



D3.2 v0.1. WSMO Use Case Modeling and Testing

WSMO Working Draft 17 May 2004

This version:

<http://www.wsmo.org/2004/d3/d3.2/v0.1/20040517/>

Latest version:

<http://www.wsmo.org/2004/d3/d3.2/v0.1/>

Previous version:

<http://www.wsmo.org/2004/d3/d3.2/v0.1/20040430/>

Editors:

Michael Stollberg

Authors:

Michael Stollberg
Michael Breu
Holger Lausen
Uwe Keller
Rubén Lara
Axel Polleres

This document is also available in non-normative [PDF](#) version.

Copyright © 2004 [DERI](#)®, All Rights Reserved. [DERI](#) liability, trademark, document use, and software licensing rules apply.

Abstract

This document exemplifies the usage of the Web Service Modeling Ontology WSMO for modeling possible Web Service driven applications. The intent of this document is to exemplify use cases with usage scenarios of Semantic Web Services on the one hand, and on the other to showcase modeling with WSMO as an evaluation with real-world testing as support for recursive development of WSMO. For use case modeling, we stick to the latest final working draft of [Web Service Modeling Ontology WSMO, Version 0.2](#).

Related Documents

WSMO Standard: [D2 v0.2 Web Service Modeling Ontology \(WSMO\)](http://www.wsmo.org/2004/d2/v0.3/) , current version at: <http://www.wsmo.org/2004/d2/v0.3/>

WSMO Primer: [D3.1 v01. WSMO Primer](#)

WSMO Reasoning: [D5.1 v01. Inferencing Support for Semantic Web Services: Proof Obligations.](#)

Table of contents

1. Introduction

[1.1 Semantic Web Services](#)

[1.2 The Web Service Modeling Ontology WSMO](#)

2. Use Cases

[2.1 B2C - Virtual Travel Agency](#)

[2.1.1 Description](#)

[2.1.2 Scope](#)

[2.1.3 Actors, Roles and Goals](#)

[2.4 Usage Scenarios](#)

[2.1.5 System Architecture](#)

[2.2 B2B - Integration with Semantic Web Services](#)

[2.2.1 Description](#)

[2.2.2 Scope](#)

[2.2.3 Actors, Roles and Goals](#)

[2.2.4 Usage Scenarios](#)

[2.2.5 System Architecture](#)

3. WSMO Use Case Modeling

[3.1.VTA for International Online Train Ticket](#)

[3.1.1 Requirements Analysis](#)

[3.1.2 WSMO Modeling](#)

[Ontologies](#)

[Goal](#)

[Web Services](#)

[Mediators](#)

[3.1.3 SWS mechanisms based on WSMO models](#)

[3.1.4 Conclusions](#)

[3.2 B2B Integration with Semantic Web Services](#)

[3.2.1 Requirements Analysis](#)

[3.2.2 WSMO Modeling](#)

[Ontologies](#)

[Goal](#)

[Web Services](#)

[Mediators](#)

[3.2.3 SWS mechanisms based on WSMO models](#)

[3.2.4 Conclusions](#)

4. Conclusions and Future Work

References

Acknowledgements

Appendix A: Flora2-F-logic for the VTA - Use Case

1. Introduction

This document exemplifies the usage of the Web Service Modeling Ontology WSMO for describing relevant aspects for Semantic Web Services. Therefore, we describe possible use cases of Semantic Web Services and showcase how these can be modeled with WSMO, especially for support of the Semantic Web Service usage scenarios in particular use cases. We briefly replicate the objectives and the approach of WSMO and outline use cases within possible usage scenarios of Semantic Web Services. Then, we showcase how specific use cases can be modeled in WSMO along with explanations on the modeling decisions. Besides, we provide the WSMO models in a computational format.

This Deliverable is intended to evolve in accordance to the ongoing development of the WSMO project, serving as a testing environment and providing input for a recursive, real world testing development of WSMO. In the longer run, additional use cases will be added in order to widen possible solutions for Semantic Web Service technologies around WSMO.

This document is organized as follows: the remainder of [Section 1](#) replicates the objectives and approach of WSMO; [Section 2](#) discusses possible application areas of Semantic Web Services. [Section 3](#) provides the modeling of the use cases in WSMO, pointing out the WSMO approach for Semantic Web Service technologies. [Section 4](#) concludes the document. The complete WSMO models as computational resources are provided in the [Appendices](#).

The use case modeling in this document relies on the latest final working draft of the [Web Service Modeling Ontology WSMO, Version 0.2](#).

1.1. Semantic Web Services

A Web Service is a piece of software accessible via the Web. Current Web Service technologies allow exchange of messages between Web Services [[SOAP](#)], describing the technical interface [[WSDL](#)], and advertising a Web Services in a registry [[UDDI](#)]. These technologies do not provide any information about the meaning of information used, neither do they explicitly describe the functionality of a services as needed for automated usage of Web Services. There are also an alternative and complement specifications such as for example [[ebXML](#)], which are still perceived to have much potential for widespread adoption as Web Services. ebXML has become the global electronic business specification, that defines a framework for global electronic business. Enhanced Web Service technologies aim at more sophisticated techniques to describe Web Services, emphasizing the concept of Semantic Web Services. In our understanding, a Semantic Web Service is defined as a “self-contained, self-describing, semantically marked-up software resources that

can be published, discovered, composed and executed across the Web in a task driven automatic way” [Arroyo et al., 2004]. By machine-processable descriptions of the relevant information and by means of automated mechanisms that utilize this information, the following functionalities for Web Services shall be achieved.

- **Automatic Web Service Discovery:** finding Web Services that abide to a service requester's specification of a desired functionality
- **Automatic Web Service Composition:** assembly of services based on its functional specifications in order to achieve a given task and provide a higher order of functionality
- **Automatic Web Service Execution:** invocation of a concrete set of services, arranged in a particular way following programmatic conventions that realizes a given task.

1.2 The Web Service Modeling Ontology WSMO

The aim of WSMO and its surrounding efforts is to define a coherent technology for Semantic Web Services (short: SWS). WSMO defines the modeling elements for describing several aspects of Semantic Web Services. The conceptual basis of WSMO is the Web Service Modeling Framework [WSMF], wherein four main components are defined that are needed for a full coverage framework for Semantic Web Services (see Figure 1). The first component is Ontologies which provide the formal semantics of the information used by all other components. The second component is Goals that specify objectives that a client may have when he consults a web service. The third component is Web Services. For supporting automated discovery, composition, and execution of Web Services, descriptions are required on the functionality provided by a Web Service (called “Capability” in WSMO). For supporting automated choreography and execution compensation of Web Services, particular information on the external visible behavior of a Web Service are needed (called “Interface” in WSMO), including information on the technical accessibility and the actual message exchange of Services. The fourth component of WSMO is Mediators, which are used as connectors between particular components and include possibly required mediation facilities needed to make connected components interoperable. WSMO distinguishes different types of Mediators. The components of WSMO along with exhaustive explanations are presented in the [WSMO Primer](#).

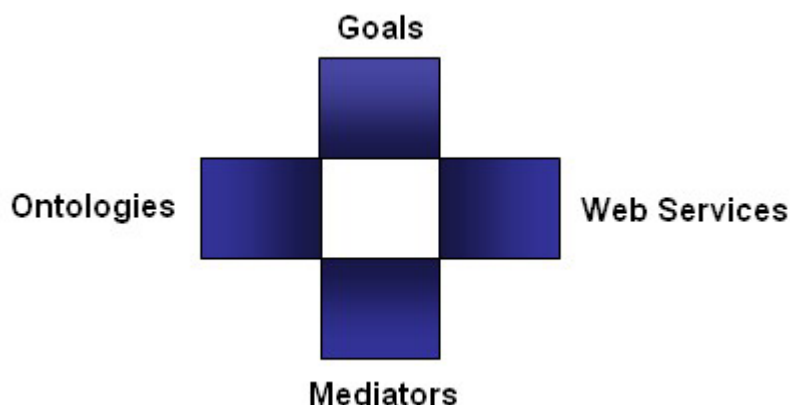


Figure 1. WSMO Components

2. Use Cases

Semantic Web Services can be used in manifold application fields. In accordance with the use cases defined in [Web Services Architecture Usage Scenarios](#) by the [W3C Web Services Architecture Working Group](#), we discuss two most common use case scenarios to exemplify the usage of SWS technologies:

1. A "Virtual Traveling Agency" that provides end-user services for e-Tourism by aggregating Web Services of different tourism service providers. This is a "B2C" use case, i.e. wherein a third party provides a service to end users as a Client / Service model as an aggregation of Semantic Web Services.
2. The second example is concerned with B2B Integration wherein a business entity, e.g. a business document, is exchanged between enterprises. Therein, different aspects of EAI might arise which shall be handled by Semantic Web Services technology.

For describing the use cases, we slightly modify the methodology of the W3C Use Case descriptions and extend by the requirements arising for Semantic Web Services technologies. The following lists the aspects we use for the use case definitions below.

- **Description:** describes the overall scenario
- **Scope:** defines the scope of the application scenario described
- **Actors, Roles and Goals:** identifies the actors in the scenario, their roles (i.e. what they do in the scenario) and their goals (i.e. what they want to achieve by participating in the scenario).
- **Usage Scenarios:** the W3C Service Architecture Working Group defines a [use case](#) as "... a sequence of interactions between a service requestor and one or more services, which achieve measurable results for the requestor", and a [usage scenario](#) as "... an atomic step in a path through a use case", i.e. an activity that has to be performed during execution of the use case and which can be automated by appropriate Semantic Web Service technologies. For each use case we describe the particular usage scenario by the following information:
 - participating actors and their goals
 - activities to be performed
 - technological requirements for this
 - and possible extensions of the scenario.
- **System Architecture:** In addition to the use-case oriented aspects of the W3C methodology, we also outline the general requirements and possible architecture of the respective SWS-based application.

2.1 B2C - Virtual Travel Agency

In [Web Services Architecture Usage Scenarios](#), the travel agency use case is separated into two use cases - one with static discovery and one with automated discovery. With Semantic Web Services we clearly want to support automated discovery, thus we restrict the first WSMO use case to a Virtual Travel Agency scenario that supports automated discovery of Web Services.

2.1.1 Description

Imagine a “Virtual Traveling Agency”, called VTA for short, that is an end user platform providing eTourism services to customers. These services can cover all kind of information services concerned with tourism information - from information about events and sights in an area to services that support booking of flights, hotels, rental cars, etc. online. Such VTAs are already existent, but as this point of time they mostly are an information portal along with some web-based customer services (e.g. eTourism.at). By applying Semantic Web Services, a VTA will invoke Web Services provided by several eTourism suppliers and aggregate them into new customer services. Such VTAs will provide automated eTourism services to end users, thus tremendously enhancing the functionality of currently existing VTAs.

The overview of the use case for VTAs that aggregate Web Services of different tourism service providers looks like this: a customer uses the VTA as the entry point for his request. This request must fit to an end-user service that the VTA provides. These end-user services are aggregated by the VTA by invoking and combining Web Services offered by several tourism service providers. Therefore, there must be some kind of contract between the service providers and the VTA for regulating usage and allowance of the Web Services. Figure 2 shows this overview (modified and extended from [W3C Travel Agent Use Case overview](#)).

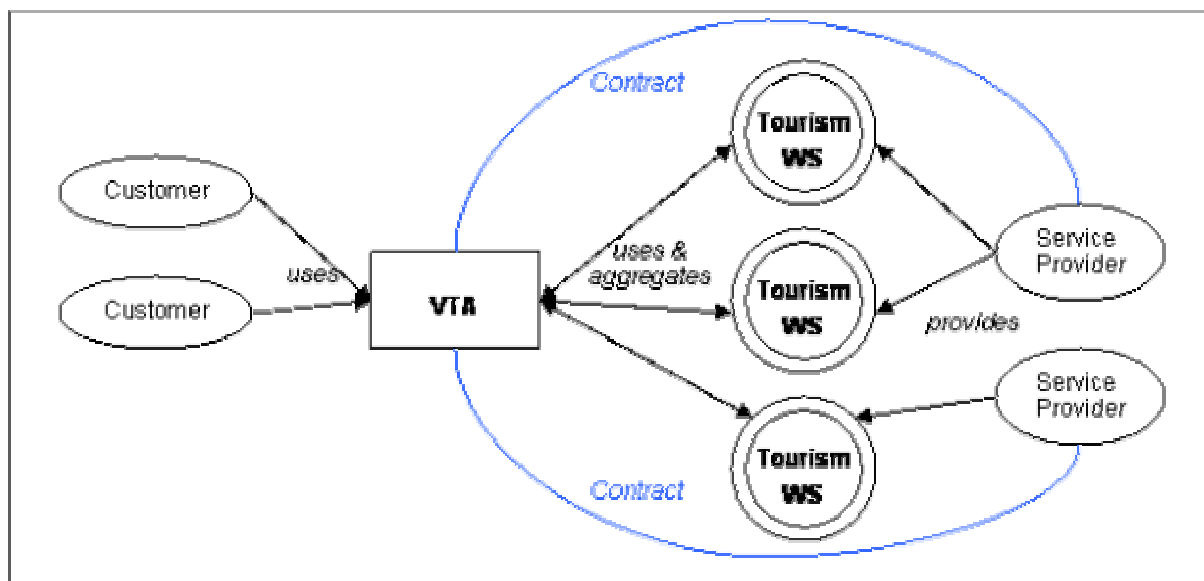


Figure 2. Use Case Overview: Virtual Travel Agency based on Semantic Web Services

2.1.2 Scope

The overview described above can be seen as a general structure for VTAs that can be extended to more complex scenarios wherein the customer can be a Web Service itself, thus creating a network of composed services that offer complex tourism services. For example, one VTA can provide flight booking services for an airline union, another VTA aggregates booking service for a worldwide hotel chain, and a third VTA provides booking services for rental cars by combining the services of several worldwide operating car rental agencies. Then, another VTA uses these VTA-

services for providing an end-user service for booking complete holiday trips worldwide.

In order to showcase and test the applicability of WSMO and not to get lost in real-world modeling of eTourism use cases, we restrict ourselves to a simple VTA use case from booking international online train tickets. This use case is described in more detail in section [3.1.VTA for International Online Train Ticket](#)

2.1.3 Actors, Roles and Goals

In general use case there are 3 actors. The following defines what they are, why they participate in this use case (goal), and with whom they need to interact in what way (role).

1. **Customer:** the end-user that requests a end-user service provided by the VTA
 - *Goal:* automated resolution of the request by a user-friendly tourism service
 - *Role:* end-user, interacts with VTA for service usage, payment, and non-computational assets (e.g. receiving the actual ticket when booking a trip)
2. **Tourism Service Provider:** a commercial company that provides specific tourism services
 - *Goal:* sell service to end customers, maximize profit as a commercial company
 - *Role:* provides tourism service as a Web Service (also provides the necessary semantic descriptions of the Web Services), has a usage and allowance contract with the VTA
3. **VTA:** the intermediate between the Customer and the Tourism Service Provider. Provides high-quality tourism services to customers by aggregating the separate services provided by the Service Providers.
 - *Goal:* provide high-quality end-user tourism services, use existing tourism services and aggregate them into new services, maximize profit as a commercial company / represent union of service providers (depending on the owners of the VTA).
 - *Role:* interacting with customer via user interface (can be web-based for human customers or and Interface / API for machine-users), usage and allowance contract for Web Services offered by Service Providers, centrally holding all functionalities for handling Semantic Web Services (mechanisms for discovery, composition, execution, etc.)

2.1.4 Usage Scenarios

We identify the following usage scenarios

1. *VTA interacts with Service Providers on contract and Web Service usage and allowance*
 - **Participating Actors:** VTA and Service Providers
 - **Activities:** business contract negotiation
 - **Technological Requirements:** contract information is displayed in system, i.e. Web Service usage is implemented via Policies
 - **Possible Extensions:** contract negotiation can be supported by automated mechanisms

2. *Customer requests VTA for searching tourism service offers, VTA detects suitable Web Services for searching tourism service offers and displays results to Customer*
 - **Participating Actors:** Customer and VTA
 - **Activities:**
 - (1) Customer selects "Search" services as provided by the VTA
 - (2) VTA discovers, invokes and executes corresponding Web Services
 - **Technological Requirements:**
 - (1) VTA has to pre-define a "Search" functionality that can be requested by a Customer
 - (2) Web Services must be semantically described in order to support dynamic discovery (assuming that single Web Services can perform the search functionality)
 - (3) VTA has to hold mechanisms for automated Service Discovery
 - **Possible Extensions:**
 - o the Customer specifies his request in natural language and the requested VTA-service is detected automatically
 - o several Web Services are aggregated for functionality
3. *Customer selects a concrete offer and requests booking for this offer (interacting with the VTA), VTA detects and aggregates Web Services for booking (incl. booking, payment, etc.), displays result to Customer and handles complete execution of customer-interaction (computational part)*
 - **Participating Actors:** Customer and VTA
 - **Activities:**
 - (1) Customer selects one concrete offer out of the Search results of usage scenario 2
 - (2) VTA discovers and composes available Web Services from Service Providers for
 - (3) VTA executes the Web Services in the sequence determined, controls the execution (handles errors and detects alternative paths if a Web Service fails)
 - (4) VTA interacts with Customer during execution when further information is needed (e.g. a creditcard number for payment)
 - **Technological Requirements:** contract information is displayed in system, i.e. Web Service usage is implemented via Policies
 - (1) Web Services must be semantically described in order to support dynamic discovery, composition, and execution
 - (2) VTA has to hold mechanisms for automated Service Discovery, Composition, and Execution
 - (3) VTA has to provide an interaction interface for contingent Customer-interaction during Service execution
 - **Possible Extensions:** advanced mechanisms for automated execution of aggregated Web Services
4. *VTA interacts with Customer and Service Provider for non-computational parts (e.g. delivery of actual tickets)*
 - **Participating Actors:** Customer, VTA
 - **Activities:** customer notification, accounting, good delivery (out of computational system), etc.
 - **Technological Requirements:** mechanisms for notification and accounting
 - **Possible Extensions:** Web Services can be used for:
 - o customer notification

- VTA-Service Provider interaction on accounting and good delivery mandate

2.1.5 System Architecture

In this use case, the VTA is the central point of interaction between the Customer and Web Services. Regarding the technological requirements, it gets obvious from the Usage Scenario descriptions that (1) the Web Services offered by the Service Providers have to carry sufficient descriptive information to support automated Web Service usage, and (2) that the VTA has to hold all mechanisms to handle Semantic Web Services. The basic architecture of such a VTA as a central entity for Semantic Web Services handling is shown in Figure 3. The essential functionalities of Semantic Web Service enabled VTAs – with special regard to the requirements for Semantic Web Service technologies – are:

- It has to provide a user interface for customer interaction (for both human and machine users)
- It has to hold generic end-user services that users can “instantiate”
- It has to discover suitable Web Services for an “instantiated” user request
- It has to invoke and combine external Semantic Web Services
- It has to provide a Web Service Execution Environment with control functions, error handling, and support for optional user interaction
- It has to have to deal with properly heterogeneous resources, thus holding appropriate mediation facilities.
- It has to provide an Interface for cooperation with Service Providers.

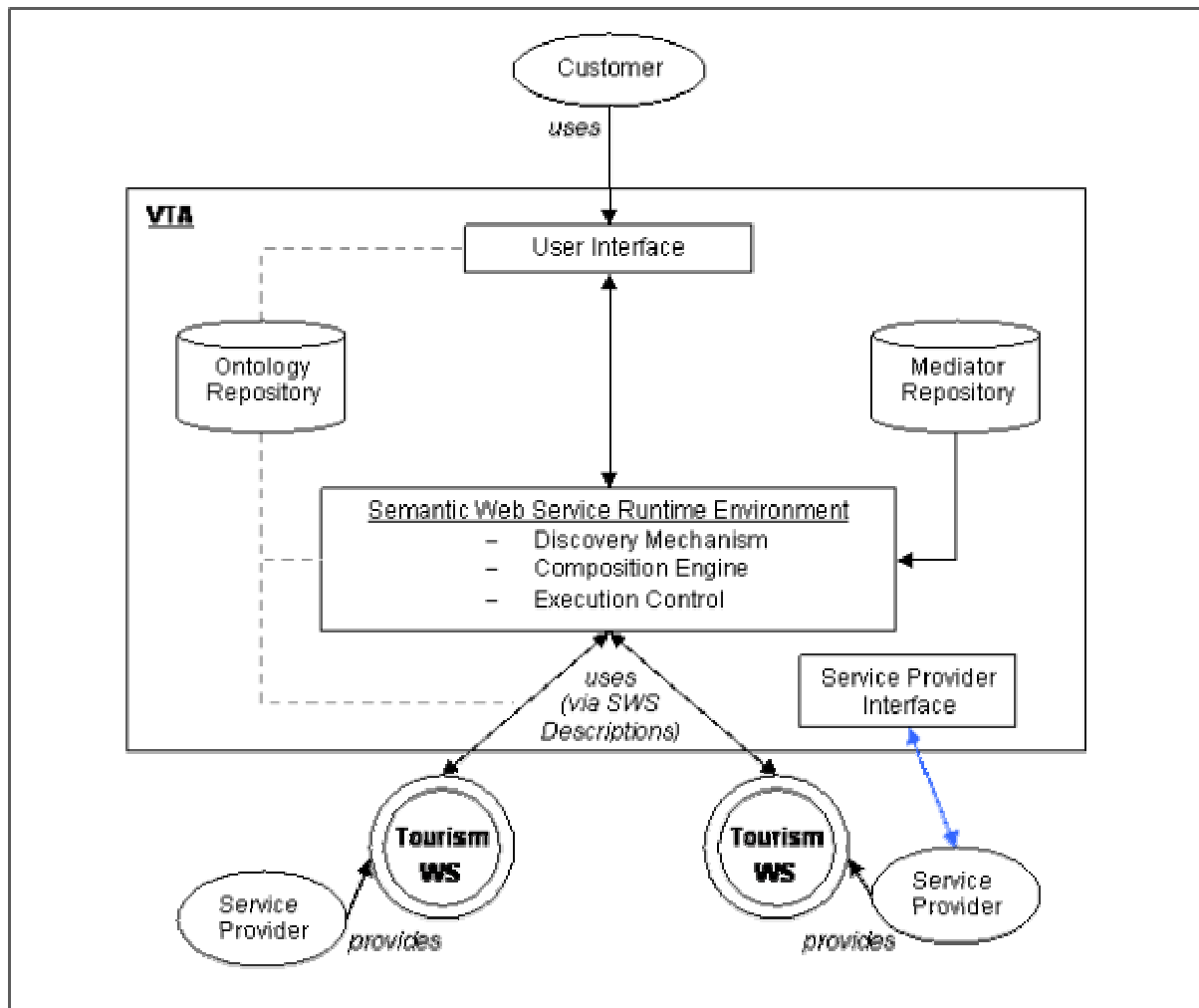


Figure 3. General Architecture of a SWS-enabled VTA

Summarizing, the VTA is a SWS-enabled B2C application that provides an end-user service following a C/S Model. In order to support coherent functionality of the VTA and ensure that the descriptions of Web Services are compatible to this, an overall framework for SWS technologies is needed. This is provided by WSMO. [Section 3.1](#) exemplifies the modeling of the WSMO components for a real world VTA use case in detail.

2.2 B2B - Integration with Semantic Web Services

The second use case is concerned with the integration of possible heterogeneous resources in B2B settings which is considered as one of the most important application fields of the Web Service technology.

2.2.1 Description

In the B2B use case, two enterprises called E1 and E2 want electronically exchange business documents across the network. It is assumed that partners may not know each other before carrying business transaction and that is why contract negotiation and contract agreement are essential elements of this use case. It is assumed that each of the partners expose a set of web services with the given capabilities, which

can handle conversation using any possible known B2B protocols (e.g. ebXML [ebXML], RosettaNet [RosettaNet] etc.). The contract agreement defines roles of enterprises in the conversation e.g. one of the enterprise E1 becomes the seller and the second enterprise E2 becomes the buyer. Agreement also predefines the order of the message exchange pattern e.g. buyer first sends purchase order (PO) and after that it receives purchase order acknowledgement (POA). Differently than in the previous B2C use case, where the client/server model of interactions has been adopted, the peer-to-peer model is used in this use case - partners are equal and they carry the conversation. Each of the companies has own orchestration and the set of web services, which enables to exchange business documents electronically. Infrastructure provided by SWS takes care for any necessary mediation between web services (links web services), ontologies (resolves possible representation mismatches between ontologies used by these two enterprises), goals (links goals) and web services and goals. SWS infrastructure supports the execution of the contract to fulfill approved agreement.

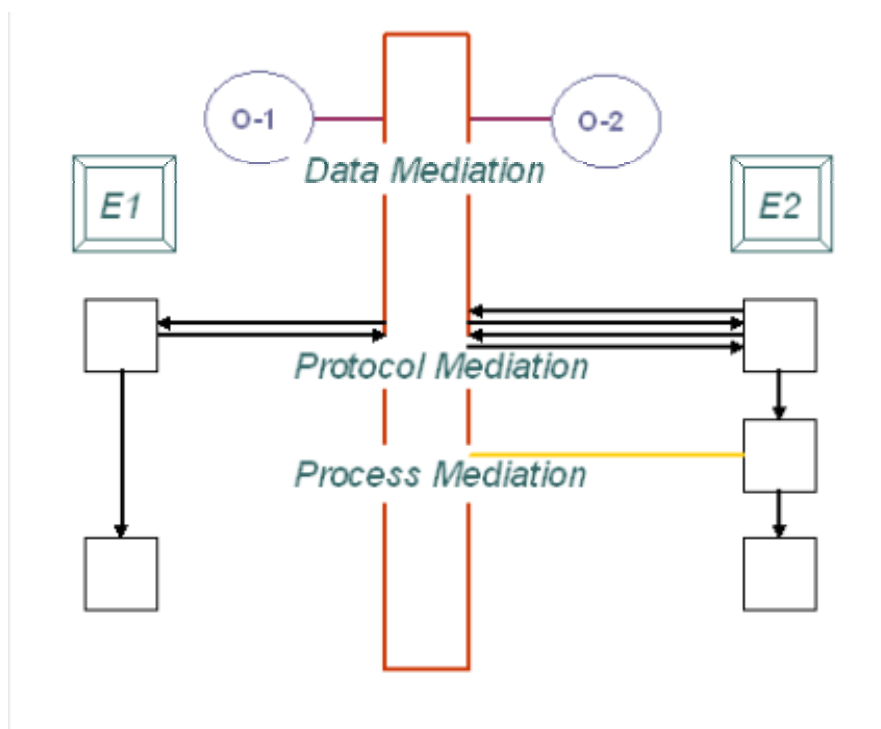


Figure 4. B2B Integration with Semantic Web Services

In this use case an ultimate goal of an enterprise E1 is to integrate its own back-end system with the back-end system of an enterprise E2. Once integrated, SWS software enables back-end systems of both companies to interact and to preserve the message, process and protocol semantic. The information systems used by enterprises E1 and E2 are **autonomous**, **heterogenous** and **distributed**. Semantic Web Services address each of these three properties and the software based on SWS enables companies to cooperate.

The back-end systems in E1 and E2 are **autonomous** since each of them changes its state without informing other system about it. SMS software enables to track state changes of back-end applications to facilitate coordination between systems of E1 and E2.

The back-end systems in E1 and E2 are **heterogonous**, because each of them has different conceptual model for expressing business semantics. SWS software takes care of appropriate mediation of the representation and meaning of the back-end system to the equivalent representation and meaning of the other system. The SWS software ensures to maintain the same semantics between back-end systems of E1 and E2.

The back-end systems in E1 and E2 are **distributed** because each of them maintains its own state independently from the other system. Back-end applications in companies E1 and E2 do not share data or state at all. SWS software implemented in both companies takes care of transporting data between the systems.

2.2.2 Scope

The use case assumes peer-to-peer relationships between two business partners carrying conversation about purchasing/selling of goods. The B2B use case focuses on the technical infrastructure based on the SWS technology, which enable any business company to automatically discover web services which are capable to fulfill its goals, compose simple web services into complex web services to achieve a given goal and to automatically execute given services in a particular order. This use case assumes that there may be no prior business relationships between two enterprises before the discovery. Enterprise E1 must find enterprise E2 and they must agree and enforce the contract in their companies. Agreement should define roles of each of them in the agreed business process – e.g. one of them would become a buyer and one of them would become a seller. The agreement can lead to only one time execution of the agreed business process (e.g. request purchase order) or to long time relationships based on the multiply execution of the agreed contract. Payments are sent through financial institutions and at this stage they are out of the scope of this use case. The same situation concerns the shipment of the goods. This use case consider sending documents as for example purchase orders or invoices, but the physical shipment of goods is out of the scope of this use case.

2.2.3 Actors, Roles and Goals

There are two actors in the B2B use case – actors, which represent two business entities. The size and the importance of companies are not predefined in this use case. They might differ in size but from the perspective of this use case it should not matter which one of them is a more dominant partner. Both of the enterprises undertake a predefined role in the use case. These are:

1. **Buyer:** the company, which initiates the use case by searching for a partner, which is capable to sell goods.
 - *Goal:* Finding a business partner, who is capable to provide goods. Signing the contract, discovering capabilities of the seller, composing provided web services and executing them.
 - *Role:* A business entity, which seeks business partner to achieve given goal by establishing new business relationships. Once the contract is signed it must be executed and as the result of contract execution, the buyer should receive goods. Buyer initiates the process described in this use case.
2. **Seller** - seller provides goods. It waits for buyers, responds to their requests, signs the contract and ships goods.

- *Goal*: Providing goods. Signing the contract, discovering capabilities of the buyer, composing provided web services and executing them.
- *Role*: A business entity, which waits for the partner to establish business relationships. As the result of the execution of the contract, the seller should send goods the seller.

2.2.4 Usage Scenarios

In this use case the following usage scenarios have been identified:

1. *Contract negotiation and implementation of agreement between buyer and seller.*
 - **Participating actors** - buyer and seller
 - **Activities** - business contract negotiation and implementation
 - **Technological Requirements** - The technology should enable matching goals of a buyer with capabilities of a seller. But matching goals of capabilities is not sufficient, because once goal is matched with the capability, the interfaces of two businesses should be matched as well.
 - **Possible Extensions** -contract is negotiated and implemented completely automatically by appropriate infrastructure
2. *Typical business messages exchange (e.g. PO & POA exchange);*
 - **Participating actors** - buyer and seller
 - **Activities** - buyer sends PO to seller. Buyer can at any time check the status of processed order. Seller sends back POA. Lower level acknowledgments messages for each of the PO and POA can be also exchanged.
 - **Technological Requirements** - The technology should enable conversation between business partners, which supports different process models to achieve given task e.g. buying a product. For example system of one business partner might require a synchronized confirmation for each business document send out, while the system of the other business partner assumes that once the document is send, it does not have to be confirmed. SWS platform should provide appropriate process mediation mechanism to resolve this issue.
 - **Possible Extensions** - System of one of the business partner might failed and drop in the middle of conversation (e.g. it receives PO, but never sends a POA). SWS platform, similarly to workflow engines, takes care to recover from deadlock and livelock errors.
3. *SWS infrastructure crashes - once it recovers, it reliable commence its operations*
 - **Participating actors** - buyer and seller
 - **Activities** - Because of some internal (e.g. lack of power supply for the server) or external (e.g. lack of network connection) failure, the SWS system becomes temporary unavailable. Once it is back online, it commence from the point where the execution has been dropped. None of the messages are lost, none of the processes are executed from the beginning.
 - **Technological Requirements** - Reliable and event driven architecture.
 - **Possible Extensions** - SWS infrastructure informs all interested parties that it is back online.
4. *E1 and E2 want to deploy a new integration definition type (described in WSML). The developer responsible for the SWS software writes a new integration type, which is next deployed by SWS infrastructures in both enterprises.*

- **Participating actors** - buyer and seller
- **Activities** - Any new integration type can be compiled and deployed by SWS infrastructure.
- **Technological Requirements** - Standard interface which allows carrying conversation between SWS infrastructure and WSML editor. New integration definition types can be saved and retrieved from the system.
- **Possible Extensions** - Public interface which enables any external party to provide own definitions.

2.2.5 System Architecture

Web Services Modeling Execution (WSMX) is the infrastructure hosted by each of the enterprises to support services following a peer-to-peer model. WSMX is software implementation of a web service execution environment supporting the development, management and execution of Semantic Web enabled Web Services. WSMX platform does not differentiate between calls coming from the back-end application systems (intra-company information systems) and from the information systems of other enterprises. WSMX can also communicate directly with other WSMX platforms hosted by other enterprises as shown on figure 5.

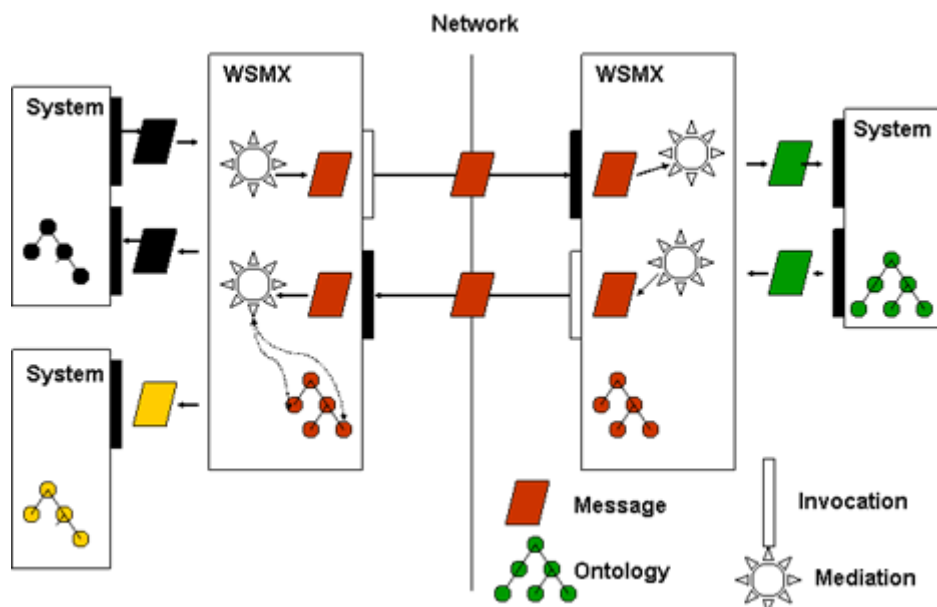


Figure 5. B2B Use Case System Architecture

3. WSMO Use Case Modeling

In the following we model the two use cases described above in WSMO in order to explicate the practical usage and the design of WSMO. We apply the following methodology for this:

1. describing the use case, especially the functionality of the Semantic Web Services handling component of the system
2. define a list of requirements on what has to be known to the application in order to make SWS applicable

3. model the use case in WSMO, including all building blocks of WSMO, along with detailed explanations of the modeling and the building blocks of WSMO
4. all WSMO models as downloadable computational resources for testing of SWS-technologies developed on WSMO.

For step number 3, the models of the respective WSMO components are described in Listings in this document. Therein, the Listings specify the different WSMO components as conceptual models in accordance to the specifications in [WSMO Standard, V0.2](#). These models have to be transformed for usage within a specific technology, with respect to syntax and technological constructs. For step 4, we provide the models as computational resources for download, testing and development in the [Appendices](#). Currently, we support FLORA-2 as a F-Logic reasoner (see [FLORA-2 homepage](#)). The models provided for download in the Appendices are runnable as separate FLORA-2-programms; the structure and the connections of these programs as WSMO-components have to be defined by the application developer (e.g. transforming the "usedMediators"-constructs as well as namespaces in the respective technology support by the tool). Besides, the identifier of every WSMO element is an URI. In the use case, the URL "http://www.wsmo.org/2004/d3/d3.2/v0.1/20040419/resources/FILE" is the general ID for all models of the use case, wherein FILE is replaced by the actual file name of the component as linked to in the Appendices. The elements of specific components have then the identifier ".../resources/FILE#element", which make the identifier-concept of WSMO conformant to the concept of URIs in web technologies. With regard to readability of the listings, we only specify the file name as the component identifier in the Listings.

For modeling of the WSMO components in the Listings in this document we use [WSML-U](#) as a human readable syntax, based on the syntax for F-Logic as defined in [\[Kifer et al., 1995\]](#).

3.1 VTA for International Online Train Tickets

According to the general VTA use case described in Section [2.1 B2C - Virtual Travel Agency](#) we define the following use case here:

- A customer wants to book an online ticket for an international train connection, more precisely from Innsbruck in Austria to Frankfurt in Germany.
- A VTA provides a search service for international train connections in Europe (here: only Austria and Germany) and an aggregated service for booking the tickets for international train connections in Europe online. (At design time of this use case the national train operators of Austria and Germany only provided online ticket booking facilities for national train connections; so we assume that the VTA has to compose the online ticket booking services from the Austrian and the German train operators.)
- Both the Austrian national train operator "ÖBB" as well as the German national train operator "DB" provides Web Services for searching international train connections and for booking tickets for national train connections online. (These services exists as conventional Internet Services, see Figure 6; here we assume that they are provided as Web Services).

The course of the use case is the following:

- the customer poses a request for an international train connection from Innsbruck to Frankfurt on 23rd May 2004, at 16.00 local time
- the VTA returns a set of possible connections
- the user selects one of these connections and poses a request for booking the ticket online
- the VTA combines the online train ticket booking services from ÖBB and DB, executes the booking and payment process, and sends the online ticket per email to the Customer.

For the aggregated service, the VTA has to determine the itineraries of the international connections, and to split them at the border stations into national itineraries. The VTA has to mediate between the following web services:

- The timetable service, providing timetable information and itineraries on national and international connections.
- The national ticketing services, providing online tickets for national itineraries.

The rationale for choosing this use case is that it showcases a possible VTA use case as described above within all the components identified in WSMO. The components are simple, thus this use case allows showing the modeling of WSMO elements without getting lost in complicated definitions of specific elements.

Connections

Your choices
 from: LINZ/DONAU Date: We, 25.02.04 Time: 09:00 (Departure)
 to: Frankfurt(Main)Hbf

Change query New timetable request New station query Return journey Continue journey
 Connection graphic Timetable Booklet PDA-timetable

Overview << Earlier | Later >>

Details	Station/Stop	Date	Time	Duration	Chg.	Products	Ticket
<input checked="" type="checkbox"/>	ÖBB Linz/Donau Hbf Frankfurt(Main)Hbf	24.02.04 25.02.04	dep 22:21 arr 06:02	7:41	0	CNL	Ticket by Post
<input checked="" type="checkbox"/>	ÖBB Linz/Donau Hbf Frankfurt(Main)Hbf	25.02.04	dep 10:23 arr 15:39	5:16	0	ICE	Ticket by Post
<input checked="" type="checkbox"/>	ÖBB Linz/Donau Hbf Frankfurt(Main)Hbf	25.02.04	dep 14:13 arr 19:41	5:28	0	EC	Ticket by Post

Print view Overview Detailed Journey view << Earlier | Later >>

Details for selection selected connections Show Show Show
 Details for all all connections Show Show Show

Journey guide

Station/Stop	Date	Arr.	Dep.	Platform	Products	Comments
Linz/Donau Hbf	24.02.04		22:21		CNL CNL 312	CityNightLine
Salzburg Hbf			23:44			Subject to compulsory reservation, Bicycles conveyed - subject to reservation, Number of bicycles conveyed limited, Sleeping-car, Couchettes, Sleeperette, BordRestaurant, CityNightLine
Mannheim Hbf	25.02.04	04:49				
Frankfurt(Main)Hbf		06:02		3		

Duration: 7:41; runs daily
 Ticket by Post

Hide intermediate stops Print view Further information Top of page

Journey guide

Station/Stop	Date	Arr.	Dep.	Platform	Products	Comments
Linz/Donau Hbf	25.02.04		10:23		ICE ICE 28	InterCityExpress
Passau Hbf		11:22	11:26			Part of journey with EC/IC, BordRestaurant
Plattling		11:57	11:59			
Regensburg Hbf		12:31	12:33			
Nürnberg Hbf		13:32	13:34			
Würzburg Hbf		14:26	14:28			
Aschaffenburg Hbf		15:08	15:10			
Frankfurt(Main)Hbf		15:39		6		

Figure 6. ÖBB Train Connection Itinerary Service

3.1.1 Functional Requirements

The following lists the requirements analysis for modeling the use case. For each of the components of WSMO, a list of requirements is defined that are needed in order to enable the requested functionality of the VTA for online search and booking of international train tickets, regarding the use case described above.

O1	We need ontological information on international train itineraries and on the purchasing process. This information should be kept in separated ontologies, following the design principle of modular ontology design.
O2	An itinerary is described by a Start- and End-Location, date and time of departure and arrival, the station where the border is crossed, and the fare.
O3	There has to be customer that buys a train ticket
O4	An itinerary describes a valid international train connection.
O5	There exists a concept that defines whether a location is located at the border between 2 countries
O6	A ticket is valid for exactly 1 itinerary
O7	A ticket is valid for exactly 1 customer
O8	A location description consists of a Location identifier, a Country identifier, and an indicator that the location has a train station
O9	The purchase ontology has to identify the buyer and seller roles, a product with a price, and valid payment methods
O10	The only valid payment method for online tickets is credit card payment
O11	Information on Date and Time should be defined generally in a separate ontology

G1	Booking a Online Train Ticket
G1.1	From Innsbruck to Frankfurt
G1.2	Start time: 17th May 2004, at 16.00 local time

W1	Each National Train Operator provides a Web Service that offers an international train connection timetable and national online ticket booking
W2.1	The international train connection timetable takes a start location and an end location and a departure date and returns a set of itineraries
W2.1.1	Exceptions are: - Service not available - Start or End Location does not exist
W2.2	The national online ticketing service takes an itinerary with start location and end location in the national country, a credit card number and returns a ticket for this itinerary.
W2.2.1	Exceptions are:

	<ul style="list-style-type: none"> - Service not available - Credit card not accepted
--	---

Table 4. Requirements Mediators	
M1	There exists a WG Mediator to link Goal G1 to a train connection timetable Web Service.
M2	An OO-Mediator has to integrate the two domain ontologies
M2.2	OO-Mediator: All WSMO components apply the integrated ontology
M3	a WW-Mediator mediates between the national online ticketing services.

3.1.2 WSMO Modeling

The following provides the modeling of the use case in WSMO with respect to the requirements determined above. The models are presented in the same structure as in the requirements analysis. As explained above, the Listings below specify conceptual models of the distinct WSMO components which have to be transformed into the technology provided by the tool used for reasoning on the ontologies. The full WSMO models as downloadable computational resources for this use case are provided in [Appendix A](#). The models apply the syntax defined in [section 3](#).

The use case modeling in this document relies on the [Web Service Modeling Ontology WSMO, Version 0.3](#). This version of WSMO does not provide elaborated description elements for all WSMO components, especially for Web Services a sophisticated specification of the WSMO modeling primitives only exists for Web Service Capabilities. As we can only apply such elaborated description structures for use case modeling, we restrict this version of the use case modeling to the WSMO description elements defined.

Ontologies

We define 3 domain ontologies that provide the terminology definitions for the use case. The first ontology "International Train Ticket" describes the domain of train tickets, the second ontology "Date and Time" defines a general model for specifying time and dates and relationships of them, and the third ontology "Purchase" describes generic elements of purchasing a product between a buyer and a seller.

We apply the following conventions in the Listings for ontology specifications (those which are not ontology-specific hold for all other WSMO component specifications as well):

- **Identifier:** As defined above, WSMO applies URIs as identifiers for all WSMO elements, thus being conform to web technologies. All URIs for this use have the URL "http://www.wsmo.org/2004/d3/d3.2/v0.1/20040419/resources/FILE" is the general ID for all models of the use case, wherein FILE is replaced by

the actual file name (without file type extension) of the component as linked to in the Appendices. The elements of specific components have then the identifier ".../resources/FILE#element". For the ease for readability, we only specify the FILE-name in the use case Listings. Moreover, we only specify the identifier for the top-level WSMO elements explicitly, as the identifiers for the sub-elements are implicitly defined.

- **"usedMediators"-element:** this is the general construct for connecting different WSMO components. According to specifications in WSMO Standards, only specific types of Mediators can be WSMO "imported" into a specific WSMO component. In general, a WSMO Mediator handles all integration and mediation issues required in between the connected resources. Especially OO Mediators, which allow to use an ontology as the terminological basis in a WSMO component, resolve all ontology integration aspects (including namespaces, resolution of heterogeneities, etc.). The targetComponent of a OO Mediator, i.e. the user of an ontology, receives his required information space, meaning all knowledge that is provided within the OO Mediator, and simply uses this without regard to the actually used ontology. The ontologies below are connected as the use constructs of each other. Therefore OO Mediators are defined. These, along with further explanations on the used Mediators in the use case, are defined in [Mediators](#) below.
- **Axiom Definitions:** all axioms in the Listings are written in FLORA-2 syntax, see above for syntactical conventions. The axioms are the same as provided in the executable resources in Appendix A.
- **Ontology Notions:** In the ontology specifications below, we only specify the corresponding WSMO elements for non-functional properties, usedMediators, and axiomDefinitions. For the other ontology notions the semantics of F-Logic are exactly the same as the ontology modelling primitives defined in WSMO Standard, thus we do not have to specify the correspondence to the WSMO Standard meta-ontology explicitly.

The "International Train Ticket" Ontology defines an itinerary and the surrounding concepts as defined in Listing 1. Additionally, a axiom is defined that checks the validity of the traveling dates (the start date / time has to be in the future and the arrival date / time as to be later than the starting date / time) as well as some instances needed in the further use case modeling. Listing 1 shows the F-Logic specification of the ontology.

Listing 1. Domain Ontology "International Train Ticket"

Ontology

International Train Connections Domain Ontology

non-functional Properties

title

International Train Connections Domain Ontology

creator

DERI International

subject

International, Train Itineraries, Ticket Booking

description

International Train Itineraries for Online Ticket

```

Booking
  publisher
    DERI International
  contributor
    Michael Stollberg, Rubén Lara, Holger Lausen, Axel
Polleres
  date
    20040419
  type
    domain ontology
  format
    text
  identifier

http://www.wsmo.org/2004/d3/d3.2/v0.1/20040517/resources/tc
.flr
  source

http://www.wsmo.org/2004/d3/d3.2/v0.1/20040517/resources/tc
.flr
  language
    English
  relation

http://www.wsmo.org/2004/d3/d3.2/v0.1/20040419/resources/dt
.flr
  coverage
    Europe
  rights
    DERI
  version
    1.3

usedMediators
  // OO Mediator for using the Date and Time Ontology
  VTA-OOM-trainConnection.wsml

CONCEPTS

// a 'location' is the general notion of all locations in a
country
location
  name oftype string
  locatedIn oftype set country

/* Axioms for location */
// 'locatedIn' is a transitive property. this allows to
specify only the city for a desired train connection
X ofclass location[locatedIn ofvalues Y] :-

```

```
X ofclass location[locatedIn ofvalues Z] and
Z ofclass location[locatedIn ofvalues Y].

// Integrity Constraint
invalid(X) :- X ofclass location[locatedIn ofvalues X].

//subclasses of location
station subclassof location

country subclassof location

city subclassof location

village subclassof location

// 'borderStation' defines a train station at the border
// 'borderToCounty' denotes all adjacent countries
borderStation subclassof station
  borderToCounty oftype set (country)

itinerary
  startLocation oftype station
  endLocation oftype station
  via oftype set station
  departure oftype dateAndTime
  arrival oftype dateAndTime

// Integrity Constraint: departure has to be before arrival
invalid(I) :-
  I ofclass itinerary[
    departure ofvalue X2 and
    arrival ofvalue X3]
  and after(X2,X3).

traveller
  name oftype string

// 'ticket' defines a ticket that relates an itinerary to a
customer
ticket subclassof product
  itinerary oftype itinerary
  traveller oftype traveller

// below, some instances are defined that are needed
throughout the use case modeling

germany subclassof country
  name ofvalue 'Germany'
```

```

austria subclassof country
  name ofvalue 'Austria'

innsbruckHbf subclassof station
  name ofvalue 'Innsbruck Hbf'
  locatedIn ofvalues innsbruck
innsbruck subclassof city
  name ofvalue 'Innsbruck'
  locatedIn ofvalues austria
boesby subclassof village
  name ofvalue 'Boesby'
  locatedIn ofvalues germany

frankfurtHbf subclassof station
  name ofvalue 'Frankfurt Hbf'
  locatedIn ofvalues germany
kufsteinHbf subclassof borderStation
  name ofvalue 'Kufstein Hbf'
  locatedIn ofvalues austria
  borderToCounty ofvalues germany
salzburgHbf subclassof borderStation
  name ofvalue 'Salzburg Hbf'
  locatedIn ofvalues austria
  borderToCounty ofvalues germany
freilassingBf subclassof borderStation
  name ofvalue 'Freilassing Bf'
  locatedIn ofvalues germany
  borderToCounty ofvalues austria
kiefersfeldenBf subclassof borderStation
  name ofvalue 'Kiefersfelden Bf'
  locatedIn ofvalues germany
  borderToCounty ofvalues austria

```

The "Date and Time" Ontology in Listing 2 defines models for dates (i.e. certain days) and time (i.e. definition of certain points in time). Further, it defines axioms that represent conventional aspects of date and time, like 'before' and 'after', etc. In the use case, this is needed to determine validity of train connections, e.g for ensuring that a ticket is not for an itinerary that is in the past. It also can be used generally for expressing dates and time and relationships between them. Listing 2 only displays the ontology schema and the algebra for date and time, while the downloadable file contains instances and queries for testing.

The main ontology taken into consideration for developing this representation in F-Logic is an entry sub-ontology of time, available at <http://www.isi.edu/~pan/damlltime/time-entry.owl>. This ontology uses abstract temporal concepts like instant, interval and event and uses the Gregorian calendar as representation (partly using own encoding and partly using XSD encoding). Axioms are defined in first order logic in the accompanying paper [Pan and Hobbs]; there also is a LISP version of these axioms available at <http://www.cs.rochester.edu/~ferguson/daml/daml-time-20030728.lisp>. Other ontologies like COBRA calendarclock ontology

(<http://daml.umbc.edu/ontologies/cobra/0.4/calendarclock>) are only a straight forward representation of the Gregorian calendar, without any abstraction of concepts and description of axioms. Widely used concrete representations for date and time are defined in ISO 8601 (Numeric representation of Dates and Time) and in the XML Schema Definition (<http://www.w3.org/TR/xmlschema-2/>), which is based on ISO 8601. The ontology defined in Listing 2 uses the Gregorian calendar with the representation based on the definition in <http://www.w3.org/TR/xmlschema-2/>.

Listing 2. Domain Ontology "Date and Time"

```

Ontology
  Date and Time Ontology

non-functional properties
  title
    Date and Time Ontology
  creator
    DERI International
  subject
    Date, Time, Date and Time Algebra
  description
    generic representation and algebra for date and time
  publisher
    DERI International
  contributor
    Holger Lausen, Axel Polleres, Rubén Lara,
  date
    20040517
  type
    domain ontology
  format
    text
  identifier

  http://www.wsmo.org/2004/d3/d3.2/v0.1/20040517/resources/dt
  .flr
  source

  http://www.wsmo.org/2004/d3/d3.2/v0.1/20040517/resources/dt
  .flr
  language
    English
  relation
    http://www.isi.edu/~pan/damlttime/time-entry.owl,
    http://daml.umbc.edu/ontologies/cobra/0.4/calendarclock,
    http://www.w3.org/TR/xmlschema-2/
  coverage
    general
  rights
    DERI
  version

```

1.15

Concepts and Integrity Constraints

```

// An instant represents a particular point in time
instant

// An interval represents a duration between 2 points in
time
interval
  start oftype instant
  end oftype instant

/* Axioms for Interval */
//computes if a interval X contains a second interval Y
contains(X and Y) :-
  X ofclass interval and Y ofclass interval and
  (before(X.start and Y.start) or equal(X.start and
Y.start)) and
  (after(X.end and Y.end) or equal(X.end and Y.end)).

//computes if a interval X contains a instant Y
contains(X and Y) :-
  X ofclass interval and Y ofclass instant and
  (before(X.start and Y) or equal(X.start and Y)) and
  (after(X.end and Y) or equal(X.end and Y)).

// class date and its representation according to the
Gregorian calendar"
date[
  dayOfMonth oftype dayOfMonth
  monthOfYear oftype monthOfYear
  year oftype year

// Integrity Constraints for date
invalid(X) :-
  X ofclass date and
  (invalid(X.dayOfMonth) or invlaid(X.monthOfYear) or
invalid(X.year)).

// day of a month is represented by an integer
dayOfMonth subclassof integer.

//integrity constraint for valid dayOfMonths:
invalid(X) :-
  X ofclass dayOfMonth and
  (X<0 or X>31).

//a year is represented by an integer
year subclassof integer.

```

```
//a monthOfYear is represented by an integer and has
additional properties
monthOfYear subclassof integer
  name oftype string
  daysAfterBeginOfYear oftype integer

//integrity constraint for valid monthOfYear:
invalid(X):-
  X ofclass monthOfYear and
  (X<0 or X>12).

// concrete month are defined as instances
1 ofclass monthOfYear
  daysAfterBeginOfYear ofvalue 31
  name ofvalue 'January'
2 ofclass monthOfYear
  daysAfterBeginOfYear ofvalue 59
  name ofvalue 'February'
3 ofclass monthOfYear
  daysAfterBeginOfYear ofvalue 90
  name ofvalue 'March'
4 ofclass monthOfYear
  daysAfterBeginOfYear ofvalue 120
  name ofvalue 'April'
5 ofclass monthOfYear
  daysAfterBeginOfYear ofvalue 151
  name ofvalue 'May'
6 ofclass monthOfYear
  daysAfterBeginOfYear ofvalue 181
  name ofvalue 'June'
7 ofclass monthOfYear
  daysAfterBeginOfYear ofvalue 212
  name ofvalue 'July'
8 ofclass monthOfYear
  daysAfterBeginOfYear ofvalue 243
  name ofvalue 'August'
9 ofclass monthOfYear
  daysAfterBeginOfYear ofvalue 273
  name ofvalue 'September'
10 ofclass monthOfYear
  daysAfterBeginOfYear ofvalue 304
  name ofvalue 'October'
11 ofclass monthOfYear
  daysAfterBeginOfYear ofvalue 334
  name ofvalue 'November'
12 ofclass monthOfYear
  daysAfterBeginOfYear ofvalue 365
  name ofvalue 'December'

// class time
time
```

```

hourOfDay oftype hourOfDay
minuteOfHour oftype minuteOfHour
secondOfMinute oftype secondOfMinute

// integrity constraint for valid time:
invalid(X) :-
    X ofclass time and
    (invalid(X.hourOfDay) or invalid(X.minuteOfHour) or
    invalid(X.secondOfMinute)).

// a secondOfMinute is represented by an integer
secondOfMinute subclassof integer.

// integrity constraint for valid secondOfMinute:
invalid(X) :-
    X ofclass secondOfMinute and
    (X<0 or X>59).

// a minuteOfHour is represented by an integer
minuteOfHour subclassof integer.

// integrity constraint for valid minuteOfHour:
invalid(X) :-
    X ofclass minuteOfHour and
    (X<0 or X>59).

// a hourOfDay is represented by an integer
hourOfDay subclassof integer.

// integrity constraint for valid hourOfDay:
invalid(X) :-
    X ofclass hourOfDay and
    (X<0 or X>23).

// class date and time and representing together a specific
point of time (instant)
dateAndTime subclassof instant
    date oftype date
    time oftype time

//integrity constraint for valid dateAndTimes:
invalid(X) :-
    X ofclass dateAndTime and
    (invalid(X.date) or invalid(X.time)).

/*****
    algebra for date and time
    *****/
// computes equality of a date
equal(X and Y) :-
    Y ofclass date and X ofclass date and

```

```

X.dayOfMonth = Y.dayOfMonth and
X.monthOfYear = Y.monthOfYear and
X.year = Y.year.

// computes if a given date X is before another date Y
before(X and Y) :-
  Y ofclass date and X ofclass date and
  ((X.dayOfMonth < Y.dayOfMonth and X.monthOfYear =
Y.monthOfYear and X.year = Y.year) or
  (X.monthOfYear < Y.monthOfYear and X.year = Y.year) or
  (X.year < Y.year)).

// computes if a given date X is after another date Y
after(X and Y) :-
  Y ofclass date and X ofclass date and
  ((X.dayOfMonth > Y.dayOfMonth and X.monthOfYear =
Y.monthOfYear and X.year = Y.year) or
  (X.monthOfYear > Y.monthOfYear and X.year = Y.year) or
  (X.year > Y.year)).

/* this is simplified and ignores leap years a proper
algorithm as e.g. found at
http://quasar.as.utexas.edu/BillInfo/JulianDatesG.html
calculates correctly for Gregorian Calendar dates
(after 1582 - at least for most countries, see
http://members.brabant.chello.nl/~h.reints/cal/whenjul2greg
.htm
for an exact dates per country (e.g. Yugoslavia changed
1919)) can be represented in f-logic as follows: */

julianDayNumber(X and JDN) :-
  X ofclass date and
  X.monthOfYear < 3 and
  Y is X.year - 1 and
  M is X.monthOfYear + 12 and
  D is X.dayOfMonth and
  A is truncate(Y and 100) and
  B is truncate(A and 4) and
  C is 2 - A + B and
  E is floor(365.25 * (Y + 4716)) and
  F is floor(30.6001 * (M + 1)) and
  JDN is C + D + E + F - 1524.

julianDayNumber(X and JDN) :-
  X ofclass date and
  X.monthOfYear > 2 and
  Y is X.year and
  M is X.monthOfYear and
  D is X.dayOfMonth and
  A is truncate(Y and 100) and
  B is truncate(A and 4) and
  C is 2 - A + B and

```

```

    E is floor(365.25 * (Y + 4716)) and
    F is floor(30.6001 * (M + 1)) and
    JDN is C + D + E + F - 1524.

/* however due to a bug in XSB (float number arithmetics
do not work proper) this could not be implemented
and a simplified version that aproximates the days after
Christ is used. */

daysAfterChrist(D and X) :-
    D ofclass date and Z is D.monthOfYear and
    X is (D.dayOfMonth + Z.daysAfterBeginOfYear +
(D.year*365)).

// the difference in days between 2 dates
daysBetween(D1 and D2 and X) :-
    D1 ofclass date and D2 ofclass date and
daysAfterChrist(D1 and DAC_D1) and
daysAfterChrist(D2 and DAC_D2) and
X is DAC_D1 - DAC_D2.

// computes if two given times are the same
equal(X and Y) :-
    X ofclass time and Y ofclass time and
X.secondOfMinute = Y.secondOfMinute and
X.minuteOfHour = Y.minuteOfHour and
X.hourOfDay = Y.hourOfDay.

// computes if a given time X is before another time Y
before(X and Y) :-
    X ofclass time and Y ofclass time and
((X.secondOfMinute < Y.secondOfMinute and X.minuteOfHour
= Y.minuteOfHour and X.hourOfDay = Y.hourOfDay) or
(X.minuteOfHour < Y.minuteOfHour and X.hourOfDay =
Y.hourOfDay) or
(X.hourOfDay < Y.hourOfDay)).

// computes if a given time X is after another time Y
after(X and Y) :-
    X ofclass time and Y ofclass time and
((X.secondOfMinute > Y.secondOfMinute and X.minuteOfhour
= Y.minuteOfhour and X.hourOfDay = Y.hourOfDay) or
(X.minuteOfhour > Y.minuteOfhour and X.hourOfDay =
Y.hourOfDay) or
(X.hourOfDay > Y.hourOfDay)).

// computes the amount of seconds from midnight
secondsFromMidnight(T and X) :-
    T ofclass time and
X is T.secondOfMinute + (T.minuteOfHour*60) +
(T.hourOfDay*60*60).

```

```

// the difference in seconds between 2 times
secondsBetween(T1 and T2 and X) :-
  T1 ofclass time and T2 ofclass time and
  secondsFromMidnight(T1 and SFM_T1) and
  secondsFromMidnight(T2 and SFM_T2) and
  X is SFM_T1 - SFM_T2.

// computes if Date and Time are equal
equal(X and Y) :-
  X ofclass dateAndTime and Y ofclass dateAndTime and
  equal(X.date and Y.date) and
  equal(X.time and Y.time).

// computes if a given date and time X is before another
date and time Y
before(X and Y) :-
  X ofclass dateAndTime and Y ofclass dateAndTime and
  ((equal(X.date and Y.date) and before(X.time and Y.time))
or
  before(X.date and Y.date)).

// computes if a given date and time X is after another
date and time Y
after(X and Y) :-
  X ofclass dateAndTime and Y ofclass dateAndTime and
  ((equal(X.date and Y.date) and after(X.time and Y.time))
or
  after(X.date and Y.date)).

// computes the difference in seconds between two different
DateAndTime
secondsBetween(D1 and D2 and X) :-
  D1 ofclass dateAndTime and D2 ofclass dateAndTime and
  daysAfterChrist(D1.date and DAC_D1) and
  daysAfterChrist(D2.date and DAC_D2) and
  secondsFromMidnight(D1.time and SFM_T1) and
  secondsFromMidnight(D2.time and SFM_T2) and
  X is SFM_T1 + DAC_D1 * 24 * 60 * 60 -
      (SFM_T2 + DAC_D2 * 24 * 60 * 60).

// the difference in (decimal) days between two different
DateAndTime
daysBetween(D1 and D2 and X) :-
  D1 ofclass dateAndTime and D2 ofclass dateAndTime and
  secondsBetween(D1 and D2 and Z) and
  X is Z/60/60/24.

//The current Date is defined here since we do not have
build in function yet

```

```

currentDate ofclass dateAndTime
date ofvalue CurrentDate ofclass date
  dayOfMonth ofvalue 28 and
  monthOfYear ofvalue 2 and
  year ofvalue 2001 and
time ofvalue CurrentTime ofclass time
  hourOfDay ofvalue 23 and
  minuteOfHour ofvalue 40 and
  secondOfMinute ofvalue 12

```

The "Purchase" ontology defines general concepts for purchasing a product (there is a buyer, a seller, a product with a price, a payment method, and delivery).

Listing 3. Domain Ontology "Purchase"

```

Ontology
  Purchase Domain Ontology

non-functional properties
  title
    Purchase Domain Ontology
  creator
    DERI International
  subject
    buyer, seller, product, price, payment method, delivery
  description
    general purchase ontology
  publisher
    DERI International
  contributor
    Michael Stollberg, Axel Polleres, Rubén Lara, Holger
    Lausen
  date
    20040517
  type
    domain ontology
  format
    text
  identifier

  http://www.wsmo.org/2004/d3/d3.2/v0.1/20040517/resources/po
  .flr
  source

  http://www.wsmo.org/2004/d3/d3.2/v0.1/20040517/resources/po
  .flr
  language
    English
  relation
  coverage
    general
  rights

```

DERI
version
1.7

CONCEPTS

address

name **oftype** string
street **oftype** string
number **oftype** integer
zipcode **oftype** string
city **oftype** city
state **oftype** state
country **oftype** country

country

name **oftype** string

location

name **oftype** string,
country **oftype** country

state

name **oftype** string
partOfCountry **oftype** country

buyer

shipTo **oftype** address
billTo **oftype** address
purchaseIntention **oftype set** (tradeItem)
hasPayment **oftype set** (paymentMethod)

seller

address **oftype** address
saleIntention **oftype set** (tradeItem)

tradeItem

product **oftype** product
price **oftype** price // Note that the item price given by
the buyer denotes a price *limit*

product

name **oftype** string

price

amount **oftype** real
currency **oftype** string

paymentMethod

name **oftype** string

```

// A trade is an actual agreement on trading items between
two partners.
trade
  items ofclass set (tradeItem)
  buyer oftype buyer
  seller oftype seller
  payment oftype paymentMethod

// Delivery of a good as an effect of a purchase
delivery
  products oftype set (product)
  receiver oftype buyer
  sender oftype seller

```

AXIOMS

```

/* purchasing only possible if
1) Only products can be traded/delivered if the Buyer
requires them (purchaseIntention),
   and if the seller provides them (saleIntention), and if
the selling price is not higher than the price intended by
the buyer: */

```

```

invalid(T) :- T ofclass trade[items ofvalues I] and tnot
T.seller[saleIntention ofvalues I].

```

```

priceok(I) :- T ofclass trade[items ofvalues I] and
T.buyer[purchaseIntention ofvalues I1] and
  I.product = I1.product and
  I.price.currency = I1.price.currency and
  I.price.amount =< I1.price.amount.

```

```

/* 2. Only products can be delivered if
the receiver requires them (purchaseIntention) and if
the sender provides them (saleIntention) */

```

```

invalid(D) :- D ofclass delivery[products ofvalues P] and
tnot D.receiver.purchaseIntention[product ofvalue P].

```

```

invalid(D) :- D ofclass delivery[products ofvalues P] and
D.receiver.saleIntention[product ofvalue P].

```

INSTANCES (needed throughout the use case modeling)

```

creditcard subclassof paymentMethod

```

```

creditcard
  name oftype string
  number oftype string
  expmonth oftype month
  expyear oftype year

```

```

type oftype string

cash ofclass paymentMethod
  currency oftype currency

currency

// some sample currencies
euro ofclass currency
usd ofclass currency
gbp ofclass currency

```

Goals

Goals denote what a user wants to receive when using a Web Service. More precisely, a Goal describes constraints of an object that the user desires. A Goal is modeled as a fact, i.e. it specifies the ontological structure of the object of desire without variables. Goals in WSMO are understood as "instantiated Goals", i.e. a concrete desire to be resolved. For WSMO-based applications, separation of Goal Schemas and Goal Instances seems to be an appropriate solution for handling of Goals. The former shall describe the general structure of a Goal, while a Goal Instances instantiate a Goal Schema by specifying certain attributes for the which is instantiated by a user for expressing a certain desire. Goal Schemas can also be thought of as pre-defined Goals, while Goal Instances are concrete requests for Web Services during runtime.

In the use case, we have one Goal: a user wants to buy a ticket online for a train connection from Innsbruck to Frankfurt on a certain date. The Goal Template states that the desire is to get a train ticket for an itinerary and for a customer, according to the knowledge defined in the ontologies. The Goal Instance specifies concrete values for the template structure. Listing 4 shows this Goal with the following elements:

- **postcondition:** A ticket for a train itinerary from Innsbruck to Frankfurt on May, 17th 2004, valid for the customer Dieter Fensel.
- **effect:** there shall be a trade for the ticket, to be payed with creditcard.

Listing 4: Goal - buying a train ticket online

```

Goal
  buying online train ticket

non-functional Properties
  title
    buying online train ticket
  creator
    DERI International
  subject
    Train Tickets, Online Ticket Booking
  description
    a desire for booking an international train ticket online
  publisher

```

```

    DERI International
    contributor
      Michael Stollberg, Rubén Lara, Holger Lausen, Axel
Polleres
    date
      20040517
    type
      WSMO Goal
    format
      text
    identifier

http://www.wsmo.org/2004/d3/d3.2/v0.1/20040517/resources/goal
.flr
    source

http://www.wsmo.org/2004/d3/d3.2/v0.1/20040517/resources/goal
.flr
    language
      English
    relation

http://www.wsmo.org/2004/d3/d3.2/v0.1/20040517/resources/tc.f
lr

http://www.wsmo.org/2004/d3/d3.2/v0.1/20040419/resources/dt.f
lr

http://www.wsmo.org/2004/d3/d3.2/v0.1/20040419/resources/po.f
lr
    coverage
      Europe
    rights
      DERI
    version
      1.14

usedMediators
  // OO Mediator for using the ontologies
  VTA-OOM-Goal1.wsml

// define an instance of Goal

mygoal ofclass goal.

// Goal Postcondition:
// a ticket for an itinerary from Innsbruck to Frankfurt

mygoal[postcondition ofvalue
  myTicket ofclass ticket[

```

```

    itinerary ofvalue exists itinerary ofclass itinerary[
      startLocation ofvalue innsbruckHbf
      endLocation ofvalue frankfurtHbf
      departure ofvalue exists dateandtime ofclass
dateAndTime[
  date ofvalue exists date ofclass date[
    dayOfMonth ofvalue 17
    monthOfYear ofvalue 5
    year ofvalue 2004
  ],
  time.hourOfDay ofvalue 18 ofclass hourOfDay
]
],
traveller ofvalue exists traveller ofclass traveller[
  name ofvalue 'Dieter Fensel'
]
].

// Goal Effect: a trade for the ticket, payed by creditcard

mygoal[effect ofvalue
mytrade ofclass trade[
  items ofvalues myTicket,
  buyer ofvalue exists buyer ofclass buyer[
    shipTo ofvalue myAddress
    billTo ofvalue myAddress
  ],
  payment ofvalue myCreditCard ofclass creditCard[
    name ofvalue 'Dieter Fensel'
    number ofvalue 1234567890
    expMonth ofvalue 9
    expYear ofvalue 2006
    type ofvalue 'MasterCard'
  ]
],
myAddress ofclass address[
  name ofvalue 'Dieter Fensel'
  street ofvalue 'Technikerstrasse'
  number ofvalue 13
  zipcode ofvalue 6020
  city ofvalue innsbruck
  state ofvalue tirol
  country ofvalue austria
]
].

```

Web Services

For our use case we define one Web Service: an (imaginary) online train ticket booking services for international train itineraries, offered by the Austrian national train operator ÖBB. Of course, this Web Service can be split up into several Web Services wherefore technologies for composition and Orchestration would be needed. But as a our intention within the current version of this use case modeling is to test and showcase the basic modeling of Web Services, we restrict ourselves to only one Web Services at this point in time.

As the referenced version of WSMO does only provide specifications for the description elements for Web Service Capability modeling, we restrict the Web Services models to the Capabilities at this point in time. A Web Service Capability in WSMO is described by pre- and postconditions, assumptions and effects. The primary information for suitability of a Web Service for satisfying a given Goal is the postcondition (the Web Service postcondition has to logically satisfy the Goal postcondition, which is the core of the discovery mechanisms). The other description elements are secondary information for determining suitability, i.e. filtering the set of Web Services that potentially match the Goal. More detailed discussion of the Discovery mechanism of WSMO Goals and Capabilities is provided in section [3.1.3](#).

Listing 5 shows the Capability specification for the Web Service, including the following WSMO description elements for Web Service Capabilities. Each notion is modeled as a rule that says when there is an instance that fulfills its body than the specific notion is satisfied:

- **precondition:** the input is an itinerary and is valid when the start- and the endlocation of the itinerary are located in Austria or in Germany.
- **assumption:** accepted payment method is creditcard, and the creditcard must be valid (expire date)
- **postcondition:** the Service returns instances of ticket an itinerary and the traveller as specified in the input.
- **effect:** there is a trade for the ticket of the output, wherefore only creditcard payment is support.

Listing 5: Capability of ÖBB Web Service for Booking Online Train Tickets for Austria and Germany

```

Web Service Capability
  selling online train tickets for Austria and Germany

non-functional Properties
  title
    selling online train tickets for Austria and Germany
  creator
    DERI International
  subject
    Train Tickets, Online Ticket Booking
  description
    a Web Service Capability for selling international train
    tickets online
  publisher
    DERI International
  contributor

```

```

    Michael Stollberg, Rubén Lara, Holger Lausen, Axel Polleres
date
    20040517
type
    WSMO Web Service Capability
format
    text
identifier

http://www.wsmo.org/2004/d3/d3.2/v0.1/20040517/resources/capability
.flr
source

http://www.wsmo.org/2004/d3/d3.2/v0.1/20040517/resources/capability
.flr
language
    English
relation
    http://www.wsmo.org/2004/d3/d3.2/v0.1/20040517/resources/tc.flr
    http://www.wsmo.org/2004/d3/d3.2/v0.1/20040419/resources/dt.flr
    http://www.wsmo.org/2004/d3/d3.2/v0.1/20040419/resources/po.flr
coverage
    Europe
rights
    DERI
version
    1.3

usedMediators
    // OO Mediator for using the Date and Time Ontology
    VTA-OOM-WS1Cap.wsml

// define an instance of Web Service Capability

oebbCap ofclass capability.

Precondition
// the input has to be an itinerary wherefore
// the start- and endlocation have to be in Austria or in Germany
and
// the departure date has to be later than the current Date

oebbCap[precondition] :-
    X ofclass itinerary[
        startLocation ofvalue StartLoc
        endLocation ofvalue EndLoc
        departure ofvalue Departure
    ] and
    (StartLoc.locatedIn = austria or StartLoc.locatedIn = germany)
and

```

```
(EndLoc.locatedIn = austria or EndLoc.locatedIn = germany) and
after(Departure,currentDate) .
```

Assumption

```
// there needs to be a buyer that accepts payment by cerditcard
// and this creditcard has to be valid (not expired).
```

```
oebbCap[assumption] :-
  X ofclass buyer[
    acceptsPayment =>> Payment
  ] and
  Payment ofclass creditCard and
  (currentDate.date.year < Payment.expYear) or
  ((currentDate.date.year = Payment.expYear) and
  ((currentDate.date.monthOfYear < Payment.expMonth) or
  (currentDate.date.monthOfYear = Payment.expMonth))).
```

Postcondition

```
// the output of the service is a ticket (train ticket)for an
itinerary wherefore
// the start- and endlocation have to be in Austria or in Germany
and
// the departure date has to be later than the current Date
```

```
oebbCap[postcondition] :-
  X ofclass ticket[
    itinerary ofvalue Itinerary ofclass itinerary[
      startLocation ofvalue StartLoc
      endLocation ofvalue EndLoc
      departure ofvalue Departure
    ]
  ] and
  (StartLoc.locatedIn = austria or StartLoc.locatedIn = germany)
and
  (EndLoc.locatedIn = austria or EndLoc.locatedIn = germany) and
  after(Departure,currentDate) .
```

Effect

```
// there shall be a trade for the train ticket of the postcondition
```

```
oebbCap[effect] :-
  Y ofclass trade[
    items ofvalues Ticket ofclass ticket[
      itinerary ofvalue Itinerary ofclass itinerary[
        startLocation ofvalue StartLoc
        endLocation ofvalue EndLoc
      ]
    ] and
    payment ofvalue Payment
  ] and
  Payment ofclass creditCard and
  (StartLoc.locatedIn = austria or StartLoc.locatedIn = germany)
```

```

and
    (EndLoc.locatedIn = austria or EndLoc.locatedIn = germany) .

```

Mediators

Regarding the requirement analysis for our use case, we have identified the need the following types of Mediators:

- OO-Mediators which connect the specific WSMO components and provide the needed ontology integration facility as a mediation service .
- WG Mediators for connecting the Goals and Web Services after successful discovery of suitable Web Services for searching valid train itineraries and online booking of tickets for an train itinerary.

In the following we model the concrete mediators needed for the use case. These Mediators are applied by the different components described before using the “usedMediators” modeling element.

OO-Mediators

OO Mediators "connect" the ontology/ies with the component that is using it. According to the WSMF/O-principle of strong de-coupling, all issues related to integrate (meaning to resolve all namespace and access issues) and mediate (meaning to resolve heterogeneities) the ontologies to be used are located in the OO Mediator. The user who uses a OO Mediator with the "usedMediators"-tag should therefore not care about which ontology he refers to - he just gets his "Information Space" by the OO Mediator and uses it.

In order to import the ontologies into all our components, we need three OO Mediators:

1. importing the Date and Time Ontology into the Train Connection Ontology
2. importing all ontologies in to the Goal
3. importing all ontologies in to the Web Service Description, here the Capability

Listings 6-8 specify these three OO Mediators. The general structure of an OO Mediator is that the sourceComponent is the imported ontologies (can also be other OO Mediators), and the targetComponent is the user of the imported ontologies. The internal structure and functionality of a OO Mediator is not stable specified yet, thus we only specify the connection facility in the Listings.

Listing 6: OO-Mediator 1

```

OO Mediator
    Train Connection Ontology uses Date and Time Ontology

non-functional Properties
    title

```

```
Train Connection Ontology uses Date and Time Ontology
creator
  DERI International
subject
description
  importing the Date and Time Ontology into the Train
  Connection Ontology
publisher
  DERI International
contributor
  Michael Stollberg
date
  20040430
type
  WSMO OO Mediator
format
  text
identifier

http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/VT
A-OOM-trainConnection.wsml
source

http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/VT
A-OOM-trainConnection.wsml
language
  English
relation

http://www.wsmo.org/2004/d3/d3.2/v0.1/20040517/resources/tc.
flr

http://www.wsmo.org/2004/d3/d3.2/v0.1/20040419/resources/dt.
flr
coverage
rights
  DERI
version
  1.1

sourceComponent
  // identifier of "Date an Time Ontology"

http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/dt
.flr

http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/po
.flr

targetComponent
  // identifier of the "Train Connection Ontology"
```

```
http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/tc
.flr
```

mediationService

```
// not needed here
```

Listing 7: OO-Mediator 2

OO Mediator

```
OO Mediator for Goal
```

non-functional Properties

title

```
OO Mediator for Goal
```

creator

```
DERI International
```

subject

description

```
importing all ontologies in to the Goal
```

publisher

```
DERI International
```

contributor

```
Michael Stollberg
```

date

```
20040430
```

type

```
WSMO OO Mediator
```

format

```
text
```

identifier

```
http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/VTA-
OOM-Goal1.wsml
```

source

```
http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/VTA-
OOM-Goal1.wsml
```

language

```
English
```

relation

```
http://www.wsmo.org/2004/d3/d3.2/v0.1/20040517/resources/tc.fl
r
```

```
http://www.wsmo.org/2004/d3/d3.2/v0.1/20040419/resources/po.fl
r
```

coverage

rights

```
DERI
```

version

1.1

sourceComponent

// identifiers of "Train Connection Ontology" and
"Purchase Ontology"

<http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/tc.flr>

<http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/po.flr>

targetComponent

// identifier of the Goal

<http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/goal.flr>

mediationService

// not needed here

Listing 8: OO-Mediator 3

OO Mediator

OO Mediator for Web Service Capability

non-functional Properties**title**

OO Mediator for Web Service Capability

creator

DERI International

subject**description**

importing all ontologies in to the Web Service Capability

publisher

DERI International

contributor

Michael Stollberg

date

20040430

type

WSMO OO Mediator

format

text

identifier

<http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/VTA-OOM-WS1Cap.wsml>

source

<http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/VTA-OOM-WS1Cap.wsml>

language

English

relation<http://www.wsmo.org/2004/d3/d3.2/v0.1/20040517/resources/tc.flr><http://www.wsmo.org/2004/d3/d3.2/v0.1/20040419/resources/po.flr>**coverage****rights**

DERI

version

1.1

sourceComponent

comment: identifiers of "Train Connection Ontology" and "Purchase Ontology"

<http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/tc.flr><http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/po.flr>**targetComponent**

comment: identifier of the Web Service Capability

<http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/capability.flr>**mediationService**

comment: not needed here

WG-Mediators

A WG Mediator denotes the differences between a Goal and a Web Service Capability in order to make them matching by restricting the range of valid information to be exchanged between the Goal Owner and the Web Service. The difference is stated in a reduction. In fact, the reduction defines the intersection between the information space of the Goal and the information space of Capability. Thereby, only such information will be interchanged between the Goal and the Web Service that are valid, meaning that with all outputs, postconditions, and effects of the Web Service the Goal is satisfiable.

In our use case, we only need 1 WG Mediator for connecting the Goal and the Web Service, specified in Listing 9. The WG Mediator, as all other WSMO components, has to be aware of the ontologies used in the components to be connected. As the source component is the Goal, and the target component is the Web Service with the Capability, the the OO Mediators defined for the Goal and for the Web Service implicitly. The "usedMediators" tag specifies usage of additional ontologies that are needed for specifying the reduction. The Reduction in the WG Mediator restricts the set of values for the Goal and the Web Service to those for which the usage of the Web Service is valid for satisfying the Goal. More precisely, the Reduction is the intersection of valid knowledge items of the Goal and of valid items of the Web Service.

Listing 9: WG-Mediator between Goal and Web Service

WG Mediator

WG Mediator

non-functional Properties

title

Web Service - Goal connection Capability uses all ontologies

creator

DERI International

subject

description

connecting the Goal and the Web Service Capability

publisher

DERI International

contributor

Michael Stollberg

date

20040430

type

WSMO OO Mediator

format

text

identifier

<http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/VTA-WGM1.wsml>

source

<http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/VTA-WGM1.wsml>

language

English

relation

<http://www.wsmo.org/2004/d3/d3.2/v0.1/20040517/resources/capability.flr>

<http://www.wsmo.org/2004/d3/d3.2/v0.1/20040419/resources/goal.flr>

coverage

rights

DERI

version

1.1

usedMediators

// no additional Mediators needed

sourceComponent

// identifier of the Goal

<http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/goal.flr>

targetComponent

// identifier of the Web Service

<http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/ws.flr>

reduction

comment: to be specified

GG-Mediators

A GG Mediator connects Goals by specifying a reduction between them. For example, a GG Mediator would connect a Goal "buy a ticket" with another Goal "buy a train ticket" by stating the ontological correspondance between the Goals as a reduction. If 'train ticket' is a subclass of 'ticket', than the reduction in the GG Mediator would specify that valid instances for the second Goal have to be 'train ticket **subclassof** ticket'.

There is no GG Mediator needed in the use case.

WW-Mediators

A WW Mediator connects Web Services used by another Web Service in ther Orchestration, resolving heterogeneities at all levels (data, process, protocol). There is no WW Mediator in this use case, since the Web Service does not apply other Web Services in order to realize its functionality (at least this is not modelled at the moment). .

3.1.3 SWS Mechanisms based on WSMO component models

On basis of the models for the WSMO components specified above, we can define the following automated mechanisms: Web Service Discovery, Web Service Composition, and Web Service Execution. In the following we explain how these mechanisms work and which parts of the WSMO models they use.

Web Service Discovery

Web Service Discovery is concerned with inference-based mechanisms that detect suitable Web Service for a given Goal. This means that the discovery mechanism searches available Web Service descriptions and determines whether these can be used to fulfill a certain Goal. The overall structure of WSMO supports Web Service discovery explicitly by introducing the notions of Goals and Web Services as top level building blocks. The requirements and the approach for Web Service Discovery in WSMO is exhaustively discussed in [\[Keller et al., 2004\]](#). Here, we shortly summarize the most important aspects and explain how the discovery mechanism works in the use case models as specified above.

The functionality of the discovery mechanism can be separated into three major aspects:

1. The Core: Goal-Capability-Matching

Goal-Capability matching determines whether the Capability of a Web Service Description can satisfy the given Goal, i.e. if the Web Service can be used for solving the Goal. Therefore, it basically has to be proven that the Capability logically entails the Goal with the premise that the conditions for successful usage of the Web Service are fulfilled at invocation time.

Goal-Capability-Matching is considered as the heart of Web Service

Discovery, as it determines whether some Web Service can satisfy the Goal at all. Upon this, different "Discoverers" can be build that modify the discovery results with regard to specific requirements for the application. We discuss this in more detail below.

2. Heuristics for establishing Goal-Capability Matching

Goal-Capability matching is only successful, i.e. it returns a set of suitable Web Services to solve a given Goal, if the Goal and the Web Service Capability match perfectly. This means that for all states where the postconditions and effects of the Web Service are fulfilled the Goal is satisfied. This might not hold for many cases, as there might be semantic differences between the Goal and the Capability; but a Web Service might be usable for solving Goal when the valid set of outputs (as well as the set of halting states) of the Web Service is restricted – also vice versa, i.e. the Web Service can be used to solve a Goal within certain restrictions on the Goal.

In WSMO, these differences are explicitly stated in a WG Mediator. This restricts the valid values between a Goal and a Web Service Capability, thereby ensures Goal-Capability-Matching between Goals and Capabilities that only match partly, and thus broadens the set of possible usable Web Services for solving a Goal. For determining the required reduction in a WG-Mediator, specific heuristics can be applied.

3. Filter mechanisms for improving discovery results

The Discovery mechanism in general returns a set of suitable Web Services for solving a Goal. In order to improve the quality of the results set, additional filter mechanism can be applied. These can be based on the non-functional properties of Web Services, or can take some preferences of the customer into account.

With regard to the WSMO models of the use case defined above, we can now show how the heart of Web Service Discovery in WSMO, i.e. the Goal-Capability-Matching works. Therefore, we have to show that the Proof Obligation for Goal-Capability-Matching holds for the models of the Goal and the Capability defined in the use case.

The Proof Obligation for Goal-Capability-Matching is defined as follows

$$PO_{gcm}^*(G, C) \equiv \{O_C, O_G, M_C, M_G\} \models Cl_{\exists}(\exists in_1, \dots, in_n: ((\Psi^{pre} \wedge \Psi^{ass}) \wedge (\Psi^{post} \rightarrow \Phi^{post}) \wedge (\Psi^{eff} \rightarrow \Phi^{eff})))$$

This Proof Obligation states that under consideration of all Ontologies and Mediators used in the Goal and the Capability description (2nd line), if the user is able to provide concrete values for the input parameters of a service such that the Preconditions and Assumptions defined in the Capability (3rd line) are satisfied, and if the Capability postconditions imply the Goal postconditions and the Capability effects imply the Goal effects, that the Capability matches the Goal. In the use case, we have defined 1 Goal and 1 Capability (see Listing 5). Obviously, the Proof Obligation is fulfilled in this case because the Goal and Capability are homogenous.

The realization of the Goal-Capability-Matching works as follows - all computational resources for this are provided in [Appendix A](#):

- The Goal is defined as a fact, meaning it describes an ontological structure with out variables. The Goal is only specified partially, meaning that concrete values are only specified for the subset of the properties of the ontological concepts that is needed to express the desire; e.g., for the itinerary that a ticket is desired for, no information is specified for 'arrival' (see Listing 4) .
- The description elements of the Web Service Capability are modeled as rules. Here also, only the subset of properties of the ontological notions that is needed to describe the information structure and conditions is specified; besides, the body of each rule contains conditions that restrict the range of valid values.

The idea for general Goal-Capability-Matching, based on the construction of Goals and Capabilities, is that there is a query that asks whether the Capability is fulfilled. This means there has to be fact that satisfies the body of the distinct description elements of the Capability. In the use case, this fact is the Goal. Considering the Goal as specified in Listing 4, the Goal postcondition is fact that satisfies the body of the Capability postcondition in Listing 5: the Goal Postcondition is a ticket for a specific itinerary (from Innsbruck to Frankfurt with a certain departure), and the body of the Capability postcondition requires a ticket for an itinerary with start- and endlocation in Austria or Germany, respectively, and with a departure that has to be later than the current date (which is explicitly modeled in the Date and Time Ontology). The same holds for the Goal Effect in correlation to the Capability Effect. Therein, the domain knowledge specified in the ontologies is applied by the reasoner in order to determine the matching; for instance, the Goal specifies "innsbruckHbf" as the start location - and instance defined in the Train Connection Ontology, and the Capability postcondition states that the start location of the itinerary has to be located in Austria or Germany. The instance "innsbruckHbF" is located in "innsbruck", and "innsbruck" is located in Austria - so "innsbruckHbf" satisfies the condition of the Capability postcondition. Thus, the following query returns "oebbCap" as a Capability that can satisfy the Goal:

```
?- X:capability[postcondition] and X:capability[effect].
```

This structure of Goal and Capability description also supports detection of "overspecified" Capabilities for resolving a Goal. Imagine a Capability with a postcondition that contains a planeticket from A to B, and a train ticket within Austria and Germany. Here, also the Goal would satisfy the Capability postcondition; in order to ensure that only train tickets are booked, the reduction in the WG Mediator that connects this "overspecified" Web Service to the Goal restricts the set of valid information to train tickets only (see an example for this in Appendix A as well).

Observant readers will notice that the realization for Goal-Capability-Matching depicted here does only consider the Capability Postcondition and the Capability Effect with respect to all used Ontologies and Mediators for Goal-Capability-Matching - in contrast to the general Proof Obligation introduced above. The reason for this is both on a conceptual level and on the modeling realization. The underlying understanding of a Goal is that it describes a desire "I want this" without any respect to how this desire could be solved. This desire is modeled as a specific ontology

structure - the Goal model it says that the desire is a "ticket for a specific itinerary", and that there also should be a trade for this ticket. The Goal does not state anything about the input that will be provided to a suitable Web Service, neither the Goal does specify any functional capabilities that might be needed for interacting with a Web Service. On the other hand, the underlying understanding of the Web Service Capability is that this is a functional description of the Web Service, i.e. it describes what the Web Service does. In general, the information provided in the Web Service Capability are "If an input is provided for which certain conditions hold, then the Web Service will return some result in relation to the input wherefore also some conditions hold". With regard to this, the Capability description elements are conceptually separated into two groups: elements that define conditions that have to hold before the Web Service is executed, and elements that describe conditions that hold after the Service is executed. The former group consists of the precondition (describes the requested input along with conditions on it) and the assumptions (arbitrary state of the world that has to hold before the Web Service can be executed), and the latter group is the postcondition (describes the output of the service after execution along with conditions on it) and the effect (arbitrary state of the world that holds after the execution of the Web Service, i.e. changes in the world).

With respect to this underlying understanding of the WSMO description elements it is obvious that for Goal-Capability-Matching only the 2nd group of Capability description elements has to be considered, i.e. Capability postcondition and Capability effect. The reason is that these information state what the Web Service can provide in the end, and this is what we want to know within Goal-Capability-Matching. This means by Goal-Capability-Matching we get the information that certain Web Services (or a set of Web Services) can be used to solve the Goal; in the next step we have to check whether we really can use the Web Service (can we provide the required input? Do we support other technical features? etc.). Besides, from a modeling perspective, you always need a counterpart for a description notion that shall be taken into account for Goal-Capability-Matching. Within the understanding of Goals outlined above and the description elements existing for Goals in WSMO, there is not such a counterpart for Capability preconditions and assumptions. The conceptual reason for this we explained above.

The solution for Goal-Capability-Matching outlined and implemented here seems to be a proper approach towards a generic Goal-Capability-Matching in WSMO. Nevertheless, it is not finished by now, and we briefly explain the background and the arising challenges for further elaboration of this approach.

In the solution outlined above, the Goal is a fact. Such facts are needed as they have to satisfy the body of the respective Capability notions. But the restriction of modeling Goals as facts obviously limits the expressiveness of Goals - in the end, we should be able to define an arbitrary object as Goal that might contain variables and axioms. Also, from an application perspective, the facts that will be applied to check satisfaction of the Capability postcondition and effects would be generated by the internal functionality of the Web Service (or at least by the Web Service owner), and not be stated by the Goal.

Such an approach wherein Goals and Capabilities are modeled as arbitrary logical expressions requires a different, more complex and advanced mechanism for Goal-Capability-Matching. More precisely, we would have to test whether a Capability logically entails a Goal (as stated in the general Proof Obligation above). The result of Goal-Capability-Matching would be the same as in the solution outlined here: When there are facts that satisfy the Capability postcondition and effect, and if the

Capability logically entails the Goal, than those facts will also satisfy the Goal, so the Capability matches the Goal. This approach seems to be the most reasonable solution for Goal-Capability-Matching within WSMO

Unfortunately, there are some problems for such a solution. Logical entailment can be realized in different ways - by implication (the Capability postcondition implies the Goal), or by query containment when the Goal and the Capability are modeled as queries. But, logical entailment is not decidable, nor satisfiable for First Order Logic in general. For a proper solution, the logical language to be used for modeling Goals and Capabilities needs to be restricted to a decidable subset of FOL - such as Horn Logic or Description Logic, or a combination of these. Furthermore, there does not exist a proper implementation for determining logical entailment (at least for the existing reasoners that we have considered so far).

In conclusion, the approach for Goal-Capability-Matching presented here seems to be a promising solution because it is heading towards the right direction. For further development of the Goal-Capability-Matching technology within WSMO as the heart of Web Service Discovery, very complex and challenging efforts have to be faced: definition of a decidable subset for the specification language of WSMO as well as implementation of logical entailment for the reasoner to be supported within WSMO.

3.1.4 Conclusions

We have described a real-world setting of using Semantic Web Services for a Virtual Travel Agency (VTA) that provides an end-user service for booking international train tickets, thereby aggregating Web Services of different e-Tourism Service Providers. The set up of this use case and the system architecture of the VTA here is conform to the general structure of the VTA use case described in [Section 2.1](#).

Within the WSMO models defined in this use case we have shown how to model the different components of WSMO, with regard to the stable WSMO modeling elements available at this point of time:

- Ontologies: all needed domain knowledge is provided in the ontologies.
- Goals: We have specified a Goal in the Use Case, with respect to the description elements defined for Goals in WSMO
- Web Service Capability: We have derived a specification on how to model Web Service Capabilities
- Mediators: We have modeled the needed WSMO Mediators needed for the Use Case, namely OO Mediators and a WG-Mediator.

Furthermore, we have outlined the general workflow of the WSMO Discovery mechanism that works on the WSMO models for Goals and Capabilities.

The outcome of the first use case modeling are manifold. First of all, it shows how the different WSMO components are modeled concretely. This gives answers to many questions that have been arising within WSMO: a more concrete understanding of Goals in WSMO and what they actually express, what is defined in a Web Service Capability with special regards to the difference between preconditions and assumptions, postconditions and effects respectively, and the concept of Mediators in WSMO. Further major outcomes of the use case and testing efforts so far is a concrete specification of how to model the different types of axiom definitions in

WSMO, as well as further insights on Goal-Capability Matching as the heart of Web Service Discovery mechanism that work on the WSMO models for Goals and Web Service Capabilities. The scope of the use case is restricted to the most essential building blocks of WSMO at this point of time, and it will be updated and extended in the future for testing and showcasing further WSMO constructs.

3.2 B2B Integration with Semantic Web Services

[not in this version]

3.2.1 Functional Requirements

similar to 3.1.1

3.2.2 WSMO Modeling

similar to 3.1.2

Ontologies

Goals

Web Services

Mediators

3.2.3 SWS Mechanisms based on WSMO component models

similar to 3.1.3

3.2.4 Conclusions

similar to 3.1.4

4. Conclusions and Further Work

Appart from discussing possible usage scenarios of Semantic Web Services, the major interest in this deliverable is to test and verify WSMO modeling for recursive development of WSMO, and to serve as a testbed for development of WSMO-based

technologies. The deliverable is intended to exemplify and showcase the usage of WSMO for modeling different aspects related to Semantic Web Services, and it will continuously be updated according to further development of WSMO.

According to the current status of WSMO, the most interesting aspects are:

- to showcase concrete modeling of a real-world scenario in WSMO
- thereby gain a better understanding of WSMO, its distinct components, and how these are related
- a more precise definition of specific elements (esp. axioms), their meaning and how to model them.

The major outcome of the Use Case modeling provided in this deliverable are:

- exemplification of modeling the core components in F-Logic according to the current specification of WSMO modeling elements
- understanding and specification for modeling different types of axioms in WSMO
- insights on the requirements and workflow of the WSMO Discovery mechanism, with special attention to Goal-Capability-Matching as the heart of Discovery.

The directions for future work in this deliverable are:

- enhance the scope of the Use Case modeling.
- provide complete modeling of other use cases, esp. for the "B2B Integratin with Semantic Web Services" as described in [Section 2.2](#)
- update the WSMO models according to new developments or changes in the WSMO Specification

References

[Arroyo et al., 2004] Arroyo, S.; Lara, R.; Gómez, J.; Berka, D.; Ding, Y.; Fensel, D. (2004): Semantic Aspects of Web Services. In Munindar. P. Singh (Ed.), Practical Handbook of Internet Computing. Baton Rouge: Chapman Hall and CRC Press, Baton Rouge. 2004.

[Arroyo and Stollberg, 2004] Arroyo, S.; Stollberg, M.: *WSMO Primer*. WSMO Deliverable D3.1, available at: <http://www.wsmo.org/2004/d3/d3.1/v0.1/>

[ebXML] ebXML Deliverables <http://www.ebxml.org/specs/>

[Fensel & Bussler, 2002] D. Fensel and C. Bussler: *The Web Service Modeling Framework WSMF*, Electronic Commerce Research and Applications, 1(2), 2002.

[Keller et al., 2004] Keller, U.; Lara, R.; Polleres, A.; Lausen, H.; Stollberg, M.: *Inferencing Support for Semantic Web Services: Proof Obligations*. WSMO

Deliverable D5.1 v0.1, WSML Working Draft 05 April 2004. available at http://www.wsmo.org/2004/d5/d5.1/v0.1/20040405//d5.1v0.1_20040405.pdf

[Kifer et al., 1995] M. Kifer, G. Lausen, and James Wu: *Logical foundations of object oriented and frame-based languages*. Journal of the ACM, 42(4):741-843, 1995.

[Pan and Hobbs] F. Pan and R. Hobbs: *Time in OWL-S*, available at: <http://www.isi.edu/~pan/damlttime/AAAsymp2004.pdf>.

[Oren et al, 2004] Oren, E. (Ed.): *BNF grammar for WSML user language.*, WSMO Deliverable D16.1, Working Draft 18 April 2004 available at: <http://www.wsmo.org/2004/d16/d16.1/v0.2>.

[Roman et al., 2004] D. Roman, U. Keller, H. Lausen (eds.): *Web Service Modeling Ontology - Standard (WSMO - Standard)*, version 0.2 available at <http://www.wsmo.org/2004/d2/v02/>

[RosettaNet] RosettaNet <http://www.rosettanet.org/>

[SOAP] Mitra, N.: *SOAP Version 1.2 Part 0: Primer*. W3C Recommendation 24 June 2003. available at: <http://www.w3.org/TR/soap12-part0/>

[UDDI] Bellwood, T.; Clément, L.; von Riegen, C. (Ed.): *UDDI Version 3.0.1*. UDDI Spec Technical Committee Specification, Dated 20031014. available at: http://uddi.org/pubs/uddi_v3.htm

[WSDL] Chinnici, R.; Gudgin, M.; Moreaum, J.-J.; Weerawarana, S. (2003): *Web Services Description Language (WSDL) Version 1.2*. W3C Working Draft 3 March 2003. available at <http://www.w3.org/TR/wsdl20/>.

Acknowledgements

The work is funded by the European Commission under the projects DIP, Knowledge Web, Ontoweb, SEKT, and SWWS; by Science Foundation Ireland under the DERI-Lion project; and by the Austrian government under the CoOperate programme.

The editors would like to thank to all the members of the [WSMO working group](#) for their advises and inputs to this document. Very special thanks go to Michael Kifer who supported the development of the models for Goals and Capabilities and especially on elaboration of generic solutions for the Goal-Capability-Matching.

Appendix A: Flora2-F-Logic for the VTA - Use Case

Here, the complete WSMO models for the VTA Use Case described in Section can be download as computational resources for testing and development of WSMO with FLORA-2, an F-Logic reasoner. The files below contain the WMSO models of the use case in Flora2-compatible syntax. For testing and development, the files can be loaded into different Flora modules.

Information and download of Flora2 is provided [here](#).

NOTE: the WSMO models for Flora2 are currently under construction. The most recent resources can be accessed via CVS web-interface at: <http://cvs.deri.at/cgi-bin/viewcvs.cgi/wsmo//d3/d32/resources/>.

Ontology 1: "International Train Connections Domain Ontology"

```
// International train connections domain ontology

// Concept definition: location
location[
  name=>string,
  locatedIn=>>country
].

//Concept definition: station
station::location.

country::location.

city::location.

village::location.

//locatedIn is a transitive property
X:location[locatedIn->>Y] :-
  X:location[locatedIn->>Z],
  Z:location[locatedIn->>Y].

invalid(X, 'cyclic dependency locatedIn') :-
  X:location[locatedIn->>X].

// Concept definition: borderStation
// Defines a train station at the border
borderStation::station[
  borderToCountry =>> country
].

//Concept definition: itinerary
itinerary[
  startLocation => station,
  endLocation => station,
  via =>> station,
  departure => dateAndTime,
  arrival => dateAndTime
].

//Concept definition: customer
traveller[
  name => string
].

//Concept definition: ticket
//Defines a ticket that relates an itinerary to a customer
```

```

ticket::product[
  itinerary => itinerary,
  traveller => traveller
].

// Axiom definition: axiomDateTime
// departure has to be before arrival

//axiomDepartureArrival:axiomDefinition[
//  definedBy -> X1:itinerary[departure->X2, arrival->X3], X2[before(X3)->true]
//].

//This is the F-Logic processable rule
invalid(I) :- I:itinerary[departure->X2, arrival->X3], after(X2,X3).

// below, some instances are defined that are needed throughout the use case
modeling

germany:country[name -> 'Germany'].
austria:country[name -> 'Austria'].

innsbruckHbf:station[name ->'Innsbruck Hbf', locatedIn ->> innsbruck].
innsbruck:city[name->'Innsbruck',locatedIn->> austria].
boesby:village[name->'Boesby', locatedIn->>germany].

frankfurtHbf:station[name ->'Frankfurt Hbf', locatedIn ->> germany].
kufsteinHbf:borderStation[name -> 'Kufstein Hbf', locatedIn ->> austria,
borderToCounty ->> germany].
salzburgHbf:borderStation[name -> 'Salzburg Hbf', locatedIn ->> austria,
borderToCounty ->> germany].
freilassingBf:borderStation[name -> 'Freilassing Bf', locatedIn ->> germany,
borderToCounty ->> austria].
kiefersfeldenBf:borderStation[name -> 'Kiefersfelden Bf', locatedIn ->> germany,
borderToCounty ->> austria].

```

Ontology 2: "General Date and Time Ontology"

```

/* EVERYTHING WHAT CAN BE REPRESENTED IN FLORA IS NOT IN COMMENTS
namespace
  default:http://www.wsmo.org/2004/d3/d3.2/v0.1/dateAndTime.wsml#
  xsd:http://something.w3c.org/withXSD#

ontology
  http://www.wsmo.org/2004/d3/d3.2/v0.1/dateAndTime.wsml
  non-functional-properties
  title
    "Date and Time Ontology"
  creator
    "Holger Lausen"
  version
    "$Version$"
  description
    "The main ontologies taken into consideration for developing this
representation in F-Logic:
  An entry sub-ontology of time: http://www.isi.edu/~pan/damlltime/time-
entry.owl
  This ontology uses abstract temporal concepts like instant, interval and
event and
  uses the Gregorian calendar as representation (partly using own encoding and
partly
  using XSD encoding). Axioms are defined in the accompanying paper (in first
order
  logic.) and there is also a LISP version of these axioms available at:
  http://www.cs.rochester.edu/~ferguson/daml/daml-time-20030728.lisp

```

Other ontologies like COBRA calendarclock ontology (<http://dam1.umbc.edu/ontologies/cobra/0.4/calendarclock>) are only a straight forward representation of the Gregorian calendar, without any abstraction of concepts and description of axioms.

Widely used concrete representations are defined in ISO 8601 (Numeric representation of Dates and Time) and in the XML Schema Definition (<http://www.w3.org/TR/xmlschema-2/>), which is based on ISO 8601.

This ontology uses the Gregorian calendar (representation based on the definition in

```

http://www.w3.org/TR/xmlschema-2/
concept
  instant
  non-functional-properties
  description
    "An instant represents a particular point in time [1]"
*/
instant[
].
/*

concept
  interval
  non-functional-properties
  description
    "An interval represents a duration between 2 points in time [1]"
  start oftype instant
  end oftype instant
*/
interval[
  start=>instant,
  end=>instant
].
/*

concept
  date
  non-functional-properties
  title
    "Date Class"
  description
    "conceptdefintion for the class date and its representation
    according to the Gregorian calendar"
  dayOfMonth oftype dayOfMonth
  monthOfYear oftype monthOfYear
  year oftype year
*/
date[
  dayOfMonth=>dayOfMonth,
  monthOfYear=>monthOfYear,
  year=>year
].
/*

axiom
  axiomValidDate
  non-functional-properties
  totöe "Axiom for a valid date"
  description
    "A date is valid if the day, month and year are valiud,
    leap years and checks on days (28/29/30) are not yet done"
  definedBy

```

```

    invalid(X) :- X oftype data AND (invalid(X.dayOfMonth) OR
invalid(X.monthOfYear) OR invalid(X.year))

//michael you may skip the "axiomValidDate" thing, this is a necessary flora hack
in order to identify what axiom is violated
*/
invalid(X,"axiomValidDate")
:- X:date, (invalid(X.dayOfMonth); invlaid(X.monthOfYear); invalid(X.year) ).
/*

concept
  dayOfMonth subclassOf integer
  non-functional-properties
  title
    "Day of the Month"

*/
dayOfMonth::integer.

//integrety constraint for valid dayOfMonths:
invalid(X,"invalid dayOfMonth")
:- X:dayOfMonth, (X<0; X>31).

//a year is represented by an integer
year::integer.

//a monthOfYear is represented by an integer
//and has additional properties, concrete month are defined as instances
//according to this class definition:
monthOfYear::integer[
  name=>string,
  daysAfterBeginOfYear=>integer
].
//integrety constraint for valid monthOfYear:
invalid(X, "monthOfYear"):- X:monthOfYear, (X<0; X>12).

/*
instance
  1
  non-functional-properties
  description
    "describing a particulat month of a year"
  daysAfterBeginOfYear hasValue 31
  name hasValue "January"
*/
1:monthOfYear[daysAfterBeginOfYear->31, name-> 'January'].
2:monthOfYear[daysAfterBeginOfYear->59, name-> 'February'].
3:monthOfYear[daysAfterBeginOfYear->90, name-> 'March'].
4:monthOfYear[daysAfterBeginOfYear->120, name-> 'April'].
5:monthOfYear[daysAfterBeginOfYear->151, name-> 'May'].
6:monthOfYear[daysAfterBeginOfYear->181, name-> 'June'].
7:monthOfYear[daysAfterBeginOfYear->212, name-> 'July'].
8:monthOfYear[daysAfterBeginOfYear->243, name-> 'August'].
9:monthOfYear[daysAfterBeginOfYear->273, name-> 'September'].
10:monthOfYear[daysAfterBeginOfYear->304, name-> 'October'].
11:monthOfYear[daysAfterBeginOfYear->334, name-> 'November'].
12:monthOfYear[daysAfterBeginOfYear->365, name-> 'December'].

//conceptdefintion of the time class
time[
  hourOfDay=>hourOfDay,
  minuteOfHour=>minuteOfHour,
  secondOfMinute=>secondOfMinute
].

//integrety constraint for valid time:
invalid(X, "invalidTime") :- X:time, (invalid(X.hourOfDay);
invalid(X.minuteOfHour); invalid(X.secondOfMinute)).

```

```

//a secondOfMinute is represented by an integer
secondOfMinute::integer.
//integrety constraint for valid secondOfMinute:
invalid(X,"invalidsecondOfMinute") :- X:secondOfMinute, (X<0; X>59).

//a minuteOfHour is represented by an integer
minuteOfHour::integer.
//integrety constraint for valid minuteOfHour:
invalid(X,"invalidminuteOfHour") :- X:minuteOfHour, (X<0; X>59).

//a hourOfDay is represented by an integer
hourOfDay::integer.
//integrety constraint for valid hourOfDay:
invalid(X,"invalidhourOfDay") :- X:hourOfDay, (X<0; X>23).

//conceptdefintion of the date and time class
//this represents together a specific point of time (instant)
dateAndTime::instant[
    date=>date,
    time=>time
].

//integrety constraint for valid dateAndTimes:
invalid(X) :-
    X:dateAndTime, (invalid(X.date); invalid(X.time)).

// computes equality of a date
equal(X,Y) :-
    Y:date, X:date,
    X.dayOfMonth = Y.dayOfMonth,
    X.monthOfYear = Y.monthOfYear,
    X.year = Y.year.

// computes if a given date X is before another date Y
before(X, Y) :-
    Y:date, X:date,
    ((X.dayOfMonth < Y.dayOfMonth, X.monthOfYear = Y.monthOfYear, X.year = Y.year);
    (X.monthOfYear < Y.monthOfYear, X.year = Y.year);
    (X.year < Y.year)).

// computes if a given date X is after another date Y
after(X, Y) :-
    Y:date, X:date,
    ((X.dayOfMonth > Y.dayOfMonth, X.monthOfYear = Y.monthOfYear, X.year = Y.year);
    (X.monthOfYear > Y.monthOfYear, X.year = Y.year);
    (X.year > Y.year)).

// this is simplified and ignores leap years a proper algorithm as
// e.g. found at http://quasar.as.utexas.edu/BillInfo/JulianDatesG.html
// calculates correctly for Gregorian Calendar dates (after 1582
// (at least for most countries, see
// http://members.brabant.chello.nl/~h.reints/cal/whenjul2greg.htm
// for an exact dates per country (e.g. Yugoslavia changed 1919))
// can be represented in f-logic as follows:

julianDayNumber(X,JDN) :-
    X:date,
    X.monthOfYear<3,
    Y is X.year-1,
    M is X.monthOfYear+12,
    D is X.dayOfMonth,
    A is '//'(Y,100)@prolog(),
    B is '//'(A,4)@prolog(),
    C is 2-A+B,
    E is floor(365.25*(Y+4716))@prolog(),
    F is floor(30.6001*(M+1))@prolog(),
    JDN is C+D+E+F-1524.

```

```

julianDayNumber(X, JDN) :-
    X:date,
    X.monthOfYear>2,
    Y is X.year,
    M is X.monthOfYear,
    D is X.dayOfMonth,
    A is '//'(Y,100)@prolog(),
    B is '//'(A,4)@prolog(),
    C is 2-A+B,
    E is floor(365.25*(Y+4716))@prolog(),
    F is floor(30.6001*(M+1))@prolog(),
    JDN is C+D+E+F-1524.

// however due to a bug in XSB (float number arithmetics do not work proper)
// this could not be implemented and a simplified version that aproximates the days
// after Christ is used.

daysAfterChrist(D,X) :-
    D:date, Z is D.monthOfYear,
    X is (D.dayOfMonth + Z.daysAfterBeginOfYear + (D.year*365)).

//the difference in days between 2 dates
daysBetween(D1,D2,X) :-
    D1:date, D2:date,
    daysAfterChrist(D1,DAC_D1),
    daysAfterChrist(D2,DAC_D2),
    X is DAC_D1 - DAC_D2.

// computes if two given times are the same
equal(X,Y) :-
    X:time, Y:time,
    X.secondOfMinute = Y.secondOfMinute,
    X.minuteOfHour = Y.minuteOfHour,
    X.hourOfDay = Y.hourOfDay.

// computes if a given time X is before another time Y
before(X,Y) :-
    X:time, Y:time,
    ((X.secondOfMinute < Y.secondOfMinute, X.minuteOfHour = Y.minuteOfHour,
X.hourOfDay = Y.hourOfDay);
    (X.minuteOfHour < Y.minuteOfHour, X.hourOfDay = Y.hourOfDay);
    (X.hourOfDay < Y.hourOfDay)).

// computes if a given time X is after another time Y
after(X,Y) :-
    X:time, Y:time,
    ((X.secondOfMinute > Y.secondOfMinute, X.minuteOfhour = Y.minuteOfhour,
X.hourOfDay = Y.hourOfDay);
    (X.minuteOfhour > Y.minuteOfhour, X.hourOfDay = Y.hourOfDay);
    (X.hourOfDay > Y.hourOfDay)).

//computes the amount of seconds from midnight
secondsFromMidnight(T,X) :-
    T:time,
    X is T.secondOfMinute + (T.minuteOfHour*60) + (T.hourOfDay*60*60).

//the difference in seconds between 2 times
secondsBetween(T1, T2, X) :-
    T1:time, T2:time,
    secondsFromMidnight(T1,SFM_T1),
    secondsFromMidnight(T2,SFM_T2),
    X is SFM_T1 - SFM_T2.

//computes if Date and Time are equal
equal(X,Y) :-
    X:dateAndTime, Y:dateAndTime,
    equal(X.date,Y.date),
    equal(X.time,Y.time).

```

```

//computes if a given date and time X is before another date and time Y
before(X,Y) :-
  X:dateAndTime, Y:dateAndTime,
  ((equal(X.date,Y.date), before(X.time,Y.time));
  before(X.date,Y.date)).

//computes if a given date and time X is after another date and time Y
after(X,Y) :-
  X:dateAndTime, Y:dateAndTime,
  ((equal(X.date,Y.date), after(X.time,Y.time));
  after(X.date,Y.date)).

//computes the difference in seconds between two different DateAndTime
secondsBetween(D1, D2, X) :-
  D1:dateAndTime, D2:dateAndTime,
  daysAfterChrist(D1.date,DAC_D1),
  daysAfterChrist(D2.date,DAC_D2),
  secondsFromMidnight(D1.time,SFM_T1),
  secondsFromMidnight(D2.time,SFM_T2),
  X is SFM_T1 + DAC_D1 * 24 * 60 * 60 -
  (SFM_T2 + DAC_D2 * 24 * 60 * 60).

//the difference in (decimal) days between two different DateAndTime
daysBetween(D1, D2, X) :-
  D1:dateAndTime, D2:dateAndTime,
  secondsBetween(D1,D2,Z),
  X is Z/60/60/24.

#####STUFF FOR intervall#####

//computes if a interval X contains a second interval Y
contains(X,Y) :-
  X:interval, Y:interval,
  (before(X.start, Y.start);equal(X.start, Y.start)),
  (after(X.end, Y.end);equal(X.end, Y.end)).

//computes if a interval X contains a instant Y
contains(X,Y) :-
  X:interval, Y:instant,
  (before(X.start, Y);equal(X.start, Y)),
  (after(X.end, Y);equal(X.end, Y)).

// ##### FLORA SPECIFIC #####
// Integrity Constraint to enforce signatures:
// If there is an instance X of Class, that has an Attribute Y and
// there is a signature definition corresponding to "Attribute" then
// Y has to be an instance of RangeofAttribute

// This is actually questionable. Should this treated relaxed or
// strict, i.e. is a fact invalid when it does not explicit state
// that one of the attribute values belongs to the coressponding range defintion
// or should it be infered that the attribute value is member of the range given in
the signature
//
Y:Range :-
  X:Class, X[Attribute->Y], Class[Attribute=>Range],
  (Class=date;Class=time;Class=dateAndTime).

// Integrity Constraint to invalidate instance not corresponding to a signature:
// If there is an instance X of Class, that has an Attribute Y and
// there is NO signature definition corresponding to Attribute then
// X is invalid
X[valid(Attribute)] :- X:Class, X[Attribute->_Value], Class[Attribute=>_Range].
invalid(X, Y) :-
  X:Class, X[Attribute->_Value], tnot X[valid(Attribute)],
  Y = 'no signature for ' + Attribute + ' in Class ' + Class.

```

```

invalid(X) :- invalid(X,_Y).

// REFERENCES
// [1] F. Pan and R. Hobbs: Time in OWL-S, available at:
http://www.isi.edu/~pan/damlttime/AAAI symp2004.pdf

truncate(X,Y) :- X:float, Y is truncate(X)@prolog().

//test instance:
//The current Date manually since we do not have build in function yet

currentDate:dateAndTime[
  date->date1:date[dayOfMonth->28,monthOfYear->2,year->2001],
  time->time1:time[hourOfDay->23,minuteOfHour->40,secondOfMinute->12]
].

currentDate2:dateAndTime[
  date->date2:date[dayOfMonth->29,monthOfYear->2,year->2001],
  time->time2:time[hourOfDay->23,minuteOfHour->40,secondOfMinute->00]
].

currentDate3:dateAndTime[
  date->date3:date[dayOfMonth->1,monthOfYear->3,year->2001],
  time->time3:time[hourOfDay->22,minuteOfHour->40,secondOfMinute->00]
].

currentDate4:dateAndTime[
  date->date4:date[dayOfMonth->2,monthOfYear->3,year->2001],
  time->time4:time[hourOfDay->23,minuteOfHour->40,secondOfMinute->12]
].

int1:interval[
  start->currentDate,
  end->currentDate4
].

int2:interval[
  start->currentDate2,
  end->currentDate3
].

```

Ontology 3: "Purchase Ontology"

```

/*****
* ONTOLOGY
* IDENTIFIER <po.flr> END
* NONFUNCTIONALPROPERTIES
* TITLE "Purchase Ontology" END
* CREATOR "Axel Polleres" END
* VERSION $Version$ END
* END
*****/

/*
At the current state this is an artificial ontology for modeling
purchases products and payment methods.
This should be aligned with existing ontologies in a later step:

IOTP ( The Internet Open Trading Protocol)
IOTP also has an additional supplement for modelling paymenti methods)
http://www.ietf.org/html.charters/trade-charter.html
Thi framework seems to be suitable for our needs in principal although
there does not seem to exist stable OWL or similar ontology formulations
of these ontologies

```

ebXML <http://www.ebxml.org> is another effort for aligning standards in the modelling of business processes, but we didn't find an ontology formulation of this standards either so far.

One suggestion by Dieter is to do an ontology based on EDI in the course of WSMO.

Integrating the views of these existing standards will be a tedious task and the suggested business models are quite complex. So, for the moment also in order to keep the usecase simple and readable, we decided to stick to this small ontology which obviously makes some simplifying assumptions but suffices our current needs. Redesign might be taken into account for further versions.
*/

```
// Concept and relation definitions
```

```
address[
  name    => string,
  street  => string,
  number  => integer,
  zipcode => string,
  city    => location,
  state   => state,
  country => country
].
```

```
// Concepts Country and Location to be used from tc-ontology
```

```
country[
  name=>string
].
```

```
location[
  name=>string,
  country=>country
].
```

```
state[
  name=>string,
  partOfCountry=>country
].
```

```
buyer[
  shipTo => address,
  billTo => address,
  // Note that the item price given by the buyer denotes a price *limit*
  purchaseIntention =>> tradeItem,
  acceptsPayment =>> paymentMethod
].
```

```
seller[
  address => address,
  saleIntention =>> tradeItem
].
```

```
tradeItem[
  product => product,
  price   => price
].
```

```
// This is the superclass for ticket:
```

```
product[
  name => string
].
```

```
price[
  // do we use float or real? shall we use xsd-datatypes?
  amount => float,
```

```

    currency => string
  ].

//paymentMethod[
// name => string
//].

// A trade is an actual agreement on trading items between two partners.
trade[
  items    =>> tradeItem,
  buyer    => buyer,
  seller    => seller,
  payment   => paymentMethod
].

// Do we need the delivery? Maybe in the effects.
delivery[
  products =>> product,
  receiver => buyer,
  sender   => seller
].

// Axioms: purchasing only possible if
// 1) Only products can be traded/delivered
//    if the Buyer requires them (purchaseIntention) and if the
//    seller provides them (saleIntention)

// note that in this notation the id is clearly outside the
// nonfunctional properties!

// 1. Only products can be traded
//    if the Buyer requires them (purchaseIntention) and if the
//    seller provides them (saleIntention)
//    and if the selling price is not higher than the price intended
//    by the buyer:

invalid(T) :- T:trade[items->>I], tnot T.seller[saleIntention->>I].
//T[invalid] :- T:trade[items->>I], tnot priceok(I).
priceAccepted(I) :- T:trade[items->>I], T.buyer[purchaseIntention->>I1],
                    I.product = I1.product ,
                    I.price.currency = I1.price.currency ,
                    I.price.amount =< I1.price.amount.

// 2. Only products can be delivered
//    if the receiver requires them (purchaseIntention) and if the
//    sender provides them (saleIntention)
invalid(D) :- D:delivery[products->>P],
              tnot D.receiver.purchaseIntention[product->P].
invalid(D) :- D:delivery[products->>P],
              D.receiver.saleIntention[product->P].

// below, some instances are defined that are needed throughout the use case
// modeling
// The axioms have to be changed
creditcard::paymentMethod.
creditcard[
  name      => string,
  number    => string,
  expmonth  => month,
  expyear   => year,
  // type is something like visa, masterCars, amex, diners, ...
  type      => string].

cash:paymentMethod.
cash[
  currency => currency].
// Concept currency not specified here.

```

```
// Some sample currencies:
euro:currency.
usd:currency.
gbp:currency.
```

Goal: "buying a train ticket from Innsbruck to Frankfurt on May 17th, 2004, starting at 6 p.m."

```
// define an instance of Goal
mygoal:goal.

// Goal Postcondition:
// a ticket for an itinerary from Innsbruck to Frankfurt
mygoal[postcondition->
  myTicket:ticket[
    itinerary -> _#:itinerary[
      startLocation -> innsbruckHbf,
      endLocation -> frankfurtHbf,
      departure -> _#:dateAndTime[
        date -> _#:date[
          dayOfMonth->17,
          monthOfYear->5,
          year->2004
        ],
        time.hourOfDay -> 18:hourOfDay
      ]
    ],
    traveller->_#:traveller[name -> 'Dieter Fensel']
  ]
].

// Goal Effect: a trade for the ticket, payed by creditcard
mygoal[effect->
  mytrade:trade[
    items ->> myTicket,
    buyer -> _#[
      shipTo -> myAddress,
      billTo -> myAddress
    ],
    payment-> myCreditCard:creditCard[
      name -> 'Dieter Fensel',
      number -> 1234567890,
      expMonth -> 9,
      expYear -> 2006,
      type -> 'MasterCard'
    ]
  ],
  myAddress:address[
    name -> 'Dieter Fensel',
    street -> 'Technikerstrasse',
    number -> 13,
    zipcode -> 6020,
    city -> innsbruck,
    state -> tirol,
    country -> austria
  ]
].
```

Web Service Capability: "selling international train tickets online for Austria and Germany"

```
// define an instance of Web Service Capability
```

```

oebbCap:capability.

// precondition: the input has to be an itinerary wherefore
// the start- and endlocation have to be in Austria or in Germany and
// the departure date has to be later than the current Date
oebbCap[precondition] :-
  _X:itinerary[
    startLocation -> StartLoc,
    endLocation -> EndLoc,
    departure -> Departure
  ],
  (StartLoc..locatedIn=austria ; StartLoc..locatedIn=germany),
  (EndLoc..locatedIn=austria ; EndLoc..locatedIn=germany),
  after (Departure,currentDate).

// assumption: there needs to be a buyer that accepts payment by creditcard
// and this creditcard has to be valid (not expired).
oebbCap[assumption] :-
  _X:buyer[
    acceptsPayment =>> Payment
  ],
  Payment:creditCard,
  (currentDate.date.year < Payment.expYear);
  ((currentDate.date.year = Payment.expYear),
  ((currentDate.date.monthOfYear < Payment.expMonth);
  (currentDate.date.monthOfYear = Payment.expMonth))).

// Postcondition: the output of the service is a ticket (train ticket)for an
itinerary wherefore
// the start- and endlocation have to be in Austria or in Germany and
// the departure date has to be later than the current Date
oebbCap[postcondition] :-
  _X:ticket[
    itinerary->_Itinerary:itinerary[
      startLocation -> StartLoc,
      endLocation -> EndLoc,
      departure -> Departure
    ]
  ],
  (StartLoc..locatedIn=austria ; StartLoc..locatedIn=germany),
  (EndLoc..locatedIn=austria ; EndLoc..locatedIn=germany),
  after (Departure,currentDate).

// Effect: there shall be a trade for the train ticket of the postcondition
oebbCap[effect] :-
  _Y:trade[
    items->> _Ticket:ticket[
      itinerary->_Itinerary:itinerary[
        startLocation-> StartLoc,
        endLocation-> EndLoc
      ]
    ],
    payment-> Payment
  ],
  Payment:creditCard,
  (StartLoc..locatedIn=austria ; StartLoc..locatedIn=germany),
  (EndLoc..locatedIn=austria ; EndLoc..locatedIn=germany).

```

For testing the Goal-Capability-Matching for this use case within FLORA-2, you have to download all the files above into a directory. Then, you have to set up the right environment for running the example; therefore we provide the following additional resources (please note that Flora is sometimes "buggy", and you should make sure

that all the resources are compiled with their latest status before you run the Goal-Capability-Matching program. It also sometimes happen that you do not receive the correct answer for some arbitrary runtime reasons)::

- [make.sh](#): this is a shell script that cleans the directory and pre-compiles all resources.

```
rm *.fld
rm *.P
rm *.xwam

runflora -e [tc]. -e flHalt.

runflora -e [dt]. -e flHalt.

runflora -e [po]. -e flHalt.

runflora -e [goal]. -e flHalt.

runflora -e [cp]. -e flHalt.

runflora -e [gcmatch].
```

- [flr.flr](#): FLORA-specific (checks signatures, etc.)

```
// ##### FLORA SPECIFIC #####
// Integrity Constraint to enforce signatures:
// If there is an instance X of Class, that has an Attribute Y and
// there is a signature definition corresponding to "Attribute" then
// Y has to be an instance of RangeofAttribute

// This is actually questionable. Should this treated relaxed or
// strict, i.e. is a fact invalid when it does not explicit state
// that one of the attribute values belongs to the coressponding range defintion
// or should it be infered that the attribute value is member of the range given in
the signature
//
Y:Range :-
  X:Class, X[Attribute->Y], Class[Attribute=>Range],
  (Class=date;Class=time;Class=dateAndTime).

// Integrity Constraint to invalidate instance not corresponding to a signature:
// If there is an instance X of Class, that has an Attribute Y and
// there is NO signature definition corresponding to Attribute then
// X is invalid
X[valid(Attribute)] :- X:Class, X[Attribute->_Value], Class[Attribute=>_Range].
invalid(X, Y) :-
  X:Class, X[Attribute->_Value], tnot X[valid(Attribute)],
  Y = 'no signature for ' + Attribute + ' in Class ' + Class.

invalid(X) :- invalid(X, _Y).
```

- [gcmatch.flr](#): this file includes all the resources and throws the query for Goal-Capability-Matching. The "X" returned is the identifier of the Capability that matches the Goal.

```
//includes flora specific stuff like signature checking
#include "flr.flr"
//including resources
#include "dt.flr"
#include "po.flr"
#include "tc.flr"
```

```
#include "cp.flr"
#include "goal.flr"

// queries for G-C-Matching
?- X:capability[postcondition], X:capability[effect].
```

- [otherCP.flr](#): this file contains other Capabilities for showcasing the functionality of Goal-Capability-Matching - one Capability that does not match and one Capability that "over-matches" the Goal.

```
// the following are different capability specifications
// for testing & showcasing Goal-Capability-Matching

// a test capability, will not be retrieved by gcmatch.flr
grCap:capability.

grCap[postcondition] :-
  _X:ticket[
    itinerary->_Itinerary:itinerary[
      startLocation -> greece,
      endLocation -> greece,
      departure -> Departure
    ]
  ],
  after(Departure,currentDate).

// a capability that "oversatisfies" the goal
// ews means "oiar legende Wollmilchsau"
ews:capability.

//I am selling any product
ews[postcondition] :-
  _X:product.
//I am doing evry trade
ews[effect] :-
  _X:trade.
```

We do not provide the Mediators as FLORA-2 resources for download because the connection facility has to be adopted to the Flora-module technology. The files below are simple text files that contain the listings defined in this document. A transformation script will be added in the future.

[OO Mediator 1: "Train Connection Ontology uses Date and Time Ontology"](#)

```
OO Mediator
  Train Connection Ontology uses Date and Time Ontology

non-functional Properties
  title
    Train Connection Ontology uses Date and Time Ontology
  creator
    DERI International
  subject
  description
    importing the Date and Time Ontology into the Train Connection Ontology
  publisher
    DERI International
  contributor
```

```

Michael Stollberg
date
  20040430
type
  WSMO OO Mediator
format
  text
identifier
  http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/VTA-OOM-
trainConnection.wsml
source
  http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/VTA-OOM-
trainConnection.wsml
language
  English
relation
  http://www.wsmo.org/2004/d3/d3.2/v0.1/20040517/resources/tc.flr
  http://www.wsmo.org/2004/d3/d3.2/v0.1/20040419/resources/dt.flr
coverage
rights
  DERI
version
  1.1

sourceComponent
  // identifier of "Date an Time Ontology"
  http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/dt.flr
  http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/po.flr

targetComponent
  // identifier of the "Train Connection Ontology"
  http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/tc.flr

mediationService
  // not needed here

```

OO Mediator 2: "Goal uses all ontologies"

```

OO Mediator
  OO Mediator for Goal

non-functional Properties
  title
    OO Mediator for Goal
  creator
    DERI International
  subject
  description
    importing all ontologies in to the Goal
  publisher
    DERI International
  contributor
    Michael Stollberg
  date
    20040430
  type
    WSMO OO Mediator
  format
    text
  identifier
    http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/VTA-OOM-Goal1.wsml
  source
    http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/VTA-OOM-Goal1.wsml
  language
    English
  relation
    http://www.wsmo.org/2004/d3/d3.2/v0.1/20040517/resources/tc.flr

```

```

    http://www.wsmo.org/2004/d3/d3.2/v0.1/20040419/resources/po.flr
coverage
rights
  DERI
version
  1.1

sourceComponent
  // identifiers of "Train Connection Ontology" and "Purchase Ontology"
  http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/tc.flr
  http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/po.flr

targetComponent
  // identifier of the Goal
  http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/goal.flr

mediationService
  // not needed here

```

OO Mediator 3: "Web Service Capability uses all ontologies"

```

OO Mediator
  OO Mediator for Web Service Capability

non-functional Properties
  title
    OO Mediator for Web Service Capability
  creator
    DERI International
  subject
  description
    importing all ontologies in to the Web Service Capability
  publisher
    DERI International
  contributor
    Michael Stollberg
  date
    20040430
  type
    WSMO OO Mediator
  format
    text
  identifier
    http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/VTA-OOM-WS1Cap.wsml
  source
    http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/VTA-OOM-WS1Cap.wsml
  language
    English
  relation
    http://www.wsmo.org/2004/d3/d3.2/v0.1/20040517/resources/tc.flr
    http://www.wsmo.org/2004/d3/d3.2/v0.1/20040419/resources/po.flr
  coverage
  rights
    DERI
  version
    1.1

sourceComponent
  comment: identifiers of "Train Connection Ontology" and "Purchase Ontology"
  http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/tc.flr
  http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/po.flr

targetComponent
  comment: identifier of the Web Service Capability
  http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/capability.flr

mediationService

```

comment: not needed here

WG Mediator: "connecting Goal and Capability"

```

WG Mediator
  connecting the Goal and the Web Service

non-functional Properties
  title
    Web Service - Goal connection Capability uses all ontologies
  creator
    DERI International
  subject
  description
    connecting the Goal and the Web Service Capability
  publisher
    DERI International
  contributor
    Michael Stollberg
  date
    20040430
  type
    WSMO OO Mediator
  format
    text
  identifier
    http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/VTA-WGM1.wsml
  source
    http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/VTA-WGM1.wsml
  language
    English
  relation
    http://www.wsmo.org/2004/d3/d3.2/v0.1/20040517/resources/capability.flr
    http://www.wsmo.org/2004/d3/d3.2/v0.1/20040419/resources/goal.flr
  coverage
  rights
    DERI
  version
    1.1

usedMediators
  comment: no additional Mediators needed

sourceComponent
  comment: identifier of the Goal
  http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/goal.flr

targetComponent
  comment: identifier of the Web Service
  http://www.wsmo.org/2004/d3/d3.2/v0.1/200400517/resources/ws.flr

reduction
  comment: to be specified

```