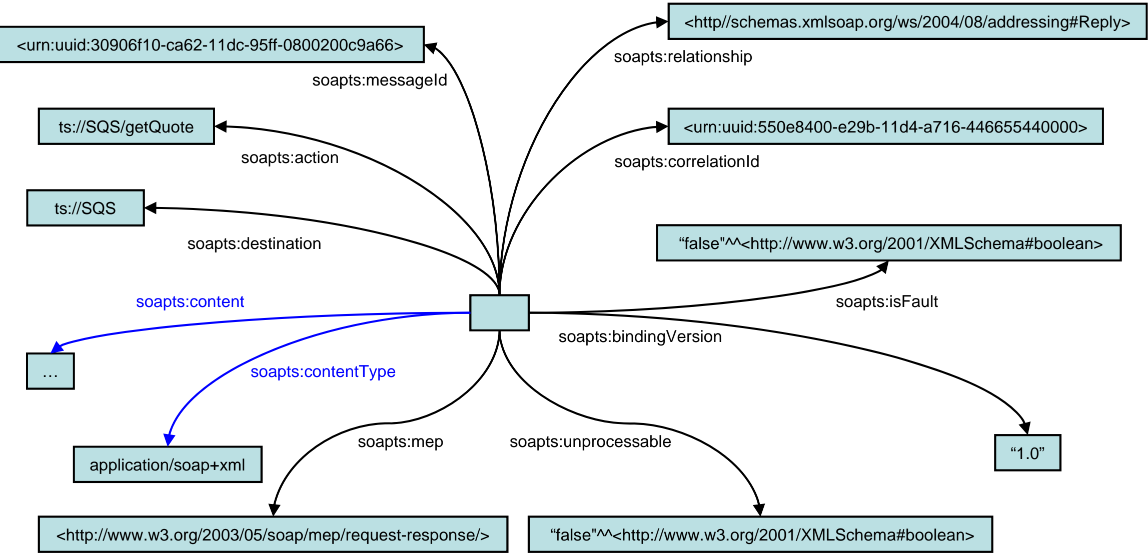


Figure 3.2: Example: RDF representation of a SOAP envelope.

(e.g. HTTP, JMS) and reconstructed by the receiver (or the next hop/node in case of multi-hop interactions). Furthermore, they (ii) describe how the services of the underlying transport protocol (i.e. its interface) are used to transmit the chosen serialization of the SOAP infoset between SOAP processing nodes and describe potential failure scenarios that can be anticipated within the binding. Since multiple SOAP bindings for different network transport protocols can be employed along the message path from sender to receiver, the mechanism of SOAP bindings essentially enables multi-protocol message exchange. In this section, the initial work carried out in D4.1 is continued by providing a description of a Triple Space grounding (i.e. RDF graphs) for SOAP messages to enable their transmission over Triple Space. This section is complemented by Section 3.3.4 where a description how various Web Service message exchange patterns can be realized using Triple Space API operations is provided.

### 3.2.1 Mapping SOAP envelopes to RDF

Figure 3.2 shows an example of a SOAP envelope encapsulated in an RDF graph. Each SOAP envelope, i.e. a sequence of zero or more headers and one or more body parts, is represented as an RDF blank node of type `soap-env` in namespace `http://tripcom.org/ns/soapts`[1], also referred to as SOAP message graph root. To retain the exact structure of the SOAP envelope during transmission (including e.g. arbitrary header structure or binary attachments) and to ensure compatibility of the defined SOAP Triple Space binding with other specifications of the Web Services stack of standards, the RDF representation of SOAP envelopes is based on the following basic principles:

- The original SOAP message as received by the Web Service runtime implementing the SOAP Triple Space binding must not be modified.
- Certain information contained in the SOAP message that is necessary for delivery of the message, such as information related to addressing of communication

[1]In the remainder of the document, this namespace is referred to as `soapts`.

partners and message correlation is "propagated" from the SOAP envelope to the RDF graph that encapsulates the original SOAP envelope. As a result, this information can be used by clients to retrieve SOAP envelopes from Triple Space by using its query functionality.

- Information that is not directly related to message delivery or message content but important for implementation of the defined binding and administrative purposes may be added to the RDF graph representing a SOAP message.

Conceptually, the RDF representation of a SOAP envelope consists of the actual SOAP envelope as received from the Web Service runtime (related properties are depicted in blue in Figure 3.2) and various metadata properties necessary for message delivery (related properties are depicted in black in Figure 3.2). These are explained in detail in the following sections.

### SOAP Message Graph Properties

The metadata necessary for a Triple Space-based delivery of SOAP envelopes consists of a number of properties that can be classified according to 3 categories:

1. Headers representing **addressing-related information**; these are *Destination*, *Action* and *Reference parameters*
2. Headers representing **information related to message identification and message correlation**; these are *MessageId*, *MEP*,*Relationship* and *CorrelationId*
3. Headers representing **information related to binding implementation and administration**; these are *BindingVersion*, *Unprocessable* and *IsFault*

In the following, a description of each of the aforementioned properties is given. The property description is the also basis for the presentation of the realization of Web Service message exchange patterns using Triple Space in Section 3.3.

**Destination** The *Destination* property is used to identify the destination of a message. The destination is identified via a Internationalized Resource Identifier (IRI) which in case of Triple Space-based Web Service invocation could e.g. be a IRI identifying the service provider a service invocation request message is directed to. In the example (Figure 3.2), the destination of the message is a "Stock Qoute Service" identified via IRI ts://SQS. The destination IRI property is linked to the SOAP message graph root via the soapts:destination property. This property is propagated to the SOAP message graph level from the WS-Addressing *Destination* property on the SOAP envelope level. The destination property is used by Web Service providers who offer their services over Triple Space for retrieving messages directed to them by issuing a corresponding query through the TS API.

**Action** The *Action* property conveys the intent of a message in form of a IRI, i.e. it indicates the operation that will be executed when the message is processed; this may e.g. be a reference to an input, output or fault message within a WSDL

interface. In the example, the action that will be carried out by message processing is identified via IRI ts://SQS/getQuote. The action IRI property is linked to the SOAP message graph root via the soapts:action property. This property is propagated to the SOAP message graph level from the WS-Addressing *Action* property on the SOAP envelope level. The action property provides Web service providers with finer grained message retrieval granularity than using the destination property only.

**Reference parameters** Certain complex communication scenarios might involve multiple instances of the same service type; however, instances of a certain service type share a common destination property. *Reference parameters* enable a message sender to provide additional information to allow a set of multiple message receivers to dispatch input messages accordingly. Reference parameters are represented as an RDF-graph that is linked to the SOAP message graph via the soapts:referenceParameters property. As reference parameters are purely application-defined and can have arbitrary structure, their definition is beyond the scope of this specification and has to be defined as a contract between message senders and receivers.

**MessageId** The *MessageId* property uniquely identifies each SOAP message graph via an IRI. It is propagated to the SOAP message graph level from the WS-Addressing *MessageId* property from the SOAP envelope level and linked to the SOAP message graph root by the soapts:messageId property.

**MEP** The *MEP* property identifies the message exchange pattern that governs the message exchange a message belongs to via an IRI. Possible values are e.g. are the message exchange pattern identifiers defined as part the WSDL 2.0 specification [3]. In the example, the message is part of a request-response interaction. The MEP IRI property is linked to the SOAP message graph root by the soapts:mep property.

**CorrelationId** To enable correlation of messages as part of interactions that involve more that one message exchange between communication partners, the *CorrelationId* property of a message can contain the message identifier of another message which is “in relation” to the given message. In case of a request-response interaction as presented in the example, the CorrelationId property refers to the original request message that cause the creation of response message. The CorrelationId property is linked to the SOAP message graph root by the soapts:correlationId property.

**Relationship** How a message relates to the message with the identifier given in the CorrelationId property is defined by the *Relationship* property. Valid values for the relationship type property are dependent on the message exchange pattern the message belongs to; the relationship type values defined as part of the WS-Addressing specification are used where possible. In case of the interaction presented in the example, the message has a relationship property that identifies it as a reply message to the message with the message id given in the CorrelationId property. The relationship property is linked to the SOAP message graph root by the soapts:relationship property. In combination with the CorrelationId property, this property enables e.g. a service requester to retrieve a response

message corresponding to a certain request message in a request-reply interaction. This can be achieved by the service requester querying Triple Space for a message that has its Relationship property set to "reply" and the original request message's message identifier is the CorrelationId property.

**Binding Version** The bindingVersion property reflects the version of the SOAP Web Service binding for Triple Space that was used for transmitting the respective message and has been added to enable future extension of the binding while retaining backward compatibility of implementations. The BindingVersion property is linked to the SOAP message graph root by the soapts:bindingVersion property and contains a literal value identifying the binding version used. The version as described is "1.0".

**Is Fault** In case the message encapsulated in the content property is a SOAP fault message, the *Is fault* property contains a boolean *true* value, otherwise it has the value of a boolean *false*. Similar to e.g. the message identifier, this information has been propagated to the tuple level to make it accessible for template matching, i.e. enabling convenient retrieval of all fault or non-fault messages in a triplespace. The property is linked to the SOAP message graph root by the soapts:isFault property.

**Unprocessable** In case a SOAP processor encounters any errors while processing the SOAP message, e.g. while parsing the message, the *Unprocessable* property is set to a boolean value *true*. This facilitates simple retrieval of un-processable messages by administrators for debugging purposes and can be used to prevent repeated consumption of un-processable messages. Note that in contrast to the *Is fault* property which indicates a fault on the application level, the *Unprocessable* property indicates an error that results from the SOAP message processor e.g. not being able to dispatch the message to a service implementation.

**Content** The *content* property contains the complete SOAP envelope, comprising both the SOAP headers and the SOAP body of the original SOAP message as defined by the original sender of the SOAP message (potentially with modifications carried out by SOAP intermediaries along the message path the SOAP message has taken). The SOAP envelope represented as a literal and linked to the SOAP graph root by the soap:content property.

**Content Type** The encoding of the SOAP message is specified by the *Content Type* property. In most cases, the preferred content type is "application/soap+xml" where the SOAP message is transferred as a XML string in UTF-8 encoding; however other other encodings are possible such as e.g. "multipart/related" in case of a MIME encoded SOAP message with binary attachments.

### SOAP Body Part Mapping

In addition to the SOAP RDF mapping as described in the previous section, it might be interesting in certain scenarios to transform the actual SOAP message payload, i.e. the body part of the SOAP envelope, to RDF to allow Triple Space clients to retrieve messages by querying on message content rather than message metadata. As the SOAP body is defined by the application sending the SOAP message and may

be of arbitrary structure and content, there is no generic way to transform it to an RDF graph while retaining compatibility to existing Web Service tooling. In the following, we describe two approaches that can be employed to enable an application-specific RDF representation of the SOAP body part. In the first approach, called *client-side RDF encoding*, the message payload is RDF-encoded by Triple Space clients (i.e. the applications interacting through Triple Space). In this case, the application sending a message provides the message payload in form of an RDF graph and the application consuming the message is able to interpret the RDF-encoded message payload. Consequently no mapping to or from RDF has to be performed by Triple Space. This way of message body encoding can be used e.g. for enabling interaction between WSMO Web Services, using client-side RDF encoding of WSMO instances as defined in the WSML RDF mapping [8]. In the second approach, called *Triple Space-side RDF encoding*, used for integration of legacy Web Services (Web Services that are not aware of the Triple Space Web Service transport mechanism), Triple Space Web Service bridges as described in D4.1 transform the message body provided by the message sender (e.g. provided in form of an XML document) to RDF, which can be communicated over Triple Space, using e.g. SA-WSDL's lifting/lowering schema mappings. No changes to the already existing Web Service or the Web Service client are necessary. All communication between clients and services is carried out over the Triple Space Web Service bridge, which must be set up with the appropriate mappings, that can take the XML data from the client and transform it to RDF for the Triple Space; and similarly, the response data from the Triple Space can be taken from the Triple Space and transformed to XML data to be sent back to the client in a response SOAP envelope.

In case one of the approaches described above is employed, the resulting RDF graph is linked to the SOAP message graph root by the soapts:body property.

## 3.3 Enabling Web Service interactions over Triple Space

The term Web Service interaction as it is used in this section refers to a *message exchange* between two or more *participants* (e.g. a service requester and one or more service providers), which is governed by a so-called *message exchange pattern (MEP)* that "identifies the sequence and cardinality of messages sent and/or received as well as who they are logically sent to and/or received from." [3]. The description in this section is based on the introduction of MEPs as presented in D4.1, however it extends the work of D4.1 by (i) providing an analysis of various aspects relevant for enabling Triple Space-based Web Service interaction such as addressing of participants and message correlation and (ii) presenting a detailed description of how the Web Service MEPs as defined in the WSDL 1.1 and WSDL 2.0 specification can be mapped to Triple Space API operations. Since Triple Space exhibits certain unique properties not found in current technologies employed for Web Service communication, three examples for custom extended Web Service message exchange patterns are described and their realization with TS API operations is presented in addition.

### 3.3.1 WSDL 1.1 Message Exchange Patterns

The WSDL 1.1 specification [4] defines four MEPs (called "transmission primitives" in the specification) which characterize incoming and outgoing messages of operations

provided by Web Service endpoints (as part of the description of the Web service Port Type): *One-way*, *Request-response*, *Solicit-response*, and *Notification*.

When an operation is defined as being of type *One-way*, the Web Service endpoint is capable of receiving an input message, it does however not produce any response (or fault message in case of errors). *Request-response* interactions are the most common type of Web Service interaction today; they comprise a message being received by the service (in the following called "request") and one response message being returned to the requester (in the following called "response"). In case of a processing error on the side of the Web Service provider, a fault message is returned to the requester instead of the response message. In addition, WSDL 1.1 defines two more MEPs, namely *Notification* and *Solicit-response*, which are inverted versions of the *One-way* and *Request-response* patterns respectively, where the message exchange is initiated by the Web Service endpoint (i.e. the first operation to be performed on the side of the Web service endpoint is an "output" rather than an "input").

Since all WSDL 1.1 MEPs are also supported in WSDL 2.0 as *In-only*, *In-out*, *Out-only*, and *Out-in*, they are covered as their respective WSDL 2.0 variants in Section 3.3.4).

### 3.3.2 WSDL 2.0 Message Exchange Patterns

The WSDL 2.0 specification [3, 16] comes with a set of eight pre-defined MEPs which can be classified into four groups:

- The *In-only* and *Out-only* MEPs are uni-directional message exchange patterns and correspond to the *One-way* and *Notification* transmission primitives in WSDL 1.1.
- The *In-out* and *Out-in* MEPs are bi-directional message exchange patterns and correspond to the *Request-response* and *Solicit-response* transmission primitives in WSDL 1.1.
- The *Robust In-only* and *Robust Out-only* message exchanges are variants of the uni-directional *In-only* and *Out-only* where participants receive fault messages in case an error occurs.
- The *In-optional-out* and *Out-optional-in* message exchanges are variants of the bi-directional *In-out* and *Out-In* where the service provider MAY send a response or fault message to the service requester.

In addition to the predefined MEPs, WSDL 2.0 allows definition and use of arbitrary MEPs which are described narratively, identified uniquely, and have to be implemented by all involved communication partners.

In Section 3.3.4 a realization of the aforementioned WSDL 2.0 MEPs using Triple Space API operations is presented.

### 3.3.3 Requirements for Triple Space-based Web Service interactions

In this section, a discussion of requirements for enabling Web Service interactions over Triple Space is presented, covering "cross-cutting" functionality that is required

for realizing arbitrary Web Service interactions over a Triple Space communication infrastructure. For the case that multiple service providers (each of them offering a different type of service) are observing the same triplespace for request messages, it is required that each service provider type is identified via a unique identifier (e.g. a IRI). This enables service providers to use the template matching mechanism provided by Triplespace to filter request messages during consumption. This also facilitates easy implementation of the *replicated worker* [11] pattern where multiple instances of the same service provider (where it is irrelevant for a service provider to pick one particular service provider from a set of functionally equivalent service providers). In the SOAP RDF mapping described in Section 3.2 this is enabled by the *Destination* and *Action* properties. tuplespace for request messages, it might be desired to address a certain service provider instance. This is e.g. required for enabling selecting a certain service provider instances based on non-functional properties (that are not reflected by the service type) or for enabling complex message exchanges where it has to be ensured that a service requester always interacts with one particular service provider, due to internal state being held by the service provider. In the SOAP RDF mapping described in Section 3.2 this is enabled by the *Reference parameters* property. Messages exchanged as part of a message exchange pattern have to be uniquely identified. To enable MEPs beyond simple one-way interactions, the messages belonging to a complex MEP need to be correlated, i.e. related to each other. A common scenario for message correlation are request-response interactions where the initiator of multiple concurrent message exchanges with the same service provider must be able to identify the response message belonging to a particular request message. In the SOAP RDF mapping described in Section 3.2 this is enabled by the *MessageId*, *CorrelationId* and *Relationship* properties.

### 3.3.4 Mapping Web Service message exchange patterns to TS API operations

In the following paragraphs, the actions required to implement the WSDL 2.0 MEPs are presented. For this description it is assumed that messages are serialized to RDF triples by the sender according to the mapping defined above prior to storing them into Triple Space and assembled on the side of the receiver after retrieval from Triple Space.

The following description covers only MEPs initiated by a service requester (i.e. where the first operation of the service is an “input” operation). The inverted variants of the MEPs where the message exchange is initiated by the service provider can be realized analogously.

#### In-only MEP

The *In-only* MEP as shown in Figure 3.3 comprises only one message exchange from a service requester to a service provider. The service requester addresses the service provider either directly using the WS-Addressing Destination property or in-directly using the Action property for a more detailed description.

The request message is stored in Triple Space by the service requester using the *out* operation and retrieved by the service provider using (i) either a blocking *in* operation or a corresponding subscription for receiving an asynchronous notification

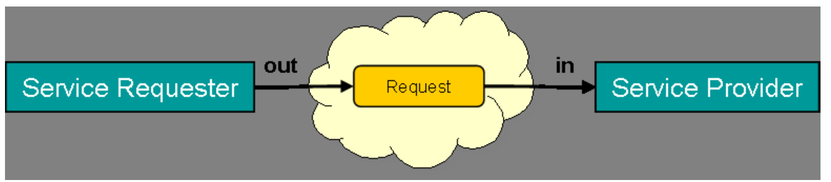


Figure 3.3: Mapping the In-Only MEP to TS API operations.

when a request message is available. The destructive consumption operation (*in*) is used rather than its non-destructive variant (*rd*), since the message should be delivered to the service provider only once.

### In-out MEP

The *In-out* MEP as shown in Figure 3.4 comprises two message exchanges. First, the service requester sends a request message to the service provider. This is achieved following the same procedure as described in the *In-only* MEP.

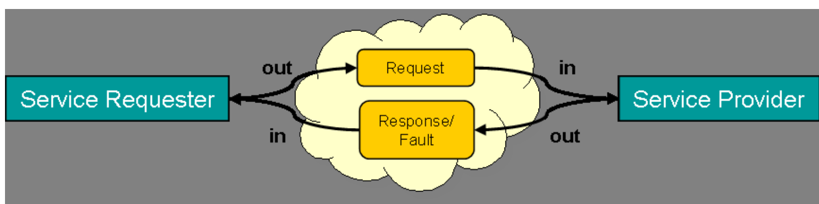


Figure 3.4: Mapping the In-Out MEP to TS API operations.

When the request message has been processed by the service provider, the service provider constructs a corresponding response or fault message. A Relationship property is added to the response message which relates the response or fault message to the IRI given in the message id property of original request message. The service provider stores the response or fault message in Triple Space using the *out* operation. The service requester retrieves the response message for its original request message by executing either (i) a blocking *in* operation, waiting for a message that contains the necessary correlation information that identifies the message as a response to the client's original request message, or (ii) a corresponding subscription. Similarly to the *In-only* pattern, destructive operations are used for message consumption throughout the MEP.

### In-optional-out MEP

The *In-optional-out* MEP as shown in Figure 3.5 is a variant of the *In-out* MEP. However, the response or fault message from service provider to service requester is optional in this pattern.

### Robust In-only MEP

The *Robust In-only* MEP as shown in Figure 3.6 is a variant of the *In-optional-out* MEP. However, a service provider only sends a (fault) message to the service provider in case an error occurs during request processing. In case of successful request processing, the MEP is equal to the *In-only* MEP.

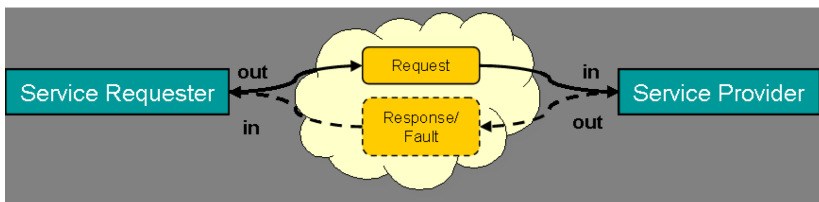


Figure 3.5: Mapping the In-Optional-Out MEP to TS API operations.

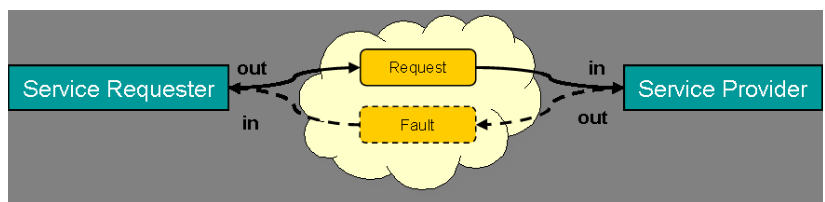


Figure 3.6: Mapping the Robust In-Only MEP to TS API operations.

### 3.3.5 Extended non-standard MEP

In the following paragraphs, three examples for custom message exchange patterns that can be implemented elegantly over Triple Space are presented.

#### One-to-many MEP

The *One-to-many* MEP as shown in Figure 3.7 is essentially a multi-cast operation. The service requester performs the same execution steps as in the *In-only* MEP.

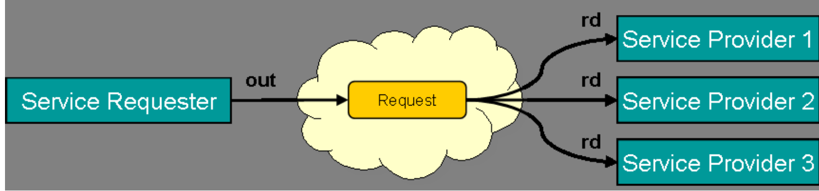


Figure 3.7: Mapping the One-to-Many MEP to TS API operations.

The service providers however, retrieve request messages through the non-destructive *rd* operation instead of the destructive *in* operation. As a result, multiple service providers can retrieve and process the same request message.

#### Continuous-update MEP

The *Continuous-update* MEP as shown in Figure 3.8 is a variant of the *In-only* and *One-to-many* MEPs. Similarly to the *In-only* MEP, the service requester does not expect to receive any response or fault messages.

A service provider retrieves the request message using the non-destructive *rd* operation, hence the request message remains in the space (and can therefore be retrieved in a similar way by other service providers). The service requester however, may destructively retrieve the request message (*in*), update the request message, and store the request message back in Triple Space (*out*).

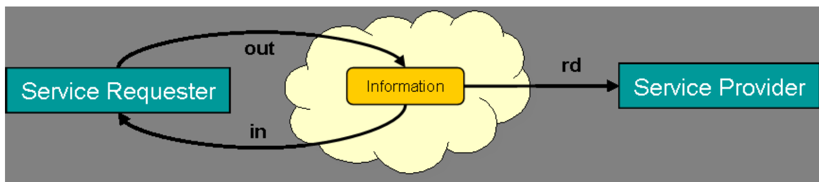


Figure 3.8: Mapping the Continuous-update MEP to TS API operations.

### Request-for-bid MEP

The *Request-for-bid* MEP as shown in Figure 3.9 is a variant of the *Continuous-update* and *In-out* MEPs, where potentially multiple service providers non-destructively consume a request message (*rd*) and each service provider sends a corresponding response message to the service requester.

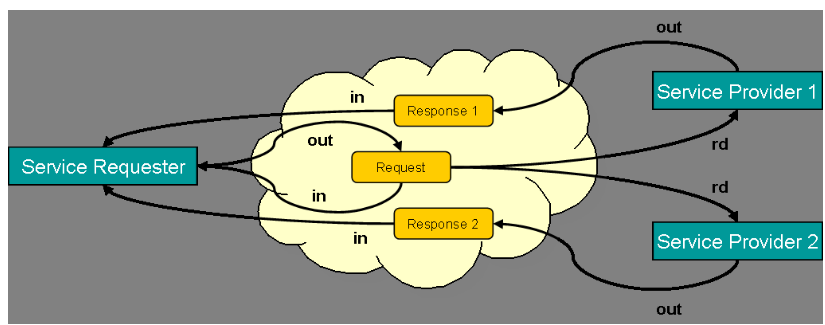


Figure 3.9: Mapping the Request-for-Bid MEP to TS API operations.

The MEP is terminated by the service requester by destructively retrieving (*out*) the request message from Triple Space.

## 3.3.6 Summary

In Table 3.3.6 all previously described Web Service message exchange patterns are listed along with the necessary TS API operation to be performed by service requester and service provider in each step of the respective message exchange pattern.

| MEP | Step | Requester Op. | Provider Op. |
|---|---|---|---|
| In-only | 1 | out(request) | |
| | 2 | | in(request) |
| Out-only | 1 | | out(request) |
| | 2 | in(request) | |
| In-out | 1 | out(request) | |
| | 2 | | in(request) |
| | 3 | | out(response) |
| | 4 | in(response) | |
| Out-in | 1 | | out(request) |
| | 2 | in(request) | |
| | 3 | out(response) | |
| | 4 | | in(response) |
| Robust In-only | 1 | out(request) | |
| | 2 | | in(request) |
| | 3 | | if fault: out(fault) |
| | 4 | in(fault) | |
| Robust Out-only | 1 | | out(request) |
| | 2 | in(request) | |
| | 3 | if fault: out(fault) | |
| | 4 | | in(fault) |
| In-optional-out | 1 | out(request) | |
| | 2 | | in(request) |
| | 3 | | if response: out(response) |
| | 4 | in(response) | |
| Out-optional-in | 1 | | out(request) |
| | 2 | in(request) | |
| | 3 | if response: out(response) | |
| | 4 | | in(response) |
| One-to-many | 1 | out(request) | |
| | 2 | | each provider: rd(request) |
| Contiuous-update | 1 | out(request) | |
| | 2 | | each provider: rd(request) |
| | 3 | update: in(request) | |
| | 4 | update: out(request) | |
| Request-for-bid | 1 | out(request) | |
| | 2 | | each provider: rd(request) |
| | 3 | | out(response) |
| | 4 | for each response: in(response) | |
| | 5 | in(request) | |

Table 3.1: Mapping Web service message exchange patterns to TS API operations

# 4 Triple Space Grounding in Semantic Web Service Execution Environment

The Triple Space grounding with respect to Web Services communication is described in this chapter. The grounding requirements and specification in Semantic Web Service Execution Environment (WSMX), i.e. for the end-point Web Service invoker, client-endpoint, resource and execution management are provided in Sections 4.2, 4.3, 4.4 and 4.5 respectively. These sections discuss grounding and mapping APIs of the respective components to TS API and provide guidelines for implementing such mappings.

## 4.1 Grounding component in Web Service Execution Environment (WSMX)

In this section, we describe the current state of grounding component in Web Service Execution Environment (WSMX). Apart from discovering Web Services and composing them, the Semantic Execution Environment (SEE) also needs to invoke the services send the necessary service request messages and receive the responses. Such invocations, including any necessary data transformations, will be taken care of by this component.

Because internal communication within the SEE uses semantic data and practically all currently deployed Web Services use their specific XML formats, the Grounding and Invocation component needs to translate between the involved data forms. This translation is also known as data grounding. Above that, this component also needs to support concrete network protocols (HTTP, SOAP, other bindings) to be able to exchange messages with the Web Service.

The Grounding and Invocation component will also provide Triple Space based communication as an additional communication channel between the SEE and external Web Services.

The Grounding and Invocation component provides the following functionalities (only to be used where applicable):

- data grounding  two-way transformations between semantic data within SEE and the XML data used in external communication network
- protocol binding  based on the WSDL description of the target Web service, the best supported protocol binding will be selected for communication
- Triple Space grounding for communication of the SEE with Web services over Triple Space

As a part of theoretical work done in the area of Grounding for Semantic Web Services, a way of distinguishing between data grounding approaches has been devised [15]. There are three different paths have been found between the XML data and the semantic quadrant where the transformations can be implemented: on the XML level, on the semantic level, and a direct option spanning the two levels.

Since Semantic Web ontologies can be serialized in XML, an XSLT (or similar) transformation can be created between the XML data and the XML serialization of

the ontological data. In case the XML format of the ontological data is actually suitable for a particular Web service, the transformation can be avoided.

Moreover, an ad-hoc ontology can be generated from the XML Schema present in the WSDL document, with automatic transformations (lifting/lowering) between the XML data and their equivalent in the ad-hoc ontology. Then a transformation using an ontology mapping language can be designed to get to the target ontology.

## 4.2 TripCom grounding for Web Service invoker

The *WSMX Web Service invoker* is part of the WSMX communication manager component and is responsible for mapping an invocation of a WSMX-external Web Service provider to the transport protocol supported by the Web Service. In combination with the WSMX grounding component, which implements a bi-directional mapping between semantic data communication between WSMX component (in form of WSMO objects) and the message and data format supported by WSMX-external Web Service endpoints, it enables invocation facilitates invocation of WSMX-external Web Service endpoints. Note that this section describes how the WSMX Web service Invoker component has to be extended to support invocation of Web services over Triple Space from within WSMX. Further issues of Triple Space based Web service invocation without WSMX are discussed in D6.3.

### 4.2.1 State of the art

When a Web Service is to be invoked by WSMX, the invoker component receives (i) the WSMO description of the Web Service to invoke and (ii) the request instance data in form of a WSMO object through the communication manager's invoke operations (Listing 4.1).

```
1  public List<Entity> invoke (
2       WebService service,
3       List<Entity> data,
4       String grounding
5  ) throws ComponentException, UnsupportedOperationException
```

Listing 4.1: WSMX Web Service invoker

The invoker uses the WSMX grounding component to extract the request parameters necessary for service invocation from the WSMO Web Service description (e.g. the service's WSDL description and the porttype/operation to invoke) and to ground the request data to the data/message format and communication protocol supported by the destination Web Service by executing the corresponding adapter. After execution of the grounding component, the transformed invocation request message is transmitted to the WSMX-external Web service using the communication primitives of the transport protocol supported by the corresponding Web Service.

If the Web Service is exposed as e.g. an HTTP endpoint, the Web service invoker sends the invocation request message to the destination Web Service by performing a synchronous HTTP-POST operation. After reception of the response message of the Web service invocation, the grounding component is invoked again to lift the response message to the semantic level again (i.e. transformed to a WSMO object).

In its current form, the WSMX invoker supports synchronous invocation of Web Service over a HTTP transport (i.e. the WSMX invoker is blocked until the response

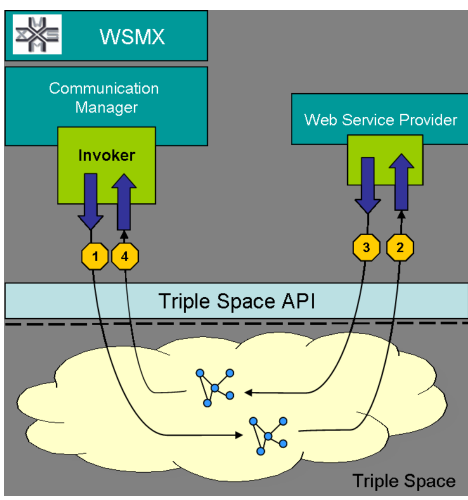


Figure 4.1: Triple Space based Web Service invocation in WSMX

of the Web Service invocation request is received). Due to its communication paradigm of persistent publication, Triple Space however provides a platform for Web service communication that is inherently asynchronous in nature. In Section 4.2.2, a description of extensions to the WSMX invoker that are necessary to support Triple Space as a transport protocol for Web Service invocation in WSMX is presented.

### 4.2.2 Mapping the WSMX invoker to TS API operations

To facilitate Triple Space based Web Service invocation in WSMX, an alternative relatization of the WSMX invoker interface (see Listing 4.1) is provided that maps the operations of the invoker interface to the communication primitives provided by the Triple Space (TS) API.

The extended WSMX invoker with support for the Triple Space transport protocol is dependent on an extended grounding component that implements the transformation of the invocation request from WSMO objects communicated inside WSMX to (i) a message format (e.g. in form of the SOAP-RDF representation proposed in Section 3.2) and (ii) message payload format that can be processed by the service provider.

Figure 4.1 depicts the sequence of TS API operations performed during asynchronous Triple Space based Web Service invocation by WSMX.

1. Upon execution of the `invoke` method of the Triple Space based invoker implementation, the WSMX grounding component is invoked for transforming the payload of the Web Service invocation (passed in through `data` parameter) to RDF format to facilitate its transmission over triplespace and its processing by the service provider. The `grounding` string provides (i) information on the data representation used by the service to send and receive messages – interpreted by the grounding component – and (ii) information the identifies the service provider endpoint – interpreted by the invoker component – in case of Triple Space based Web Service invocation the URI of the Triple Space the service provider monitors for Web Service invocation requests.

   After construction of the invocation request RDF graph, it is stored in the corresponding Triple Space using the `out` TS API operation.

2. The Web Service invocation request is consumed by the service provider by either (i) registering a callback at the triplespace for a template describing the message format it can process through the TS API operation `subscribe` or (ii) by issuing blocking or non-blocking `read` operation calls.

   After message consumption, the invocation request message is processed by the the service provider.

3. This step is dependent on the message exchange pattern the communication between WSMX and service provider conforms to. The following description applies to a request-response interaction.

   After processing of invocation request message of a request-reply interaction, a response message created, if necessary transformed to RDF representation and stored in triplespace using the `out` TS API operation.

   Since the communication between WSMX and the service provider is conducted over Triple Space and therefore asynchronous, correlation information has to be added to a response message to enable WSMX (i.e. the initiator of the message interaction) to retrieve the corresponding response message for a particular request message. This is e.g. facilitated by the WS-Addressing SOAP header RelatesTo (and the RelationshipType set to Reply) as described in Section 3.2.1.

4. To retrieve the response message for a Web Service invocation request, the WSMX invoker registers a template based on the aforementioned correlation information at the Triple Space using the `subscribe` TS API operation.

### 4.2.3 Implementation guidelines for extendending the WSMX Invoker with support for Triplespace

In this section, an architecture for supporting Triple Space based Web Service invocation in the WSMX invoker component is presented. The description comprises two different aspects: first, an architecture for the implementation of the Web Service binding for Triple Space as defined in Sections 3.2 and 3.3 is described. Subsequently, it is explained how this binding implementation can be used in the WSMX invoker component to support Web Service invocation over Triple Space.

Currently, it is presumed that the implementation of the Web Service binding for Triple Space will be based on *Apache Axis 2*[1]. *Apache Axis 2* is an open-source (Apache License 2.0[2]) implementation of the SOAP messaging protocol. It provides a framework for defining handlers for processing of SOAP messages, along with a set of pre-defined message handlers for performing certain typical SOAP message processing tasks (such as message correlation and processing of standard WSDL 2.0 message exchange patterns). Handlers can be registered for being called during processing of inbound or outbound messages by attaching them to the *In Pipe* or *Out Pipe* of the Axis engine respectively. Upon handler invocation, a *message context* object is passed to each handler implementation which consists of the actual SOAP message plus internal information necessary for message processing. The SOAP processing parts of Axis 2 are transport agnostic which allows for implementation of SOAP bindings

[1] `http://ws.apache.org/axis2`
[2] `http://www.apache.org/licenses/LICENSE-2.0.html`

to different network transport protocols. The individual bindings are implemented as part of the Axis transport framework as *transport sender* and *transport receiver* components: outbound messages that are to be sent over the respective transport are handled by the transport sender, the message receiver is responsible for consuming inbound messages and triggering message processing by the Axis engine. In Figure 4.2 an overview of the extensions of Apache Axis 2 that are necessary for supporting the Triple Space Web service binding is depicted. The *SOAP RDF Mapping* component implements the RDF representation of SOAP envelopes as outlined in Section 3.2. The *Triple Space Transport Sender* and *Triple Space Transport Receiver* components implement the Web Service message exchange patterns as described in Section 3.3 by storing Web Service invocation messages to and retrieving Web Service invocation messages from Triple Space.

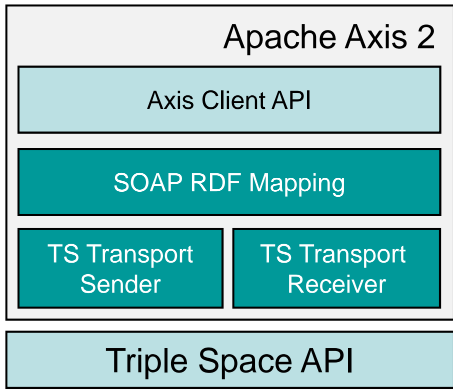


Figure 4.2: Enabling Triple Space based Web Service interactions over Triple Space through a Web Service binding

To perform a Web Service call, the user (i.e. the client application) interacts with the Axis Client API – either directly by calling API functions or through a generated client stub. The Axis API hands the Web Service invocation request message to the Axis engine which subsequently invokes each registered message handler in the out pipe with the message context object as parameter. Finally, the message context is handed over to the Triple Space transport sender. The Triple Space transport sender wraps the message to be transmitted over the Triple Space in an RDF graph as described in Section 3.2, establishes a connection to the Triple Space stores the request message in the destination triplespace. According to the MEP that governs the Web service invocation, the corresponding operations described in Section 3.3.4 and 3.3.5 are used to publish and consume messages to and from triplespace. On the side of the Web Service provider, a connection to triplespace is established by the Triple Space transport listener. The service-side Axis implementation registers a template for entries matching its service name property (and possibly further reference parameters). A subscription for the corresponding template is registered with triplespace in order to receive notifications upon insertion of matching entries; alternatively an equivalent blocking consumption operation could be performed. Once a matching message graph is inserted, the Web service implementation is notified by triplespace. Depending on the message exchange pattern that governs the Web Service interaction, messages are consumed by the triplespace transport listener either destructively or non-destructively. Once the message has been consumed, the encapsulated SOAP message is extracted by

the SOAP RDF mapping component, added to the Axis *MessageContext* and passed through the registered handlers on the service-side Axis implementation which dispatches the Web Service invocation request to the destination service implementation and triggers execution of the actual service application logic. In case of a bi-directional message exchange pattern (e.g. a request-response interaction), the Web Service produces a response message which is sent back to the service provider analogously; i.e. the service provider acts similarly to a service requester in that case.

While the aforementioned description of the Triple Space Web Service binding implementation can be used independently of WSMX, it can be used for extending the WSMX Invoker component with support for Triple Space based Web Service invocation as well. The architecture of the extended WSMX Invoker component is depicted in Figure 4.1.

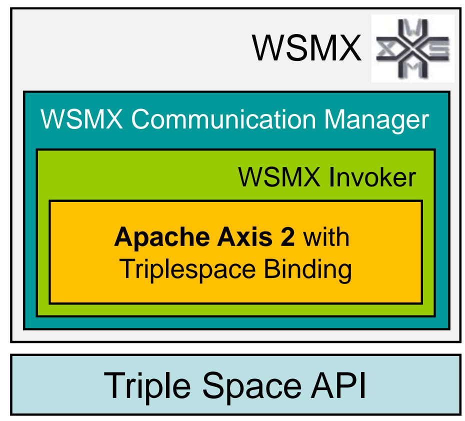


Figure 4.3: Extending the WSMX Invoker component with support for the Triple Space Web Service binding

For this purpose, the current WSMX invoker implementation will be extended with the aforementioned Triple Space Web Service binding implementation: in case a Web Service provider offering its services over Triple Space is to be invoked by WSMX the invocation is carried out using the Triple Space binding implementation as described before, i.e. WSMX acts similar to a non-WSMX Web Service client invoking a Web Service over Triple Space.

## 4.3 TripCom grounding for client-endpoint

The application services or the middleware systems are the typical examples of Triple Space clients. This communication is concerned with Triple Space API. In this scenario, a client can either be a service provider or a service requester involving either a read operation or a write operation to the Triple Space. In this document a typical scenario from WSMX environment is taken to describe Triple Space client-end point grounding. In this scenario, users communicate with WSMX either to find the services they are interested in or to register description of services they are offering. Users invoke *achieveGoal*( ) operation of WSMX communication manager to find services that they are interested in. The description of offered services are registered with WSMX by invoking *publish*( ) operation of WSMX communication manager.

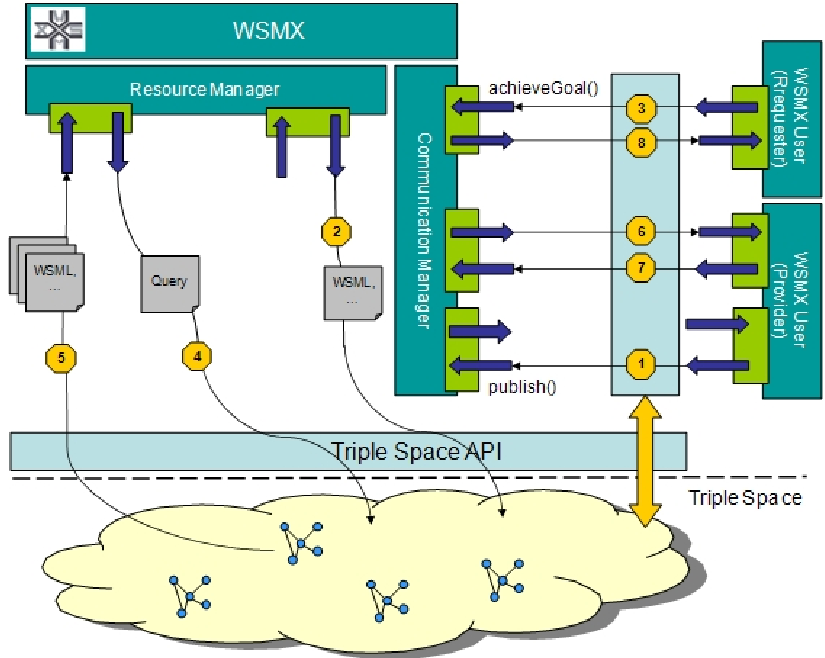


Figure 4.4: User Communicating with WSMX through Triple Space

Triple Space API provides a simple set of *read*, *write* operations to allow interaction between interacting parties. In addition, operations for subscribing to a particular message type (represented as rdf graph pattern) is provided by the Triple Space API. For each subscription, if the matching graph pattern is available, Triple Space notifies to all who subscribed for this particular graph pattern. These simple operation can be mapped to the operations provided by WSMX communication manager API as described in Section 4.3.1. Having such mapping at hand, the communication scenario described above can be mediated through Triple Space thereby avoiding synchronicity and reducing communication overhead. Additionally, this approach offers the benefit of effectively hiding the complexity of service interaction, combining discovery and execution of one or more services to achieve a certain goal, from the service requester. Figure 4.4 shows a high-level description of such communication.

Assuming that WSMX has already registered its endpoint descriptions at Triple Space together with expected message graph, following logical sequence narrates the overall communication process. These sequences are indicated by numbered arrow-lines in Figure 4.4.

1. If a WSMX user wishes to register the descriptions of offered services with WSMX, the service provider invokes the write operation through Triple Space API and writes them to the Triple Space. WSMX user also registers a callback at the Triple Space with a template matching the requests it can process, so as to be notified by the Triple Space upon reception of a matching request message graph.

2. The publish endpoint of WSMX communication manager is triggered by notification which in turn invokes read operation through Triple Space API and reads the message to register with it resource manager.

3. If a WXMX user wishes to obtain required services, the request is submitted by invoking write operation through Triple Space API and writes them to the Triple Space and registers a callback for the response.

4. The achieveGoal endpoint of WSMX communication manager is triggered by notification which in turn invokes read operation through Triple Space API and reads the message to be processed in order to discover the requested services. This request is passed to the WSMX resource manager which invokes the query operation provided by Triple Space API to retrieve the descriptions of services capable of handling the service requesters requirements.

5. The results of the query on the Triple Space are returned to the WSMX resource manager.

6./7. WSMX selects the other WSMX users (i.e. providers), and interacts with them on behalf of the requester (i.e. WSMX user) through the Triple Space API.

8. When the requesters goal is achieved, WSMX returns the result of the invocation back to the service requester through Triple Space API.

### 4.3.1 Mapping WSMX Communication Manager API with Triple Space API

The communication between WSMX and Triple Space is enabled by mapping the WSMX Communication Manager (CM) API with the Triple Space API. The Communication Manager API provides necessary entry points for the clients to communicate with it. Table 4.3.1 shows the corresponding method of Triple Space API which can be used by the methods of the WSMX Communication Manager API while communicating with Triple Space.

| **Communication Manager API operation** | **TS API option to be used** |
|---|---|
| Context achieveGoal (WSMLDocument wsmlMessage) | URI Space create(URI space) followed by void out(Set<Triple> t, URI space) |
| Context invokeWebService (WSMLDocument wsmlMessage) | URI Space create(URI space) followed by Set<Triple> in (Template t, URI space) |
| Context invokeWebService (WSMLDocument wsmlMessage, Context context) | URI Space create(URI space) followed by Set<Triple> s rd (Template t, URI space) |
| Context store (WSMLDocument wsmlMessage) | URI Space create(URI space) followed by void out(Set<Triple> t, URI space) |

Table 4.1: Mapping operations of Communication Manager and TS APIs

### 4.3.2 Implementation Guidelines

In this section we introduce the implementation approach for client end-point grounding. The main grounding task is performed by the component implementing the

*Web Service API*. The grounding service itself is divided into *publication/registration grounding* and *query/request grounding*. The former is concerned with storing client data or registering client with the triplespace where as the latter is concerned with handling user requests such as discovery and invocation of Web Services.

The implementation approach taken should follow the component based implementation approach and separate the concerns of external developers. To assist in implementation of the aforementioned services, following guidelines are provided.

- Provide interface for accessing grounding services by WS API.
- Follow the guidelines provided in 4.3.1 to map operations of WSMX Communication manager to those of Triple Space.
- Since WSMX operations are required to return a Context for each invocation of operation, use Space URI as context
- Use guidelines provided in D4.1 [19] for transforming data formats.

## 4.4 TripCom grounding for Resource Manager

The Resource Manager in WSMX currently manages the persistent storage of data in the repositories. The Resource Manager provides a heterogeneous interface for WSMX. The component implementing this interface is responsible for storing every data item WSMX uses. The WSMO API provides a set of Java interfaces that can be used to represent the domain model defined by WSMO. WSMO4J [1] provides both the API itself and a reference implementation but it is not a prerequisite that implementations of the Resource Manager use WSMO4j. Currently WSMX defines interfaces for six repositories. Four of these repositories correspond to the top level concept of WSMO i.e. Web Services, ontologies, goals, and mediators. The fifth repository is used by WSMX for non-WSMO data items e.g. events and messages. Finally the sixth repository stores WSDL documents used to ground WSMO service descriptions to SOAP or SOAP/HTTP.

The first four repositories Web Service, Goals, Mediators and Ontologies can be provided grounding to Triple Space for the top level entities. The Resource Manager will be provided with RDF grounding support so that while storing data, the local repositories could be bi-passed and the WSML based data could be stored in the Triple Space with URIs (to identify the data afterwards). It will help in making the process of persistent storage independent from WSMX. Moreover, it will help is further exploiting the large Triple Space storage to store the data. There will be no need to maintain local repositories as well. It will not effect the current design of components since the Resource Manager interface will remain same. However, the grounding extensions will help in transforming and storing data on Triple Space.

Reference to WSMO Goals, Ontologies, WS descriptions and Mediators representation (and examples) have to be provided here.

### 4.4.1 Mapping Resource Manager API with TS API

In this section, the Resource Manager API has been mapped with the TS API. The following table shows the corresponding method of TS API which can be used by the

| Resource Manager API operation | TS API operation to be used |
|---|---|
| void saveGoal (Goal goal) | void out (Set<Triple> t, URI space, Time lease) |
| void removeGoal (Goal goal) | Set<Triple> in (Template t, URI space, Time timeout) |
| Set<identifier> getGoalIdentifiers () | Set<Triple> s rd (Template t, Time timeout) |
| Set<Identifier> getGoalIdentifiers (Set<Object>searchTerms, boolean conjunctive) | Set<Triple> s rd (Template t, Time timeout) |
| Set<Identifier> getGoalIdentifiers (Namespace namespace) | Set<Triple> s rd (Template t, Time timeout) |
| boolean containsGoal (Identifier identifier) | Set<Triple> s rd (Template t, Time timeout) |
| Goal loadGoal (Identifier identifier) | Set<Triple> s rd (Template t, Time timeout) |
| Goal loadAllGoals () | Set<Triple> s rd (Template t, Time timeout) |

Table 4.2: Mapping Goal operations of Resource Manager API and TS API

methods of Resource Manager API to ground the data on Triple Space. The table 4.4.1 shows the mapping to Resource Manager API operations mapping with TS API operations for WSMO Goals only. Rest of the tables can be found in annex as 6.4, 6.4 and 6.4.

### 4.4.2 Using ORDI for mapping WSMO top level elements to RDF

ORDI (Ontology Representation and Data Integration) is an open-source ontology middleware that enables enterprise data integration. The ORDI data model introduces basic notions and RDF-like data-structures optimized for ontology and data representation. The model is no way ontology specific and does not prescribe any sort of semantics and epistemology - therefore it is left to the higher levels to impose further interpretations over it.

WSMO4RDF implementation realizes a bidirectional mapping between WSML and WSML RDF - a serialization format defined by [8]. The reference implementation of the WsmoRepository interface uses the named graph and tripleset elements to specify the TopEntity and the Entity that produced the triple statement respectively.

Creating an instance of WsmoConnection (implements WsmoRepository interface, defined in wsmo4j) with default settings

```
1  import com.ontotext.ordi.Factory;
2  import com.ontotext.ordi.wsmo4rdf.Repository;
3
4          ...
5
6  // Instantiate WsmoSource
7  WsmoSource wsmoSource = Factory.createTSource(WsmoSource.class, null);
8
9  // WsmoConnection implements WsmoRepository interface
```

```
10  WsmoConnection wsmoConnection = wsmoSource.getConnection();
```

Creating an instance of WSML RDF serializer

The WSMLTripleSerializer serializes wsmo4j TopEntity types (Ontology, WebService, Goal, Mediator, Capability and Interface) to WSML RDF according to the WSML to RDF mapping specification, [8]. The example below demonstrates how to transform an array of TopEntity objects to a stream of RDFXML.

```
1       import java.io.File;
2       import java.io.FileReader;
3       import java.io.FileWriter;
4       import java.io.IOException;
5       import java.util.HashMap;
6       import java.util.Map;
7
8       import org.wsmo.common.TopEntity;
9       import org.wsmo.wsml.Serializer;
10
11      import com.ontotext.ordi.wsmo4rdf.WSMLTripleSerializer;
12
13      ...
14
15      TopEntity[] topEntities = ...;
16
17      // Construct a WSMLRDF serializer
18      Map<String, Object> createParams = new HashMap<String, Object>();
19      createParams.put(org.wsmo.factory.Factory.PROVIDER_CLASS,
20              WSMLTripleSerializer.class.getName());
21      Serializer rdfparser = org.wsmo.factory.Factory
22              .createSerializer(createParams);
23
24      // Generate the output WSMLRDF
25      try {
26          rdfparser.serialize(topEntities, writer);
27      } catch (IOException e) {
28          throw new RuntimeException("Error while generating the WSMLRDF!", e);
29      }
```

The WSMLTripleParser is responsible for the transformation of WSML RDF to wsmo4j types. The input of the parser is RDFXML document formatted according to the WSML to RDF specification, [8]. The example below demonstrates how to process a WSML RDF file and transform it to WSMO types.

```
1       import java.io.FileReader;
2       import java.io.IOException;
3       import java.util.HashMap;
4       import java.util.Map;
5
6       import org.wsmo.common.TopEntity;
7       import org.wsmo.factory.Factory;
8       import org.wsmo.wsml.Parser;
9
10      import com.ontotext.ordi.wsmo4rdf.WSMLTripleParser;
11
12      ...
13
14      // Create WSMLRDF parser
15      Map<String, Object> createParams = new HashMap<String, Object>();
16      createParams.put(org.wsmo.factory.Factory.PROVIDER_CLASS,
17              WSMLTripleParser.class.getName());
18      Parser wsmlrdfParser = Factory.createParser(createParams);
19
20      // Parse the input WSMLRDF file
21      TopEntity[] topEntities = null;
22      try {
23          topEntities = wsmlrdfParser.parse(new FileReader(file));
24      } catch (Exception e) {
25          throw new RuntimeException(String.format(
26                  "Could not processes the input file %s!", file
```

```
27                                  .getAbsolutePath()), e);
28              }
```

WSMO4RDF implementation realizes a bidirectional mapping between WSML and WSML RDF - a serialization format defined by [8]. The reference implementation of WsmoRepository interface uses the named graph and tripleset elements to specify the TopEntity and the Entity to result the triple statement.

Command line tools can also be used to convert from and to WSMLRDF. In WSMO4RDF data service distribution two tools wsml-to-wsmlrdf and wsmlrdf-to-wsml are provided. To start using them set ORDI-HOME variable to the directory you have unzipped WSMO4RDF distribution, so ORDI-HOME bin should be accessible. Conversion of WSML document to WSMLRDF document is performed by executing:

```
1   wsml2wsmlrdf <path-to-file-name>
```

The command above will result path-to-file-name.rdfxml file to contain the output WSMLRDF content in the directory of the input file.

To perform the reverse conversion from WSMLRDF to WSML use the command:

```
1   wsmlrdf2wsml <path-to-file-name>
```

A new output file will be created with the name path-to-file-name.wsml in the directory where the input file is located.

### 4.4.3 Implementation guidelines

In this section, implementation guidelines and concrete steps have been devised in order building the Triple Space grounding for Resource Manager in WSMX. It will allow it to carry out its data storage and management tasks through triplespace, rather than maintaining individual data repositories within WSMX. A mapping between Resource Management API and the TS API has been devised which has to be carried out. Conversion of WSML data to RDF will be carried out using ORDI framework API which will be used in implementing the proposed mapping.

Considering the well-defined components in WSMX, the implementation approach should be component oriented and should only concern with Resource Manager. The TripCom grounding for Resource Manager should not be visible to other components. Resource Manager will provide the same standard interface for resource management to all other components in between, but it will be connected to the Triple Space underline. Following are the points that should be considered while carrying out the implementation in the next phase of the project:

- Interfaces of the Resource Manager API and TS API should be mapped as proposed in section 4.4.1
- Implementation of the mapping should be carried out with a clear separation kept between Resource Manager and Triple Space middleware
- RDF mapping from WSML to RDF should be carried out using ORDI as mentioned in section 4.4.2
- Resource Manager should only keep the list of URIs for of all the elements stored in the Triple Space

## 4.5 TripCom grounding for Component Management

WSMX has a management component that manages the over all execution of the system by coordinating different components based on dynamic execution semantics. In this way there has been made a clear separation between business and management logic in WSMX. The individual components have clearly defined interfaces and have component implementation well separated with communication issues. Each component in WSMX have wrapper to handle the communication issues. The WSMX manager and individual components wrappers are needed to be interfaced with Triple Space in order to enable the WSMX manager to manage the components over Triple Space. The communication between manager and wrappers of the components will be carried out by publishing and subscribing the data as a set of RDF triples over triplespace. The wrappers of components that handle communication will be interfaced with Triple Space middleware. The WSMX manager has been designed in such a way that it could distinguish between the data flows related with the business logic (execution of components based on the requirements of a concrete operational semantic) and the data flows related with the management logic (monitoring the components, load-balancing, instantiation of threads, etc).

The architecture of WSMX [9], defines a clear separation of communication issues of components and the WSMX internal business logic (can also be called as Execution Semantics [23]). Our goal is that adapting Triple Space Computing for communication in WSMX should not change any of its execution semantics or WSMX internal component logic, not interfaces of components, as well as nor the implementation of components. It does not defines that which component to be invoked, and when. It rather defines that how to invoke (or communicate, in general) with a particular component, by reading and writing data on Triple Space.

Anytime a triple is published in the space, this module will check if related subscriptions are stored. In case that there are related subscriptions, publish-subscribe module will notify to the consumers of those subscriptions that there are triples available. Based on the management information collected by the management module, the publish-subscribe module can prioritize the order of notifications and deliver first to those components which have less workload. The query module will verify the correctness (syntax level) of the query received based on a standard query language. The operation layer in the TripCom Kernel will execute all the operations that are related with the manipulation of data in the space (basically writing, modifying and deleting the triples). It manages version of message communicated over Triple Space and provide URI to it to be able to refer when required, rather that sending the message again. It helps in a way that if a message or semantic description of a Web Service is bring communicated frequently between two or more participants, instead of sending it again and again, a URI of message could be used to save the time.

In order to enable the WSMX Management Component perform component management over Triple Space, a certain set of grounding is needed to be defined where the current status of information could be grounded and published in the Triple Space.

### 4.5.1 Architectural details for integration and grounding TripCom with WSMX Component Manager

The current state of component wrappers consists of reviver and proxies. Reviver Reviver subscribes to a proper event type template to receive event notification. Each wrapper contains the Web Service based proxy of all other components. The component currently being executed might need to invoke other component functionality via Proxy by specifying a component name, a method to be invoked and parameters. The current state of component wrappers, while adding a new component in WSMX, also poses a requirement of adding the proxy of newly added component in the wrappers of all other components to make it recognizable.

Based on all the issues mentioned above, and our proposal on integration of WSMX with Triple Space Computing, we further propose the interfacing of WSMX manager and individual components wrappers with TripCom Kernel in order to enable the WSMX manager to manage the components over Triple Space. The communication between manager and wrappers of the components will be carried out by publishing and subscribing the data as a set of RDF triples over triplespace. All the data that has been exchange between the WSMX component are Web Service Modeling Language (WSML) [7] based Goals and Web Service descriptions. In order to publish the WSML description on Triple Space, WSML description are serialized as a set of RDF triples to be stored as identifiable Named Graphs. The RDF serialization of WSML is performed based on the guidelines and recommendations by Web Service Modeling Language (WSML) Working Groups working draft [7].

There are two ways for the WSMX components to access a TS core, i.e. heavy clients embed the TS core as a Java package and the application and TS core run in the same Java Virtual Machine. The second way is to deploy a standalone TS kernel as a server, which may be accessed by multiple light clients via remoting. Both scenarios can work. However, in order to ensure maximum decoupling WSMX and Triple Space Computing middleware, we have used the light clients. The light clients of Triple Space have to be embedded in the wrappers of each of the WSMX components. It will also keep the complexity level of components wrapper to a limit. It will also give the flexibility that Triple Space light clients embedded in wrappers can either be local or remote to the Triple Space Kernel.

The WSMX can also have the TripCom Kernel local to it which will be the most simplified scenario. In this case, sophisticated mechanisms for providing remote access (through RMI or HTTP) as well as distributed security and trust is not necessary. As shown in the figure above, the Triple Space clients will be embedded in the components wrappers. The Transport module of the wrapper of each component will access the Triple Space through simple APIs. These APIs will implement the operations provided by TS API [18]. WSMX components will get benefit from all major aspects of TS Kernel, i.e. publish-subscribe mechanism, RDF based semantic template matching, data and resource handling, as well as persistent storage. WSMX manager will coordinate all the reading and writing requests received; will dispatch the request to the appropriate functional module (data module or query module); will monitor the appropriate execution of the rest of the elements of the system; and will periodically check the coherence of the information stored in the space.

### 4.5.2 Implementation guidelines

The architecture of WSMX [9], defines a clear separation of communication issues of components and the WSMX internal business logic (can also be called as Execution Semantics [23]). The idea is to make the fact clear that adapting Triple Space for communication in WSMX will not change any of its execution semantics or WSMX internal component logic, not interfaces of components, as well as nor the implementation of components. It doesn't define for WSMX manager, which component to invoke, and when. It rather defines how to invoke (or communicate, in general) with a particular component, by reading and writing data on Triple Space.

Anytime a triple is published in the space, this module will check if related subscriptions are stored. In case that there are related subscriptions, publish-subscribe module will notify to the consumers of those subscriptions that there are triples available. Based on the management information collected by the management module, the publish-subscribe module can prioritize the order of notifications and deliver first to those components which have less workload. The query module will verify the correctness (syntax level) of the query received based on a standard query language. The operation layer in the TripCom Kernel will execute all the operations that are related with the manipulation of data in the space (basically writing, modifying and deleting the triples). It manages version of message communicated over Triple Space and provide URI to it to be able to refer when required, rather that sending the message again. It helps in a way that if a message or semantic description of a Web Service is bring communicated frequently between two or more participants, instead of sending it again and again, a URI of message could be used to save the time. Moreover, using Triple Space Computing for component management, also bring other advantages like, complexity reduction, as underline Triple Space middleware incorporates features like publish-subscribe mechanisms which WSMX manager can reuse for its inter-mediate even data management.

# 5 Run Through Example

This chapter aims to provide a run through example for the use of the Triple Space capabilities of dealing with Web services and Semantic Web services. This example is inspired to the European Patient Summary (EPS) scenario described in details in D8b.1 [2].

In order to increase the efficiency of patient care delivery in Europe, healthcare parties must be able to access and exchange patient information independent of organizational and technological heterogeneities. The European Commission is performing a first step in this direction by defining guidelines for the *European Patient Summary* (EPS): a strategic initiative towards the realization of a European eHealth infrastructure capable of integrating information and applications in order to ensure the pervasive delivery of high quality care services.

But such a goal demands very strong requirements for a technological platform that can support the scale of the scenario. We identify several requirements for an efficient EPS infrastructure:

- **decentralization/distribution** is a prerequisite for the realization of **multilaterality** and **subsidiarity**. A highly distributed EPS infrastructure allows arbitrary healthcare parties to publish and retrieve patient information efficiently and ensures a feasible level of fault-tolerance.
- support for **asynchronous** and **anonymous interaction** among institutions is equally important. The coordination of information should happen independently of communication partners.
- to cope with the inherent heterogeneity problems (e.g. different encoding schemes and languages) and to align different eHealth systems the middleware should provide means for flexible **data and application integration**.
- support for appropriate **security** mechanisms is important with respect to **privacy**. The privacy of citizens must be respected, according to EU and specific country policies that guarantee that only authorized care givers will have access to sensitive data.

Such very demanding requirements are met by the Triple Space infrastructure. Furthermore, the autonomy features of the Triple Space approach match the EPS scenario requirements because of:

**Time autonomy:** each medical care provider is able to store data and retrieve it at any time, even if a prior data provider is currently not available.

**Location autonomy:** the EPS space becomes a virtually closed unit of information since all medical institution participating in a given space for the patient summaries provide parts of the logic and parts of the storage.

**Reference autonomy:** medical care providers do not know what other provider may care for a patient, and thus need the data, in the future.

**Schema autonomy:** each medical care provider may continue to use their own internal schema and terminology. Data mediation capabilities are able to convert data to and from such schemas as appropriate.

Finally, the link with Web services and Semantic Web services makes possible to use the Triple Space for connecting the EPS to a "cloud" of eHealth services spread all over the Europe. For example, eHealth applications that provide booking services to citizens and practitioners for examinations and hospitalizations can be registered into the EPS infrastructure and can be invoked when needed. In this way, the triplespace provides, firstly, a European-wide registry for Web services and Semantic Web services descriptions associated to the eHealth applications and, secondly, an infrastructure for interacting with such applications independently by location, time, reference and schema.

A shared care path use case is reported here, divided into four subsequent steps:

1. The Toothache and the Urgency to visit a Dentist while Abroad,
2. The Laboratory Examination for Additional Control of Patient Status,
3. The Surgical Operation at a Regional Hospital, and
4. The Notification of the General Practitioner at Home.

These episodes are briefly described from a pure use case perspective and, for each step, we then provide a detailed description about the interactions with Web services and Semantic Web services.

## 5.1 Step 1: The Toothache and the Urgency to visit a Dentist while Abroad

*Mr. Christian, an English citizen on holidays, suddenly feels a strong toothache and he finds a small dentist ambulatory. The dentist accesses and reads his patient summary from the EPS infrastructure, administers the correct drug, books a laboratory exam for better inspecting the disease, and prescribes a periodontist inspection.*

Figure 5.1 reports graphically the inner steps performed between the eHR and the triplespace. Those steps are detailed hereafter.

**1.1** The dentist logs into her eHR and connects the eHR to the triplespace so that the triplespace becomes aware of the availability of the eHR.

**1.2** The eHR retrieves the information by submitting an appropriate query via the TS API for retrieving all records in the patient summary with all the relevant information.

**1.3** The triplespace executes such query in the subspace that stores the summary of the citizen. The result of the query is an RDF graph expressed in terms of the EPS and coding systems ontologies and it represents the desired health record of the summary. The graph is returned to the eHR that elaborate it internally for showing the records to the dentist via its own GUI.

**1.4** For selecting a lab that can provides medical examinations, the eHR retrieves all desired Web services (e.g. Web services in WSML format) by submitting a goal (e.g. a goal in WSML format) via Web service discovery API with all the relevant information.

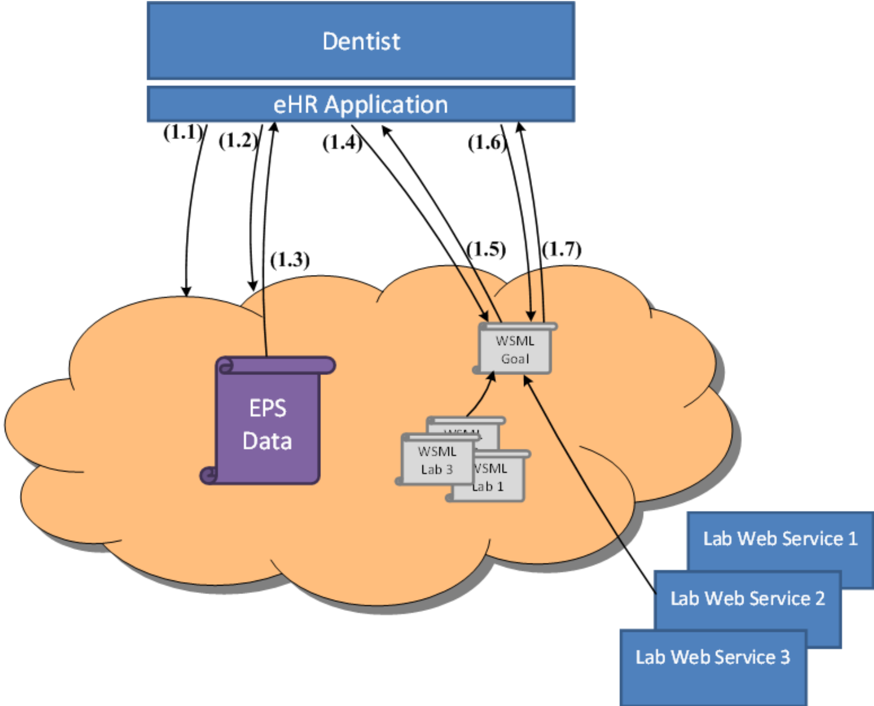


Figure 5.1: Step 1: The Toothache and the Urgency to visit a Dentist while Abroad

**1.5** The Web service discovery component in triplespace executes such a discovery in the subspace that performs matchmaking of the Goal with all the stored Web services registered in the triplespace. The result of such discovery operation is the set of all relevant Web services (e.g. in WSML format). These Web services are returned to the eHR that elaborates them internally to extract the laboratories descriptions. Thus descriptions are presented to the dentist that chooses the most suitable laboratory for the medical examination purposes.

**1.6** For booking the exam to the selected lab, the eHR submits a invocation request (e.g. in WSML format) via Web service invocation component with all the relevant information.

**1.7** The Web service invocation component in triplespace executes such an invocation request by invoking the external Web service. The result of the invocation request is the confirmation of the laboratory booking (e.g. in WSML format). Such confirmation is returned to the eHR that elaborates it internally and shows it to the dentist via its own GUI.

Figure 5.1 shows that during the step 1, eHR acting as a client of the triplespace interacts with the triplespace using TS API, Web service discovery API, and Web service invocation API. The purpose for Step 1.1 is user authentication. Step 1.2 and step 1.3 can be seen as equivalent to a request and response MEP: the client submits a particular request to the service (triplespace in this case) and the service returns desired result. Step 1.4 and step 1.5 can be seen as equivalent to Web service discovery procedure that: the client submits a discovery request to the registry (triplespace in this case) and the registry returns desired Web services. Step 1.6 and step 1.7 can be seen as equivalent to Web service invocation procedure that: via triplespace, the client

submits an invocation request to the Web service, and Web service returns invocation result.

Sample WSML descriptions of Goals and Web Service descriptions are shown in listings 5.1 and 5.2 below:

```
1  wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-dl"
2  namespace { _"http://example.org/"
3  ,
4      eHealth _"http://hospitals.example.org/" }
5
6  webService DentistLab
7
8      nonFunctionalProperties
9          _"http://wiki.wsmx.org/images/4/4a/DiscoveryOntology.wsml#typeOfMatch" hasValue _"http://wiki.
              wsmx.org/images/4/4a/DiscoveryOntology.wsml#IntersectionMatch"
10         _"http://wiki.wsmx.org/images/4/4a/DiscoveryOntology.wsml#usedDiscoveryStrategy" hasValue _"http
              ://wiki.wsmx.org/images/4/4a/DiscoveryOntology.wsml#lightweight"
11     endNonFunctionalProperties
12
13 capability DentistLabCap
14
15 sharedVariables ?type
16
17 precondition DentistLabPre
18
19     definedBy
20             ?type memberOf eHealth#Lab
21
22     and forall ?lab ( (?type[eHealth#Lab hasValue ?lab] implies ?lab memberOf eHealth#Dentist ))
23
24     .
```

Listing 5.1: Sample WSML Goal description

```
1  wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-dl"
2  namespace { _"http://example.org/"
3  ,
4      eHealth _"http://hospitals.example.org/" }
5
6  goal searchDentistLab
7
8      nonFunctionalProperties
9          _"http://wiki.wsmx.org/images/4/4a/DiscoveryOntology.wsml#typeOfMatch" hasValue _"http://wiki.
              wsmx.org/images/4/4a/DiscoveryOntology.wsml#IntersectionMatch"
10         _"http://wiki.wsmx.org/images/4/4a/DiscoveryOntology.wsml#usedDiscoveryStrategy" hasValue _"http
              ://wiki.wsmx.org/images/4/4a/DiscoveryOntology.wsml#lightweight"
11     endNonFunctionalProperties
12
13 capability searchDentistLabCap
14
15 sharedVariables ?type
16
17 precondition searchDentistLab
18
19     definedBy
20             ?type memberOf eHealth#Lab
21
22     and forall ?location ( (?type[eHealth#Lab hasValue ?location] implies ?location memberOf eHealth#
           PlaceInEurope ))
23     and forall ?lab ( (?type[eHealth#Lab hasValue ?lab] implies ?lab memberOf eHealth#Dentist ))
24     .
```

Listing 5.2: Sample WSML Web Service description

## 5.2 Step 2: The Laboratory Examination for Additional Control of Patient Status

*Mr. Christian goes to the laboratory and takes the laboratory examination. Since the batch processing of the examination takes some hours, Mr. Christian leaves the laboratory and goes to the hospital to visit the specialist. When the laboratory will complete the processing of the examination, the LIS*[1] *will send a report of the result of the examination to the specialist.*

Figure 5.2 depicts the steps necessary to perform Steps 2 and 3 of the use case scenario.

**2.1** The LIS starts processing the examination. When this is completed, the examination is sent to the responsible specialist's HIS (see Step 3) over triplespace. For this purpose the result of the examination is wrapped in a SOAP message which is RDF-encoded as described in Section 3.2 (an example for the sample SOAP message and its equivalent RDF-encoded SOAP message are presented in Listings 5.4 and 5.5.

**2.2** The interaction between the LIS and the HIS follows the *in-only* WSDL 2.0 message exchange pattern, meaning that the service requester (i.e. the LIS) sends one Web service invocation message to exactly one service provider (i.e. the HIS) which retrieves the message destructively from triplespace (see Section 3.3). The information necessary for interacting with the service provider is encoded in the service provider's WSDL file according to the description in Section 3.1.3 and can extracted from there by the LIS (an example for this WSDL description is presented in Listing 5.3. The WSDL description is enabled with Triple Space binding as proposed in section 3.1. This information comprises the message format that the service provider can process, which subspace the service provider monitors for request messages and the service identifier. In accordance with the in-only message exchange pattern, the request message is stored in the subspace monitored by the service provider's Web service runtime through TS API operations by the service requester's Web service runtime.

In this step, the Laboratory Information System (LIS) acts as client and the Hospital Information System (HIS) acts as service. HIS is a Web Service which is to be accessed by LIS. LIS invokes the HIS Web Service over the triplespace. It requires the WSDL description of the HIS Web Service with Triple Space bindings. Moreover, the SOAP invocation request to be sent by LIS have also to be RDF encoded. The example WSDL description (with proposed Triple Space bindings) as well as SOAP message (original in XML and encoded in RDF) have been presented below:

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <wsdl:definitions targetNamespace="http://localhost:8080/axis/HIS.jws" xmlns:apachesoap="http://xml.apache.
      org/xml-soap" xmlns:impl="http://localhost:8080/axis/HIS.jws" xmlns:intf="http://localhost:8080/axis/HIS
      .jws" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:wsdl="http://schemas.xmlsoap.
      org/wsdl/" xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org
      /2001/XMLSchema">
3
4
5 <wsdl:message name="getSurgicalInformationDataResponse">
```

[1] LIS: the *Laboratory Information System* is the application used by the laboratory to manage the examination.

```
6     <wsdl:part name="getSurgicalInformationDataReturn" type="xsd:string" />
7     </wsdl:message>
8
9   <wsdl:message name="submitPatientReportResponse">
10    <wsdl:part name="submitPatientReportReturn" type="xsd:boolean" />
11    </wsdl:message>
12
13  <wsdl:message name="submitPatientReportRequest">
14    <wsdl:part name="report" type="xsd:string" />
15    </wsdl:message>
16
17  <wsdl:message name="getSurgicalInformationDataRequest">
18    <wsdl:part name="id" type="xsd:int" />
19    </wsdl:message>
20
21  <wsdl:portType name="HIS">
22
23  <wsdl:operation name="submitPatientReport" parameterOrder="report">
24    <wsdl:input message="impl:submitPatientReportRequest" name="submitPatientReportRequest" />
25    <wsdl:output message="impl:submitPatientReportResponse" name="submitPatientReportResponse" />
26    </wsdl:operation>
27
28  <wsdl:operation name="getSurgicalInformationData" parameterOrder="id">
29    <wsdl:input message="impl:getSurgicalInformationDataRequest" name="getSurgicalInformationDataRequest"
         />
30    <wsdl:output message="impl:getSurgicalInformationDataResponse" name="getSurgicalInformationDataResponse
         " />
31    </wsdl:operation>
32    </wsdl:portType>
33
34  <wsdl:binding name="HISSoapBinding" type="impl:HIS">
35    <wsdlsoap:binding style="rpc" transport="http://tripcom.org/ns/soap-tripcom-binding" />
36
37  <wsdl:operation name="submitPatientReport">
38    <wsdlsoap:operation soapAction="tripcom://his/actions/submitPatientRecord" />
39
40  <wsdl:input name="submitPatientReportRequest">
41    <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="http://
         DefaultNamespace" use="encoded" />
42    </wsdl:input>
43
44  <wsdl:output name="submitPatientReportResponse">
45    <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="http://localhost
         :8080/axis/HIS.jws" use="encoded" />
46    </wsdl:output>
47    </wsdl:operation>
48
49  <wsdl:operation name="getSurgicalInformationData">
50    <wsdlsoap:operation soapAction="tripcom://his/actions/getSurgicalInformationData" />
51
52  <wsdl:input name="getSurgicalInformationDataRequest">
53    <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="http://
         DefaultNamespace" use="encoded" />
54    </wsdl:input>
55
56  <wsdl:output name="getSurgicalInformationDataResponse">
57    <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="http://localhost
         :8080/axis/HIS.jws" use="encoded" />
58    </wsdl:output>
59    </wsdl:operation>
60    </wsdl:binding>
61
62  <wsdl:service name="HISService">
63
64  <wsdl:port binding="impl:HISSoapBinding" name="HIS">
65
66    <wsdlsoap:address targetclient="tripcom://www.ABCHospital.org/his" subspace="tripcom://www.hospitals.org/
         hospitals_information_systems"/>
67    </wsdl:port>
68    </wsdl:service>
69
70    </wsdl:definitions>
```

Listing 5.3: Sample WSDL description for HIS Web Service

```
1  <soap:Envelope
2    xmlns:soap="http://www.w3.org/2001/12/soap−envelope"
3    xmlns:wsa="http://www.w3.org/2005/08/addressing">
4
5    <soap:Header>
6      <wsa:To>tripcom://www.ABCHospital.org/his</wsa:To>
7      <wsa:Action>tripcom://his/actions/submitPatientRecord</wsa:Action>
8      <wsa:MessageID>59d1c740−eeaf−11dc−95ff−0800200c9a66</wsa:MessageID>
9    </soap:Header>
10
11   <soap:Body xmlns:eHealth="http://www.example.org/stock">
12   <eHealth:submitPatientReport>
13     <eHealth:report>"Report ..."</eHealth:report>
14   </eHealth:submitPatientReport>
15   </soap:Body>
16
17 </soap:Envelope>
```

Listing 5.4: Sample SOAP message for HIS Web Service

```
1  @prefix soapts: <http://tripcom.org/ns/soapts#>
2
3  # typing
4  _:root rdf:a soapts:envelope .
5
6  # addressing
7  _:root soapts:to "tripcom://www.ABCHospital.org/his" .
8  _:root soapts:action "tripcom://his/actions/submitPatientRecord" .
9  _:root soapts:messageId "59d1c740−eeaf−11dc−95ff−0800200c9a66" .
10
11 # delivery and processing
12 _:root soapts:mep <http://www.w3.org/ns/wsdl/in−only> .
13 _:root soapts:bindingVersion "1.0" .
14 _:root soapts:isFault false^^<http://www.w3.org/2001/XMLSchema#boolean> .
15 _:root soapts:correlationId "" .
16 _:root soapts:relationship "" .
17
18 # content
19 _:root soapts:contentType "application/soap+xml" .
20 _:root soapts:content "<soap:Envelope> ... </soap:Envelope>" .
```

Listing 5.5: RDF encoded sample SOAP message for HIS Web Service

## 5.3 Step 3: The Surgical Operation at a Regional Hospital

*Mr. Christian reaches the periodontist at the hospital. She uses her information system to retrieve the examination report sent by the remote LIS. In the case that the exam is not ready, she keeps the citizen in the waiting room until the operation is completed. When the examination report is retrieved by the specialist's information system, the periodontist reads the examination report. Having a full picture of the clinical situation of Mr. Christian, the specialist proceeds with the necessary surgical operation. When the operation finishes, the specialist updates the summary of the citizen in the EPS infrastructure with the relevant information about the just executed treatment.*

In this step of the use case scenario, two different ways of clients interacting with the triplespace are presented in two sub-scenarios.

In the first sub-scenario, the HIS of the doctor performing the surgical operation retrieves the examination result message sent by the LIS in Step 2 of the use case scenario to gather all information necessary for the surgery. Here, the triplespace is

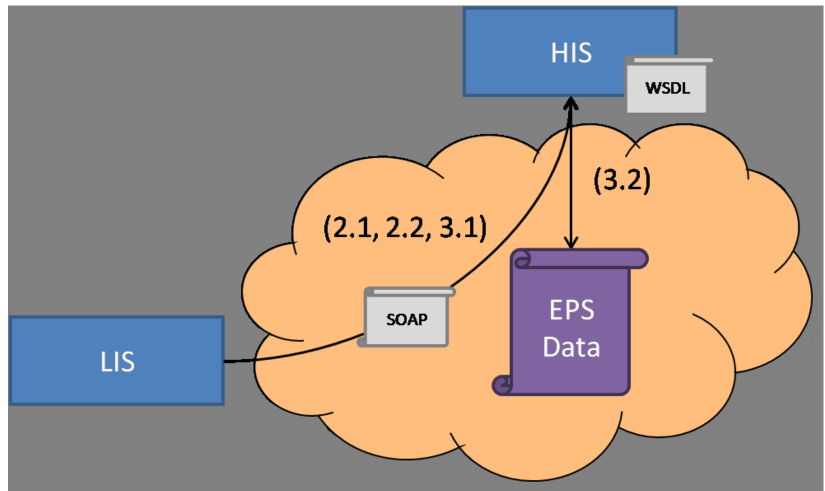


Figure 5.2: Steps 2 and 3: The Laboratory Examination for Additional Control of Patient Status and the Surgical Operation at a Regional Hospital

used as a communication platform for enabling Web service interactions; hence the actual interaction with the triplespace (by calling the Triple Space API operations) is performed by the HIS's Web service middleware implementation.

After the surgery has been performed, the doctors uses the HIS to modify the health record in the patient summary of the citizen by inserting a new RDF graph via the Triple Space API in the second sub-scenario. Here triplespace is used in a "data-oriented" way similar to a database; data stored in the triplespace is modified by the HIS application directly calling Triple Space API operations to access, modify and insert data. The individual steps necessary to realize both sub-scenarios outlined above are depicted in Figure 5.2 and described in the following:

**3.1** Following the in-only message exchange pattern employed for the communication between LIS and HIS, the Web service runtime of the HIS monitors a certain sub-space of the root triplespace for request messages directed to the HIS. The information that is required for clients to send Web service requests to the HIS is contained in the HIS's WSDL description. As soon as an RDF-encoded SOAP message matching the HIS's template is available in the triplespace, it is retrieved by the HIS's Web service runtime asynchronously, the data contained in it is unwrapped and presented to the doctor performing the surgery.

**3.2** For appending the information of the course of the surgery to the citizen's patient record, the HIS (i) retrieves the patient's record using a query expressed in the query language supported by TripCom's query processor, (ii) creates an RDF graph of the information to be appended internally, and (iii) inserts the information to be added using Triple Space API operations and links it to the citizen's patient record.

## 5.4 Step 4: The Notification of the General Practitioner at Home

*The EPS infrastructure notifies the English GP*[2] *responsible for Mr. Christian whenever she connects her information system to the EPS.*

Figure 5.3 reports graphically the inner steps performed between the eHR and the triplespace. Those steps are detailed hereafter.

**4.1** The GP logs into her eHR and connects the eHR to the triplespace so that the triplespace becomes aware of the availability of the eHR.

**4.2** Since the eHR has been subscribed to be notified whenever an update to the summary of the specific citizen occurs, the triplespace notifies the eHR via the notification features of the TS API.

**4.3** The eHR accesses the new information by submitting an appropriate query via the TS API for retrieving the last inserted record in the patient summary with all the relevant information.

**4.4** The triplespace executes such query in the subspace that stores the summary of the citizen. The result of the query is an RDF graph expressed in terms of the EPS and coding systems ontologies and it represents the desired health record of the summary. The graph is returned to the eHR that elaborate it internally in order to visualize the new record to the GP via its own GUI.

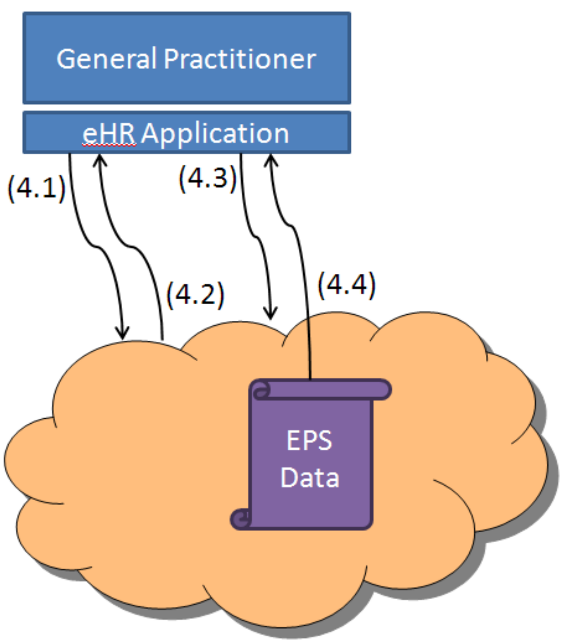


Figure 5.3: Step 4: The Notification of the General Practitioner at Home

The picture shows that during this step 4 there is one client of the triplespace that interact with it using the Triple Space API. Step 4.1 represents the enabling factor in

[2]General Practitioner

order to let the triplespace to perform the step 4.2 by notifying the eHR. Step 4.3 and step 4.4 can be seen as equivalent to a request and response MEP: the client submit a particular request to the service (the triplespace in this case) and the service returns the desired result.

# 6 Conclusions

Whenever a Web Service has to be invoked it is necessary to run some grounding processes that translate the semantic representation of the data that we want to pass to the service into the syntactic form that the service accepts. In other words, grounding must be defined from the semantic descriptions to the underlying WSDL and XML Schema definitions.

TripCom can be exploited as an asynchronous, scalable and reliable medium for communication of Web Services. For this reason, this deliverable addresses all possible grounding issues that may arise when Triple Space Computing is used for communication and coordination of Web Services as well as Semantic Web Services.

There are two main steps in the life cycle of Web Services: after its development a Web Service is published into public registries (such as UDDI or ebXML) and its description is exposed in platform and implementation neutral way (i.e. WSDL) using which it can be invoked with the help of communication protocols (like SOAP). In this deliverable we focused on these two actions and we tried to solve grounding issues in case such actions are performed using TripCom.

The chapter 2 provides basis for the development of a Web service registry based on TripCom. For this purpose, methods to insert into the triplespace both the meta-information used by registries and the service descriptions have been defined.

Grounding between Triple Space and the commonly used parts of the registry information model (which are entities from UDDI registry that also capture corresponding entities from ebXML registry) have been focused as a first step. In particular, a method for translating a tModel (which is the main entity used by UDDI) into an RDF format (which can be easily stored into the triplespace as a set of triples) is described and several guidelines for implementing a component that performs such translation are given.

In order to allow TripCom to store the WSDL description of Web services, they have to be represented as RDF triples. For that purpose, W3C WSDL 2.0 RDF Mapping has been used that defines a method to translate WSDL documents in RDF. It has further been extended accordingly to a list of requirements delineated for the RDF form of WSDL.

The chapter 3 deals with grounding issues concerning Web service communication over TripCom. Web service communication is performed by sending messages using one of the admissible protocols stated in the "Binding" section of the WSDL descriptions of the services involved in the communication. In order to allow a service to send messages by using TripCom as transport protocol, new WSDL binding for the Triple Space has been proposed in order to allow Web Services to be invoked using Triple Space.

Moreover, representing SOAP messages as RDF triples has also been described in order to enable their transmission over Triple Space. In particular, mappings for SOAP headers to RDF have been defined and SOAP body part has been handeled by distinguishing two approaches. In the *client-side RDF encoding* case, no mapping to or from RDF has to be performed by Triple Space because the SOAP payload is in form of an RDF graph already. In the *Triple Space-side RDF encoding* case, Triple Space Web service bridges transform the message body to RDF using SAWSDL's lifting/lowering schema mappings.

In addition, a detailed description has been given about how various Web Service message exchange patterns (MEPs) can be implemented using Triple Space API operations.

After describing all possible grounding issues, Semantic Web Services Execution Environment (WSMX) has been investigated to be able to use TripCom for communication and coordination. In particular, WSMX components have been identified that are required to adapt the proposed TripCom grounding. For each of the identified WSMX components, mappings have been proposed to Triple Space API operations followed by implementation guidelines.

The deliverable ends with a European Patient Summary (EPS) based eHealth use case example that shows the working of Web Services as well as Semantic Web Services based on grounding to triplespaces. Such use case has been described in four following steps which each one emphasizes a specific type of interactions with the triplespace via the TS API and WS API.

# References

[1] Wsmo4j - wsmo api for java. Available at: http://wsmo4j.sourceforge.net.

[2] Dario Cerizza, Emanuele Della Valle, doug foxvog, David de Francisco, Reto Krummenacher, Henar Munoz, Martin Murth, and Elena Simperl. State of the art and requirements analysis for sharing health data in the triplespace. TripCom Project Deliverable D8b.1, March 2007.

[3] R. Chinnici, M. Gudgin, J. J Moreau, J. Schlimmer, and S. Weerawarana. Web services description language (wsdl) version 2.0 part 1: Core language. *W3C Working Draft*, 26, 2004.

[4] E. Christensen, F. Crubera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1, March 2001. Available at: urlhttp://www.w3.org/TR/wsdl.

[5] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1. W3C Note 15 March 2001. Available at http://www.w3.org/TR/2001/NOTE-wsdl-20010315, March 2001.

[6] Luc Clement, Andrew Hately, Claus von Riegen, and Tony Rogers, editors. *UDDI Spec Technical Committee Draft, Version 3.0.2*. OASIS, 2004.

[7] J. de Bruijn, J. Kopeck, and R. Krummenacher. Rdf representation of wsml, d32 wsml working draft, 2006. Available at: `http://www.w3.org/Submission/WSMX`.

[8] J. de Bruijn, J. Kopecký, and R. Krummenacher. D32 RDF Representation of WSML. WSML Working Draft, DERI, February 2006. Available at: `http://www.wsmo.org/TR/d32/v0.1`.

[9] C. Bussler et al. Web service execution environment (wsmx), w3c member submission, 2005. Available at: `http://www.w3.org/Submission/WSMX`.

[10] Apache Software Foundation. Axis user guide, 2005. Available at: `http://ws.apache.org/axis/java/user-guide.html`.

[11] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice*. Pearson Education, 1999.

[12] M. Gudgin, M. Hadley, N. Mendelsohn, J. Moreau, and C. H. F. Nielsen. SOAP Version 1.2. *W3C Recommendation*, June 2003.

[13] M. Gudgin, M. Hadley, and J. J. Moreau. SOAP Version 1.2 Part 1: Messaging Framework. *W3C Recommendation 27 April 2007*, 2007.

[14] Jacek Kopeck, Matthew Moran, Tomas Vitvar, Dumitru Roman, and Adrian Mocan. Wsmo grounding. WSMO Deliverable D24.2, April 2007.

[15] Jacek Kopeck, Dumitru Roman, Matthew Moran, and Dieter Fensel. Semantic web services grounding. In *Proceedings of the International Conference on Internet and Web Applications and Services (ICIW'06)*, 2 2006.

[16] A. A Lewis. Web services description language (wsdl) version 2.0: Additional meps. *W3C Working Draft 23 May 2007*, 2007.

[17] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, and T. R. Payne. OWL-S: Semantic Markup for Web Services, 2004.

[18] L. Nixon, E. Simperl, R. Krummenacher, F. Martin-Recuerda, V. Momtchev, M. Murth, G. Joskowicz, and e. Kuhn. Specification and implementation of a semantic linda model, tripcom project deliverable d3.1, 2007.

[19] Brahmananda Sapkota, Doug Foxvog, Daniel Wutke, Daniel Martin, Martin Murth, Omair Shafiq, Andrea Turati, Emanuele Della Valle, Nuria Sanchez, and Jacek Kopecky. Architectural Integration of Triple Spaces with Web Service Infrastructures. Technical report, TripCom, FP6 - 027324, 2007.

[20] Martin Treiber Schahram Dustdar. A View Based Analysis onWeb Service Registries. *Distributed and Parallel Databases*, July 2005.

[21] Semantic Annotations for WSDL and XML Schema. Recommendation, W3C, August 2007. Available at `http://www.w3.org/TR/sawsdl/`.

[22] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2005.

[23] M. Zaremba and C. Bussler. Towards dynamic execution semantics in semantic web services. In *", in Proceedings of the Workshop on Web Service Semantics: Towards Dynamic Business Integration, International Conference on the World Wide Web (WWW), Chiba, Japan*, 2005.

# Appendix A

## 6.1 Grounding Registry Data Model to RDF

```
1  <rdf:RDF
2
3      xmlns:rdf="http://www.w3.org/1999/02/22−rdf−syntax−ns#"
4      xmlns:t="http://www.w3.org/2001/XMLSchema#">
5
6      <rdf:Description rdf:about="http://www.tripcom.org/tmodel.xsd#uddi">
7
8      <t:element rdf:parseType="Resource">
9        <t:name>tModel</t:name>
10       <t:type>uddi:tModel</t:type>
11     </t:element>
12     <t:complexType rdf:parseType="Resource">
13       <t:sequence rdf:parseType="Resource">
14         <t:element rdf:parseType="Resource">
15           <t:ref>uddi:description</t:ref>
16           <t:minOccurs>0</t:minOccurs>
17           <t:maxOccurs>unbounded</t:maxOccurs>
18         </t:element>
19         <t:element rdf:parseType="Resource">
20           <t:minOccurs>0</t:minOccurs>
21           <t:ref>uddi:overviewURL</t:ref>
22         </t:element>
23       </t:sequence>
24       <t:name>overviewDoc</t:name>
25     </t:complexType>
26     <t:targetNamespace>urn:uddi−org:api_v2</t:targetNamespace>
27     <t:element rdf:parseType="Resource">
28       <t:type>uddi:description</t:type>
29       <t:name>description</t:name>
30     </t:element>
31     <t:complexType rdf:parseType="Resource">
32       <t:simpleContent rdf:parseType="Resource">
33         <t:extension rdf:parseType="Resource">
34           <t:attribute rdf:parseType="Resource">
35             <t:ref>xml:lang</t:ref>
36           </t:attribute>
37           <t:base>string</t:base>
38         </t:extension>
39       </t:simpleContent>
40       <t:name>description</t:name>
41     </t:complexType>
42     <t:element rdf:parseType="Resource">
43       <t:type>uddi:categoryBag</t:type>
44       <t:name>categoryBag</t:name>
45     </t:element>
46     <t:element rdf:parseType="Resource">
47       <t:name>overviewURL</t:name>
48       <t:type>string</t:type>
49     </t:element>
50     <t:simpleType rdf:parseType="Resource">
51       <t:restriction rdf:parseType="Resource">
52         <t:base>string</t:base>
53       </t:restriction>
54       <t:name>tModelKey</t:name>
55     </t:simpleType>
56     <t:element rdf:parseType="Resource">
57       <t:name>keyedReference</t:name>
58       <t:type>uddi:keyedReference</t:type>
59     </t:element>
60     <t:attributeFormDefault>unqualified</t:attributeFormDefault>
61     <t:elementFormDefault>qualified</t:elementFormDefault>
62     <t:element rdf:parseType="Resource">
63       <t:type>uddi:overviewDoc</t:type>
64       <t:name>overviewDoc</t:name>
65     </t:element>
66     <t:element rdf:parseType="Resource">
```

```
67        <t:name>name</t:name>
68        <t:type>uddi:name</t:type>
69      </t:element>
70      <t:complexType rdf:parseType="Resource">
71        <t:attribute rdf:parseType="Resource">
72          <t:use>optional</t:use>
73          <t:type>string</t:type>
74          <t:name>operator</t:name>
75        </t:attribute>
76        <t:name>tModel</t:name>
77        <t:attribute rdf:parseType="Resource">
78          <t:use>optional</t:use>
79          <t:name>authorizedName</t:name>
80          <t:type>string</t:type>
81        </t:attribute>
82        <t:sequence rdf:parseType="Resource">
83          <t:element rdf:parseType="Resource">
84            <t:minOccurs>0</t:minOccurs>
85            <t:ref>uddi:overviewDoc</t:ref>
86          </t:element>
87          <t:element rdf:parseType="Resource">
88            <t:ref>uddi:categoryBag</t:ref>
89            <t:minOccurs>0</t:minOccurs>
90          </t:element>
91          <t:element rdf:parseType="Resource">
92            <t:ref>uddi:identifierBag</t:ref>
93            <t:minOccurs>0</t:minOccurs>
94          </t:element>
95          <t:element rdf:parseType="Resource">
96            <t:ref>uddi:name</t:ref>
97          </t:element>
98          <t:element rdf:parseType="Resource">
99            <t:minOccurs>0</t:minOccurs>
100           <t:ref>uddi:description</t:ref>
101           <t:maxOccurs>unbounded</t:maxOccurs>
102         </t:element>
103       </t:sequence>
104       <t:attribute rdf:parseType="Resource">
105         <t:name>tModelKey</t:name>
106         <t:use>required</t:use>
107         <t:type>uddi:tModelKey</t:type>
108       </t:attribute>
109     </t:complexType>
110     <t:complexType rdf:parseType="Resource">
111       <t:simpleContent rdf:parseType="Resource">
112         <t:extension rdf:parseType="Resource">
113           <t:attribute rdf:parseType="Resource">
114             <t:use>optional</t:use>
115             <t:ref>xml:lang</t:ref>
116           </t:attribute>
117           <t:base>string</t:base>
118         </t:extension>
119       </t:simpleContent>
120       <t:name>name</t:name>
121     </t:complexType>
122     <t:element rdf:parseType="Resource">
123       <t:name>identifierBag</t:name>
124       <t:type>uddi:identifierBag</t:type>
125     </t:element>
126     <t:complexType rdf:parseType="Resource">
127       <t:sequence rdf:parseType="Resource">
128         <t:element rdf:parseType="Resource">
129           <t:maxOccurs>unbounded</t:maxOccurs>
130           <t:ref>uddi:keyedReference</t:ref>
131         </t:element>
132       </t:sequence>
133       <t:name>categoryBag</t:name>
134     </t:complexType>
135     <t:version>2.03</t:version>
136     <t:complexType rdf:parseType="Resource">
137       <t:name>keyedReference</t:name>
138       <t:attribute rdf:parseType="Resource">
139         <t:use>required</t:use>
```

```
140             <t:type>string</t:type>
141             <t:name>keyValue</t:name>
142          </t:attribute>
143          <t:attribute rdf:parseType="Resource">
144             <t:name>keyName</t:name>
145             <t:type>string</t:type>
146             <t:use>optional</t:use>
147          </t:attribute>
148          <t:attribute rdf:parseType="Resource">
149             <t:use>optional</t:use>
150             <t:name>tModelKey</t:name>
151             <t:type>uddi:tModelKey</t:type>
152          </t:attribute>
153       </t:complexType>
154       <t:complexType rdf:parseType="Resource">
155          <t:name>identifierBag</t:name>
156          <t:sequence rdf:parseType="Resource">
157             <t:element rdf:parseType="Resource">
158                <t:maxOccurs>unbounded</t:maxOccurs>
159                <t:ref>uddi:keyedReference</t:ref>
160             </t:element>
161          </t:sequence>
162       </t:complexType>
163       <t:import rdf:parseType="Resource">
164          <t:schemaLocation>http://www.w3.org/2001/xml.xsd</t:schemaLocation>
165          <t:namespace>http://www.w3.org/XML/1998/namespace</t:namespace>
166       </t:import>
167    </rdf:Description>
168 </rdf:RDF>
```

Listing 6.1: XML RDF representation of the tModel structure.

## 6.2 Representing Registry Data Model in RDF

```
1  <?xml version="1.0"?>
2  <rdf:RDF
3    xmlns:p1="http://www.owl-ontologies.com/Ontology1194353329.owl#"
4    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
6    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
7    xmlns:owl="http://www.w3.org/2002/07/owl#"
8    xml:base="http://www.owl-ontologies.com/Ontology1194353329.owl">
9
10   <owl:Ontology rdf:about=""/>
11   <owl:Class rdf:ID="uddi_identifierBag"/>
12   <owl:Class rdf:ID="uddi_keyedReference"/>
13   <owl:Class rdf:ID="uddi_categoryBag"/>
14   <owl:Class rdf:ID="uddi_overviewDoc"/>
15   <owl:Class rdf:ID="uddi_tModel"/>
16
17   <owl:ObjectProperty rdf:ID="identifierBag">
18     <rdfs:domain rdf:resource="#uddi_tModel"/>
19     <rdfs:range rdf:resource="#uddi_identifierBag"/>
20   </owl:ObjectProperty>
21
22   <owl:ObjectProperty rdf:ID="keyedReference">
23     <rdfs:domain>
24       <owl:Class>
25         <owl:unionOf rdf:parseType="Collection">
26           <owl:Class rdf:about="#uddi_identifierBag"/>
27           <owl:Class rdf:about="#uddi_categoryBag"/>
28         </owl:unionOf>
29       </owl:Class>
30     </rdfs:domain>
31     <rdfs:range rdf:resource="#uddi_keyedReference"/>
32   </owl:ObjectProperty>
33
34   <owl:ObjectProperty rdf:ID="categoryBag">
35     <rdfs:domain rdf:resource="#uddi_tModel"/>
```

```
36     <rdfs:range rdf:resource="#uddi_categoryBag"/>
37   </owl:ObjectProperty>
38
39   <owl:ObjectProperty rdf:ID="overviewDoc">
40     <rdfs:range rdf:resource="#uddi_overviewDoc"/>
41     <rdfs:domain rdf:resource="#uddi_tModel"/>
42   </owl:ObjectProperty>
43
44   <owl:DatatypeProperty rdf:ID="keyName">
45     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
46     <rdfs:domain rdf:resource="#uddi_keyedReference"/>
47   </owl:DatatypeProperty>
48
49   <owl:DatatypeProperty rdf:ID="operator">
50     <rdfs:domain rdf:resource="#uddi_tModel"/>
51     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
52   </owl:DatatypeProperty>
53
54   <owl:DatatypeProperty rdf:ID="overviewURL">
55     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
56     <rdfs:domain rdf:resource="#uddi_overviewDoc"/>
57   </owl:DatatypeProperty>
58
59   <owl:DatatypeProperty rdf:ID="authorizedName">
60     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
61     <rdfs:domain rdf:resource="#uddi_tModel"/>
62   </owl:DatatypeProperty>
63
64   <owl:DatatypeProperty rdf:ID="keyValue">
65     <rdfs:domain rdf:resource="#uddi_keyedReference"/>
66     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
67   </owl:DatatypeProperty>
68
69   <owl:DatatypeProperty rdf:ID="name">
70     <rdfs:domain rdf:resource="#uddi_tModel"/>
71     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
72   </owl:DatatypeProperty>
73
74   <owl:DatatypeProperty rdf:ID="description">
75     <rdfs:domain>
76       <owl:Class>
77         <owl:unionOf rdf:parseType="Collection">
78           <owl:Class rdf:about="#uddi_tModel"/>
79           <owl:Class rdf:about="#uddi_overviewDoc"/>
80         </owl:unionOf>
81       </owl:Class>
82     </rdfs:domain>
83     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
84   </owl:DatatypeProperty>
85
86   <owl:DatatypeProperty rdf:ID="tModelKey">
87     <rdfs:domain>
88       <owl:Class>
89         <owl:unionOf rdf:parseType="Collection">
90           <owl:Class rdf:about="#uddi_keyedReference"/>
91           <owl:Class rdf:about="#uddi_tModel"/>
92         </owl:unionOf>
93       </owl:Class>
94     </rdfs:domain>
95     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
96   </owl:DatatypeProperty>
97 </rdf:RDF>
```

Listing 6.2: XML snippet of UDDI structure

## 6.3 WSDL RDF Mapping Example

```
1  <description
2     xmlns="http://www.w3.org/ns/wsdl"
```

```
3      targetNamespace="http://greath.example.com/2004/wsdl/resSvc"
4      xmlns:tns="http://greath.example.com/2004/wsdl/resSvc"
5      xmlns:ghns="http://greath.example.com/2004/schemas/resSvc"
6      xmlns:wsoap="http://www.w3.org/ns/wsdl/soap"
7      xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
8      xmlns:sawsdl="http://www.w3.org/ns/sawsdl"
9      xmlns:wsdlx="http://www.w3.org/ns/wsdl-extensions">
10
11   <documentation>
12     This document describes the GreatH Web service. Additional
13     application-level requirements for use of this service --
14     beyond what WSDL 2.0 is able to describe -- are available
15     at http://greath.example.com/2004/reservation-documentation.html
16   </documentation>
17
18   <types>
19     <xs:schema
20         xmlns:xs="http://www.w3.org/2001/XMLSchema"
21         targetNamespace="http://greath.example.com/2004/schemas/resSvc"
22         xmlns="http://greath.example.com/2004/schemas/resSvc">
23
24       <xs:element name="checkAvailability" type="tCheckAvailability"
25             sawsdl:loweringSchemaMapping="http://greath.example.com/lowering.xslt" />
26
27       <xs:complexType name="tCheckAvailability">
28         <xs:sequence>
29           <xs:element name="checkInDate" type="xs:date"/>
30           <xs:element name="checkOutDate" type="xs:date"/>
31           <xs:element name="roomType" type="xs:string"/>
32         </xs:sequence>
33       </xs:complexType>
34
35       <xs:element name="checkAvailabilityResponse" type="xs:double"/>
36       <xs:element name="invalidDataError" type="xs:string"/>
37     </xs:schema>
38   </types>
39
40   <interface name="reservationInterface"
41       sawsdl:modelReference="http://greath.example.com/ontology#reservation" >
42
43     <fault name="invalidDataFault"
44         element="ghns:invalidDataError"/>
45
46     <operation name="opCheckAvailability"
47             pattern="http://www.w3.org/ns/wsdl/in-out"
48             style="http://www.w3.org/ns/wsdl/style/iri"
49             wsdlx:safe="true">
50         <input messageLabel="In"
51               element="ghns:checkAvailability" />
52         <output messageLabel="Out"
53               element="ghns:checkAvailabilityResponse" />
54         <outfault ref="tns:invalidDataFault" messageLabel="Out"/>
55     </operation>
56
57   </interface>
58
59   <binding name="reservationSOAPBinding"
60           interface="tns:reservationInterface"
61           type="http://www.w3.org/ns/wsdl/soap"
62           wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/">
63
64     <fault ref="tns:invalidDataFault"
65       wsoap:code="soap:Sender"/>
66     <operation ref="tns:opCheckAvailability"
67       wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response"/>
68   </binding>
69
70   <service name="reservationService"
71        interface="tns:reservationInterface">
72
73      <endpoint name="reservationEndpoint"
74                binding="tns:reservationSOAPBinding"
75                address="http://greath.example.com/2004/reservation"/>
```

```
76    </service>
77  </description>
```

Listing 6.3: WSDL 2.0 primer example, with added SAWSDL annotations

```
1   @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
2   @prefix sawsdl: <http://www.w3.org/ns/sawsdl#> .
3   @prefix whttp: <http://www.w3.org/ns/wsdl/http#> .
4   @prefix wsdl: <http://www.w3.org/ns/wsdl-rdf#> .
5   @prefix wsdlx: <http://www.w3.org/ns/wsdl-extensions#> .
6   @prefix wsoap: <http://www.w3.org/ns/wsdl/soap#> .
7   @prefix tsxs: <http://tripcom.org/ns/tsxs#> .
8
9
10  <http://greath.example.com/2004/wsdl/resSvc#wsdl.description()>
11    a wsdl:Description ;
12    wsdl:interface
13      <http://greath.example.com/2004/wsdl/resSvc#wsdl.interface(reservationInterface)> ;
14    wsdl:binding
15      <http://greath.example.com/2004/wsdl/resSvc#wsdl.binding(reservationSOAPBinding)> ;
16    wsdl:service <http://greath.example.com/2004/wsdl/resSvc#wsdl.service(reservationService)> ;
17    rdfs:isDefinedBy <http://greath.example.com/2004/wsdl/resSvc.wsdl> .
18
19
20  <http://greath.example.com/2004/wsdl/resSvc#wsdl.interface(reservationInterface)>
21    a wsdl:Interface ;
22    rdfs:label "reservationInterface" ;
23    wsdl:interfaceOperation
24      <http://greath.example.com/2004/wsdl/resSvc#wsdl.interfaceOperation(reservationInterface/
            opCheckAvailability)> ;
25    wsdl:interfaceFault
26      <http://greath.example.com/2004/wsdl/resSvc#wsdl.interfaceFault(reservationInterface/invalidDataFault)> ;
27    sawsdl:modelReference <http://greath.example.com/ontology#reservation> .
28
29  <http://greath.example.com/2004/wsdl/resSvc#wsdl.interfaceOperation(reservationInterface/opCheckAvailability)
        >
30    a wsdl:InterfaceOperation ;
31    rdfs:label "opCheckAvailability" ;
32    sawsdl:modelReference wsdlx:SafeInteraction ;
33    wsdl:interfaceMessageReference
34      <http://greath.example.com/2004/wsdl/resSvc#wsdl.interfaceMessageReference(reservationInterface/
            opCheckAvailability/In)> ,
35      <http://greath.example.com/2004/wsdl/resSvc#wsdl.interfaceMessageReference(reservationInterface/
            opCheckAvailability/Out)> ;
36    wsdl:interfaceFaultReference
37      <http://greath.example.com/2004/wsdl/resSvc#wsdl.interfaceFaultReference(reservationInterface/
            opCheckAvailability/Out/invalidDataFault)> ;
38    wsdl:messageExchangePattern <http://www.w3.org/ns/wsdl/in-out> ;
39    wsdl:operationStyle <http://www.w3.org/ns/wsdl/style/iri> .
40
41  <http://greath.example.com/2004/wsdl/resSvc#wsdl.interfaceMessageReference(reservationInterface/
        opCheckAvailability/In)>
42    a wsdl:InterfaceMessageReference , wsdl:InputMessage ;
43    wsdl:elementDeclaration <urn:uuid:ead76f4f-f5bc-4ca7-af8c-460a9c99fa53> ;
44    wsdl:messageContentModel wsdl:ElementContent ;
45    wsdl:messageLabel <http://www.w3.org/ns/wsdl/in-out#In> .
46
47  <urn:uuid:ead76f4f-f5bc-4ca7-af8c-460a9c99fa53>
48    a wsdl:QName ;
49    wsdl:localName "checkAvailability" ;
50    wsdl:namespace <http://greath.example.com/2004/schemas/resSvc> .
51
52  <urn:uuid:c3220a0a-3a1b-4f51-8635-cfe6273024b4>
53    a tsxs:ElementDeclaration ;
54    tsxs:qname <urn:uuid:ead76f4f-f5bc-4ca7-af8c-460a9c99fa53> ;
55    sawsdl:loweringSchemaMapping <http://greath.example.com/lowering.xslt> .
56
57  <http://greath.example.com/2004/wsdl/resSvc#wsdl.interfaceMessageReference(reservationInterface/
        opCheckAvailability/Out)>
58    a wsdl:InterfaceMessageReference , wsdl:OutputMessage ;
59    wsdl:elementDeclaration <urn:uuid:3066fc8c-d060-4bc5-a479-44d752de54ac> ;
60    wsdl:messageContentModel wsdl:ElementContent ;
61    wsdl:messageLabel <http://www.w3.org/ns/wsdl/in-out#Out> .
```

```
62
63  <urn:uuid:3066fc8c-d060-4bc5-a479-44d752de54ac>
64    a wsdl:QName ;
65    wsdl:localName "checkAvailabilityResponse" ;
66    wsdl:namespace <http://greath.example.com/2004/schemas/resSvc> .
67
68  <http://greath.example.com/2004/wsdl/resSvc#wsdl.interfaceFaultReference(reservationInterface/
        opCheckAvailability/Out/invalidDataFault)>
69    a wsdl:InterfaceFaultReference , wsdl:OutputMessage ;
70    wsdl:interfaceFault
71      <http://greath.example.com/2004/wsdl/resSvc#wsdl.interfaceFault(reservationInterface/invalidDataFault)> ;
72    wsdl:messageLabel <http://www.w3.org/ns/wsdl/in-out#Out> .
73
74  <http://greath.example.com/2004/wsdl/resSvc#wsdl.interfaceFault(reservationInterface/invalidDataFault)>
75    a wsdl:InterfaceFault ;
76    rdfs:label "invalidDataFault" ;
77    wsdl:elementDeclaration <urn:uuid:2e72002d-9472-4c4a-a5b8-c6614b609355> ;
78    wsdl:messageContentModel wsdl:ElementContent .
79
80  <urn:uuid:2e72002d-9472-4c4a-a5b8-c6614b609355>
81    a wsdl:QName ;
82    wsdl:localName "invalidDataError" ;
83    wsdl:namespace <http://greath.example.com/2004/schemas/resSvc> .
84
85
86  <http://greath.example.com/2004/wsdl/resSvc#wsdl.binding(reservationSOAPBinding)>
87    a wsdl:Binding , <http://www.w3.org/ns/wsdl/soap> ;
88    rdfs:label "reservationSOAPBinding" ;
89    wsdl:binds
90      <http://greath.example.com/2004/wsdl/resSvc#wsdl.interface(reservationInterface)> ;
91    wsdl:bindingOperation
92      <http://greath.example.com/2004/wsdl/resSvc#wsdl.bindingOperation(reservationSOAPBinding/
            opCheckAvailability)> ;
93    wsdl:bindingFault
94      <http://greath.example.com/2004/wsdl/resSvc#wsdl.bindingFault(reservationSOAPBinding/invalidDataFault)
            > ;
95    whttp:defaultQueryParameterSeparator "&" ;
96    wsoap:protocol <http://www.w3.org/2003/05/soap/bindings/HTTP/> ;
97    wsoap:version "1.2" .
98
99  <http://greath.example.com/2004/wsdl/resSvc#wsdl.bindingOperation(reservationSOAPBinding/
        opCheckAvailability)>
100   a wsdl:BindingOperation ;
101   wsdl:binds
102     <http://greath.example.com/2004/wsdl/resSvc#wsdl.interfaceOperation(reservationInterface/
            opCheckAvailability)> ;
103   wsoap:soapMEP <http://www.w3.org/2003/05/soap/mep/soap-response> .
104
105 <http://greath.example.com/2004/wsdl/resSvc#wsdl.bindingFault(reservationSOAPBinding/invalidDataFault)>
106   a wsdl:BindingFault ;
107   wsdl:binds
108     <http://greath.example.com/2004/wsdl/resSvc#wsdl.interfaceFault(reservationInterface/invalidDataFault)> ;
109   wsoap:faultCode <urn:uuid:2fe62e8f-154a-4d6d-ae16-c8c062c5c60b> .
110
111 <urn:uuid:2fe62e8f-154a-4d6d-ae16-c8c062c5c60b>
112   a wsdl:QName ;
113   wsdl:localName "Sender" ;
114   wsdl:namespace <http://www.w3.org/2003/05/soap-envelope> .
115
116
117 <http://greath.example.com/2004/wsdl/resSvc#wsdl.service(reservationService)>
118   a wsdl:Service ;
119   rdfs:label "reservationService" ;
120   wsdl:endpoint
121     <http://greath.example.com/2004/wsdl/resSvc#wsdl.endpoint(reservationService/reservationEndpoint)> ;
122   wsdl:implements
123     <http://greath.example.com/2004/wsdl/resSvc#wsdl.interface(reservationInterface)> .
124
125 <http://greath.example.com/2004/wsdl/resSvc#wsdl.endpoint(reservationService/reservationEndpoint)>
126   a wsdl:Endpoint ;
127   rdfs:label "reservationEndpoint" ;
128   wsdl:address <http://greath.example.com/2004/reservation> ;
129   wsdl:usesBinding
```

```
130 <http://greath.example.com/2004/wsdl/resSvc#wsdl.binding(reservationSOAPBinding)> .
```

Listing 6.4: WSDL 2.0 primer example in RDF form

## 6.4 Mapping Resource Manager API with TS API

| Resource Manager API operation | TS API operation to be used |
| --- | --- |
| void saveMediator (Mediator mediator) | void out (Set<Triple> t, URI space, Time lease) |
| void removeMediator (Mediator mediator) | Set<Triple> in (Template t, URI space, Time timeout) |
| Set<identifier> getMediatorIdentifiers () | Set<Triple> s rd (Template t, Time timeout) |
| Set<Identifier> getMediatorIdentifiers (Set<Object>searchTerms, boolean conjunctive) | Set<Triple> s rd (Template t, Time timeout) |
| Set<Identifier> getMediatorIdentifiers (Namespace namespace) | Set<Triple> s rd (Template t, Time timeout) |
| boolean containsMediator (Identifier identifier) | Set<Triple> s rd (Template t, Time timeout) |
| Goal loadMediator (Identifier identifier) | Set<Triple> s rd (Template t, Time timeout) |
| Goal loadAllMediators () | Set<Triple> s rd (Template t, Time timeout) |

Table 6.1: Mapping Mediator operations of Resource Manager API and TS API

| Resource Manager API operation | TS API operation to be used |
| --- | --- |
| void saveWebService (WebService webService) | void out (Set<Triple> t, URI space, Time lease) |
| void removeWebService (WebService webService) | Set<Triple> in (Template t, URI space, Time timeout) |
| Set<identifier> getWebServiceIdentifiers () | Set<Triple> s rd (Template t, Time timeout) |
| Set<Identifier> getWebServiceIdentifiers (Set<Object>searchTerms, boolean conjunctive) | Set<Triple> s rd (Template t, Time timeout) |
| Set<Identifier> getWebServiceIdentifiers (Namespace namespace) | Set<Triple> s rd (Template t, Time timeout) |
| boolean containsWebService (Identifier identifier) | Set<Triple> s rd (Template t, Time timeout) |
| Goal loadWebService (Identifier identifier) | Set<Triple> s rd (Template t, Time timeout) |
| Goal loadAllWebServices () | Set<Triple> s rd (Template t, Time timeout) |

Table 6.2: Mapping Web Service operations of Resource Manager API and TS API

| Resource Manager API operation | TS API operation to be used |
| --- | --- |
| void saveOntology (Ontology ontology) | void out (Set<Triple> t, URI space, Time lease) |
| void removeOntology (Ontology ontology) | Set<Triple> in (Template t, URI space, Time timeout) |
| Set<identifier> getOntologyIdentifiers () | Set<Triple> s rd (Template t, Time timeout) |
| Set<Identifier> getOntologyIdentifiers (Set<Object>searchTerms, boolean conjunctive) | Set<Triple> s rd (Template t, Time timeout) |
| Set<Identifier> getOntologyIdentifiers (Namespace namespace) | Set<Triple> s rd (Template t, Time timeout) |
| boolean containsOntology (Identifier identifier) | Set<Triple> s rd (Template t, Time timeout) |
| Goal loadOntology (Identifier identifier) | Set<Triple> s rd (Template t, Time timeout) |
| Goal loadAllOntologys () | Set<Triple> s rd (Template t, Time timeout) |

Table 6.3: Mapping Ontology operations of Resource Manager API and TS API