Dissertation:

ODEG – Ontology Driven E-Government
Combining MDA and Semantic Technologies to efficiently provide E-Government Services

Peter Salhofer

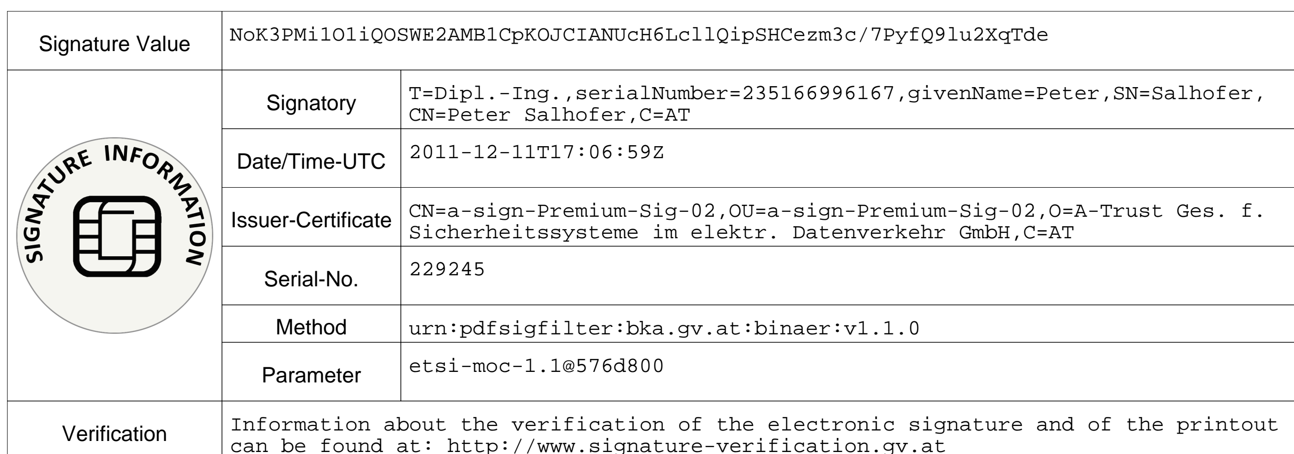
Supervisor: Prof. Dr. Reinhard Posch

Graz University of Technology,
Institute for Applied Information Processing and Communications

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, 11.12.2011



Signature Value	NoK3PMi1O1iQOSWE2AMB1CpKOJCIANUcH6Lcl1QipSHCezm3c/7PyfQ9lu2XqTde	
SIGNATURE INFORMATION	Signatory	T=Dipl.-Ing.,serialNumber=235166996167,givenName=Peter,SN=Salhofer, CN=Peter Salhofer,C=AT
	Date/Time-UTC	2011-12-11T17:06:59Z
	Issuer-Certificate	CN=a-sign-Premium-Sig-02,OU=a-sign-Premium-Sig-02,O=A-Trust Ges. f. Sicherheitssysteme im elektr. Datenverkehr GmbH,C=AT
	Serial-No.	229245
	Method	urn:pdfsigfilter:bka.gv.at:binaer:v1.1.0
	Parameter	etsi-moc-1.1@576d800
Verification	Information about the verification of the electronic signature and of the printout can be found at: http://www.signature-verification.gv.at	

Table of Contents

1 Motivation & Research Question ... 1
2 The Semantic Web ... 4
2.1 Ontologies ... 4
2.2 Open vs. Closed World Assumption ... 6
3 Semantic Web Technologies ... 7
3.1 Resource Description Framework – RDF ... 7
3.1.1 RDF Abstract Syntax ... 7
3.1.2 RDF XML Syntax ... 9
3.1.3 RDF Schema – The RDFS Vocabulary ... 11
3.1.3.1 RDFS Classes ... 11
3.1.3.2 RDF Properties ... 12
3.1.4 RDF Semantics ... 13
3.1.4.1 Interpretation ... 13
3.1.4.2 Entailment ... 15
3.1.4.3 RDF Vocabulary Interpretation ... 15
3.1.4.4 RDFS Interpretation ... 16
3.1.4.5 Entailment Rules ... 17
3.1.5 Conclusions ... 20
3.2 The Web Ontology Language (OWL) ... 20
3.2.1 SHOE ... 21
3.2.2 OIL ... 21
3.2.3 DAML ... 22
3.2.4 OWL Language Variants ... 22
3.2.5 Important OWL Constructs ... 23
3.2.5.1 OWL Classes ... 23
3.2.5.2 OWL Properties ... 24
3.2.5.3 Property Restrictions ... 25
3.2.6 Discussion ... 25
3.3 OWL 2 ... 26
3.3.1 Syntaxes ... 26
3.3.2 OWL 2 Features ... 26
3.3.2.1 Negative Property Assertions ... 26
3.3.2.2 Qualified Cardinality Restrictions ... 27
3.3.2.3 Property Chain Inclusion ... 27
3.3.2.4 Keys ... 27
3.3.3 OWL 2 Sub-Languages ... 28
3.3.3.1 OWL EL ... 28
3.3.3.2 OWL QL ... 28
3.3.3.3 OWL RL ... 28
3.3.4 Discussion ... 28
3.4 The Web Service Modeling Language WSML ... 29
3.4.1 WSML Syntax and Structure ... 30
3.4.2 WSML Semantics ... 31
3.4.2.1 WSML DL Extension ... 32
3.4.2.2 WSML Core, Flight and Rule Semantic ... 33
3.5 Comparing OWL and WSML ... 33
3.5.1 The WSML Solution ... 35
3.5.2 The OWL Solution ... 38
3.5.3 Comparison of Results ... 42
4 Semantic Web Services ... 43

4.1 Web Services ... 43
4.1.1 WSDL 1.1 ... 44
4.1.2 WSDL 2.0 ... 48
4.2 Semantic Markup for Web Services (OWL-S) ... 49
4.2.1 Service Profiles ... 51
4.2.2 Service Model ... 58
4.2.3 Service Grounding ... 61
4.3 Semantic Web Service Framework (SWSF) ... 62
4.4 Web Service Modelling Ontology (WSMO) ... 63
4.4.1 The WebService Element ... 64
4.4.2 The Goal Element ... 69
4.4.3 WSMO Grounding ... 72
4.5 Comparison ... 75
4.5.1 Goal based discovery ... 76
4.5.2 Service Choreography ... 78
4.5.3 Service Execution ... 79
4.5.4 Summary ... 79
5 Model Driven Architecture ... 79
5.1 Idea/Motivation ... 80
5.1.1 Computational Independent Model ... 81
5.1.2 Platform Independent Model ... 82
5.1.3 Platform Specific Model ... 83
5.1.4 Model Transformation ... 83
5.2 Meta Object Facility (MOF) ... 85
5.3 Object Constraint Language (OCL) ... 88
5.3.1 Invariants ... 88
5.3.2 Pre- and Postconditions ... 89
5.3.3 Initial and Derived Values ... 90
5.3.4 Operation Body Expressions ... 90
5.4 Ontology Definition Metamodel (ODM) ... 93
5.5 Discussion ... 94
6 Ontology Modelling ... 96
6.1 General Ontology Modelling Guidelines ... 96
6.2 Governance Enterprise Architecture (GEA) ... 97
6.2.1 GEA Object Model for Service Provisioning ... 97
6.3 Discussion ... 99
7 Ontology Driven E-Government ... 99
7.1 Initial Feasibility Study ... 101
7.1.1 Prototype Requirements and Example Scenario ... 101
7.1.2 Semantic Service Model and Ontologies ... 101
7.1.3 Generating Forms to Access the Permanent Parking Permit Service ... 103
7.1.4 Lessons Learned ... 105
7.2 Technology Selection ... 106
7.3 Meta-Model ... 106
7.3.1 How to create the ODEG meta-model ... 107
7.3.2 WMSO-PA – An WSMO implementation of GEA-PA ... 107
7.3.3 GEA-SeGoF – Specialising WSMO/GEA-PA ... 111
7.3.4 PersonData Ontology ... 114
7.4 Service Locator ... 115
7.4.1 Selecting a Desire ... 117
7.4.2 Refining a Desire ... 119
7.4.3 The Service Finding Algorithm ... 125
7.5 Semantic Forms ... 126
7.5.1 Determining Required Service Input ... 126

7.5.2 Rendering the Electronic Forms 128
7.5.3 Marking the Model 134
7.6 Auxiliary Service Modelling 140
7.6.1 The Auxiliary Service Ontology 141
7.6.2 Implementing Auxiliary Services 144
7.6.3 Enabling Auxiliary Services 146
7.7 WSDL and XSD Generation 149
7.7.1 Converting Ontologies to XML Schema 149
7.7.2 Generation of WSDL Files 156
7.8 Implementing ODEG web services 161
7.9 The Big Picture 166
8 Related Work 167
8.1 Goal Oriented Discovery for Semantic Web Service 167
8.2 Domain Knowledge-Based Automatic Workflow Generation 167
8.3 SemanticGov 169
8.4 TerreGov 172
8.5 SUPER - Semantics Utilized for Process management within and between Enterprises 173
8.6 Access-eGov 174
9 Conclusion & Outlook 178

List of relevant Publications

All papers listed represent parts of this dissertation that have already been published and are accepted submissions to international conferences, journals and books that have all undergone a qualified peer review process.

- Bernd Stadlhofer and Peter Salhofer„„ Automatic Generation of E-Government Forms from Semantic Descriptions" in *Proceedings of the 1st International Conference on Theory and Practice of Electronic Governance*, Macao, China, pp 12-19, 10-13 December 2007

  This paper presents the approach that was chosen for the prototypic implementation. The main author was also the author of the diploma thesis. My contribution was to define the structure of the paper and proofreading.
  All other papers were entirely written by me. The co-authors are members of the ODEG software development team and were responsible for providing screen-shots and proofreading.

- Peter Salhofer and Bernd Stadlhofer, "e-Government Service Discovery based on Citizens' Desires" in *Proceedings of the 4th International Conference on e-Government*, RMIT University Melbourne, Australia, pp 371-380, 23-24 October 2008

  This paper describes the algorithm behind the service finder component.

- Peter Salhofer, Gerald Tretter and Bernd Stadlhofer "Goal-Oriented Service Selection" in *Proceedings of the 2nd International Conference on Theory and Practice of Electronic Governance*, Cairo, Egypt, pp 60-66, 1-4 December 2008

  In this paper the overall service identification approach was presented.

- Peter Salhofer, Bernd Stadlhofer, "Ontology Modeling for Goal Driven E-Government" in *Proceedings of the 42nd Hawaii International Conference on System Sciences, HICCS 42*, 5-8 January 2009, Big Island Hawaii, USA, IEEE, pp 1-9, 2009

  This paper focuses on the meta-model used for the service identification component.

- Peter Salhofer, Bernd Stadlhofer, Gerald Tretter and Barbara Meyer, "www.SeGoF.org - Semantic E-Government Forms" in *Proceedings of Ongoing Research, General Development Issues and Projects of EGOV 09, 8th International Conference*, Linz, August 31 - September 5, 2009, pp 289-296

  This paper describes the form generation process.

- Peter Salhofer, Bernd Stadlhofer and Gerald Tretter, "Ontology Driven E-Government" in *Politics, Democracy and E-Government: Participation and Service Delivery* (Reddick, Ch., Ed.), Information Science Reference, April 2010, pp 383-401

  This article outlines the idea behind ODEG.

- Peter Salhofer, Bernd Stadlhofer, "Knowledge-first Web Services - An E-Government Example" in *Proceedings of the 6th International Conference on Networked Computing and Advanced Information Management, NCM2010*, August 16-18 2010, Seoul, Korea, pp 218-223

  The focus of this paper lies on ODEG's support for creating web-services based on semantic models.

- Peter Salhofer, Bernd Stadlhofer, “Semantic MDA for E-Government Service Development “ in *Proceedings of the 45th Hawaii International Conference on System Sciences, HICCS 45*, 4-7 January 2012, Grand Wailea, Maui, USA, IEEE [to be published in 2012]

  This paper represents a short version of the entire dissertation and covers almost all aspects of the approach presented here.

Abstract

This paper describes a new comprehensive approach to the creation of public electronic services called Ontology Driven E-Government (ODEG). The concepts applied incorporate principles from Model Driven Architecture (MDA) and Semantic Web Services (SWS). Both of these techniques have been intensively hyped but for the time being could not keep up with the expectations. By combining these two approaches, a new way to create E-Government services could be established that tries to overcome the disadvantages of either of the underlying technologies. MDA, a software engineering practice, aims at reducing system development effort by automatically transforming models of systems into running applications. Semantic Web Services provide machine interpretable descriptions of web services that allow so called software agents to automatically and autonomously achieve specific goals for their users. The very principle of the approach presented here is to provide a running system based on a model expressed by semantic technologies. Whereas classical MDA uses models based on the Unified Modelling Language (UML), ODEG uses ontologies. The clear focus on E-Government allows for a well-defined and simple meta-model that can be used as a scaffold for new public services. The resulting system model is directly interpreted by a runtime environment. An automatic semantic reasoner represents the core of this system allowing to utilise the immense expressiveness of semantic technologies not only during design time but also during the execution of the system. This allows for a whole new quality of electronic services. Semantic descriptions of public procedures can be used to identify relevant services in a new intuitive way that solely focuses on the citizen's point of view. Since every service holds a description of the information that is necessary in order to consume it, the system interactively gathers this information whenever a service is about to be used. Since the input to a service is represented by a concept model that makes up the specific application domain of the service, the information elicitation process automatically adapts to the specific situation of the current user. This leads to electronic dialogues that are dynamically rendered based on concepts that reflect the current need of the citizens and only contain information that is specific and relevant. Since the model is interpreted every change to the model will show immediate effect in the system. Thus, changes to the system can be easily achieved by simple modifying the model without any programming effort at all. Besides supporting service identification and service utilisation ODEG also supports the implementation of the actual electronic procedures. Therefore ODEG automatically generates a WSDL description for every single electronic service together with all XML datatypes needed. This service interface can be used to model a BPEL process that executes the procedure in a service oriented environment.

Illustration Index

Figure 1: Schematic Overview of Ontology Driven E-Government (own illustration) 3
Figure 2: The Semantic Web Tower ([23], Copyright © 2000 World Wide Web Consortium. All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231) 7
Figure 3: Graphical Representation of an RDF-Triple ([28] Copyright © 2004 World Wide Web Consortium. All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231) 8
Figure 4: RDF graph presenting the situation described in Listing 3 (own illustration). 9
Figure 5: OWL and its predecessors (own illustration based on [36]) 21
Figure 6: WSML variants [73]... 29
Figure 7: Sample scenario travel agency business (own illustration) 34
Figure 8: WSML travel agency example ontology. Screenshot from WSMO Toolkit 35
Figure 9: OWL 2 travel agency example ontology. Protegé screenshot. 38
Figure 10: A Web Service sample Scenario (own illustration) 44
Figure 11: WSDL 1.1 Structure (own illustration) 45
Figure 12: Top-level description elements in WSDL 2.0 (own illustration) 48
Figure 13: OWL-S top-level classes ([96], Copyright © 2004 World Wide Web Consortium. All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231) 50
Figure 14: The OWL-S ServiceProfile ([96], Copyright © 2004 World Wide Web Consortium. All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231 50
Figure 15: The OWL-S variable hierarchy (own illustration, restored from http://www.ai.sri.com/daml/services/owl-s/1.2/Process.owl using Protegé). 51
Figure 16: Definition of input variables in a service profile (own illustration) 52
Figure 17: Class hierarchy to model preconditions in OWL-S (own illustration, extracted from http://www.daml.org/services/owl-s/1.2/generic/Expression.owl using Protogè) 53
Figure 18: RDF-graph-like structure of the Result class an its properties (own illustration based on [97]) 55
Figure 19: The top level OWL-S process ontology[97]. 58
Figure 20: Basic OWL-S classes needed to model a composite process (own illustration based on [97]). 59
Figure 21: Data binding classes to define the input of component processes (own illustration constructed from http://www.ai.sri.com/daml/services/owl-s/1.2/Process.owl). 59
Figure 22: Required elements to model OWL-S producer-push scenarios within a composite process (own illustration based on [97]). 60
Figure 23: Example composite process reconstructed from http://www.ai.sri.com/daml/services/owl-s/1.2/BravoAirProcess.owl (own illustration) 61
Figure 24: OWL-S classes to model WSDL service grounding (extracted from http://www.ai.sri.com/daml/services/owl-s/1.2/Grounding.owl, own illustration) 62
Figure 25: The WSMO approach to describe a web service's functional aspects (Reprinted from [113], page 87 with permission from IOS Press) 66
Figure 26: WSMO Goal Model Overview[122].. 70
Figure 27: WSMO data grounding approaches[125] 72
Figure 28: MDA's different model levels (own illustration) 80
Figure 29: MDA model taxonomy ([141] page 193) 81
Figure 30: The Waterfall Model in Software Engineering (own illustration based on [147]) 83
Figure 31: MDA Transformation Paths (own illustration) 84
Figure 32: MDA Transformation Process based on Marking ([136], p. 3-8) 84
Figure 33: Mapping of metamodels allows for direct model-to-model transformation ([136],p. 3-9) 85
Figure 34: The MOF metalevels (own illustration) 86
Figure 35: Example of four-layer metamodel hierarchy ([152],page 19) 86
Figure 36: UML Infrastructure Library: Class definition ([152], page 93) 87
Figure 37: UML model of the sample scenario that is the basis for Listing 50 (own illustration created with ArgoUML) 90
Figure 38: ODM Metamodel of an RDF triple ([156],page 35) 93
Figure 39: ODM metamodel of the OWL class ([156],page 69) 94

Figure 40: The GEA detailed object model for service provision [175] 98
Figure 41: A typical ODEG usage scenario (own illustration) 100
Figure 42: Part of the prototype's service model. The general model is shown on the left and the description of the permanent parking permit service on the right (own illustration based on [177]). 102
Figure 43: Fragment of the prototype's domain model (own illustration) 102
Figure 44: Introducing constraints by subclassing existing class with restricted properties (own illustration) 103
Figure 45: Overview of the prototype's modelling and generation process (adapted from [177], page 63) 105
Figure 46: Overview of framework provided ontologies and service specific ontologies (own illustration) 107
Figure 47: Concept hierarchy of WSMO-PA societal entities and their relations to corresponding PROTON Top module concepts (own illustration) 108
Figure 48: Part of the WSMO-PA ontology representing ServiceOutcome and ServiceInput concepts (own illustration) 109
Figure 49: Sample usage scenario of the ValidConcept service input type (own illustration) 110
Figure 50: The ServiceProcess concept and its relation to other elements (own illustration) 110
Figure 51: The ODEG specific specialisation of WSMO-PA called GEA-SeGoF ontology (own illustration) . 111
Figure 52: Concept hierarchy describing different possible implementation types of public services (own illustration) 112
Figure 53: Main concepts and structure of the PersonData ontology (own illustration) 114
Figure 54: The Desire concept and its related concepts (own illustration) 116
Figure 55: Definition of the pull down permit service and its relations to a desire and its service constraint (own illustration) 117
Figure 56: Start page of the service locator showing a selection of available desires (screen shot of the service finder application). 117
Figure 57: A fragment of the construction ontology showing parts of the construction taxonomy (own illustration) 119
Figure 58: Specialisation as one way to refine a desire (screen shot from the service finder application) 120
Figure 59: The PullDownRelevent concept and its direct sub-concept (screen shot of the WSMO Visualizer) 120
Figure 60: Dialogue to further specify the type of a residential house (screenshot of the service finder application) 122
Figure 61: Specifying the location of the pull down activity (screen shot of the service finder application) ... 122
Figure 62: Result of the service finding process (screen shot of the service finder application) 123
Figure 63: General overview of the desire refinement process (own illustration) 124
Figure 64: Schematic overview of the service matching step when looking up relevant services (own illustration) 125
Figure 65: Initial screen of the building permit application without pre-filled instances. 128
Figure 66: Initial building permit application form with data transferred from the service finder component . 128
Figure 67: Specialisation of the person concept assigned to the application property 129
Figure 68: Form used to collect information about the applicant 129
Figure 69: Input form for specifying the properties of a physical person applying for a building permit 130
Figure 70: Example of an error message caused by a model constraint that was not met 131
Figure 71: Example of registering several degrees 132
Figure 72: Definition of the applicant property after personal data was successfully collected. 132
Figure 73: Refinement path consisting of attribute value specification and classification/ specialisation (own illustration) 133
Figure 74: Part of the final overview that allows the user to review the application before it is actually submitted 134
Figure 75: Dialogue asking the current user to select different types of activities that will be offered via the new business that is about to be registered (currently available in German only). 137
Figure 76: Automatically generated form to specify a garage 138
Figure 77: Concept graph representing the path of a default value for the municipality attribute 139
Figure 78: Screenshot showing the effect of a help text 140
Figure 79: Different auxiliary services as defined in the auxiliary service ontology 141

Figure 80: A small part of the Austrian administration hierarchy according to the GEA meta-model 142
Figure 81: Screenshot of a form that uses the street name provider service 144
Figure 82: The list of values created for the disctrictCadastre property of the PieceOfLand concept. 147
Figure 83: Screenshot of the input form used to get values for an instance of type PieceOfLand. 148
Figure 84: An instance of PieceOfLand was successfully added to the application 149
Figure 85: XSD type hierarchy of the GasFiringInstallation type. 153
Figure 86: Resolving type conflicts be re-arranging of properties (own illustration) 156
Figure 87: Schematic overview of the WSDL creation process based on the building permit application example (own illustration)... 157
Figure 88: Schematic view on services and underlying processes (own illustration). 161
Figure 89: Outside view on a BPEL process and its partner links (Screenshot of the Business Registration Process opened in the Netbeans SOA Module's CASA viewer) 162
Figure 90: Schematic overview of the business registration BPEL process (own illustration) 163
Figure 91: Snippet of the business registration BPEL process showing the creation of a new file (screenshot from Netbeans BPEL Designer)... 164
Figure 92: Mapping of part of the message necessary to create a new file 164
Figure 93: Sample JBI components configuration (own illustration) 165
Figure 94: ODEG structural overview (own illustration).. 166
Figure 95: Example service component hierarchy used by the automatic workflow generation approach ([210], page 4)... 168
Figure 96: Regulation ontology used for business registration process ([210], page 5) 168
Figure 97: A blank user profile ([210], page 9).. 169
Figure 98: Composing public services based on existing service operations ([211],page 37) 170
Figure 99: A sample goal tree used for the so called Greek Naturalization Service ([212],page 4) 171
Figure 100: The TerreGov eGovernment Interoperability Centre Platform(EGIC, [213], page 5) 172
Figure 101: Business process composition based on semantic annotations of services and processes ([215], page 43)... 173
Figure 102: The SUPER architecture ([219], page 7) 174
Figure 103: Architecture of the Access-eGov platform ([224], page 3) 177

1 Motivation & Research Question

Currently public agencies are facing significant budget cuts. As a result they have to make sure that scarce resources are used most effectively and efficiently. In the field of information and communication technologies (ICT) this means that every new investment has to keep its value for the organisation for as long as possible, which requires systems and infrastructures that are capable of being adapted to steadily shifting requirements at low costs and with almost no effort. This will make sure that systems keep their benefit for the organisation. All these requirements also apply for E-Government services offered to citizens. Thus, the questions arises how electronic public services can be created and provided at high quality with low effort and in a way that facilitates maintenance. To answer this question firstly the term quality has to be characterised in the context of E-Government.

According to [1] E-Government is the execution of business processes that involve public agencies by the means of ICT and electronic media. This covers a range of qualitatively different efforts, starting from offering information about public services on public agencies' web sites and ending at fully transactional services that are conducted entirely electronically. To asses these different levels of service sophistication, several classification schemes exist [2][3][4]. Usually they distinguish between four different levels or stages similar to these defined by Layne and Lee [2]:

1. Catalogue (Online presence, catalogue presentation, downloadable forms)
2. Transaction (Services and forms on-line, working database supporting online transactions)
3. Vertical Integration (Local systems linked to higher level systems within similar functionalities)
4. Horizontal Integration (Systems integrated across different functions, real one stop shopping for citizens)

Probably the most interesting aspect of this service sophistication model is, that the availability and integration of electronic services on different governmental levels is a prerequisite to reach the final stage. As a consequence, the evolvement of electronic service maturity at lower level Governments (e.g. Municipality level) directly depends on the service maturity at higher level Governments (e.g. Provincial or Federal Governments). Thus, every E-Government environment that strives for a high service maturity level has to support the integration of different kinds of external services as well. This is why technologies that facilitate the integration of services across system boundaries play a central role in E-Government. Generally, high quality E-Government services are showing a high degree of integration, which allows them to offer comprehensive services via a single entry point.

The next question is how can these services be created with minimal effort?

In the software engineering domain exist several approaches that try to reduce the development effort for new systems. One of these approaches is known under the term agile development. This encompasses a set of methodologies, technologies and tools that are emphasising at the fast production of code and thereby trying to minimise the need for analysis, design and documentation tasks [1]. In relative short development iterations more and more parts of the system are implemented in close cooperation with the customer. This ensures compliance with user requirements and early availability of deliverables.

Model Driven Architecture (MDA) [5] also aims at minimising development time and effort but its basic idea is almost the opposite of the one behind agile methods. Instead of focusing on code production, it promotes thorough analysis and comprehensive systems models. These models should be automatically converted into running applications by code generators and can be reused for different platforms or programming languages. To facilitate re-usability of models, there exist different layers of abstraction within the MDA based on an approach called Meta Object Facility (MOF) [6]. At least theoretically, applying changes to the system is about changing the model and re-generation of the application. Thus, the MDA seems to be good choice for a methodology that allows for the fast creation of E-Government services that can also easily be modified.

1 http://www.agilemanifesto.org/

Eventually the technology stack and the overall system architecture paradigm need to be defined.

As already pointed out at the beginning of this chapter, the ability to easily integrate other services regardless whether they are provided within the same organisation or by any other governmental organisation is key to reach high service maturity. Thus the technologies selected should facility integration of services even across organisational boundaries.

One very popular technical approach to enable the integration of different services, regardless whether they are offered at different governmental levels or not, is the use of web services [7]. Web service technology can be used to offer services as a set of operations to arbitrary or distinct service consumers. One big advantage of web services is their platform and programming language independency. This characteristic makes them a first class candidate for every type of system integration. To facilitate the use of web services by various types of clients, they typically come with a detailed technical description. This description is an XML file expressed in the so called Web Service Description Language (WSDL) [8]. WSDL files contain the definition of all operations that are offered by a particular service as well as the structure of input messages consumed and output messages returned by them. Since this description is text it can be interpreted on any platform and it is also detailed enough to automatically generate client code that can be used to access a specific service. Overall system architectures that are based on such web services are known under the term Service Oriented Architecture (SOA). This represents an approach that allows for the aggregation of otherwise de-coupled services to more complex processes. Due to the minimal coupling between individual services a software system that provides particular services can be replaced without causing an enormous impact on the rest of the ICT landscape. This facilitates the adoption of new technologies and reduces the risk of vendor lock-in situations. As a result SOA and web services seem to be an appropriate architectural choice also for public agencies. So, web services are one excellent option to offer E-Government services at high maturity levels.

However, when offering a (large) set of services there has to exist some mean to lookup appropriate services that are suited to support citizens or businesses. Although there exists a mechanism that is called UDDI (Universal description, discovery, and integration) [9] to facilitate the process of identifying and locating web services that are offered on the web, it is hardly adopted anywhere. So called semantic web services [10], however, address the problem of service discovery by introducing an additional semantic layer to existing web services. While web services already include comprehensive descriptions of their technical structure (syntax), additional semantic markup is needed to make them machine interpretable (semantic). This for example allows so called software agents [11] to understand what a service does and what input data is required to use it. Therefore semantic web services describe the so called IOPEs (Inputs, Outputs, Preconditions and Effects) [12] of a service. Inputs and outputs are closely related to the input and output datatypes of the actual web service, but refer to logical concepts that are represented by the datatypes used. The set of preconditions describes the status of the world that has to be true in order to correctly use the service. Effects describe how the service might change the status of the world and what is returned by the service. This additional logical description is created by means of ontologies (see section 2.1). Based on the availability of this information, appropriate services that serve a specific task or cause a specific effect can be located and executed. Since every semantic web service contains a description of necessary preconditions as well as its outcome, this allows for intelligent software agents that figure out a combination or orchestration of several services to achieve even more complex goals.

Thus, while web services are apt to integrate E-Government services and processes that might span several agencies, capabilities of semantic web services go far beyond this. The ability to link them to specific intentions or goals together with their machine understandable nature could heavily facilitate identification and localisation of appropriate public services. This, however, requires public services to be augmented with semantic annotations as well as the creation of suitable user interfaces, since (semantic) web services are primarily designed for machine-to-machine communication. Besides this, semantic annotations require the creation of properly defined knowledge bases, so called ontologies. Since ontologies have to capture the knowledge of a given domain, they can become very expressive but also probably complex, depending on the chosen semantic framework and language. There are currently several different semantic modelling languages available that could be used to describe semantic web services. The two most important ones are OWL [13] and WSML [14]. Each of them exists in different variants, reflecting various trade-offs between expressiveness and decidability resulting in different levels of complexity.

Despite their inherent complexity semantic web services are a great option for implementing E-Government services, since the semantic description can be used to intelligently identify relevant services as well as for their utilisation. Thus the question arises how the additional effort of creating a semantic knowledge base can be compensated by the overall development process for new E-Government services that are implemented as semantic web services?

The basic idea of the work presented here is to merge the concepts of MDA and semantic web services and to apply them to the generation of new E-Government services with minimal effort. The research question is, whether there exists a way to define semantic models for specific E-Government services that can be used to efficiently generate executable on-line services that can easily be found, accessed and used by citizens (see Figure 1). This would incorporate the advantages of both approaches; services that can be found based on formally expressed semantic goals as well as minimised effort to implement such solutions. Services can be developed rapidly, can be easily adapted and changes will have minimal effects on other parts of the ICT infrastructure. This new approach to the creation of E-Government services is called Ontology Driven E-Government (ODEG).

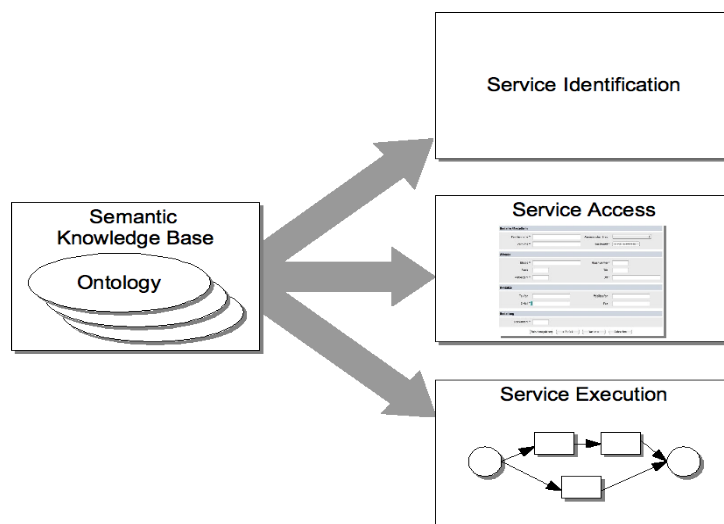


Figure 1: Schematic Overview of Ontology Driven E-Government (own illustration)

To show how ODEG can be designed to keep up with the aforementioned expectations, all relevant aspects and possible solution scenarios have to be investigated and will be covered in the rest of this work. In chapter 2 the basics of the semantic web together with a definition of the term ontology is given. To select an appropriate semantic framework for the proposed solution it is necessary to know the capabilities of ontologies in general and of existing candidate frameworks in particular. That is why chapter 3 provides a presentation of some proposed standards for semantic modelling languages and compares their features to support the decision to select one of them for use in the ODEG approach. Chapter 4 discusses the characteristics of semantic web services and compares some existing frameworks. The necessary basics of Model Driven Architecture and its various modelling levels are presented and discussed in chapter 5. Since semantic modelling languages typically allow several ways to express identical situations, it seems necessary to establish some modelling guidelines that reflect best practices and serve as a reference. Recommendations about ontology modelling in general as well as a proposed reference model for the E-Government domain are therefore presented in chapter 6. The actual approach to generate executable E-Government services based on semantic models is presented in chapter 7, whereas similar approaches and their differences to the presented one are discussed in chapter 8. Finally some essential conclusions from the presented results are drawn and possible directions for future development are highlighted.

2 The Semantic Web

"The Semantic Web is a web of data, in some ways like a global database." [15]

This was the initial definition of the semantic web given by Tim Berners-Lee one of the inventors of the Internet back in 1998. The emphasis of his paper was clearly on improving the relevance of search results by adding semantic annotations to web pages. This should enable search engines to understand the logical context of web content. In the following years that were also characterised by the introduction of web services, the understanding of the semantic web had obviously shifted. In 2001 a new vision of "The Semantic Web" was published [16]. Today this article is considered to be the hour of birth of the semantic web and describes a fictional scenario in which so called agents are able to solve rather complex tasks autonomously. The Semantic Web was no longer a global database but a repository of services that can be discovered, understood and utilised by software agents that try to achieve the goals of their human masters. One of the elements that are needed to add the necessary amount of logic to facilitate this vision are ontologies.

2.1 Ontologies

Ontologies are the basic elements of semantic systems since they describe the semantic aspects of any given domain. There are numerous definitions of the term ontology available. One that is very frequently cited, is the one by Thomas Gruber:

"An ontology is an explicit specification of a conceptualization" [17]

By citing [18] he also explains, that any approach of representing knowledge has to be based on conceptualisation, which in turn is a collection of "objects, concepts, and other entities that are assumed to exist in some area of interest and the relationships that hold among them". This makes a conceptualisation a simplified and abstract representation of the part of the world that should be modelled. All needed elements are explicitly specified by means of a representational vocabulary, thus leading to the more precise definition:

"In such an ontology, definitions associate the names of entities in the universe of discourse (e.g., classes, relations, functions, or other objects) with human-readable text describing what the names mean, and formal axioms that constrain the interpretation and well-formed use of these terms. Formally, an ontology is the statement of a logical theory." [17]

In a more recent article Gruber refines this definition and provides slightly different explanations depending on the context in which an ontology is used. For the context of computer and information sciences his definition is:

"...an ontology defines a set of representational primitives with which to model a domain of knowledge or discourse. The representational primitives are typically classes (or sets), attributes (or properties), and relationships (or relations among class members). The definitions of the representational primitives include information about their meaning and constraints on their logically consistent application." [19]

In this article Gruber also argues, that the most important reason why ontologies are considered to be at the "semantic" level rather than at the "logical" level is their expressive power when it comes to logical constraints. This expressiveness comes close to first-order logic.
A similar but rather pragmatic definition can be found in [11]:

"I define ontology as a set of knowledge terms, including the vocabulary, the semantic interconnections, and some simple rules of inference and logic for some particular topic"

A more formal definition that further refines the previous description can be found in [20]:

Definition 1: An ontology with datatypes is a structure
$O := (C, T, \leq_C, R, A, \sigma_R, \sigma_A, \leq_R, \leq_A, I, V, \iota_C, \iota_T, \iota_R, \iota_A)$ consisting of

- six disjoint sets C, T, R, A, I and V called concepts, datatypes, relations, attributes, instances and data values,
- partial orders $\leq_C$ on C called concept hierarchy or taxonomy and $\leq_T$ on T called type hierarchy,
- functions $\sigma_R : R \rightarrow C^2$ called relation signature and $\sigma_A : A \rightarrow C \times T$ called attribute signature,
- partial orders $\leq_R$ on R called relation hierarchy and $\leq_A$ on A called attribute hierarchy, respectively,
- a function $\iota_C : C \rightarrow 2^I$ called concept instantiation,
- a function $\iota_T : T \rightarrow 2^V$ called datatype instantiation,
- a function $\iota_R : R \rightarrow 2^{I \times I}$ called relation instantiation,
- a function $\iota_A : A \rightarrow 2^{I \times V}$ called attribute instantiation.

Here is a short example that will point out the meaning of the different elements used in this definition. Assume there is a simple ontology $O_{driving} := (C, T, \leq_C, R, A, \sigma_R, \sigma_A, \leq_R, \leq_A, I, V, \iota_C, \iota_T, \iota_R, \iota_A)$ that models certain aspects in the field of individual mobility where:

C={Thing, Person, Car, Driver, Drivinglicense}, T={String, Date},
≤C={(Thing, Person), (Thing, Car), (Thing, Drivinglicense), (Person, Driver)},
R={hasOwner,belongsTo}, A={hasName, hasExpirationDate},
σR={(hasOwner,(Car, Person)), (belongsTo,(Drivinglicense,Person))},
σA={(hasName,(Person,String)), (hasExpirationDate,(Drivinglicense,Date))},
≤R={}, ≤A={}, I={JohnFoo, BMW320, CarDrivingLicense4711},
V={"John Foo", 31-12-2015},
ιC={(Person, {JohnFoo}), (Car, {BMW320}),
(DrivingLicense,{CarDrivingLicense4711})},
ιT={(String,{"John Foo"}), (Date, {31-12-2015})},
ιR={(hasOwner,{(BMW320, JohnFoo)}),
(belongsTo, {(CarDrivingLicense4711, JohnFoo)})}
ιA={(hasName,{(JohnFoo, "John Foo")}),
(hasExpirationDate, {(CarDrivingLicense4711, 31-12-2005)})}

This formal definition covers most of the aspects that are mentioned in the other definitions above. There are classes, attributes, relationships among them and a vocabulary. The mapping between attributes and classes as well as the instantiation methods impose some constraints on the model that restrict the creation of valid elements.

Extracting the commonalities of the mentioned definitions leads to the following common characteristics of ontologies:

- Ontologies contain abstractions of things in a particular domain, called classes or concepts.
- These concepts are expressed in a strictly formal language. Their description might include attributes that in turn might be of a particular datatype.
- Concepts in an ontology form a taxonomy and might show addition relationships.

- Axioms can be used to further restrict the use of concepts.

2.2 Open vs. Closed World Assumption

It is important to know that most ontology modelling frameworks use the so called open world assumption (OWA) [21], which basically assumes that the knowledge represented in a model is never complete. In the context of creating an ontology, which is a model of some part of the world, these approach seems to be intuitive, since such models can hardly be complete especially if there exist references to other domains as well.

The practical consequence of this approach is, that if an assumption can not be explicitly inferred to be wrong based on the existing model, it can't be decided at all, thus the answer is unknown.

Lets assume that an ontology contains the following fact:

"Vienna" isCapitalOf "Austria" .

If we would ask a reasoner based on the open world assumption whether Berlin is the capital of Austria, the answer would be "unknown", since, unless there is any rule saying that Berlin is not the capital of Austria, this fact can't be inferred from the given information. In a system using the closed world assumption (CWA), the answer would be "false", since everything that is not known is assumed to be false. This behaviour is also known as "Negation as Failure" [22]. A fact is considered to be false if every possible proof of this fact fails. This leads to the important difference that OWA ontologies include restrictions to the world, whereas closed world systems define everything that is possible. It is also important to notice that OWA systems are monotonic. This means, that already made decisions (i.e. a result that is either false or true) will not change when additional information is added to the system:

If we would ask whether Berlin is the capital of Germany and this fact is not deducible from the given knowledge base an OWA system would respond "unknown", whereas a CWA system would answer "false". As soon as a new fact, saying that Berlin is the capital of Germany, would be added to the ontology, the OWA system would find an answer ("true"), but the CWA system would have to change it's previous answer to "true". Thus CWA systems behave non-monotonic.

The open world assumption, however, has also significant influence on modelling constraints that might limit its use in classical systems engineering. Assume the following example:

The model of a flight reservation system contains some notion of `flight` and also a notion of `seats`. Furthermore there is a cardinality constraint, limiting the number of `persons` that can be assigned to a `seat` on a particular `flight` to *one*. Now the following situation occurs:

"Seat_10A" isReservedBy "Mrs. Miller" .

"Seat_10A" isReservedBy "Mrs. Johns" .

Any CWA system would immediately detect the inconsistency that there are two persons assigned to one seat which contradicts the cardinality restriction. In an OWA system this potential conflict would be resolved by inferring that Mrs. Miller and Mrs. Johns are actually the same person. This conclusion is made possible, since OWA systems do not use the unique name assumption (UNA) either.

Even though the open world assumption seems to be most appropriate for the basic idea of modelling ontologies, it shows some critical drawbacks in the field of systems engineering. Since in the proposed approach to the model driven generation of E-Government services needs to model constraints as well, the closed world assumption seems to be favourable.

3 Semantic Web Technologies

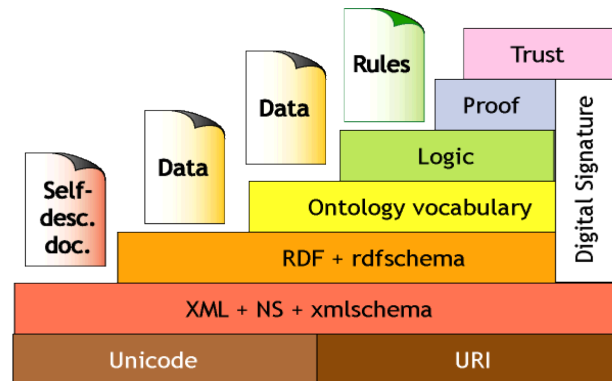


Figure 2: The Semantic Web Tower ([23], Copyright © 2000 World Wide Web Consortium. All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231)

According to Tim Berners-Lee [23], the semantic web is based on a stack of technologies (Figure 2). The base of this semantic web tower is formed by Uniform Resource Identifiers (URI) [24] that uniquely identify resources and the Unicode standard [25] that allows to represent and share information in any language. The next layer is formed by XML [26] which defines the notation used by the successive layers. This chapter will give an overview about several semantic technologies that could be used within this proposed architecture. The presented technologies are all W3C recommendations and start with the Resource Description Framework (RDF), which was one the first widely accepted semantic notations. It was also the foundation for the Web Ontology Language (OWL), which became recently available in version 2. Whereas OWL is based on description logics, the Web Service Modeling Language (WSML) also incorporates logic programming and rule paradigms.

3.1 Resource Description Framework – RDF

In October 1997 the World Wide Web Consortium has published the first RDF working draft. According to this specification RDF was designed to be

> *“... a foundation for processing metadata; it provides interoperability between applications that exchange machine-understandable information on the Web. RDF emphasizes facilities to enable automated processing of Web resources” [27]*

3.1.1 RDF Abstract Syntax

Any RDF expression represents a triple consisting of subject, predicate and object (Figure 3). Whereas the subject is either an URI or a blank node, the predicate is an URI and the object is either an URI, a literal or a blank node.

A set of RDF triples is called an RDF graph [28]. Only subjects and objects are considered nodes whereas predicates are also called arcs. Since the predicate always points from the subject to the object the resulting graph is directed (digraph). Furthermore, since the subject might only be an URI or a blank node, it has to be a web resource (e.g. an HTML document) or at least an entity that can be uniquely identified using the web (e.g. a person, a company, a product, ...). In the notion of RDF such triples are often used to describe properties of given resources (the subject). This is why RDF uses the term property as a synonym for predicate. In fact, the use of property instead of predicate is much more common in RDF [29]. Nevertheless,

since subject and object can be URI references, RDF triples can be used to describe any kind of relationship between arbitrary (web) resources.

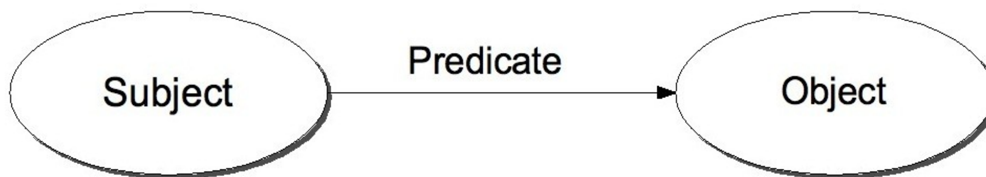


Figure 3: Graphical Representation of an RDF-Triple ([28] Copyright © 2004 World Wide Web Consortium. All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231)

RDF also distinguishes between two different types of literals:

- plain literals: a unicode string with an optional language tag that is self-denoting.
- typed literals: a unicode string with a datatype URI that maps the value to the given datatype. This also includes XML in which case the `rdf:XMLLiteral` type has to be used.

Listing 1 is a simple example that shows how to model some facts (author, subject, year of publication) about a particular paper. Beside the fact that the subject is represented by an URI reference these facts are expressed in natural language.

http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4755547&isnumber=4755314 has **creator** whose value is **Peter Salhofer**
http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4755547&isnumber=4755314 has **subject** whose value is **e-Government**
http://ieeexplore.ieee.org/stamp/stamp.jsp?tarnumber=4755547&isnumber=4755314 has **date** whose value is **2009**

Listing 1: Some simple facts expressed as triples of subject, predicate and object

These three statements, however, are not valid RDF triples since the predicates used are not of type URI. RDF uses URI references to make sure that predicates are machine processable, unique and can be easily exchanged [30]. To express these facts in valid RDF we have to rewrite these statements in the N-Triple notation [31] like shown in Listing 2.

```
<http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4755547&isnumber=4755314> \
<http://purl.org/dc/elements/1.1/creator> "Peter Salhofer" .
<http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4755547&isnumber=4755314> \
<http://purl.org/dc/elements/1.1/subject> "e-Government" .
<http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4755547&isnumber=4755314> \
<http://purl.org/dc/elements/1.1/date> "2009" .
```

Listing 2: Same facts as in Listing 1 expressed in N-Triple notation

In the next step this example will be refined in order to add more information about the author of this paper. Therefore a so called blank node is introduced (Listing 3). Compared to the previous example, subject and date of the paper identified by its URI are left unchanged. The creator property, however, now refers to a blank node, which is identified by a generated id ("_:a" in this case). The blank node is of type *http://www.w3.org/2000/10/swap/pim/contact#Person* and has the properties *firstname*, *lastname* and *organisation*. The identifier of a blank node only has the purpose to allow local references to the node but is not the label of the node. In fact, two graphs that are only distinguished by the id s of their blank nodes are considered to be equal ([28], chapter 6.3).

```
<http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4755547&isnumber=4755314> \
<http://purl.org/dc/elements/1.1/subject> "e-Government" .
_:a <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> \
<http://www.w3.org/2000/10/swap/pim/contact#Person> .
_:a <http://www.w3.org/2000/10/swap/pim/contact#firstname> "Peter" .
_:a <http://www.w3.org/2000/10/swap/pim/contact#lastname> "Salhofer" .
_:a <http://www.w3.org/2000/10/swap/pim/contact#organization> "FH JOANNEUM" .
<http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4755547&isnumber=4755314> \
<http://purl.org/dc/elements/1.1/creator> _:a .
<http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4755547&isnumber=4755314> \
<http://purl.org/dc/elements/1.1/date> "2009" .
```

Listing 3: Enhanced example using a blank node to add more information about the author (creator)

Figure 4 shows this example as an RDF graph. There exist many ways to represent RDF triples. By definition, every RDF document is a serialisation of an RDF graph into concrete syntax ([32], chapter 5.5). Also tables in a relational database can be seen as RDF triples. If there is a table with multiple columns, one row would represent a subject uniquely identified by its primary key. Every column in the table would represent an object value and the column name would represent the predicate [28]. Another way to represent RDF triple is the use of logical predicates. Since a predicate p defines a truth-value for a pair of resources, the following notion could be used as well: $R^p(x,y)$.

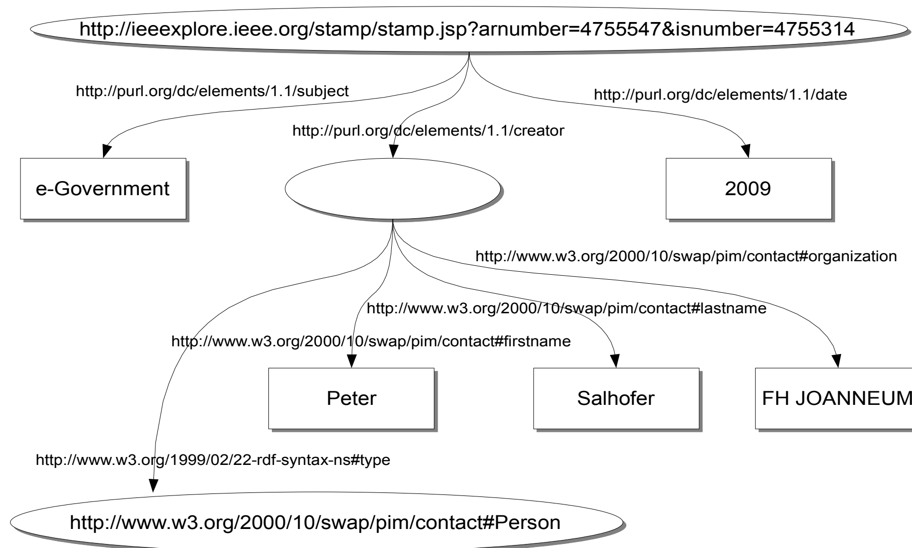


Figure 4: RDF graph presenting the situation described in Listing 3 (own illustration).

3.1.2 RDF XML Syntax

RDF uses XML to encode RDF graphs. The exact grammar for RDF/XML can be found in [32]. The XML representation of the N-triple example from Listing 3 is shown in Listing 4. Every RDF triple is contained in an RDF description tag. The *about* attribute of this tag is the URI reference of the subject of the triple and the property (predicate) is identified by its child-tag (e.g. dc:subject). The value of the child tag is the object of the triple.

Description tags containing a blank node (another *description* tag without an *about* attribute) are

representing sub-graphs rather than triples.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:contact="http://www.w3.org/2000/10/swap/pim/contact#">

    <rdf:Description rdf:about="http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=...">
        <dc:subject>e-Government</dc:subject>
    </rdf:Description>

    <rdf:Description rdf:about="http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=...">
        <dc:creator>
          <rdf:Description rdf:type="contact:Person">
           <contact:firstname>Peter</contact:firstname>
          </rdf:Description>
        </dc:creator>
    </rdf:Description>

    <rdf:Description rdf:about="http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=...">
        <dc:creator>
          <rdf:Description rdf:type="contact:Person">
           <contact:lastname>Salhofer</contact:lastname>
          </rdf:Description>
        </dc:creator>
    </rdf:Description>

    <rdf:Description rdf:about="http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=...">
        <dc:creator>
          <rdf:Description rdf:type="contact:Person">
           <contact:organization>FH JOANNEUM</contact:organization>
          </rdf:Description>
        </dc:creator>
    </rdf:Description>

    <rdf:Description rdf:about="http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=...">
        <dc:date>2009</dc:date>
    </rdf:Description>

</rdf:RDF>
```

Listing 4: XML representation of the graph shown in Figure 4

The RDF/XML syntax also allows for some abbreviations that help to make the resulting XML more compact. Here is a list of some important abbreviations:

- If an object has several properties they can be modelled as multiple child properties in the same *description* element
- If a property value is a string literal the property can be written as an attribute of the enclosing node (attribute properties)
- If all property values are string literals of the same language and occur only once, they can be made attribute properties of the enclosing element which is made an empty element

By applying these abbreviations, the XML representation can be significantly reduced like shown in Listing 5.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:contact="http://www.w3.org/2000/10/swap/pim/contact#">

  <rdf:Description rdf:about="http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=..."
    dc:subject="e-Government" dc:date="2009" >
   <dc:creator>
     <contact:Person contact:firstname="Peter" contact:lastname="Salhofer"
                contact:organization="FH JOANNEUM" />
   </dc:creator>
  </rdf:Description>
</rdf:RDF>
```

Listing 5: Shortened version of the XML representation making use of abbreviations

3.1.3 RDF Schema – The RDFS Vocabulary

The RDF abstract syntax defines RDF triples and graphs but does not provide any mechanism to describe properties and classes as well as the mapping between them. Therefore the RDF vocabulary description language, RDF Schema (RDFS) is used [29]. In the RDF graph shown in Figure 4 various properties (represented by the URI references at the arcs) and also a special type (Person) have been used. These elements are described in separate RDF documents using RDFS and are called *classes* and *properties*. Although this approach is conceptually similar to object oriented programming (OOP), there are some important differences. Unlike OOP or other frame-based systems, RDFS class descriptions do not contain the attributes of a class. Attributes are defined as separate classes on their own, are therefore global and are linked to the classes they should belong to via special properties. Since attributes are classes, they can be extended using inheritance as well.

The RDFS type system knows two different kinds of elements: classes and properties. In the following sub-sections some of the most important vocabulary elements are explained. A complete description of RDFS can be found in [29].

3.1.3.1 RDFS Classes

rdfs:Resource

This is the root of the RDFS class hierarchy. Every thing that is described in RDF is an instance of `rdf:Resource` or more precisely is an a subclass of `rdf:Resource`.

rdfs:Class

This element represents classes in RDF. Like all other elements it is a subclass of `rdfs:Resource` which means that all classes are resources as well. Classes are used to form groups of things with common characteristics and can be used as types in RDF. In fact, everything that is referenced by an `rdf:type` attribute is an instance of `rdfs:Class` (compare the example in Listing 4). Listing 6 shows a simple class declaration.

```
<Class rdf:about="http://www.w3.org/2000/10/swap/pim/contact#SocialEntity">
    <comment>
            The sort of thing which can have a phone number.
            Typically a person or an incorporated company, or unincorporated group.
    </comment>
</Class>
```

Listing 6: Sample class declaration taken from http://www.w3.org/2000/10/swap/pim/contact.rdf (Copyright © 2000 World Wide Web Consortium. All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231)

rdfs:Property

```
<rdf:Property rdf:about="http://www.w3.org/2000/10/swap/pim/contact#birthday">
    <domain rdf:resource="http://www.w3.org/2000/10/swap/pim/contact#SocialEntity"/>
    <range rdf:resource="http://www.w3.org/2000/10/swap/pim/contact#Date"/>
</rdf:Property>
```

Listing 7: A simple property definition taken from http://www.w3.org/2000/10/swap/pim/contact.rdf
(Copyright © 2000 World Wide Web Consortium. All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231)

As already mentioned above, RDFS specifies class attributes as separate classes. The class `rdfs:Property` is used for this purpose and is a subclass of `rdf:Class`. Listing 7 shows an example of a property definition. The meaning of the domain and range tags will be explained in the next section.

3.1.3.2 RDF Properties

rdfs:range

This element is used to define the datatype of a given property. In the example shown in Listing 7, the `rdfs:range` element limits all values assigned to the property `birthday` to be instances of the class `Date`.

Like all other RDF properties `rdfs:range` itself is a subclass of `rdfs:Property`.

rdfs:domain

This element is used to link a property to one or more classes. In object oriented programming, adding a property to a class means that every instance of this class possesses an instance of this property. In RDF the semantics of relating a property to a class is vice versa. Every instance that possesses this property is an instance of the class specified by the `rdf:domain` tag. Taken the example from Listing 7 this would mean that every instance that has a `birthday` attribute associated to it is an instance of the class `SocialEntity`.

rdf:type

This element can be used to define that a given resource is an instance of a specific class (see Listing 4 or an example).

rdfs:subClassOf

This transitive property states that instances of the class enclosing this tag are also instances of the class that is assigned as a value to this property. Listing 8 shows an example, saying that all instances of `Person` are also instances of `SocialEntity`.

```
<rdf:Description rdf:about="http://www.w3.org/2000/10/swap/pim/contact#Person">
    <comment>A person in the normal sense of the word.</comment>
    <subClassOf rdf:resource="http://www.w3.org/2000/10/swap/pim/contact#SocialEntity"/>
</rdf:Description>
```

Listing 8: Defining a class Person as subclass of SocialEntity (from http://www.w3.org/2000/10/swap/pim/contact.rdf, Copyright © 2000 World Wide Web Consortium. All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231)

rdfs:subPropertyOf

```
<rdf:Description rdf:about="http://www.w3.org/2000/10/swap/pim/contact#zip">
    <subPropertyOf rdf:resource="http://www.w3.org/2000/10/swap/pim/contact#postalCode"/>
</rdf:Description>
```

Listing 9: Definition of ZIP code as a sub-property of postalCode (from http://www.w3.org/2000/10/swap/pim/contact.rdf, Copyright © 2000 World Wide Web Consortium. All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231)

Defines a transitive relationship between two properties stating that one is a sub-property of the other. The example shown in Listing 9 defines the property `zip` as a sub-property (a special form) of `postalCode`. As a consequence, a class that is related to `zip` is also related to `postalCode`, which means it also has a property of type `postalCode.`

3.1.4 RDF Semantics

Based on precise formal vocabularies defined in RDFS, RDF is a formal language to formulate assertions. A basic semantic capability is to show, whether a given proposition is true or false. Assigning a boolean value to a given sentence (e.g. an RDF triple) is also called interpretation. Thus interpretation defines a formal description to decide the truth of or falsity of any expression of a logic [33].

3.1.4.1 Interpretation

The formal semantic model of RDF is based on model theory. Model theory is used to connect a formal language with its interpretation. A key element is the truth definition (denotation) that specifies for each pair of a sentence and a model whether the sentence is true or false using the given model [34]. More formally:

> $I \models S$, I .. Interpretation, S .. Sentence
> means that S is true in I, I satisfies S or I is a model of S

An interpretation is a model for an RDF graph if it is a model for every single sentence (RDF triple) in it. As a consequence, by having given this definition it is obvious that there does not exist one single interpretation for any given RDF graph. However, the number of possible interpretations is inversely proportional to the number of assertions that are made about the world of discourse. This means that, the bigger the graph is, the fewer interpretations might be available since there exist more constraints in the assertions that have to be considered.

Before the denotation mechanism of RDF can be explained in more detail, the following definition is important:

> Def: A ground graph is an RDF graph that does not contain any blank nodes.

RDF uses the following way to define an interpretation:

“A simple interpretation I of a vocabulary V is defined by:

1. *A non-empty set I_R of resources, called the domain or universe of I.*
2. *A set I_P, called the set of properties of I.*
3. *A mapping I_{EXT} from I_P into the powerset of I_R x I_R i.e. the set of sets of pairs <x,y> with x and y in I_R.*
4. *A mapping I_S from URI references in V into (I_R union I_P)*
5. *A mapping I_L from typed literals in V into I_R.*
6. *A distinguished subset LV of I_R, called the set of literal values, which contains all the plain literals in V” [34]*

Based on this definition RDF uses the following denotation algorithm for ground graphs (@ indicates a tag in the N-triple notation which is typically used to indicate the datatype of a typed literal):

> if E is a plain literal `"aaa"` in V then $I(E) = aaa$
>
> if E is a plain literal `"aaa"@ttt` in V then $I(E) = \langle aaa, ttt \rangle$
>
> if E is a typed literal in V then $I(E) = I_L(E)$
>
> if E is a URI reference in V then $I(E) = I_S(E)$
>
> if E is a ground triple $s\ p\ o$. then $I(E) = true$ if s, p and o are in V, $I(p)$ is in I_P and $\langle I(s),I(o) \rangle$ is in $I_{EXT}(I(p))$ otherwise $I(E) = false$.

> if *E* is a ground RDF graph then *I(E) = false* if *I(E') = false* for some triple *E'* in *E*, otherwise *I(E) =true*.

To further illustrate the use of this mechanism a short example is discussed. Assume there exists a vocabulary consisting of the following terms *{<c:article>, <c:authorOf>, <c:writtenBy>, "John Doe"}*.

The given vocabulary consists of one plain literal and three URI references. For the sake of compactness instead of absolute URIs their QNames [26] consisting of a namespace prefix and a local name are used. Thus we assume that the namespace prefix used was already defined elsewhere. Now a possible interpretation can be defined. Therefore first a set of existing resources has to be defined:

I_R = *LV* ∪ *{S,T}*

Since plain literals are considered to be self-contained, they directly represent some resources (the person with the name "John Doe" in this case) and therefore are part of I_R, more precisely members of the subset *LV*. Beside the literal values two additional resources called *S* and *T* are defined in the interpretation. Hence I_R consists of the following elements: *{John Doe, S, T}*. In the next step a set of properties has to be defined along with I_{EXT}, which is an extension for the elements of I_P. This means, that all possible relations are defined.

I_P = *{T}*
I_{EXT} = *(T {<S,T>,<T,John Doe>})*

In this definition property *T* also appears in the list of resources so it can also be used as subject and object. This complies with the RDF abstract syntax, since a property is defined by an URI reference. In the next two steps we have to map the terms of our vocabulary into the sets of resources and properties. Since the interpretation defines a grammar for true sentences, this step is important from a semantic point of view.

I_S = *(<c:article> S,<c:authorOf> T,<c:writtenBy> T)*
I_L = *()*

The mapping I_L is empty since there do not exist any typed literals in this example. Now sentences can be checked whether they are true in *I*.

<c:article> <c:authorOf> <c:writtenBy> .

To prove this sentence we follow the algorithm defined above. Therefore we have to show that:

1. *{<c:article>,<c:authorOf>,<c:writtenBy>}* ⊆ *V*
2. *I(<c:authorOf>)* ∈ I_P
3. *<I(<c:article>), I(<c:writtenBy>)>* ∈ I_{EXT}*(<c:authorOf>)*

The given sentence turns out to be true since all elements are members of *V*, *I(<c:authorOf>) = T* which is element of I_P and *<I(<c:article>), I(<c:writtenBy>)> = <S,T>* is element of I_{EXT}*(I(<c:author>)) = {<S,T>,<T,John Doe>}*. On the other side the following sentence turns out to be false according to this interpretation:

<c:article> <c:writtenBy> "John Doe" .

Again all elements used in this sentence are members of *V* and *I(<c:writtenBy>)* is member of I_P. However, *<I(<c:article>),I("John Doe")> = <S,John Doe>* is not member of I_{EXT}*(I(<c:writtenBy>)) = {<S,T>,<T,John Doe}*.

This formal theory allows for automated interpretation of RDF graphs, even the interpretation used in this example does not reflect the typical human understanding of the given domain. An interpretation that would come closer to this understanding would be the following one:

I_R = *LV* ∪ *{1,2,3}*
I_P = *{2}*

I_{EXT} = (2   {<1, John Doe>})
I_S = (<c:article>   1,<c:authorOf>   3,<c:writtenBy>   2)
I_L = ()

In this interpretation the following sentence would hold true:

<c:article> <c:writtenBy> "John Doe" .

The following sentences are false within this interpretation:

<c:article> <c:authorOf> <c:writtenBy> .

"John Doe" <c:authorOf> <c:article> .

The latter sentence is false since there exists no valid extension for *I(<c:authorOf>)* in the interpretation and since the subject in this case is a literal it does not comply with the RDF abstract syntax that only allows URI references or blank nodes for subjects.

This schema can be extended to support interpretation for non-ground graphs as well [33]. Therefore a mapping *A* is used that maps blank nodes to I_R. The extended interpretation *I+A* simply uses *A* to obtain an interpretation for blank nodes. The denotation algorithm needs to be extended to support blank nodes:

"If E is a blank node and A(E) is defined then [I+A](E) = A(E)
If E is an RDF graph then I(E) = true if [I+A'](E) = true for some mapping A' from blank(E) to IR, otherwise I(E)= false." [33]

Where *blank(E)* defines the set of blank nodes in E.

3.1.4.2 Entailment

Entailment is an essential characteristic in semantic technologies. Generally *A* entails *B* if whenever *A* is true also *B* is true and if *A* is false also *B* is false. In the terms of semantics this can be taken as that the meaning of *A* already includes the meaning of *B*. If *A* entails *B* and *B* entails *A* then both mean the same thing.

One practical impact on RDF is the validity of a graph that was constructed from other graphs. A graph *E* constructed from a set of graphs *S* is valid, if every interpretation that satisfies every member of *S* also satisfies *E*. In other words, if the set *S* entails *E*.

A detailed description of entailment together with useful lemmas and their proofs can be found in [33].

3.1.4.3 RDF Vocabulary Interpretation

In section 3.1.4.1 model theoretic interpretation was discussed in general. By adding the so-called RDF vocabulary to a given vocabulary *V* and adding some constraints on valid interpretations, additional semantics can be added. Interpretations that incorporate these constraints a called rdf-interpretations.

> Def.: An *rdf-interpretation* of a vocabulary *V* is a simple interpretation *I* of (*V* union *rdfV*) which satisfies the extra conditions described in Table 1 and all the rdf axiomatic triples [33].

The RDF vocabulary *rdfV* consists of *{rdf:type, rdf:Property, rdf:XMLLiteral, rdf:nil, rdf:List rdf:Statement, rdf:subject, rdf:predicate, df:object, rdf:first, rdf:rest, rdf:Seq, rdf:Bag, rdf:Alt, rdf:_1, rdf:_2, ... , rdf:value}*.

The first condition in Table 1 only allows for properties that are of type *rdf:Property*. Additionally the rdf axiomatic triples (please see [33], chapter 3.1 for a complete list) defines the following elements as properties: *rdf:type, rdf:subject, rdf:predicate, rdf:object, rdf:first, rdf:rest, rdf:value, rdf:1...n, rdf:nil*. Since this condition requires all properties to appear in an ordered pair defined by I_{EXT}, which in turn is a mapping into $I_R x I_R$, the set of properties I_P is a subset of I_R.

The next two conditions in Table 1 define, that a literal of type XML is treated like a plain literal if the XML content is well-typed, otherwise it is ignored.

x is in I_P if and only if <x,I(rdf:Property)> is in I_{EXT}(I(rdf:type))
If "xxx"^^rdf:XMLLiteral is in V and xxx is a well-typed XML literal string, then *I_L("xxx"^^rdf:XMLLiteral) is the XML value of xxx;* *I_L("xxx"^^rdf:XMLLiteral) is in L_V;* *I_{EXT}(I(rdf:type)) contains <I_L("xxx"^^rdf:XMLLiteral), I(rdf:XMLLiteral)>*
If "xxx"^^rdf:XMLLiteral is in V and xxx is an ill-typed XML literal string, then *I_L("xxx"^^rdf:XMLLiteral) is not in LV;* *I_{EXT}(I(rdf:type)) does not contain <I_L("xxx"^^rdf:XMLLiteral), I(rdf:XMLLiteral)>.*

Table 1: Semantic conditions for rdf-interpretations[33]

3.1.4.4 RDFS Interpretation

The RDFS vocabulary *rdfsV* (also see section 3.1.3) adds additional semantics to RDF and consists of the following elements:

rdfsV = {rdfs:domain, rdfs:range, rdfs:Resource, rdfs:Literal, rdfs:Datatype, rdfs:Class, rdfs:subClassOf, rdfs:subPropertyOf, rdfs:member, rdfs:Container, rdfs:ContainerMembershipProperty, rdfs:comment, rdfs:seeAlso, rdfs:isDefinedBy, rdfs:label}

Like rdf-interpretation also rdfs-interpretation imposes additional constraints on an interpretation and defines a set of axiomatic triples. In fact, the interpretation needs to be extended to conveniently deal with the meaning of the *rdfs:Class* element. Therefore a new set I_C, consisting of all classes and a special extension I_{CEXT} are introduced.

Def.: An rdfs-interpretation of *V* is an rdf-interpretation *I* of (*V* union *rdfV* union *rdfsV*) which satisfies the semantic conditions shown in Table 2 and all the RDFS axiomatic triples [33].

1	*x* is in I_{CEXT}(y) if and only if <x,y> is in I_{EXT}(I(rdf:type)) I_C = I_{CEXT}(I(rdfs:Class)) I_R = I_{CEXT}(I(rdfs:Resource)) L_V = I_{CEXT}(I(rdfs:Literal))
2	If <x,y> is in I_{EXT}(I(rdfs:domain)) and <u,v> is in I_{EXT}(x) then *u* is in I_{CEXT}(y)
3	If <x,y> is in I_{EXT}(I(rdfs:range)) and <u,v> is in I_{EXT}(x) then *v* is in I_{CEXT}(y)
4	I_{EXT}(I(rdfs:subPropertyOf)) is transitive and reflexive on I_P
5	If <x,y> is in I_{EXT}(I(rdfs:subPropertyOf)) then *x* and *y* are in I_P and I_{EXT}(x) is a subset of I_{EXT}(y)
6	If *x* is in I_C then <x,I(rdfs:Resource)> is in I_{EXT}(I(rdfs:subClassOf))
7	If <x,y> is in I_{EXT}(I(rdfs:subClassOf)) then *x* and *y* are in I_C and I_{CEXT}(x) is a subset of I_{CEXT}(y)
8	I_{EXT}(I(rdfs:subClassOf)) is transitive and reflexive on I_C
9	If *x* is in I_{CEXT}(I(rdfs:ContainerMembershipProperty)) then: <x,I(rdfs:member)> is in I_{EXT}(I(rdfs:subPropertyOf))
10	If *x* is in I_{CEXT}(I(rdfs:Datatype)) then <x,I(rdfs:Literal)> is in I_{EXT}(I(rdfs:subClassOf))

Table 2: Semantic conditions for rdfs-interpretation [33]

The definitions in row 1 of Table 2 state that I_{CEXT} defines a set, where all members are of the same type as the argument (see first line). Thus I_{CEXT}(I(rdfs:Class)) defines a set of all classes, I_{CEXT}(I(rdfs:Resource)) a set of all resources and so forth.

Row 2 defines an interpretation for *rdfs:domain*: If there exists a subject and an object for an *rdfs:domain* property (x and y) and there also exists and extension <u,v> for x (subjects of the *rdfs:domain* property have to be properties to be valid within this interpretation), then u is of type y. Thus the object (y in this case) of an *rdfs:domain* triple defines the type. Consequently, every resource that has this property is of type y.

The next condition defines the interpretation for *rdfs:range* which is similar to row 2, except that the object of a property of type *rdfs:range* has to be an instance of *rdfs:Class* and determines the type of the corresponding subject.

The condition in row 4 is obvious. In row 5 the interpretation for *rdfs:subPropertyOf* is further specified, defining that subject and object of this property both have to be instances of class and every relation defined by the sub-property is also member of the relation defined by the super-property. The next row defines that every class is a sub-class of *rdfs:Resource*. Row 7 provides an interpretation for *rdfs:subClassOf*, stating that it can only be applied to classes and all instances of the sub-class are also instances of the super-class.

The rule in row 9 defines that instances of *rdfs:ContainerMembershipProperty* are sub-properties of *rdfs:member*.

Row 10 defines that all instances of *rdfs:Datatype* are sub-classes of *rdfs:Literal*.

Together with the rdfs axiomatic axioms (see [33], section 4.1) the rdfs-interpretation incorporates the semantic of RDF and RDFS.

3.1.4.5 Entailment Rules

All the constraints discussed in the previous sections have added semantic conditions that can in turn be used to define so called entailment rules. These rules are the basis for automatic inference and therefore are essential to every semantic framework. Rdf and rdfs entailment rules are defined as triples that represent patterns to recognise situations where these rules can be applied. They further describe a resulting triple that can be added to the graph so that the resulting bigger graph is entailed from the original one. All rules are taken from [33]. As a result the application of these rules extends the original graph. These extensions are basically “new” additional facts that were inferred from the so-called ground facts. Thus, by applying entailment rules new implicit knowledge can be derived from a set of some ground facts, which adds enormous value to semantic models. These additional triples are sometimes also called “virtual triples” and are accessible using query languages like RDQL[35].

To understand the meaning of the terms used, the following explanation is necessary:

- aaa, bbb etc., represent URI references that are typically used as predicates of a triple
- uuu, vvv etc., represent either URI references or blank node identifiers and can thus be used as subjects and objects of triples
- xxx, yyy etc., stand for URI references, blank node identifiers or literals and are used as objects
- lll stands for a literal
- _:nnn is a blank node identifier

The interpolation lemma says that a graph *S* entails a graph *E* if and only if a subgraph of *S* is an instance of *E*. An instance of a graph has no blank nodes in it. Thus a graph *E* can be entailed from a graph *G* by replacing some URI references with blank nodes that have the same meaning as the original URI references (“are allocated to” the original URI references). This fact is covered by the so-called simple entailment rules (Table 3).

Rule name	if E contains	then add
se1	uuu aaa xxx .	uuu aaa _:nnn . where _:nnn identifies a blank node allocated to xxx by rule se1 or se2.
se2	uuu aaa xxx .	_:nnn aaa xxx . where _:nnn identifies a blank node allocated to uuu by rule se1 or se2.

Table 3: Simple entailment rules[33]

A specialisation of rule *se1* is the so-called literal generalisation rule (Table 4). It simply replaces a literal with a unique blank node identifier allocated to it. The consequence of the application of this rule is important, since a blank node identifier can also be used as subject, which allows for making assertions about literals.

Rule name	if E contains	then add
lg	uuu aaa lll .	uuu aaa _:nnn . where _:nnn identifies a blank node allocated to the literal lll by this rule.

Table 4: Literal generalisation rule[33]

The inverse of rule *lg* is the literal instantiation rule *gl* (Table 5). Notice that a blank node identifier derived by rule *lg* can only be replaced by the literal if it is in an object position.

Rule name	if E contains	then add
gl	uuu aaa _:nnn . where _:nnn identifies a blank node allocated to the literal lll by rule lg.	uuu aaa lll .

Table 5: Literal instantiation rule[33]

The RDF entailment rules shown in Table 6 are straightforward consequences of the conditions for rdf-interpretations listed in Table 1.

Rule name	if E contains	then add
rdf1	uuu aaa yyy .	aaa rdf:type rdf:Property .
rdf2	uuu aaa lll . where lll is a well-typed XML literal .	_:nnn rdf:type rdf:XMLLiteral . where _:nnn identifies a blank node allocated to lll by rule lg.

Table 6: RDF entailment rules[33]

Similar to RDF entailment rules also RDFS entailment rules (see Table 7) are consequences of rdfs-interpretation conditions.

In the following paragraphs all rdfs entailment rules will be explained by their relation to corresponding rdfs-interpretation conditions in Table 2. Thus whenever an rdfs condition is referred by a number, this number is the corresponding row number in Table 2.

Rule *rdfs1* results from condition 1. To be correct, the literal has to be in L_V, thus it also has to be element of I_{CEXT}*(I(rdfs:Literal))* which is only true if there exists an extension *<lll,rdf:Literal>* in I_{EXT}*(I(rdf:type))* which means: *_nnn rdf:type rdfs:Literal* (*lg* has to be used since literals must not be used as subjects).

Rdfs2 is the straightforward application of condition 2: *<aaa,xxx>* is element of I_{EXT}*(I(rdfs:domain))* and *<uuu,yyy>* is element of I_{EXT}*(I(aaa))*, thus *uuu* is in I_{CEXT}*(I(xxx))* which means *uuu rdf:type xxx*.

Example:

<c:hasName> <rdfs:Domain> <c:person>

<anURI> <c:hasName> "John Doe"

Entailed triple:

<anURI> rdf:type <c:person>

Rdfs3 is induced by condition 3: *<aaa,xxx>* is in $I_{EXT}(I(rdfs:range))$ and *<uuu,vvv>* in $I_{EXT}(I(aaa))$, which forces *vvv* to be in $I_{CEXT}(I(xxx))$.

Example:

<c:author> rdf:range <c:person> .
<c:article> <c:author> <anURI> .
Entailed triple:
<anURI> rdf:type <c:person>

Rule Name	if E contains	then add
rdfs1	uuu aaa lll. where lll is a plain literal (with or without a language tag).	_:nnn rdf:type rdfs:Literal . where _:nnn identifies a blank node allocated to lll by rule lg.
rdfs2	aaa rdfs:domain xxx . uuu aaa yyy .	uuu rdf:type xxx .
rdfs3	aaa rdfs:range xxx . uuu aaa vvv .	vvv rdf:type xxx .
rdfs4	uuu aaa xxx .	uuu rdf:type rdfs:Resource .
rdfs4b	uuu aaa vvv.	vvv rdf:type rdfs:Resource .
rdfs5	uuu rdfs:subPropertyOf vvv . vvv rdfs:subPropertyOf xxx .	uuu rdfs:subPropertyOf xxx .
rdfs6	uuu rdf:type rdf:Property .	uuu rdfs:subPropertyOf uuu .
rdfs7	aaa rdfs:subPropertyOf bbb . uuu aaa yyy .	uuu bbb yyy .
rdfs8	uuu rdf:type rdfs:Class .	uuu rdfs:subClassOf rdfs:Resource .
rdfs9	uuu rdfs:subClassOf xxx . vvv rdf:type uuu .	vvv rdf:type xxx .
rdfs10	uuu rdf:type rdfs:Class .	uuu rdfs:subClassOf uuu .
rdfs11	uuu rdfs:subClassOf vvv . vvv rdfs:subClassOf xxx .	uuu rdfs:subClassOf xxx .
rdfs12	uuu rdf:type rdfs:ContainerMembershipProperty .	uuu rdfs:subPropertyOf rdfs:member .
rdfs13	uuu rdf:type rdfs:Datatype .	uuu rdfs:subClassOf rdfs:Literal .

Table 7: RDFS entailment rules

Rdfs4 and *rdfs4b* are a consequence of applying *rdfs3* to the following two axiomatic triples (see [33] for a complete list):

rdf:subject rdfs:range rdfs:Resource .

rdf:object rdfs:range rdfs:Resource .

Condition 4 directly leads to *rdfs5* (transitivity) and *rdfs6* (reflexivity), whereas *rdfs7* is the application of condition 5.

Since according to condition 6 every *rdfs:Class* is a sub-class of *rdfs:Resource*, *rdfs8* is a trivial application of this constraint, which is also true for *rdfs9* and condition 7 respectively. The reflexivity of *rdfs:subClassOf* (condition 8) directly leads to *rdfs10* whereas its transitivity is covered by *rdfs11*. *Rdfs12* and *rdfs13* are also straightforward applications of condition 9 and 10.

3.1.5 Conclusions

RDF/RDF-S was the first semantic markup language developed under the guidance of the W3C and specifically aiming at implementing the vision of a semantic web. Compared to the definition of an ontology presented in section 2.1, RDF/RDF-S shows relatively limited capabilities, since there's no way to express additional axioms, nevertheless, it is the basis of most modern semantic languages. One particular aspect of RDF-S is the fact, that classes can be instances of other classes (meta-classes). This opens a lot of modelling possibilities and broadens its expressiveness, but also has some negative influences on decidability and automatic reasoning support as will be shown later. In contrast to frame-based systems (like object oriented programming languages) where classes contain the description of their attributes (also referred to as slots), property definitions in RDF-S are global. Consequently every instance is a member of all classes defined by its attributes.

RDF/RDF-S's initial version did not include the formal semantics model presented in section 3.1.4. This was introduced later as a result of extensions to RDF/RDF-S that where created to overcome the limited semantic capabilities of RDF/RDF-S. Model theoretic interpretations are key to efficient reasoning support, since they can be mapped to description logics as will be shown later.

One of the most important representatives of RDF/RDF-S successors is OWL that will be discussed in the next section.

3.2 The Web Ontology Language (OWL)

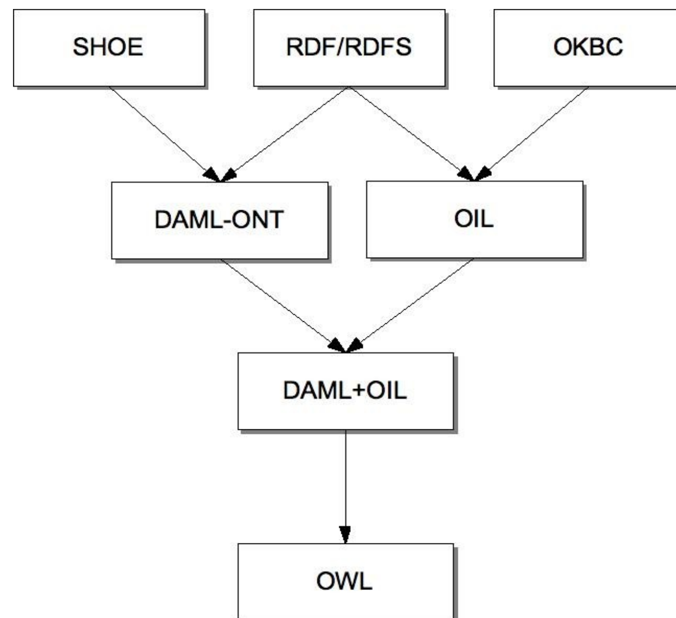


Figure 5: OWL and its predecessors (own illustration based on [36])

In November 2001 the World Wide Web Consortium (W3C) founded the Web Ontology Working Group as part of its Semantic Web Activity[2]. The goal of this group was to develop a new language for the semantic web “to extend the semantic reach of current XML and RDF meta-data efforts” that should be general and also have formal semantics [37]. The result of this effort was the Web Ontology Language (OWL) [38]. OWL is heavily based on DAML+OIL. The OWL Language Overview document even calls it a revision of the DAML+OIL web ontology language [39]. Whereas a detailed discussion of the OWL development process and how existing languages were influencing OWL design decisions can be found in [36], this work first provides a short presentation of OWL predecessors before the most important OWL features are discussed. Figure 5 provides an overview of how different semantic languages where influencing each other.

3.2.1 SHOE

The Simple HTML Ontology Extension (SHOE) [40][41] allows for embedding semantic descriptions into HTML documents. There are three properties of SHOE that have influenced other languages such as DAML-ONT and DAML-OIL:

- The usage of URI references for names
- Importing of other ontologies
- Versioning of ontologies

3.2.2 OIL

The Ontology Inference Layer (OIL) is heavily influenced by OKBC [42] and was one outcome of the EU funded On-To-Knowledge[3] project (IST-1999-10132). It is based on the following three aspects [43]:

- Description Logics (DL): To provide OIL with strong formal semantics and automatic reasoning

2 http://www.w3.org/2001/sw/
3 http://www.ontoknowledge.org

support it includes a mapping to the $\mathcal{SHIQ}$ description logic [44]. Description logics (DLs) define decidable fragments of first-order logic and, like frames, provide concepts (unary predicates) and slots (binary predicates). As distinguished from frames, they come with a formal, logic-based semantics [45].

- Frame based systems: To make OIL as intuitive as possible, it was designed as a frame-based system from the very beginning [46].
- Web standards: XML and RDF. To be compliant with other web technologies, OIL offers an XML serialisation syntax that is an extension of RDF and RDF-S. However, since RDF-S uses global properties, slots of frame-based concept have to be adapted when represented in RDF/RDF-S. OIL therefore uses class name suffixes on properties [47]. Every OIL ontology is a valid RDF-Document.

OIL is organised in several layers that are distinguished by features and complexity. At the lowest level there is “Core OIL”, which is extended by “Standard OIL” and “Instance OIL” [48]. There is also and additional layer called “Heavy OIL” that is reserved for future language developments. Since OIL was merged with the DAML initiative it is very unlikely that this layer will ever be defined.

3.2.3 DAML

In August 2000, the American Defense Advanced Research Projects Agency (DARPA) officially launched the DARPA Agent Markup Language (DAML)[4] initiative [49]. The initial release of the developed semantic markup language was called DAML-ONT [50]. Some of the shortcomings of DAML-ONT are discussed in [51].

Since the objectives of DAML and OIL have been very similar, the two efforts were joined and the newly created Joint US/EU ad hoc Agent Markup Committee developed a revised version called DAML+OIL [52]. This language was the direct predecessor of OWL.

3.2.4 OWL Language Variants

To cope with the wide range of OWL design objectives and requirements [53], the working group decided to come up with three different sublanguages. These different species of OWL represent a trade-off between ease-of-use, expressiveness, efficient reasoning and compatibility with RDF/RDF(S) and are [39]:

OWL Lite: This is the simplest version of OWL. In contrast to the next complex version it only supports cardinality restrictions in the range zero to one and does not allow to declare classes as unions or intersections of other classes and also lacks some other set based constructs. OWL Lite strongly corresponds to $\mathcal{SHIF}(\mathcal{D})$ description logics and although it is the most efficient variant when it comes to automatic reasoning it has exponential worst-case computational complexity [54].

OWL DL: DL is the acronym for description logics and expresses the close relationship between this version of OWL and description logics. OWL DL strongly corresponds to $\mathcal{SHOIN}$ [54] and therefore the worst-case complexity is non-deterministic exponential. OWL DL extends the features and capabilities of OWL Lite and guarantees that all ontologies are decidable. In fact, OWL DL is the sub-language that offers most expressiveness combined with automatic reasoning support and is a superset of OWL Lite.

OWL Full: This version is a superset of the other two languages and is completely compatible to RDF/RDF-S. Since RDF(S) can be used to express scenarios that are not decidable, OWL Full ontologies are not decidable either. One reason for the lack of decidability is RDF-S' capability to allow for classes that are instances of other classes. An explanation, why this inevitably leads to undecidability can be found in [55]. OWL Full has the highest expressiveness of all OWL languages but does not support automatic reasoning.

4 http://www.daml.org/

3.2.5 Important OWL Constructs

This section provides a short overview of the most important OWL language constructs that are extensions to RDF/RDF-S' capabilities. To unambiguously describe their meaning a short introduction to the model theoretic semantics of OWL is needed [56]. For the sake of simplicity, the interpretation of datatypes is omitted.

Every OWL vocabulary V consists of the following sets:

V_L: The set of all literals used in V.

V_C: The set of all class names in V, always includes `owl:Thing` and `owl:Nothing`.

V_D: The set of all datatype names in V.

V_I: The set of individual (instance) names in V.

V_{DP}: The set of data-valued property names in V.

V_{IP}: The set of individual-valued property names in V.

V_{AP}: The set of annotation property names in V.

V_O: The set of ontology names in V (might be used in import clauses, can be empty).

The definition of these subsets points out, that OWL explicitly distinguishes between properties of different type: `owl:ObjectProperty` (V_{IP}), `owl:DatatypeProperty` (V_{DP}) and `owl:AnnotationProperty` (V_{AP}).

An abstract OWL interpretation is a tuple of the following form:

$I = \langle R, EC, ER, L, S, LV \rangle$

R is a non empty set of resources of I, LV represents the literal values of I

EC: $V_C \rightarrow 2^O$

EC: $V_D \rightarrow 2^{LV}$

ER: $V_{DP} \rightarrow 2^{OxLV}$

ER: $V_{IP} \rightarrow 2^{OxO}$

ER: $V_{AP} \rightarrow 2^{RxR}$

ER: $V_{OP} \rightarrow 2^{RxR}$

L: $TL \rightarrow LV$ (TL is the set of typed literals)

S : $V_I \cup V_C \cup V_D \cup V_{DP} \cup V_{IP} \cup V_{AP} \cup V_O \cup$ { owl:Ontology, owl:DeprecatedClass, owl:DeprecatedProperty } $\rightarrow R$

$S(V_I) \subseteq O$

EC(owl:Thing) = $O \subseteq R$, where O is non-empty and disjoint from LV

EC(owl:Nothing) = { }

EC(rdfs:Literal) = LV

The extension function EC defines all extensions for classes that are also known as the class' instances. Thus, if an individual (instance) is part of a class' extension it is a member (instance) of this class. In the interpretation described above, the set of all possible instances is called O. OWL introduced two special classes that are by definition part of every ontology: `owl:Thing` and `owl:Nothing`. Every instance of any OWL class is also an instance of `owl:Thing`. This is not achieved by embedding `owl:Thing` into the class hierarchy, but by the definition of the model theoretic interpretation. Since every individual is a member of O (EC: $V_C \rightarrow 2^O$ and $S(V_I) \subseteq O$) and the extension of `owl:Thing` is O ($EC(owl:Thing) = O$) every instance is also a member of `owl:Thing`, whereas no single individual can be an instance of `owl:Nothing` (since $EC(owl:Nothing) = \{ \}$). This is an important semantic extension to RDF/RDFS.

3.2.5.1 OWL Classes

Since `rdfs:Class` allows for instances that are classes - which leads to undecidability - OWL has introduced a separate class construct. This construct is called `owl:Class` that is a subclass of `rdfs:Class`. In fact

owl:Class is the base class in all OWL Lite and OWL DL ontologies where *rdfs:Class* must not be used. Thus the major difference between *rdfs:Class* and *owl:Class* is that *owl:Class* does not support the definition of meta-classes.

While OWL Lite only allows for simple named classes (e.g. <owl:Class rfd:ID="Person" />), OWL DL supports a variety of class definition options, so called complex classes that are based on restrictions, set operations or enumerations. For example, it is possible to define the class of all white wines as the intersection of the class wine and those classes, that have a color property with value white (see Listing 10).

```
<owl:Class rdf:ID="WhiteWine">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Wine" />
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasColor" />
      <owl:hasValue rdf:resource="#White" />
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

Listing 10: Example of an OWL complex class definition [57]

More generally a class can be defined as the complement of another class, the intersection or union of other classes or as an enumeration of individuals. Especially when working with *owl:complementOf* it is important to know the exact semantics of this construct that is:

$EC(complementOf(C)) = O \setminus EC(C)$

This means that the complement of a class consists of all individuals that do not belong to the extension of class C. Defining a class "Male" as a complement of the class "Female" would be probably semantically wrong, since as a consequence everything that is not an instance of "Female" (e.g. Car, House, Fish, ..) would belong to the class "Male". Assertions like these typically require a common super-class, so that "Male" could be defined as the intersection of all instances of "Animal" (or "Human") and the complement of "Female".

Beside the constructs mentioned above, there exist two additional constructs to make assertions about classes. The *owl:equivalentClass* constructs defines that two classes have exactly the same set of instances whereas *owl:disjointWith* states that there exist no common instances in the extensions of two classes.

A detailed description of complex class definitions together with examples can be found in [57].

3.2.5.2 OWL Properties

Since OWL strictly separates properties that link instances of a class to other individuals and those that link instances to literal values two sub-classes of *rdf:Property* have been introduced:

- *owl:ObjectProperty* ($ER: V_{IP} \rightarrow 2^{O \times O}$)
- *owl:DatatypeProperty* ($ER: V_{DP} \rightarrow 2^{O \times LV}$)

Additionally, object properties can be further characterised by adding one of the following types to their declaration:

p rdf:type owl:TransitiveProperty. : States that a property is transitive ($ER(p) = (ER(p))^{+}, p \in V_{IP}$)

p rdf:type owl:SymetricProperty . : The property is symmetric ($ER(p) = (ER(p))^{-}, p \in V_{IP}$)

p owl:inverseOf p_0 *.* : *p* is the inverse of p_0 ($ER(p) = (ER(p_0))^{-}, p, p_0 \in V_{IP}$)

p rdf:type owl:FunctionalProperty: defines, that the property p can have at most one value (P(x,y) and P(x,z) implies y = z)

p rdf:type owl:InverseFunctionalProperty: defines, that the individual referenced by property p can be assigned to at most one instance (P(y,x) and P(z,x) implies y = z)

Only *owl:FunctionalProperty* can be applied to datatype properties as well.

3.2.5.3 Property Restrictions

Due to the compatibility with RDF/RDFS properties in OWL are global and classes on their own. Property restrictions however, can be used to express additional axiomatic constraints in a local scope. Apparently, this shows some influence of frame-based systems on the design of OWL.

```
<owl:Class rdf:ID="Wine">
  <rdfs:subClassOf rdf:resource="&food;PotableLiquid" />
  ...
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasMaker" />
      <owl:allValuesFrom rdf:resource="#Winery" />
    </owl:Restriction>
  </rdfs:subClassOf>
  ...
</owl:Class>
```

Listing 11: Example of a property restriction on the class "Wine" [57]

Listing 11 presents the use of a property restriction. This restriction only applies to the class within it is defined. As already shown in section 3.1.4, every property defines the type of the individual it belongs to via its domain attribute. Thus every instance's type represents the intersection of all the domains of its properties. This is explicitly used to apply property restrictions in OWL. In this example *owl:allValuesFrom* denotes that all values assigned to the *hasMaker* property have to be of type *Winery* in order to form an instance of type *Wine*. This restriction, however, does not require an instance of type wine to have a value assigned to its *hasMaker* property at all.

Beside *owl:allValuesFrom* there exists the *owl:someValuesFrom* restriction. Whereas the first requires all values (of fillers) of the restricted property to be instances of a given class, the latter means that at least one of the assigned values has to be a member of the given class. The exact semantics of these two restrictions is:

p owl:allValuesFrom r . : $\{x \in O \mid \forall <x,y> \in ER(p) \rightarrow y \in EC(r)\}$

p owl:someValuesFrom e . : $\{x \in O \mid \exists <x,y> \in ER(p) \wedge y \in EC(e)\}$

Additionally any property can also be restricted to a single value using the *owl:hasValue* restriction.

An additional form to restrict the usage of properties is offered by cardinality restrictions. By default every property can hold an arbitrary number of values. This can be changed either by defining a specific number of values using *owl:cardinality*, a minimum or maximum number of values using *owl:minCardinality* or *owl:maxCardinality* and a range of values by using a combination of the latter two.

3.2.6 Discussion

As shown above, OWL provides much more semantic constructs than RDF/RDF-S and therefore provides significantly higher expressiveness. Especially its class definition axioms appear to be extremely powerful. Class axioms can range from an enumeration of individuals that belong to the class to complex combinations of set-operations and restrictions. To define classes by applying restrictions to their properties is an important extension to RDF/RDF-S' capabilities, since this allows for defining different classes that do not have structural differences but have different ranges of domains assigned to their properties.

OWL is structured in several different sub-languages that offer different sets of constructs and at least OWL-

Lite and OWL-DL provide automatic reasoning support whereas OWL-Full concentrates on compatibility with RDF/RDF-S. For the purpose of creating a semantic E-Government solution only OWL-DL seams to be appropriate since OWL-Full is not decidable and therefore cannot be efficiently used in a run-time environment and OWL-Lite has some limitations (e.g. all properties can only have zero or one values) that do not reflect the characteristics of the E-Government domain.

One important issue is the fact that OWL only supports the open world assumption. As already pointed out in section 2.2 this seams to contradict some basic requirements of public services that are often characterised by limitations and constraints that have to met in order to become eligible for them. One way to get at least partly around this potential problem is the use of `owl:FunctionalProperty` and `owl:InverseFunctionalProperty`. This basically enables unique name assumption for the values of these properties. However, limitations imposed by the open world assumption might influence the future development of OWL ([51], p. 25). General, yet important potential shortcomings of OWL are discussed in [58].

3.3 OWL 2

To overcome some of the shortcomings of OWL mentioned in the previous sections an improved version of OWL called OWL 2 was recently presented[59]. OWL 2 is based on the initial version of OWL (subsequently referred to as OWL 1), thus every OWL 1 ontology is a valid OWL 2 ontology. Whereas OWL DL was based on the description logic $\mathcal{SHOIN}$ (see section 3.2.4) OWL 2 is based on $\mathcal{SROIQ}$[60]. The extended features of $\mathcal{SROIQ}$ compared to $\mathcal{SHOIN}$ are directly reflected by constructs and the semantics of OWL 2[61].

This section focuses on the most important differences between OWL version one and version two. A detailed description of these differences can be found in [60].

3.3.1 Syntaxes

OWL 1 used RDF as its official exchange language and therefore supported several ways to represent RDF graphs like the RDF/XML serialization [32], whereas the language specification was mostly based on the so-called OWL abstract syntax[56]. OWL 2 now supports a variety of syntaxes besides RDF/XML. The specification itself is based on a so called functional-style syntax[62]. To improve the processing of OWL ontologies with XML tools a special OWL/XML serialization [63] is available that is based on an XML schema[64]. For eased readability and creation of ontologies the Manchester OWL syntax [65] is provided. Optionally Turtle (The Terse RDF Triple Language [66]) can be used to represent OWL 2 ontologies as RDF graphs.

3.3.2 OWL 2 Features

In this section some of the most important new features of OWL 2 are presented, for a complete discussion see [67].

3.3.2.1 Negative Property Assertions

Like OWL 1 also OWL 2 is based on the open world assumption, thus, there is no negation-as-failure available. OWL 2, however, at least introduces a language construct that allows to model negative facts, asserting that two individuals are not part of an object property relation (NegativeObjectPropertyAssertion) or particular literals are not assigned to an individual using a data property (NegativeDataPropertyAssertion). The following statement expressed in OWL 2 functional-style syntax defines that Paris is not the capital of Austria:

```
NegativeObjectPropertyAssertion( :hasCapital :Austria :Paris)
```

This allows to answer some queries with *no* instead of *unknown*, since a reasoner can prove that certain combinations cannot occur due to negative property assertions. However, to model the behaviour of a

system that uses the closed world assumption it would be necessary to explicitly model all negative facts which is virtually impossible.

3.3.2.2 Qualified Cardinality Restrictions

Another set of very powerful constructs that are new in OWL 2 are qualified cardinality restrictions on properties. OWL 1 already allows defining cardinality restrictions on properties and therefore to define the number of fillers that can be assigned to an individual using the restricted property (see section 3.2.5.3). This could be used to define a class of persons who have for example at least three children (using *owl:minCardinality)*. OWL 2 now allows including the type of a filler to be part of the cardinality constraints. This allows for defining a class of persons that have a least two daughters:

ObjectMinCardinality(2 :hasChildren :Female)

Beside *ObjectMinCardinality* there exist *ObjectMaxCardinality*, *ObjectExactCardinality*, *DataMinCardinality*, *DataMaxCardinality* and *DataExactCardinality* restriction constructs.

3.3.2.3 Property Chain Inclusion

Property Chain Inclusion can be used to express that two individuals that are indirectly related via an arbitrary number of properties and other individuals are also part of a direct property relation. Here is a short example to illustrate this construct:

SubPropertyOf(ObjectPropertyChain(:locatedIn :partOf) :locatedIn)([67] section 2.2.5)

This means that if some individual *x* is *locatedIn y* and *y* is *partOf z* then *x* is also *locatedIn z*. So, since Graz is located in Austria and Austria is part of Europe, Graz is located in Europe as well.

3.3.2.4 Keys

Another important extension that is new in OWL 2 are Keys. This axiom allows defining a set of properties (data or object properties) as identifying attributes of class members similar to primary keys in relational database tables. This allows to axiomatically introduce something like a unique name assumption that is typically only part of systems using the closed world assumption. This axiom only applies to so called named instances. Example:

HasKey (:File :hasReferenceNumber)

ClassAssertion (:File :BuildingPermitApplication1234)

DataPropertyAssertion(:hasReferenceNumber :BuildingPermitApplication1234 "2009/10-2231")

This example defines that the *hasReferenceNumber* property is a key property, which means that all named instances of File with the same reference number are considered to be the same individual. In the second line a named instance (*BuildingPermitApplication1234*) is created and in the next line a reference number is defined. The general syntax of this axiom is as follows:

HasKey(CE (OPE_1 *...* OPE_m *) (* DPE_1 *...* DPE_n *))*

Where the acronyms have the following meanings

CE … Class Expression

OPE … Object Property Expression

DPE … Data Property Expression

The correct semantics of this axiom expressed as model theoretic interpretation is the following [61]:

$$\forall x, y, z_{1,} \ldots, z_m, w_{1,} \ldots, w_n : x \in (CE)^C \wedge \; ISNAMED_O(y)$$
$$\wedge \{(x, z_i) \in (OPE_i)^{OP} \wedge (y, z_i) \in (OPE_i)^{OP} \wedge ISNAMED_O(Z_i)\}_{i=1}^m$$
$$\wedge \{(x, w_j) \in (DPE_j)^{DP} \wedge (y, w_j) \in (DPE_j)^{DP}\}_{j=1}^n \Rightarrow x = y$$
$$with:$$
$$.^C \ldots Class\ Interpretation$$
$$.^{DP} \ldots Data\ Property\ Interpretation$$
$$.^{OP} \ldots Object\ Property\ Interpretation$$
$$\Delta_I \ldots Object\ Domain\ of\ the\ Interpretation\ (=(owl:Thing)^C)$$
$$ISNAMED_O(x) = true\ for\ x \in \Delta_I\ iff\ (a)^I = x\ for\ some\ named\ individual\ a$$

3.3.3 OWL 2 Sub-Languages

Just like OWL 1 (see section 3.2.4) also OWL 2 offers OWL 2 DL and OWL 2 Full. Additionally OWL 2 specifies three sub-languages also called profiles [68]. Profiles are optimized for specific usage scenarios and vary in expressiveness and reasoning performance, two dimensions that are inversely proportional to one another. Profiles therefore represent restrictions on OWL 2 and in fact every single OWL 2 profile is less expressive then OWL DL.

3.3.3.1 OWL EL

The name EL stems from the EL language family of description logics [69] that has influenced the modelling restrictions of OWL EL. Just like EL this profile is basically limited to conjunction and existential quantification. It does for example not allow the use of universal quantification and cardinality restrictions.

OWL EL is specifically recommended for ontologies that consist of very large numbers of classes and/or properties like the SNOMED CT[5] ontology[70].

3.3.3.2 OWL QL

This profile limits OWL 2 constructs to a subset that can be automatically translated into SQL for query answering. Therefore a simple query rewriting approach is used.

This profile is recommended when instances are stored in relational databases. The exact description of this profile and its limitations can be found in [68].

3.3.3.3 OWL RL

This profile represents a set of restrictions on the way in which specific constructs are used. It allows for building rule-based reasoners for OWL 2. A detailed discussion of its limitations can be found in [68].

3.3.4 Discussion

The design of this latest version of OWL was heavily influenced by the experience made with OWL 1 in various domains. Probably the most interesting features that were added to OWL 2 are bringing the new language at least a small step closer to rule based or logic programming based systems. This can be seen in the new construct to assert negative facts as well as in the keys axiom. Beside this, the introduction of OWL RL might bring description logics and rule based systems closer together.

Whereas OIL as one of the predecessors of OWL 1 used a frame-based notation for enhanced readability,

5 Systematized Nomenclature of Medicine--Clinical Terms, http://www.ihtsdo.org/snomed-ct/

OWL only offered RDF/XML serialization. OWL 2 now offers several additional serialization formats that should facilitate processing but also readability of ontologies. This is another important improvement.

3.4 The Web Service Modeling Language WSML

WSML [71] is the language used within the Web Service Modeling Ontology (WSMO), a framework that was designed to provide an integrated environment for semantic web service provisioning [72]. In contrast to OWL it was not designed as an extension to an existing technology but to optimally integrate into the WSMO framework [73]. The language is defined as a meta-meta-model based on the Meta Object Facility MOF [6] (also see section 5.2).

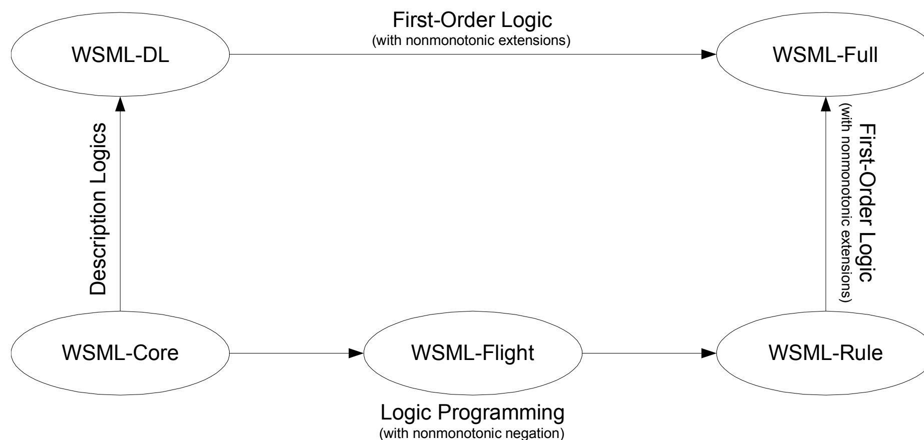


Figure 6: WSML variants [73]

Like OWL, also WSML offers different language variants. These variants are organised in two branches reflecting the different paradigms they are based on (see Figure 6). WSML is influenced by description logics as well as logic programming. Since these two paradigms use contrary concepts they can hardly be integrated in a common logic system.

WSML-Core: Is the least expressive WSML variant and contains the compatible elements of description logics and logic programming. This set is also called description logics programs (DLP) and is defined by the intersection of a mapping of description logics and the horn subset of logic programming into first-order logics. This excludes negation-as-failure as well as procedural attachments that are typically found in logic programming [74]. WSML-Core shows the best computational performance of all WSML variants, which is polynomial complexity [71].

WSML-DL: Extends WSML-CORE in the description logics dimension and is based on $\mathcal{SHIQ}(\mathcal{D})$. Since this is the same description logics OWL-DL is based on, this variant of WSML is OWL-DL compatible and there exists a mapping between these two standards [75]. WSML-DL is decidable and can be used for automatic reasoning.

WSML-Flight: This WSML variant extends WSML-Core in the logic-programming dimension. Whereas WSML logic programming is based on F-Logic [76], WSML-Flight is limited to the datalog subset of F-Logic. Since WSML-Flight includes negation-as-failure it is based on the closed-world-assumption (see section 2.2).

WSML-Rule: As a superset of WSML-Flight it extends the logic-programming capabilities and represents the Horn subset of F-Logic.

WSML-Full: Is a super-set of the description logics and the logic-programming branch of WSML. Thus it includes extensions to integrate non-monotonic logic with description logics. WSML-Full is the only

variant that is not decidable.

It is important to notice that this creates two rather independent branches starting with WSML-Core and merging in WSML-Full. The partial order relations over all variants are: Core < DL < Full and Core < Flight < Rule < Full. Since every WSML ontology allows importing existing ontologies that use other variants, this imposes certain restrictions on the way different variants could be mixed. Whenever WSML-DL and Flight/Rule ontologies occur in the same document, the resulting variant has to be WSML-Full [77].

3.4.1 WSML Syntax and Structure

By design WSML was developed to support semantic web services. This explains why ontologies are just one part of it. In general a WSML document can contain of the following parts [78]:

- goal
- ontology
- webservice
- mediator
- capability
- interface

A `goal` is a description of what a semantic web service can achieve including the pre- and postconditions as well as the actual effect. Since a goal can contain the description of a web service that is capable of accomplishing the desired functionality, it consists of `capability` and `interface` elements. Ontologies are used to define the terms and rules that are used throughout all other elements. The `webservice` section represents the actual functionality that can be invoked over the Internet. Its description therefore covers the service's capability as well as the service's interfaces. Mediators are used to bridge different terminologies and to avoid mismatches. WSML distinguishes between four types of mediators: ontology-to-ontology mediators (ooMediator), goal-to-goal mediators (ggMediator), webservice-to-goal mediators (wgMediator) and webservice-to-webservice mediators (wwMediator). A `capability` element is used to describe the functional aspects of a web service. This might include preconditions and assumption as well as postconditions and the actual effect of the service invocation, which describes how the state of the world will be changed. Interfaces are used to describe how to communicate with an actual web service (choreography) and can also be used to describe how a given service depends on other services to successfully provide its functionality (orchestration).
The rest of this section focuses on the mechanism used by WSML to describe ontologies. WSML therefore uses a frame-based syntax that is defined in [78]. The syntax consists of two parts, the conceptual syntax to describe concepts, attributes and instances and a logical expression syntax that is used to express rules and constraints[79].

```
concept Human
    annotations
        dc#description hasValue "concept of a human being"
    endAnnotations
    hasName ofType foaf#name
    hasParent inverseOf(hasChild) impliesType Human
    hasChild subAttributeOf(hasRelative) impliesType Human
    hasAncestor transitive impliesType Human
    hasRelative symmetric impliesType Human
    hasWeight ofType (1) xsd#decimal
    hasWeightInKG ofType (1) xsd#decimal
    hasBirthdate ofType (1) xsd#date
    hasObit ofType (0 1) xsd#date
    hasBirthplace ofType (1) loc#location
    isMarriedTo symmetric impliesType (0 1) Human
    hasCitizenship ofType oo#country
    isAlive ofType (1) xsd#boolean

axiom IsAlive
    definedBy
        ?x[isAlive hasValue xsd#boolean("true")] :-
            naf ?x[hasObit hasValue ?obit] memberOf Human.
        ?x[isAlive hasValue xsd#boolean("false")]
        impliedBy
            ?x[hasObit hasValue ?obit] memberOf Human.
```

Listing 12: Example WSML concept definition (WSML-Rule). Taken from [78],Appendix A.2

Listing 12 shows an example of a concept and an axiom definition in WSML. Compared to OWL there are many differences that will be explained in the next sections. Most notably however is, that WSML does not use XML but has a frame-based syntax as its preferred serialisation. This greatly improves human-readability and allows for compact models. WSML also prefers local property definitions, which means that properties are defined and only valid within the concept declaration. Whereas OWL explicitly distinguishes between object and data type properties, WSML obviously does not.

3.4.2 WSML Semantics

Like OWL also WSML comes with a formal semantic definition that is based on model theory. This complete semantic model can be found in [77]. Values used in the WSML vocabulary are either data values (literals), built-in data types (e.g. "_string") or Internationalized Resource Identifiers (IRIs, [80]) An IRI therefore indicates the use of a reference to another concept or instance.

A WSML Core interpretation is defined as a tuple of the following form [77]:

$$I = \langle U, <_U, \in_U, U^D, I_F, I_P, I_{hv}, I_{it}, I_{ot} \rangle$$

with

- a non-empty countable set U (abstract domain) and a non-empty set U^D (concrete domain) disjoint from U,
- a strict sub-concept relation $<_U: \rightarrow (U \cup U^D) \times (U \cup U^D)$,
- a concept membership relation $\in_U: \rightarrow (U \cup U^D) \times (U \cup U^D)$,
- a mapping I_F of constants and function identifiers to elements of U and functions over $(U \cup U^D)$,
- a mapping I_P of relation identifiers to relations over $(U \cup U^D)$
- mappings of binary relations I_{hv} (hasValue), I_{it} (impliesType), I_{ot} (ofType) : $(U \cup U^D) \rightarrow$

$2^{(U \cup U^D) \times (U \cup U^D)}$.

The sub-concept relation is transitive: if $a \in_U b \wedge b <_U c \quad a \in_U c$.

The *impliesType* relation is semantically equivalent to the *rdfs:range* relation (see section 3.1.4.4):

if $c,d \quad I_{it}(p)$, then for every $a \in_U c$ holds that for every $b \in U \cup U^D$ such that $\langle a,b \rangle \in I_{hv}(p)$, $b \in_U d$.

Functions and instance identifiers are interpreted as follows:

- Every instance identifier f is mapped to an element of the abstract domain U: $I_F(f) = u \in U$.
- Function identifiers are interpreted as functions over U according to their arity $i \geq 1$: $I_F(f)^i : U^i \to U$.
- Data values and datatype wrappers (e.g. *"xsd#date(2009,5,30)"*) with arity $n \geq 0$ are interpreted as functions over the concrete domain U^D: $I_F(f)^n : (U^D)^n \to U^D$

Relation identifiers are treaded as follows:

- N-ary relation identifiers p (with $n \geq 0$) are interpreted as relations over the domain $U \cup U^D$: $I_P(p)^n \subseteq (U \cup U^D)^n$.
- Identifiers of built-in predicates (e.g. "*wsml#numericSubtract(?x1,A,B)*") are interpreted as relations over the concrete domain U^D: $I_P(p)^n \subseteq (U^D)^n$.

This leads to the following set of conditions for the satisfaction of atomic formulas and molecules:

$I \models p(t_1,t_2,..t_n)$ iff $(t_1^I,t_2^I,..t_n^I) \in I_P(p)$

$I \models t_1{:}t_2$ iff $t_1^I \in_U t_2^I$ (memberOf)

$I \models t_1{::}t_2$ iff $t_1^I <_U t_2^I$ (subConceptOf)

$I \models t_1[t_2$ hasValue $t_3]$ iff $\langle t_1^I,t_3^I \rangle \in I_{hv}(t_2^I)$

$I \models t_1[t_2$ impliesType $t_3]$ iff $\langle t_1^I,t_3^I \rangle \in I_{it}(t_2^I)$

$I \models t_1[t_2$ ofType $t_3]$ iff $\langle t_1^I,t_3^I \rangle \in I_{ot}(t_2^I)$

$I \models t_1{=}t_2$ iff $t_1^I = t_2^I$

These basic rules can be extended to support the valid interpretation of arbitrary complex formulas. The interpretation of elements of the concrete domain is merely delegated to so-called concrete domain schemes. WSML is not limited to one particular concrete domain scheme but every concrete domain scheme that should be used together with WSML has to meet certain requirements and therefore has to be WSML conformal (see [77], page 29ff). WSML treats all datatypes as concepts and consequently all data values as instances of the corresponding datatype.

3.4.2.1 WSML DL Extension

When the WSML Core interpretation of the previous section is compared to the OWL interpretation from section 3.2.5 one can see, that OWL as a description language based approach strictly distinguishes between classes (concepts), instances and properties. To enable DL-based reasoning on WSML ontologies WSML DL introduces a syntactic separation of these concepts (i.e. it defines where theses different elements might occur in constructs) as well as a semantic separation (see [77], p33ff).

Therefore the domain and the interpretation functions are split up:

U^i … The non-empty set of instances of individuals

U^a ... The set of attributes

U^c … The set of concepts

I_F … maps instance identifiers to U^i, concept identifiers to U^c and attribute as well as annotation

property identifiers to U^a.

Beside this, the *subconceptOf* relation is redefined to apply to concepts only ($<_U \subset U^c \times U^c$), *memberOf* is only defined for instances and concepts ($\in_U \subset U^i \times U^c$) and *hasValue*, *impliesType* and *ofType* are only defined for attributes ($I_{hv}(u)=I_{it}(u)=I_{ot}(u)=\{\} \mid \forall u \in U^i \cup U^c$).

Finally satisfaction rules have to be extended to support quantified formulas on abstract instances:

$I \models_{DL} \forall_a y(\phi) \text{ iff } \forall_a y(\phi) \wedge y(\phi) \in U^i \wedge y(\phi) \notin U^p$

$I \models_{DL} \exists_a y(\phi) \text{ iff } \exists_a y(\phi) \wedge y(\phi) \in U^i \wedge y(\phi) \notin U^p$

With these extensions to WSML Core, a semantic model is defined that is almost equivalent to the OWL semantic model (see section 3.2.5). As already mentioned above, the interpretation of data values is delegated to a compliant concrete domain scheme. However, there exists no explicit equivalent to *owl:Thing* and *owl:Nothing*.

3.4.2.2 WSML Core, Flight and Rule Semantic

The semantic model of WSML Core, Flight and Rule is based on research to develop semantic models for logic programs. One of these approaches is called Stable Model Semantics [81]. Logic programs consist of a set of rules and a set of ground terms or facts. Based on these sets a so-called Herbrand model can be created that consists of all ground atoms (i.e. ground terms and entailed facts). For negation free programs, the minimal Herbrand model is the so-called canonical model that contains all answers to variable free queries. Programs with negation, however, do not contain a unique minimal Herbrand model. In this case a Stable Model can be identified by the following algorithm ([81], p.1073):

> *"For any set M of atoms from program Π, let Π_M be the program obtained from Π by deleting*
> *(i) each rule that has a negative literal $\neg B$ in its body with $B \in M$, and*
> *(ii) all negative literals in the bodies of the remaining rules.*
>
> *… If this model coincides with M, then we say that M is a stable set of Π. Such sets can be also described as the fixed points of the operator S_Π defined by the condition: for any set M of atoms from Π, $S_\Pi(M)$ is the minimal Herbrand model of Π_M.*
> *… The stable model semantics is defined for a logic program Π if Π has exactly one stable model, and it declares that model to be the canonical model of Π."*

Since every stable set is a minimal Herbrand model (see [81] for a formal proof), this approach is called the stable model semantics. It is essential for the existence of a minimal Herbrand model, that after reduction (i.e. the removal of all rules with negation) the set of atoms does not change. WSML enforces this by its syntactic rules that allow negation (or more precisely negation-as-failure) only for atoms [78].
How-to derive the stable model for a given WSML Core, Flight or Rule ontology and a concrete domain scheme is shown in [77] (p. 36-38).

3.5 Comparing OWL and WSML

WSML is an approach to use logic statements as well as rules to describe ontologies and therewith overcomes some of the shortcomings of OWL. Nevertheless, there also exists the description logic variant WSML DL and a mapping to OWL [75]. Thus this section focuses on differences between OWL and the logic programming variants of WSML called Flight and Rule.

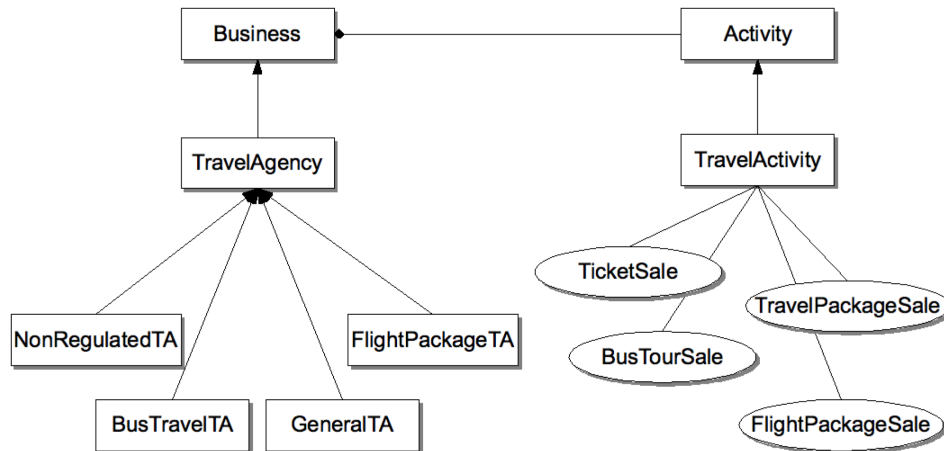


Figure 7: Sample scenario travel agency business (own illustration)

To point out the differences the following two example scenarios are modelled with both technologies:

Scenario 1

The Austrian Industrial Code defines all necessary regulations and procedures for different professions, businesses and industries. When it comes to travel agencies the law distinguishes between four different sub-types of businesses depending on the services offered by them (see Figure 7).

1. Non regulated business: If the agency only sells tickets then no special regulations apply. This kind of business can be run without the proof of special skills or knowledge.
2. Regulated (Bus Tours): If the only purpose of the agency is to organise and sell bus tours then the owner of the business has to have some minimum skills in the tourism industry
3. Regulated (Travel Agency): This represents the typical travel agency. Profound knowledge is necessary to run the business.
4. Regulated (incl. flight packages): If the agency also organises tour packages including flights, it has to proof the existence of a special type of insurance in addition to the previously mentioned requirements.

The goal is to setup an ontology that represents this knowledge and that can decide in which category any given travel agency falls into. The four types of travel agencies mentioned above are disjoint, thus any agency can only fall into only one category. Since every single travel agency can offer a set of services (i.e. pursue several activities) rules to categorise any given agency have to follow this order:

TicketSale < BusTourSale < TravelPackageSale < FlightPackageOrganization

Thus, if an agency sells tickets but also offers tour packages it falls into the third category. Any travel agency that organises flight packages automatically falls into the fourth category regardless of any other activities.

To model this situation we want to introduce a general class `Business` that represents all possible professions and businesses. `TravelAgency` should be a subclass of `Business` and should have four sub-classes representing the four different cases mentioned above.

Scenario 2

Let's assume the following situation. Some construction law distinguishes between residential houses of different size. Construction of smaller houses is eligible for a faster, less complex approval procedure, whereas construction of bigger houses requires certain additional steps and reports. Thus, it is essential to define whether a building permit application is about a big or a small house. Furthermore assume that every residential house with no more then two floors and no more than 400 square meter of effective surface is

considered small whereas every other house is big. Thus the modelled ontology has to classify every instance of a house with less than 3 floors and less than or equal 400 square meter of effective surface as small and every other house as big.

3.5.1 The WSML Solution

To model these scenarios the most expressive but still decidable WSML variant WSML Rule was selected. Let's start with the first scenario.

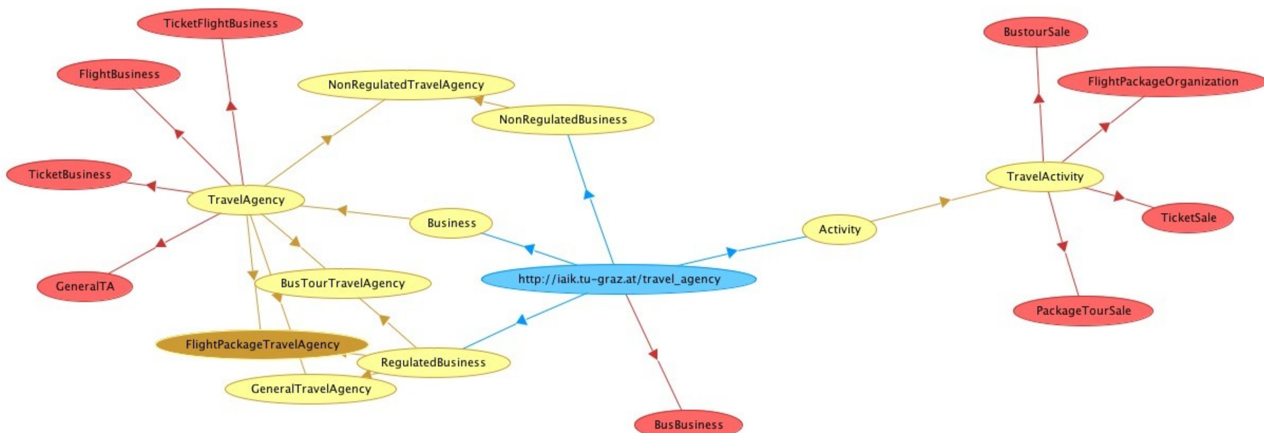


Figure 8: WSML travel agency example ontology. Screenshot from WSMO Toolkit

Whereas Figure 8 provides a structural overview of the entire ontology, Listing 13 shows the definition of the concept hierarchy.

Every `Business` "performs" at least one `Activity`. Since `TravelAgency` is a subconcept of `Business` it also has to have at least one `Activity` assigned to its instances. Since, however, a travel agency cannot perform any arbitrary activity a specialised sub-class called `TravelActivity` was introduced that represents all activities limited to travel agencies. `RegulatedBusiness` and `NonRegulatedBusiness` represent the fact that there exist businesses where certain regulations apply whereas for other businesses no regulations apply. `NonRegulatedTravelAgency` represents those agencies that merely sell tickets. It therefore is a sub-concept of `TravelAgency` and `NonRegulatedBusiness`. A `BusTourTravelAgency` is a `TravelAgency` that is specialised in organising and selling bus tours but might also sell other tickets. `GeneralTravelAgency` represents all travel agencies that organise and sell trips but do not organise flight packages. `FlightPackageTravelAgency` is the business with the strictest regulations.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-rule"
namespace { _"http://iaik.tu-graz.at/dla#",
    xsd _"http://www.w3.org/2001/XMLSchema#"
 }
ontology _"http://iaik.tu-graz.at/travel_agency"

concept Business
    hasActivity ofType (1 *) Activity
concept RegulatedBusiness
concept NonRegulatedBusiness
concept Activity
concept TravelActivity subConceptOf Activity
concept TravelAgency subConceptOf Business
concept GeneralTravelAgency subConceptOf {TravelAgency,RegulatedBusiness}
concept BusTourTravelAgency subConceptOf {TravelAgency,RegulatedBusiness}
concept FlightPackageTravelAgency subConceptOf {TravelAgency,RegulatedBusiness}
concept NonRegulatedTravelAgency subConceptOf {TravelAgency,NonRegulatedBusiness}

instance BustourSale memberOf TravelActivity
instance FlightPackageOrganization memberOf TravelActivity
instance PackageTourSale memberOf TravelActivity
instance TicketSale memberOf TravelActivity
```

Listing 13: WSML example concept hierarchy for the travel agency example

Listing 14 contains the definition of some example instances of travel agencies that have to be classified by a semantic reasoner based on the modelled ontology. Whereas most of these instances already contain a class assertion (i.e. "*memberOf TravelAgency*"), the *BusBusiness* does not. Thus there have to be rules that allow identifying this instance as a member of *TravelAgency*.

```
instance TicketFlightBusiness memberOf TravelAgency
    hasActivity hasValue {TicketSale,FlightPackageOrganization}
instance TicketBusiness memberOf TravelAgency
    hasActivity hasValue TicketSale
instance FlightBusiness memberOf TravelAgency
    hasActivity hasValue FlightPackageOrganization
instance BusBusiness
    hasActivity hasValue {TicketSale,BustourSale}
instance GeneralTA memberOf TravelAgency
   hasActivity hasValue {TicketSale,PackageTourSale}
```

Listing 14: Sample instances to test automatic classification

To decide which category any given travel agency belongs to a set of axioms has to be defined (see Listing 15). The first axiom allows identifying an instance as a member of the type *TravelAgency*, which is true as soon as it has at least one activity of type *TravelActivity* associated with it using the *hasActivity* property. The second axiom (*isFlightPackageTravelAgency*) defines the rules to classify a given instance as a *FlightPackageTravelAgency*. Every axiom exists of a head clause (i.e. the consequence) and a body (i.e. the condition). The head of the *isFlightPackageTravelAgency* axiom defines that the instance represented by the variable *?business* is a member of the concept *FlightPackageTravelAgency* if it is a member of *TravelAgency* and has one activity with the value *FlightPackageOrganization*. Thus as soon as a travel agency offers flight packages it is considered as a *FlightPackageTravelAgency* and all other potentially existing activities are ignored. The *isGeneralTravelAgency* encompasses all travel agencies that offer *PackageTourSale* but no *FlightPackageOrganization*. This is expressed by the *naf* (negation-as-failure) operator. Thus, if the proof of the existence of the value *FlightPackageOrganization* fails, this operation returns true. Notice that the use of the equality (=) operator together with the *naf* operator (e.g. "*naf ?act = FlightPackageOrganization*") is not allowed. Due to the restrictions explained in section 3.4.2.2, negation can only be applied to atoms and whereas the *hasValue* construct is an atom, every equality expression is a molecule. The remaining axioms simply exclude more and more activities using the *naf* operator and eventually the *isNonRegulatedTravelAgency* axiom defines every agency that is not covered by the other axioms as a *NonRegulatedTravelAgency*.

```
axiom isTravelAgency
     definedBy
      ?business memberOf TravelAgency
      :-
      ?business[hasActivity hasValue ?a] and ?a memberOf TravelActivity .

axiom isFlightPackageTravelAgency
 definedBy
     ?business memberOf FlightPackageTravelAgency
     :-
      ?business[hasActivity hasValue FlightPackageOrganization] memberOf  TravelAgency .

axiom isGeneralTravelAgency
 definedBy
     ?business memberOf GeneralTravelAgency
     :-
      ?business[hasActivity hasValue PackageTourSale] memberOf TravelAgency  and
      naf ?business[hasActivity hasValue FlightPackageOrganization] .

axiom isBusTourTravelAgency
 definedBy
     ?business memberOf BusTourTravelAgency
     :-
      ?business memberOf TravelAgency and
      naf ?business[hasActivity hasValue FlightPackageOrganization] and
      naf ?business[hasActivity hasValue PackageTourSale] and
      ?business[hasActivity hasValue BustourSale] .

axiom isNonRegulatedTravelAgency
 definedBy
     ?business memberOf NonRegulatedTravelAgency
     :-
      ?business memberOf TravelAgency and
      naf ?business[hasActivity hasValue FlightPackageOrganization] and
      naf ?business[hasActivity hasValue BustourSale] and
      naf ?business[hasActivity hasValue PackageTourSale] .
```

Listing 15: WSML axiom definition for the travel agency example

The entire WSML ontology was modelled with the Web Service Modeling Toolkit v2.0[6], which comes bundled with the IRIS reasoner[7]. This tool has an integrated query interface that accepts semantic queries. Thus the consistency and correctness of the ontology can easily be checked.

```
concept ResidentialHouse
     hasFloors ofType (1 1) _integer
     hasEffectiveSurface ofType (1 1) _integer

concept SmallResidentialHouse subConceptOf ResidentialHouse

concept BigResidentialHouse subConceptOf ResidentialHouse

axiom isSmallResidentialHouse
 definedBy ?house memberOf SmallResidentialHouse
 :- ?house[hasFloors hasValue ?floors,hasEffectiveSurface hasValue ?size] and
   size < 401 and ?floors < 4 .

axiom isBigResidentialHouse
 definedBy ?house  memberOf  BigResidentialHouse
  :- ?house[hasFloors hasValue ?floors, hasEffectiveSurface hasValue ?size]
   and (?floors > 3 or ?size > 400) .
```

Listing 16: WSML sample solution for the house classification problem

6 http://sourceforge.net/projects/wsmt/
7 http://www.iris-reasoner.org/

The query "`?x memberOf FlightPackageTravelAgency`" for example correctly lists all automatically classified instances (`FlightBusiness, TicketFlightBusiness`).

The solution for the second scenario is rather straight forward as shown in Listing 16. The only compromise enforced by WSML is the lack of less-than-or-equal and greater-than-or-equal datatype predicates.

3.5.2 The OWL Solution

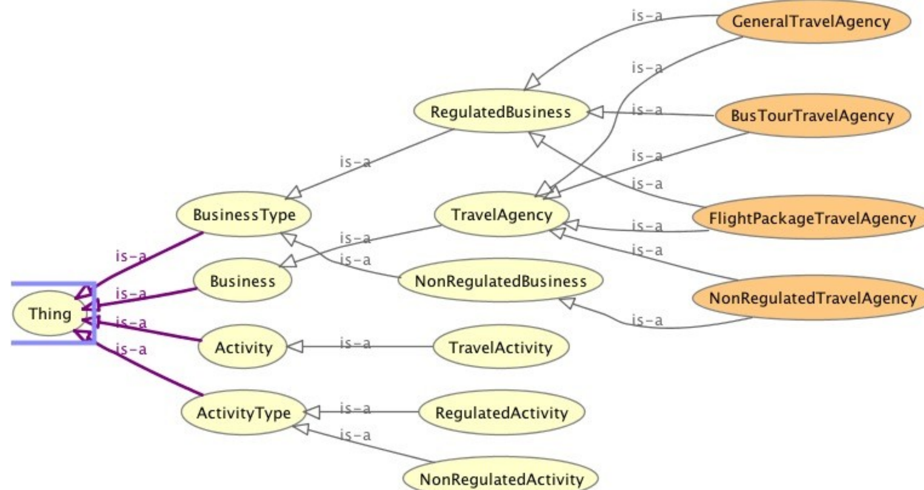


Figure 9: OWL 2 travel agency example ontology. Protegé screenshot.

The OWL implementation of the sample scenario was done using Protegé [8] version 4.0, which is the first version that supports OWL 2. Figure 9 provides an overview of the class structure of the sample ontology. In contrast to the WSML solution three additional classes (`ActivityType`, `RegulatedActivity` and `NonRegulatedActivity`) were added to the ontology. These classes are later used to find out whether a business is regulated or non-regulated.

While the default OWL/XML serialisation is definitely not human readable OWL 2 supports the so called Manchester Syntax as an alternative. Although OWL 2 is not a frame-based language, the Manchester Syntax groups assertions and axioms by properties, classes and individuals which greatly improves readability.

```
ObjectProperty: hasActivity
    Characteristics:
        Irreflexive
    Domain:
        Business
    Range:
        Activity

Class: Activity
    SubClassOf:
        owl:Thing
    DisjointWith:
        Business

Class: Business
    SubClassOf:
        hasActivity min 1 Activity
    DisjointWith:
        Activity

Class: TravelAgency
```

8 http://protege.stanford.edu/

```
    SubClassOf:
        Business

Class: TravelActivity
    SubClassOf:
        Activity

Class: BusinessType
    SubClassOf:
        owl:Thing

Class: NonRegulatedBusiness
    SubClassOf:
        BusinessType
    DisjointWith:
        RegulatedBusiness

Class: RegulatedBusiness
    SubClassOf:
        BusinessType
    DisjointWith:
        NonRegulatedBusiness
```

Listing 17: OWL 2 travel agency sample ontology, basic class hierarchy

The basic class structure, describing `Business`, `Activity` and `BusinessType`s is shown in Listing 17. Although the definition of the `Business` class contains a cardinality restriction, due to the open world assumption this restriction does not have any practical implications. This means that an instance of a business can be modelled that does not have a single activity associated to it, unless the absence of any activity is explicitly modelled (e.g. by negative property assertions).

```
Class: ActivityType

Class: NonRegulatedActivity
    SubClassOf:
        ActivityType
    DisjointWith:
        RegulatedActivity

Class: RegulatedActivity
    SubClassOf:
        ActivityType
    DisjointWith:
        NonRegulatedActivity

Class: NonRegulatedTravelAgency
    EquivalentTo:
        TravelAgency
        and (hasActivity only NonRegulatedActivity)
    SubClassOf:
        NonRegulatedBusiness,
        TravelAgency

Class: GeneralTravelAgency
    EquivalentTo:
        (not (hasActivity value FlightPackageOrganization))
        and (hasActivity value PackageTourSale)
    SubClassOf:
        RegulatedBusiness,
        TravelAgency

Class: FlightPackageTravelAgency
    EquivalentTo:
        hasActivity value FlightPackageOrganization
    SubClassOf:
        RegulatedBusiness,
        TravelAgency

Class: BusTourTravelAgency
    EquivalentTo:
        (not ((hasActivity value FlightPackageOrganization)
```

```
        or (hasActivity value PackageTourSale)))
        and (hasActivity value BusTourSale)
    SubClassOf:
        RegulatedBusiness,
        TravelAgency

DisjointClasses:
    BusTourTravelAgency,
    FlightPackageTravelAgency,
    GeneralTravelAgency,
    NonRegulatedTravelAgency
```

Listing 18: OWL 2 Travel agency example, class axioms for automatic classification

Listing 18 contains the definition of the different types of travel agencies. A `NonRegulatedTravelAgency` is a `TravelAgency` that only offers services of type `NonRegulatedActivity`, whereas a `FlightPackageTravelAgency` performs the `FlightPackageOrganization` activity (please see Listing 19 for available activities). A `GeneralTravelAgency` offers `PackageTourSale` but does not perform `FlightPackageOrganization`. A `BusTourTravelAgency` can offer `BusTourSale` but neither `PackageTourSale` nor `FlightPackageOrganization`. Additionally it is asserted that all different travel agency types are disjoint, thus any given instance of a travel agency can only belong to one single category. The `not` operator used in this notation is synonym for the `owl:ObjectComplementOf` construct. Since there is no negation-as-failure in OWL this construct is only evaluated to `true` if the presence of any negated property value is explicitly excluded. The same requirements hold true for the `only` restriction. Exclusion of the potential presence of property values can be achieved in several ways. One approach is to limit the number of fillers for an individual's properties as done for the `TicketBusiness` (see Listing 20). If the maximum number of values assigned to a property for all instances is one, this could also be indicated by defining the property itself as functional. Otherwise, if the number of fillers can vary from individual to individual as in this case, a cardinality restriction can be used.

```
Individual: BusTourSale
    Types:
        RegulatedActivity,
        TravelActivity,
        owl:Thing
    DifferentFrom:
        FlightPackageOrganization,
        PackageTourSale,
        TicketSale

Individual: FlightPackageOrganization
    Types:
        RegulatedActivity,
        TravelActivity,
        owl:Thing
    DifferentFrom:
        BusTourSale,
        PackageTourSale,
        TicketSale

Individual: PackageTourSale
    Types:
        RegulatedActivity,
        TravelActivity,
        owl:Thing
    DifferentFrom:
        BusTourSale,
        FlightPackageOrganization,
        TicketSale

Individual: TicketSale
    Types:
        NonRegulatedActivity,
        TravelActivity,
        owl:Thing
    DifferentFrom:
        BusTourSale,
        FlightPackageOrganization,
        PackageTourSale
```

Listing 19: OWL 2 instances of TravelActivity used in the example ontology

The `BusBusiness` individual has two activities assigned to it. To assert that there is no way for additional property values, the maximum cardinality is set to two. This is necessary to “close” all possible assertions about this individual's property. However, this also requires the two fillers of the `BusBusiness`'s `hasActivity` property (`BusTourSale` and `TicketSale`) to be declared different individuals (see the `DifferentFrom` assertions in Listing 19).

An alternative way to express that an individual's property does not contain specific values is the use of negative property assertions like done for `GeneralTA`.

```
Individual: TicketFlightBusiness
  Types:
    TravelAgency,
    owl:Thing
  Facts:
    hasActivity  FlightPackageOrganization,
    hasActivity  TicketSale

Individual: TicketBusiness
  Types:
    TravelAgency,
    owl:Thing,
    hasActivity max 1 owl:Thing
  Facts:
    hasActivity  TicketSale

Individual: BusBusiness
  Types:
    TravelAgency,
    owl:Thing,
    hasActivity max 2 owl:Thing
  Facts:
    hasActivity  BusTourSale,
    hasActivity  TicketSale

Individual: FlightBusiness
  Types:
    TravelAgency,
    owl:Thing
  Facts:
    hasActivity  FlightPackageOrganization,
    hasActivity  PackageTourSale

Individual: GeneralTA
  Types:
    TravelAgency,
    owl:Thing
  Facts:
    hasActivity  BusTourSale,
    hasActivity  PackageTourSale,
    hasActivity  TicketSale,
     not  hasActivity  FlightPackageOrganization
```

Listing 20: OWL 2 test individuals to check automatic classification

A solution to the problem stated in scenario two is only possible due to some new features introduced in OWL 2. With the previous version of OWL it was simply impossible to meet these requirements since there was no construct to further restrict the values of any datatype property to a particular range. With the introduction of the `DataTypeRestriction` construct (see section 7.5 in [62]) this was made possible. Thanks to this new feature the OWL solution almost directly reflects the requirements (see Listing 21).

```
DataProperty: hasFloors
   Characteristics:
      Functional
   Domain:
      ResidentialHouse
   Range:
      positiveInteger

DataProperty: hasEffectiveSurface
   Characteristics:
      Functional
   Domain:
      ResidentialHouse
   Range:
      positiveInteger

Class: ResidentialHouse

Class: SmallResidentialHouse
   EquivalentTo:
      (hasEffectiveSurface only positiveInteger[<= 400])
      and (hasFloors only positiveInteger[<= 3])
   SubClassOf:
      ResidentialHouse
   DisjointWith:
      BigResidentialHouse

Class: BigResidentialHouse
   EquivalentTo:
      ResidentialHouse
      and ((hasEffectiveSurface some positiveInteger[> 400])
      or (hasFloors some positiveInteger[> 3]))
   SubClassOf:
      ResidentialHouse
   DisjointWith:
      SmallResidentialHouse
```

Listing 21: OWL 2 solution to the house classification problem

3.5.3 Comparison of Results

When comparing the solutions based on the two different frameworks and paradigms the first and most obvious result is that they all meet the basic requirements stated in the problem descriptions. Thus this section works out the differences between these approaches and therefore possible advantages and disadvantages of one solution over the other.

As already discussed in sections 3.2 - 3.4, the major differences between the compared frameworks are as follows:

- OWL is based on the open world assumption whereas WSML is based on the closed world assumption
- OWL applies the description logics paradigm, whereas the WSML variant used here (WSML-Rule) rests on the rule-based and logic programming paradigm
- WSML uses a frame-based approach

The implication of the open world assumption is the most obvious one since this also excludes the unique name assumption. Therefore, for example, minimum cardinality restrictions cannot be checked unless it is explicitly asserted that there cannot be additional values as already pointed out in section 3.5.2. This also requires different individuals to be explicitly asserted as being different. Thus, in order to allow reasonable consistency checking, which is key in the E-Government domain, you have to “close” your world by asserting the absence of information. This does not only sound less intuitive but can also become tedious and error-prone especially when dealing with larger ontologies. Although OWL 2 has introduced new constructs to simplify this (e.g. by using negative property assertions), a lot of additional facts have to be stated. WSML on the other hand, is based on the closed world assumption and therefore assumes everything that is not explicitly stated as being wrong. This principle for example does not allow for the creation of instances

without any properties if their corresponding types (concepts) have some minimum cardinality restrictions. Since WSML also uses the unique name assumption there is no need to specify that several individuals are mutually different, which minimises the amount of assertions necessary.

Generally OWL 2 provides several useful additions. Without some of these a solution to the second problem scenario would not be possible. Besides functional and logical extensions OWL 2 also supports the Manchester Syntax as one of its serialisation formats. This makes OWL 2 ontologies easily readable for humans. Although the Manchester Syntax almost looks like the WSML syntax, OWL is not frame-based. This means for example that all properties are classes on their own and therefore global. Consequently every property name can only be used once within an ontology. Thus, if there would be a property called *hasAge*, it can only be declared and therefore also be assigned to a domain once. Age, however, is a property that might be used for many classes. People have an age but also for example buildings. Thus what should be the domain of this property? OWL suggests the use of pre- and suffixes to indicate the usage of a property, nevertheless, *hasAge* has the same semantics regardless where it is used. WSML, in contrast, is a frame-based language. Properties are part of a concept assertion and therefore local. Thus, the *hasAge* property could be used within different concepts and could even have different types (ranges) depending on the concept within it is used. On the other side, it would also be possible to use pre- and/or suffixes to make properties globally unique. Together with axioms that define that the presence of a particular property implies a specific type, the same semantics as in OWL could be achieved. Therefore the frame-based approach, apart from the fact that it greatly improves readability, can be considered an advantage.

Whether the open or the closed world assumption should be considered advantages or disadvantages merely depends on the nature of the domain that should be modelled. In the case of E-Government, where anything that can't be proved is considered to be non-existent or false, the closed world assumption seems to be the more natural or intuitive approach. Generally, every ontology modelled in an open world assumption environment can be “closed” by modelling all negative facts, although this might lead to enormously large ontologies. On the other side, closed worlds cannot be “opened” since there is no notion of “unknown”.

4 Semantic Web Services

In Tim Berners-Lee's vision of the Semantic Web[16] intelligent software agents assist people in getting relatively complex tasks done. Semantic web services are the technical backbone behind such scenarios and are semantic extensions to web services. Since a sound understanding of semantic web services is key to set-up semantic E-Government services, this chapter will present the most important initiatives in this field. Later on these approaches are compared and discussed with respect to the E-Government domain and the overall goal of Ontology Driven E-Government. Since all approaches are based on conventional web services, a brief introduction to this subject is given as well.

4.1 Web Services

There exist various definitions of the term web service[7]. One that is broadly accepted as a standard definition is the one by the W3C Web Service Activity Group[82]:

> **Definition 2**: *"A Web service is a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A Web service supports direct interactions with other software agents using XML based messages exchanged via internet-based protocols."[82]*

As covered by this definition a web service is an application that can be accessed via the exchange of XML based messages over internet using standard protocols. How messages are exchanged and how these messages have to be composed has to be defined in accessible XML documents. The XML definition of a web service is contained in a so called Web Service Description Language (WSDL) document. There exist several versions of WSDL. Still the most widely adopted version is 1.1[8] although there exists a more recent

version 2.0[83]. This section will use a simple example to illustrate the characteristics of web services. The example reflects a business integration scenario, where a service provider wants to offer some business functionality to its clients or customers as a web service (see Figure 10). For the sake of simplicity only one function that allows to look up a particular product from the service providers inventory based on a given product id is offered via the web service.

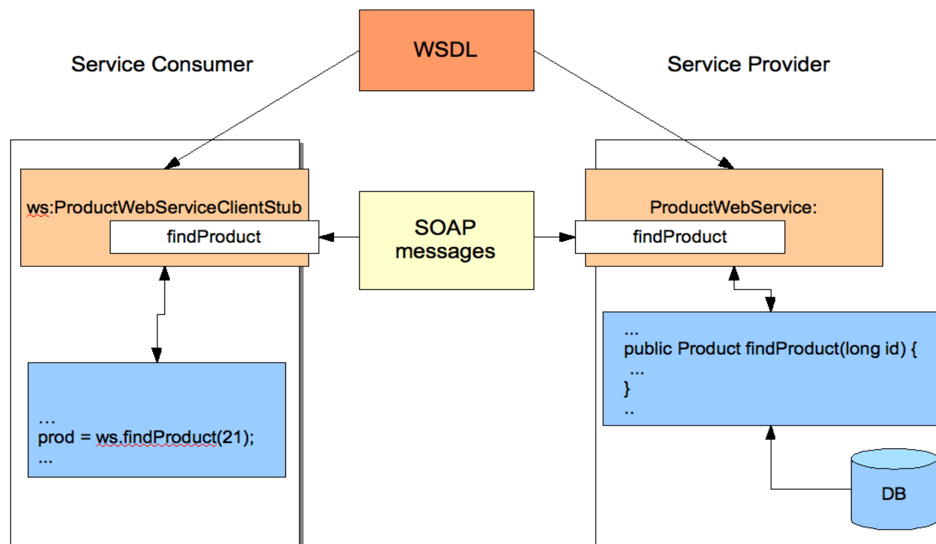


Figure 10: A Web Service sample Scenario (own illustration)

The web service is described in a WSDL file that can be used by the service consumer to generate a client side service stub. This stub is used to make local calls of the *findProduct* operation, which are transparently serialised and sent to the service provider. The response from the service provider is unmarshalled and exposed to the client side code as the return value of the method invocation. This principle is called Remote Procedure Call (RPC)[84] and was supported by standards like the Common Object Request Broker Architecture (CORBA)[85] before web services even came into existence. Like web services, CORBA can be used in integration scenarios since it allows for cross-platform and cross-language RPCs. Although CORBA has some technical advantages over web services[86], the foundation of web services which is a set of successfully adopted standard technologies like XML and HTTP is considered to be an even more important success factor[87].

4.1.1 WSDL 1.1

A WSDL document contains a web service description like required by Definition 2. It is conceptually split into an abstract and a concrete definition (see Figure 11). The top level description elements (direct child elements of the *definitions* root tag) are the following:

> *"**Types**– a container for data type definitions using some type system (such as XSD).*
> ***Message**– an abstract, typed definition of the data being communicated.*
> ***Operation**– an abstract description of an action supported by the service.*
> ***Port Type**–an abstract set of operations supported by one or more endpoints.*
> ***Binding**– a concrete protocol and data format specification for a particular port type.*
> ***Port**– a single endpoint defined as a combination of a binding and a network address.*
> ***Service**– a collection of related endpoints." [8]*

Figure 11: WSDL 1.1 Structure (own illustration)

WSDL uses XML schema data type as its intrinsic default typing system. Types can be declared directly inside the WSDL document or can be imported from existing schema files. Listing 22 provides the type entry of the example WSDL file together with the source of the imported schema file. The defined elements are then used as types in the service description. The complex type `product` represents the actual inventory entry and is made up of an id, a product name, a product description and the unit price of the product.

```
WSDL File:
<types>
    <xsd:schema>
        <xsd:import namespace="http://webservice.demo.service.iaik.tugraz.at/"
            schemaLocation="http://localhost:8080/DemoService/productWebService?xsd=1"/>
    </xsd:schema>
</types>

Imported XSD File:
<xs:schema version="1.0" targetNamespace="http://webservice.demo.service.iaik.tugraz.at/">
  <xs:element name="ProductNotFoundException" type="tns:ProductNotFoundException"/>
  <xs:element name="findProduct" type="tns:findProduct"/>
  <xs:element name="findProductResponse" type="tns:findProductResponse"/>
  <xs:complexType name="findProduct">
      <xs:sequence>
          <xs:element name="productId" type="xs:long" minOccurs="0"/>
      </xs:sequence>
  </xs:complexType>
  <xs:complexType name="findProductResponse">
      <xs:sequence>
          <xs:element name="return" type="tns:product" minOccurs="0"/>
      </xs:sequence>
  </xs:complexType>
  <xs:complexType name="product">
      <xs:sequence>
          <xs:element name="description" type="xs:string" minOccurs="0"/>
          <xs:element name="id" type="xs:long" minOccurs="0"/>
          <xs:element name="name" type="xs:string" minOccurs="0"/>
          <xs:element name="unitprice" type="xs:double" minOccurs="0"/>
      </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ProductNotFoundException">
      <xs:sequence>
          <xs:element name="message" type="xs:string" minOccurs="0"/>
      </xs:sequence>
```

```
  </xs:complexType>
</xs:schema>
```

Listing 22: WSDL type node together with the imported XSD file from the “findProduct” example

Messages that are exchanged between the service requestor and the service endpoint are defined via message elements (see Listing 23). The `part`s of these messages refer to XML elements in the `types` section and are therefore precisely typed. A message can consist of several parts.

```
<message name="findProduct">
    <part name="parameters" element="tns:findProduct"/>
</message>

<message name="findProductResponse">
    <part name="parameters" element="tns:findProductResponse"/>
</message>

<message name="ProductNotFoundException">
    <part name="fault" element="tns:ProductNotFoundException"/>
</message>
```

Listing 23: WSDL snippet from the "findProduct" example showing the message definition section

The final part of the abstract service definition is made up by the `portType` element with the embedded definitions of supported operations (see Listing 24).

```
<portType name="ProductWebService">
    <operation name="findProduct">
        <input wsam:Action="http://.../ProductWebService/findProductRequest"
        message="tns:findProduct"/>
        <output wsam:Action="http://.../ProductWebService/findProductResponse"
             message="tns:findProductResponse"/>
        <fault message="tns:ProductNotFoundException" name="ProductNotFoundException"
              wsam:Action="http://.../ProductWebService/findProduct/Fault/ProductNotFoundException"/>
    </operation>
</portType>
```

Listing 24: Porttype definition of the "findProduct" web service example

An operation represents a message exchange between the client and the service endpoint. Beside the request/response exchange pattern WSDL 1.1 also supports one-way (service endpoint receives a message), solicit-response (the endpoint sends a message to the client who has to reply) and notification (the endpoint sends a message). The actual exchange pattern used is entirely determined by the messages that occur within an operation. An input message followed by an output message indicates the request/response pattern, an output message first followed by an input message implies the solicit-response pattern. Input only or output only messages define the notification or one-way pattern respectively.
The example in Listing 24 defines a request-response message exchange pattern. Although it defines an input and an output message this does still not necessarily indicate a synchronous RPC-like operation. The actual behaviour of the operation is to be defined in the concrete section of the WSDL file when the binding is specified. The `findProduct` operation also defines a so called fault message. This is an output message that indicates the occurrence of an erroneous condition. When recursively following the elements used in this definition it becomes clear that the operation expects a number as the input message an returns a product consisting of id, name, description and unit price in case of normal termination or returns a text message in case of an error.

```
<binding name="ProductWebServicePortBinding" type="tns:ProductWebService">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="findProduct">
        <soap:operation soapAction=""/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
            <fault name="ProductNotFoundException">
                <soap:fault name="ProductNotFoundException" use="literal"/>
            </fault>
    </operation>
</binding>

<service name="productWebService">
    <port name="ProductWebServicePort" binding="tns:ProductWebServicePortBinding">
        <soap:address location="http://localhost:8080/DemoService/productWebService"/>
    </port>
</service>
```

Listing 25: Concrete definition of the "findProduct" example web service.

The concrete definition section of a WSDL file binds the abstract service description to the actual transport mechanism and defines where the system can be found in the internet (see Listing 25). The `binding` element defines the messaging protocol that should be used for the conversation between the service requester and the service provider. The default protocol used by WSDL based web services is SOAP [88], which initially was the acronym for Simple Object Access Protocol.

A SOAP message is an XML document that is sent between the communication peers. It supports various transport protocols like HTTP[89] or SMTP[90]. In the case of the `findProduct` example HTTP is used as the transport protocol, which together with the request-response message exchange pattern defines the RPC-like nature of the operation. Listing 26 shows a sample request that invokes the `findProduct` operation and the response that contains the inquired product details. The usage of XML as the messaging format is one of the success factors of web services since it allows for entirely platform and programming language independence.

```
Request:
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:findProduct xmlns:ns2="http://webservice.demo.service.iaik.tugraz.at/">
            <productId>77</productId>
        </ns2:findProduct>
    </S:Body>
</S:Envelope>

Response:
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:findProductResponse xmlns:ns2="http://webservice.demo.service.iaik.tugraz.at/">
            <return>
                <description>32&quot; LCD TV, 4xHDMI</description>
                <id>77</id>
                <name>UE32X100</name>
                <unitprice>499.98</unitprice>
            </return>
        </ns2:findProductResponse>
    </S:Body>
</S:Envelope>
```

Listing 26: A "findProduct" sample request and response

4.1.2 WSDL 2.0

WSDL 2.0 is the most recent version of the standard. The most obvious differences when compared to version 1.1 are changes in the top-level description elements (see Figure 12). Input and output messages of operations now directly refer to particular types. Thus there is no need for an additional *message* section that is used in WSDL 1.1 to compose messages out of parts. The *portType* element of WSDL 1.1 was renamed to *interface*.

Figure 12: Top-level description elements in WSDL 2.0 (own illustration)

Beside these structural changes in the XML another important difference is the increased number of supported message exchange patterns (MEPs) that has doubled to eight when compared to version 1.1 (see section 4.1.1). MEPs are used to define the interaction between clients and the service endpoints. This includes the number of messages that are sent, the potential creation of fault messages but also the timing of messages (e.g. whether an operation is synchronous or asynchronous). WSDL 2.0 supports the following MEPs[91][92]:

- In-only: This is the equivalent to WSDL's 1.1 in-only operations. The corresponding operation allows for one input message and does not produce any fault messages.
- Robust in-only: This is a special case of the in-only MEP that allows the endpoint to respond with a fault message if necessary.
- In-out: Equivalent to WSDL's 1.1 request-response operations. It is defined by one input message followed by one output message. The endpoint might also respond with a fault message
- In-optional-out: Similar to in-out, the output message, however, is optional.
- Out-only: Equivalent to WSDL 1.1. There is only an output message, no fault messages are produced.
- Robust out-only: Similar to out-only, however, a fault message is also allowed.
- Out-in: Allows for one output message followed by an input message. The client receives a message

from the endpoint and has to respond either with the appropriate output message or a fault message

- Out-optional-in: Similar to out-in, however, the response is optional

Since, in contrast to WSDL 1.1 to interaction pattern used by an operation can not be unambiguously inferred from the number and the order of messages used, the MEP has to be explicitly specified using the operation's `pattern` attribute (see Listing 21).

```
<description ...>
    ...
    <interface  name="reservationInterface">
        ...
        <operation name="opCheckAvailability" ... >

        <operation name="opLogInquiry"
                pattern="http://www.w3.org/ns/wsdl/out-only">
            <output messageLabel="Out" element="ghns:customerData" />
        </operation>

    </interface>
    ...
</description>
```

Listing 27: Sample specification of an out-only operation

Although there already exist eight predefined patterns, WSDL 2.0 provides an extension mechanism that allows for the definition of additional message exchange patterns if needed. Instructions on how to define custom MEPs is provided in [93], whereas a discussion of this feature can be found in [94].

Although WSDL 2.0 shows some major improvements compared to WSDL 1.1 is still not widely adopted by framework and tool providers. This might be due to its higher complexity and interoperability issues with existing WSDL 1.1 web services[95]. The aim of this section was to provide a brief introduction to web services and the underlying standards. In the next view sections so called semantic web service frameworks will be discussed that try to add semantics required by software agents to discover, assess and utilise web services as presented in this section.

4.2 Semantic Markup for Web Services (OWL-S)

OWL-S[96] was previously called DAML-S since its was originally based on OWL`s predecessor DAML+OIL. The latest release of OWL-S is version 1.2[9]. The aim of OWL-S is the creation of a service ontology that can be used to describe the different semantic aspects of web services. This ontology is expressed in the Web Ontology Language (OWL) and provides a computer interpretable description that allows software agents to understand the intention of a web service. In particular OWL-S wants to support the following tasks:

1. **Automatic Web service discovery**: Based on a user's intention specified as computer-interpretable semantic markup some agent process can identify those services apt to meet the given requirements and constraints
2. **Automatic Web service invocation**: Agents should be capable of executing discovered and selected web services solely based on OWL-S' declarative descriptions. Enabling agents to understand the meaning of web services' operations and messages requires a mapping to corresponding classes in OWL ontologies.
3. **Automatic Web service composition and interoperation**: Based on high-level descriptions of goals, agents should be enabled to achieve them by automatically combining different web service calls. Therefore, agents have to be aware of the effects and the preconditions of every single operation to establish a sequence of web service calls that will eventually fulfil the given objective.

9 See http://www.daml.org/services/owl-s/ for an overview of DAML-S/OWL-S releases

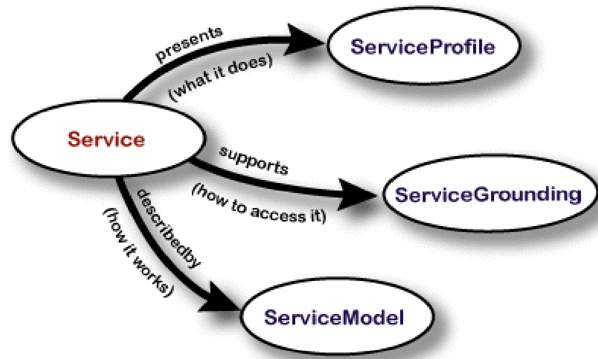


Figure 13: OWL-S top-level classes ([96], Copyright © 2004 World Wide Web Consortium. All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231)

The top-level classes that are defined in the OWL-S service ontology are shown in an RDF-graph-like style in Figure 13. The top level element is the `Service` class. In fact every semantic web service is represented by an instance of this class. It has the following three properties:

- **ServiceProfile**: This element describes what a service does. Therefore, it provides a semantic description that should allow agents to find out whether a service might be relevant for their goal.
- **ServiceModel** (**ProcessModel**): This class describes interaction patterns with the actual service in terms of processes.
- **ServiceGrounding**: A grounding describes how the semantic description of a web service is related to its technical description in the WSDL file. This part is particularly important for the actual service enactment.

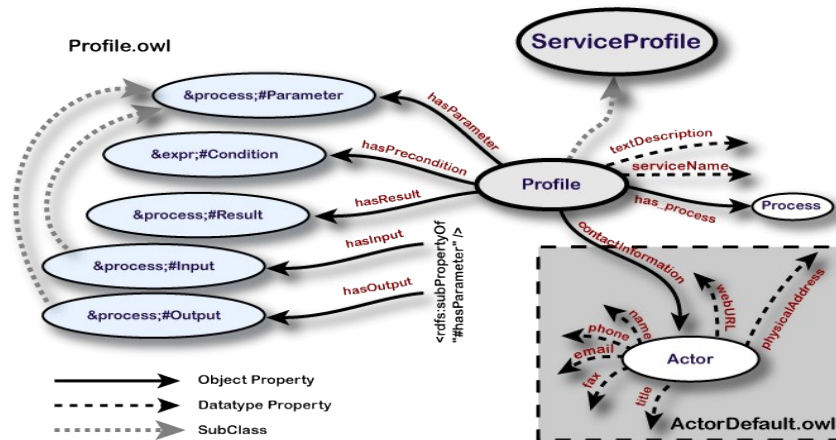


Figure 14: The OWL-S ServiceProfile ([96], Copyright © 2004 World Wide Web Consortium. All Rights Reserved. http://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231

Thus, to understand the OWL-S approach to semantic web services it is necessary to cover some of the key elements in the three service description sub-elements.

4.2.1 Service Profiles

The intention of this element is to advertise provided services and to support capability-based service discovery[12]. Therefore, the service profile has to capture a service's capabilities in a relatively generic, yet formal enough way to allow service requestors to find out what the service actually does. Therefore, OWL-S provides three different aspects: functional, non-functional and classification aspects. The functional aspect is covered by describing the inputs, outputs, necessary preconditions and effects (IOPEs) of the service. Beside these functional aspects the service profile also contains some non-functional aspects like elements that contain information about the service provider. As shown in Figure 14, the `ServiceProfile` class itself is not related to any properties but there exists a subclass called `Profile`. Instead of directly using this class, OWL-S promotes the creation of specialised subclasses of `Profile` to indicate special service types. This covers the classification aspect of capability descriptions, since the actual class memberships of a particular profile individual can already reveal the intention of a service (e.g. if an instance of a profile class is also a member of a class called `HotelBookingService`, this allows to subsume about the service's intention).

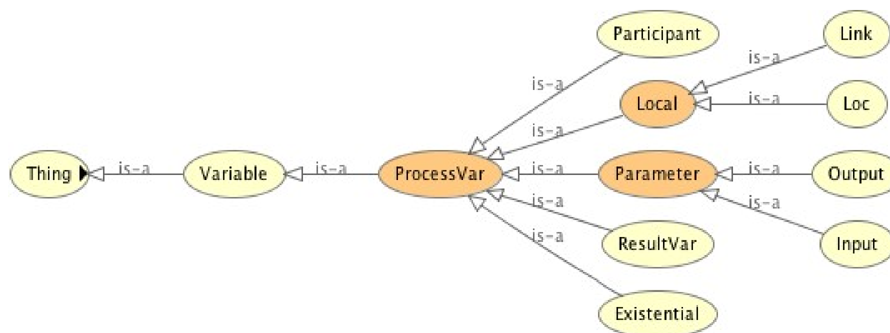


Figure 15: The OWL-S variable hierarchy (own illustration, restored from http://www.ai.sri.com/daml/services/owl-s/1.2/Process.owl using Protegé).

The profile class is related to the `Parameter`, `Condition`, `Result`, `Input` and `Output` classes via corresponding object properties. Input and output classes are subclasses of the `Parameter` class as shown in Figure 15. The actual instances of parameter, result, input and output elements are defined in the process or service model (see section 4.2.2), The service profile only refers to them. For the `ProcessVar` class exist two datatype properties called `parameterType` and `parameterValue`. The first one links process variables to elements of type `xsd#anyURI` whereas the latter one links to elements of type `xsd#XMLLiteral`.

To actually assign any OWL class as input or output to a service profile, OWL-S uses service specific individuals that are of type `Input` or `Output` respectively that link to the actual OWL input/output elements via their `prameterType` property. To illustrate this approach Figure 16 shows some aspects of the OWL-S CongoBuy[10] example – a fictional book selling company - that is part of the OWL-S reference documentation.

This OWL-S example, however, is suffering from a version mix-up between the two web sites http://www.daml.org/services/owl-s/1.2/ and http://www.ai.sri.com/daml/services/owl-s/1.2/. Whereas the official OWL-S 1.2 web site is the latter one, the OWL-S ontologies can be obtained from both sites. At least the process ontology (Process.owl), however, appears in two different versions. SRI's (Stanford Research Institute) web site offers version 1.148 (dated 2007/01/18) whereas the DAML site provides version 1.139 (dated 2005/05/18). Nevertheless, the OWL-S example ontologies available at http://www.ai.sri.com/daml/services/owl-s/1.2/examples.html import the outdated version of the process ontology from the DAML web site. The two different versions show considerable differences (e.g. in the variable class hierarchy). In this and the subsequent sections the analysis of OWL-S' features and characteristics is based on the latest version of the process ontology, which is also the basis of the technical documentation[97].

10 http://www.ai.sri.com/daml/services/owl-s/1.2/CongoService.owl

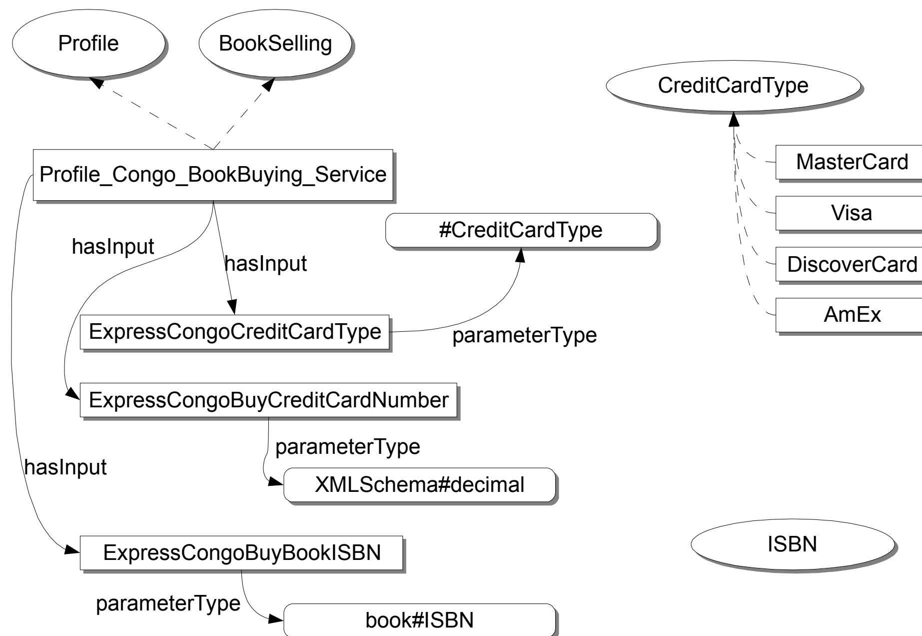


Figure 16: Definition of input variables in a service profile (own illustration)

Ellipses in Figure 16 represent classes, rectangles are individuals, rectangles with rounded corners are literals and solid arcs represent property relations. Dashed lines indicate a subclass relationship. *Profile_Congo_BookBuying_Service* is an instance of *Profile* but also of type *BookSelling*. The relation to *BookSelling* adds significant additional semantics to the service description and reveals important information about the type of the service. The profile is linked to several input elements. The input elements shown in Figure 16 are not complete, but comprise some representative examples. The service described here needs a credit card type as input information. Therefore, the service profile is not directly linked to the *CreditCardType* class, but holds a *hasInput* relation to an individual called *ExpressCongoBuyCreditCardType*. Consequently, *ExpressCongoBuyCreditCardType* is of type *Input* as well as of type *Parameter* and *ProcessVar* and therefore can have a *parameterType* property assigned to it. Since *parameterType* is a datatype property its value has to be a literal (of type *anyURI*) and it can't directly refer to the class *CreditCardType*. Instead it uses the URI of the *CreditCardType* class as its value. It is important to note that this does not create a direct semantic relationship between the class *CreditCardType* and the profile's input variables, since the value of the *parameterType* property is interpreted as an URI value and not as *CreditCardType*. Consequently you can't use a reasoner to query whether there exits a service profile that takes a *CreditCardType* as one of its inputs. However, you can query whether one of the input variables contains a *parameterType* that has the same value as the URI of the *CreditCardType* class. Thus to conclude that the URI of the *parameterType* actually is a *CreditCardType* requires additional interpretation of the URI value. Since *CreditCardType* is an enumeration of its individuals, which are different credit card brands, it can be inferred, which types of credit cards are accepted by the service.

Besides the credit card type also a credit card number is required. In the analysed example this fact is modelled by the introduction of the individual *ExpressCongoBuyCreditCardNumber*. The *parameterType* property of this individual refers to the decimal XML schema datatype. By evaluating this value, it can be inferred that the credit card number has to be a decimal number. Also here it is important to note that this description does not include any semantic relationship to any class or individual that would indicate that this number actually is the number of a credit card. The only glue about the actual nature of this number is the name of the input parameter (*ExpressCongoBuyCreditCardNumber*), although this could not be interpreted by any reasoner. Thus there is additional effort needed to indicate that this value actually means the number of a credit card as we will see later. The third example of an input parameter definition is the ISBN number of the book that should be purchased. Once again, there only exists an indirect reference to the *ISBN* class via the *ExpressCongoBuyBookISBN* individual's *parameterType* property. Like in the first example of the credit card type,

this information needs additional interpretation to figure out that the value of the URI refers to the `ISBN` class. This will lead to the information that an individual of `ISBN` is required by this service but does not include any information about the datatype or the format of this element. Assuming that there is a common understanding of this concept as well as the fact that there has to be a mapping between this element and some web service message part, this should not imply any problems. In fact, since the range of the `parameterType` property is `anyURI`, an URI literal assigned to this property can basically point to any web resource and is not limited to OWL classes or datatypes.

Although the input and output parameters are part of the functional description it is not required to list all of these parameters in the service profile. This is due to the intention of the service profile to advertise a service and to describe what a service does. Therefore, according to OWL-S' technical documentation, only relevant parameters have to be added to the service profile, even though there do not exist any rules or guidelines on how to identify the relevance of parameters. In general the example above shows that OWL-S' abilities to describe the inputs and outputs of a service are rather limited. This stems from the approach used by OWL to model properties as well as from the open world assumption. Since all properties are global, the existence of a specific property implies a certain type where as a type does not necessarily imply any properties. Thus, when there is a need to model the existence of a special datatype there has to be a reference like the one in the credit card number example, where, however, the semantic information gets lost. Otherwise there can be a reference to a class like in the ISBN example, although there are no longer any assumptions about the actual datatype or single class properties possible.

Besides input and output elements preconditions are an important part of a service profile. Preconditions in the OWL-S sense are defined as follows:

> *"A precondition is a proposition that must be true in order for the service to operate effectively" ([12], page 5)*

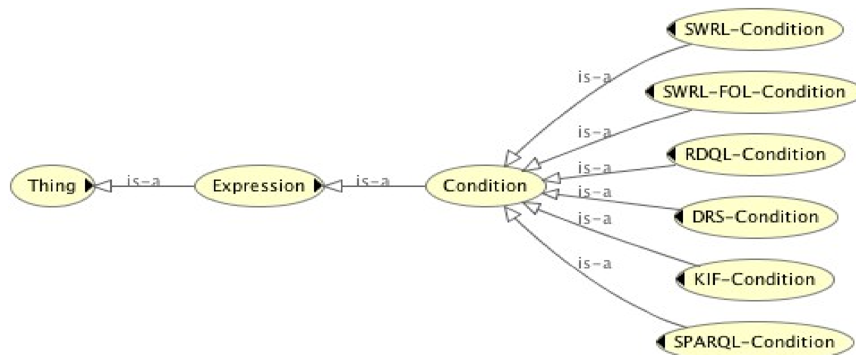


Figure 17: Class hierarchy to model preconditions in OWL-S (own illustration, extracted from http://www.daml.org/services/owl-s/1.2/generic/Expression.owl using Protogè)

Thus, preconditions can tell a client or agent whether it makes sense to call the service having given a specific situation. Preconditions, however, can also be used to tackle some of the problems that are implied by the limits of input and output variable definitions as we will see in an example bellow. OWL-S uses the `hasPrecondition` property to map service profiles to `Condition`s. Although description logics are not suited to explicitly model logical rule-like conditions, there exist ways to incorporate expressions from other languages into OWL-S. Figure 17 shows the class hierarchy that is used to model preconditions. This structure reveals how different logical expressions can be embedded into OWL-S.

The following languages are supported:

- **Semantic Web Rule Language** (SWRL)[98]: This language is the datalog (i.e. construction-function-free) sublanguage of the Rule Markup Language (RuleML) [99] restricted to unary or binary

predicates. This language allows to embed Horn-like rules consisting of heads (consequences) and bodies (conditions) into OWL.

- **Semantic Web Rule Language – First-Order Logic** (SWRL-FOL)[100]: Extends SWRL with some first-order logic constructs (e.g. universal and existence quantifiers).
- **RDF Data Query Language** (RDQL)[35]: This is a simple SQL-like query language that can match edges in RDF-graphs. It is not a classical rule language, but conditions can be stated within a query's where-clause.
- **Declarative RDF System** (DRS)[101]: This is a generalisation of SWRL that allows for predicates of arbitrary arity and quantifiers. It therefore is significantly more expressive than SWRL and SWRL-FOL.
- **Knowledge Interchange Format** (KIF)[102]: KIF was designed as interchange format between different computer systems. It can be used to state logical terms and sentences including quantifiers.
- **SPARQL Protocol and RDF Query Language** (SPARQL)[103]: SPARQL is a very recent language that was nevertheless widely adopted already[104]. Basically it is similar to RDQL since it also matches RDF-graphs but offers significantly more expressiveness (e.g. optional parts, unions, nesting, filtering, ..)

```
<process:hasPrecondition>
    <expr:SWRL-Condition rdf:ID="ExpressCongoBuyCreditExists">
        <rdfs:label>
            cardNumber(ExpressCongoBuyCreditCard, ExpressCongoBuyCreditCardNumber)
             &amp; validity(ExpressCongoBuyCreditCard, Valid)
        </rdfs:label>
    ...
   <expr:expressionObject>
      <swrl:AtomList>
        <rdf:first>
         <swrl:DatavaluedPropertyAtom>
           <swrl:propertyPredicate rdf:resource="#cardNumber"/>
           <swrl:argument1 rdf:resource="#ExpressCongoBuyCreditCard"/>
           <swrl:argument2 rdf:resource="#ExpressCongoBuyCreditCardNumber"/>
         </swrl:DatavaluedPropertyAtom>
        </rdf:first>
        <rdf:rest>
          <swrl:AtomList>
            <rdf:first>
              <swrl:IndividualPropertyAtom>
                <swrl:propertyPredicate rdf:resource="#validity"/>
                <swrl:argument1 rdf:resource="#ExpressCongoBuyCreditCard"/>
                <swrl:argument2 rdf:resource="#Valid"/>
              </swrl:IndividualPropertyAtom>
            </rdf:first>
            <rdf:rest rdf:resource="&rdf;#nil"/>
          </swrl:AtomList>
        </rdf:rest>
      </swrl:AtomList>
   </expr:expressionObject>
  </expr:SWRL-Condition>
</process:hasPrecondition>
```

Listing 28: Example of an OWL-S precondition. Taken from http://www.daml.org/services/owl-s/1.2/CongoProcess.owl.

Although all of these languages have their formal model theoretic semantics, this semantic is not understood by ordinary description logics reasoners. Thus, to actually evaluate these conditions appropriate additional reasoners have to be used. To get an idea of how rules are actually embedded into an OWL-S service profile Listing 28 provides and example. This precondition uses SWRL as its expression language and is basically a list of atomic formulae that contains two entries. The first axiom requires the two individuals `ExpressCongoBuyCreditCard` and `ExpressCongoBuyCreditCardNumber` to be linked by the cardNumber property. This conditions is essentially an RDF triple with the subject `ExpressCongoBuyCreditCard` the predicate `cardNumber` and the object `ExpressCongoBuyCreditCardNumber`. The `cardNumber` datatype property is also part

of the example and links a decimal number to the class *CreditCard*. Following an RDF and therefore also OWL interpretation of this assertion the individual *ExpressCongoBuyCreditCard* is of type *CreditCard* associated with a card number. *ExpressCongoBuyCreditCard* is not declared as an input variable but as so called local variable, which indicates its usage in conditions. Thus, this first condition is needed to add the semantics that is otherwise not covered by the declaration of the *ExpressCongoBuyCreditCardNumber* input variable.

The second axiom in Listing 28 defines that the object property *validity* has to link *ExpressCongoBuyCreditCard* to the individual *Valid*, which is an instance of *validity*'s range class *ValidityType*. This indicates that the provided credit card has to be valid. It is important to recognise that OWL and RDF reasoners would take these conditions as ordinary assertions rather than as conditions that have to be evaluated. Hence, additional tooling is needed to interpret these rules correctly as already mentioned above.

The next important elements of a service profile are the descriptions of the web service's results. This information is captured by the *Result* class. A web service is typically associated to several instances of *Result*, describing different possible outcomes[97].

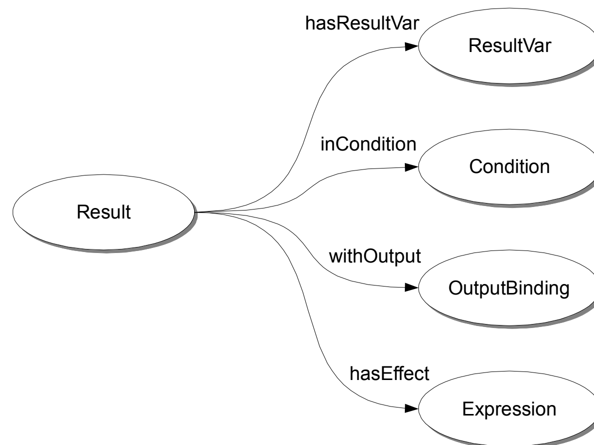


Figure 18: RDF-graph-like structure of the Result class an its properties (own illustration based on [97])

Figure 18 depicts the result class together with its properties. The *hasResultVar* property can be used to define so called *ResultVar*s. These are *ProcessVar*s like input and output variables (see Figure 15), but are only valid within the *Result* they are specified. They can be used as variables to share information between the condition, outputs and effects. Every *Result* can be associated with an expression that describes the circumstances under which this result will be the actual outcome of a service call. This condition is also used to initialise the result variables. Listing 29 shows the value of the *inCondition* property of the *ExpressCongoBuyPositiveResult*, which describes the outcome of the simple book selling service in case everything went right. The condition is expressed using SWRL and consists of two axioms. The first one defines that the individual *ExpressCongoBuyBook*, which is one the result variables, has to have the value of the input variable *ExpressCongoBuyBookISBN* associated to it using the *hasISBN* property. This defines that *ExpressCongoBuyBook* is of type *Book* and that it is exactly the one that should be bought. The second axiom requires that the *InStockBook* relationship holds true for this particular book.

```
<process:hasResult>
    ...
    <process:inCondition>
    <expr:SWRL-Condition rdf:ID="ExpressCongoBuyBookInStock">
            ...
    <expr:expressionObject>
     <swrl:AtomList>
```

```
      <rdf:first>
        <swrl:IndividualPropertyAtom>
        <swrl:propertyPredicate rdf:resource="&profileHierarchy;#hasISBN"/>
        <swrl:argument1 rdf:resource="#ExpressCongoBuyBook"/>
        <swrl:argument2 rdf:resource="#ExpressCongoBuyBookISBN"/>
       </swrl:IndividualPropertyAtom>
      </rdf:first>
      <rdf:rest>
        <swrl:AtomList>
        <rdf:first>
         <swrl:ClassAtom>
          <swrl:classPredicate rdf:resource="#InStockBook"/>
          <swrl:argument1 rdf:resource="#ExpressCongoBuyBook"/>
          </swrl:ClassAtom>
        </rdf:first>
        <rdf:rest rdf:resource="&rdf;#nil"/>
       </swrl:AtomList>
      </rdf:rest>
    </swrl:AtomList>
   </expr:expressionObject>
  </expr:SWRL-Condition>
   </process:inCondition>
</process:hasResult>
```

Listing 29: InCondition for the ExpressCongoBuyPositiveResult (taken from http://www.daml.org/services/owl-s/1.2/CongoProcess.owl).

The `withOutput` property can be used to map certain result variables to appropriate output variables, which describes the content of the messages that are returned by the service. In contrast to this, the effects of a result indicate how the state of the world is modified by this particular service outcome:

“An effect is a proposition that will become true when the service completes.” ([12], page 5)

To illustrate the relevance of effects, Listing 30 shows the effect definitions of the `ExpressCongoBuyPositiveResult`.

```
<process:Result>
...
     <process:hasEffect>
   <expr:SWRL-Expression>
            ...
    <expr:expressionObject>
     <swrl:AtomList>
         <rdf:first>
      <swrl:ClassAtom>
       <swrl:argument1 rdf:resource="#ExpressCongoBuyOutput"/>
       <swrl:classPredicate rdf:resource="#OrderShippedAcknowledgment"/>
      </swrl:ClassAtom>
     </rdf:first>
     <rdf:rest rdf:resource="&rdf;#nil"/>
    </swrl:AtomList>
   </expr:expressionObject>
  </expr:SWRL-Expression>
 </process:hasEffect>
 <process:hasEffect>
  <expr:SWRL-Expression rdf:ID="ExpressCongoOrderShippedEffect">
   ...
       <expr:expressionObject>
    <swrl:AtomList>
     <rdf:first>
      <swrl:ClassAtom>
       <swrl:classPredicate rdf:resource="#Shipment"/>
       <swrl:argument1 rdf:resource="#ExpressCongoBuyShipment"/>
      </swrl:ClassAtom>
     </rdf:first>
     <rdf:rest>
```

```
<swrl:AtomList>
 <rdf:first>
  <swrl:IndividualPropertyAtom>
   <swrl:propertyPredicate rdf:resource="#shippedTo"/>
    <swrl:argument1 rdf:resource="#ExpressCongoBuyShipment"/>
    <swrl:argument2 rdf:resource="#ExpressCongoBuyAcctID"/>
   </swrl:IndividualPropertyAtom>
  </rdf:first>
  <rdf:rest>
   <swrl:AtomList>
    <rdf:first>
     <swrl:IndividualPropertyAtom>
      <swrl:propertyPredicate rdf:resource="#shippedBook"/>
      <swrl:argument1 rdf:resource="#ExpressCongoBuyShipment"/>
      <swrl:argument2 rdf:resource="#ExpressCongoBuyBook"/>
     </swrl:IndividualPropertyAtom>
    </rdf:first>
    <rdf:rest rdf:resource="&rdf;#nil"/>
   </swrl:AtomList>
  </rdf:rest>
 </swrl:AtomList>
</rdf:rest>
</swrl:AtomList>
</expr:expressionObject>
</expr:SWRL-Expression>
</process:hasEffect>
</process:Result>
```

Listing 30: Effects of the ExpressCongoBuyPositiveResult (taken from http://www.daml.org/services/owl-s/1.2/CongoProcess.owl).

The first effect defines that the type of the output variable `ExpressCongoBuyOutput` will be `OrderShippedAcknowledgment` (see [98] for a definition of SWRL atoms). The second effect consists of three axioms. The first one defines that the type of the result variable `ExpressCongoBuyShipment` is `Shipment`. `Shipment` is an OWL class that encapsulates the recipient, the book that is shipped as well as the delivery and packaging type. The second axiom states that the recipient of the shipment is the customer identified by the current account id (`ExpressCongoBuyAcctID`). Finally the `ExpressCongoBuyBook` that was defined in the `InCondition` is selected as the one that is shipped.
Assuming that a software agent is capable of evaluating the expressions used in preconditions and results, a service's capabilities in terms of IOPEs can be figured out from the service profile. If these capabilities match the agent's intention or needs, the service model has to be evaluated in order to figure out how the agent can interact with the service.

4.2.2 Service Model

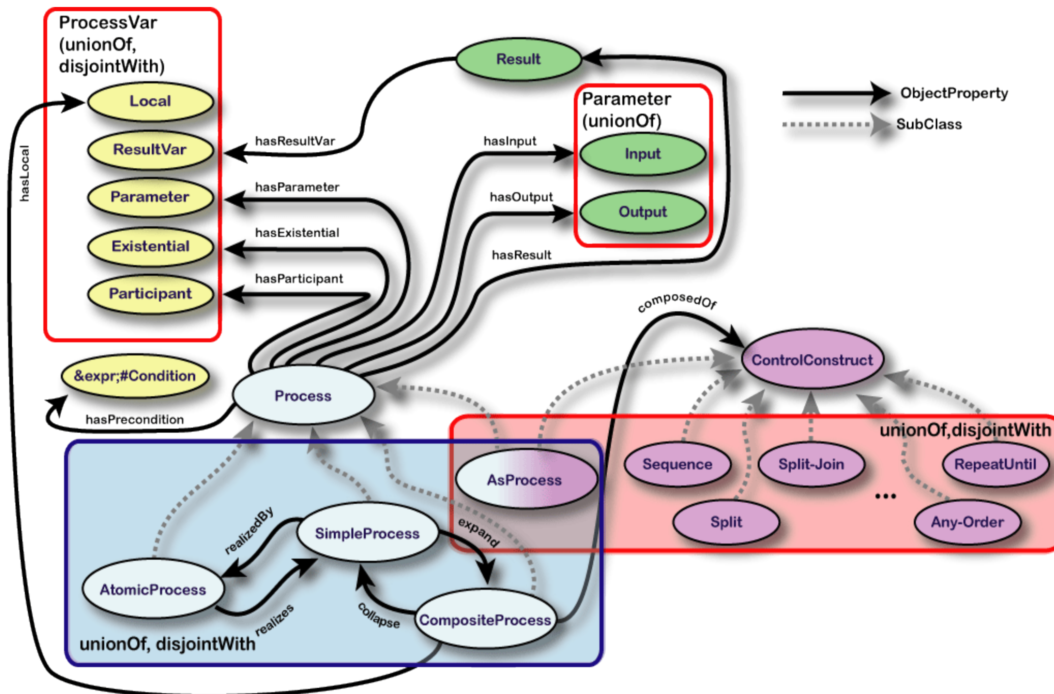


Figure 19: The top level OWL-S process ontology[97].

Whereas the service profile is used to advertise a service's capabilities, the service model describes the interaction pattern with the actual web service at a semantic level. Similar to the service model where a subclass of *ServiceModel* is used to describe a service's capabilities (*Profile*) also here a subclass is used to model the service interaction. This subclass is called *Process* and emphasises OWL-S' process oriented view on web services.

Figure 19 depicts the top level elements of the service model. Some of the elements shown here have already been discussed in the previous section. In fact the service profile refers to a process' input, output and result elements in order to describe a service's IOPEs. The service profile also has a property *has_process* that links the profile to the process (see Figure 14). There exist four different process subclasses. An *AtomicProcess* represents a single method invocation that has no internal state. In fact the *ExpressCongoBuy* process that was used as an example in the previous section is of this type. Therefore it is entirely defined by its inputs, outputs and results. A *CompositeProcess*, however, represents a composition of processes that are arranged in a programming-language-like control and data flow structure. Additionally a *SimpleProcess* provides an abstract view on atomic and composite processes. The *AsProcess* class is also a subclass of *ControlConstruct* and will be covered later in this section.

As indicated in Figure 19 a composite process is *composedOf ControlConstruct*s. OWL-S defines the following constructs to describe a control flow: sequence, split (concurrent execution that ends as soon as all composite tasks are started), split+join (concurrent execution that ends when the last composite task has finished), any-order (arbitrary selection of processing order but no concurrency), choice (one of the available options has to be selected), if-then-else, iteration, repeat-while and repeat-until. All of these constructs are subclasses of *ControlConstruct*. As shown in Figure 20 a composite process is only allowed to be composed of exactly one control construct. Every single control construct, however, may refer to any number of other classes using the *components* object property. For the base class *ControlConstruct* this property does not define a range attribute, thus it could link to any arbitrary OWL class (indicated by the question mark in Figure 20). For the specific subclasses of *ControlConstruct*, however, an object property restriction (see section 3.2.5.3) limits the range of this property either to a *ControlConstructList* or a *ControlConstructBag*. Both of these classes are lists of control constructs whereas the latter one does not impose any order.

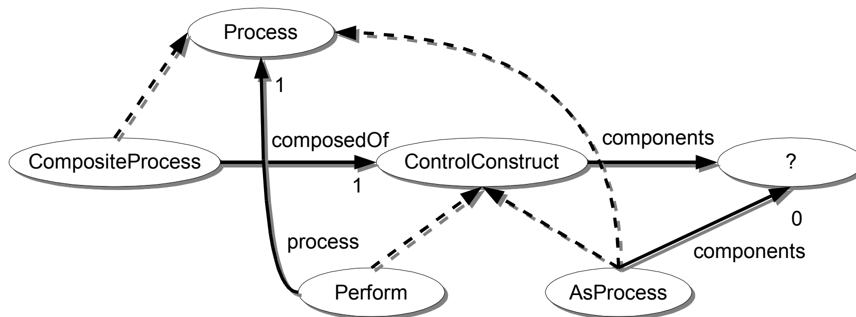


Figure 20: Basic OWL-S classes needed to model a composite process (own illustration based on [97]).

The actual control flow therefore consists of a tree of nested control constructs. The leafs of this tree represent processes that can be invoked. This can be modelled by either using a `Perform` control construct that refers to another process via its `process` property or by using an `AsProcess` control construct. The `AsProcess` class is a subclass of `ControlConstruct` as well as of `Process` but is disjoint from `SimpleProcess`. It cannot be further decomposed to other control constructs since the cardinality of its `components` property is set to zero. Thus, this element allows for the "inline" definition of processes.

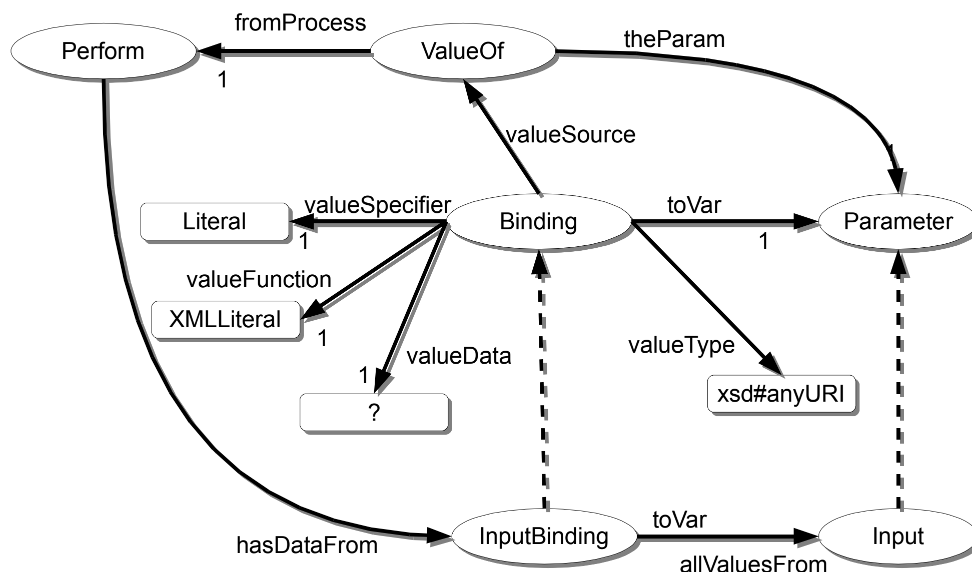


Figure 21: Data binding classes to define the input of component processes (own illustration constructed from http://www.ai.sri.com/daml/services/owl-s/1.2/Process.owl).

Like an atomic process also a composite process is described by its interface consisting of inputs, outputs and results. Thus there has to be a mean to model how the input data is passed to the individual process components and how the output of the composite process is constructed based on the component processes' outputs. Beside this, output of one component process can be necessary input for other ones. To organise this data flow OWL-S provides data binding components.

Figure 21 shows the necessary classes to model process input in a so called consumer-pull scenario, which covers the fact, that a component process defines the data needed from other processes. Therefore, a

Perform individual can use its *hasDataFrom* property to define data for the input variables of the component process it represents via its *process* property. The *hasDataFrom* property refers to an instance of *InputBinding*, a subclass of *Binding* that only allows for mappings to input variables. Consequently, the *toVar* property identifies the input variable of the *Perform* individual's process that will receive the value. To define where the data comes from *InputBinding* offers three different properties but only one of them should be used:

- *valueFunction*: This data type property is a sub-property of *valueSpecifier* and links the Binding class to an XML literal. By convention this literal is supposed to be an expression in any of the supported expression languages.
- *valueData*: Is also a sub-property of *valueSpecifier* and can be used to refer to literals that are used as constant values.
- *valueSource*: In contrast to the previous two properties *valueSource* is an object property and links the *Binding* class to a class called *ValueOf*. *ValueOf* uniquely identifies any value in the scope of the enclosing composite process by referring to a parameter (see Figure 15 for an overview of parameters) and to the process where this parameter occurs, by linking to the *Perform* element representing it. The special *Perform* individual called *ThisPerform* can be used to refer to the parent perform object of the current process. The *ValueOf* class can also be used to refer to other parameters inside an expression.

The *valueType* property is an URI that refers the type of the parameter.

Producer-Push is another approach to share information among the different steps of a composite process. The necessary classes are shown in Figure 22.

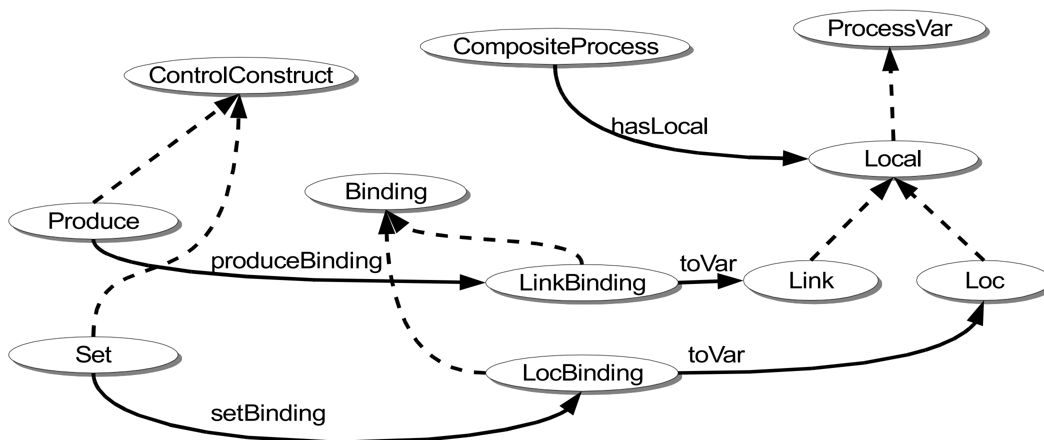


Figure 22: Required elements to model OWL-S producer-push scenarios within a composite process (own illustration based on [97]).

A composite process can define so called *Local* variables via its *hasLocal* property. These variables can be used to share information along the flow of the composite process and can be accessed via their names. There are two subclasses of *Local*: *Link* values can be written only once where as *Loc* values can be written as often as necessary.

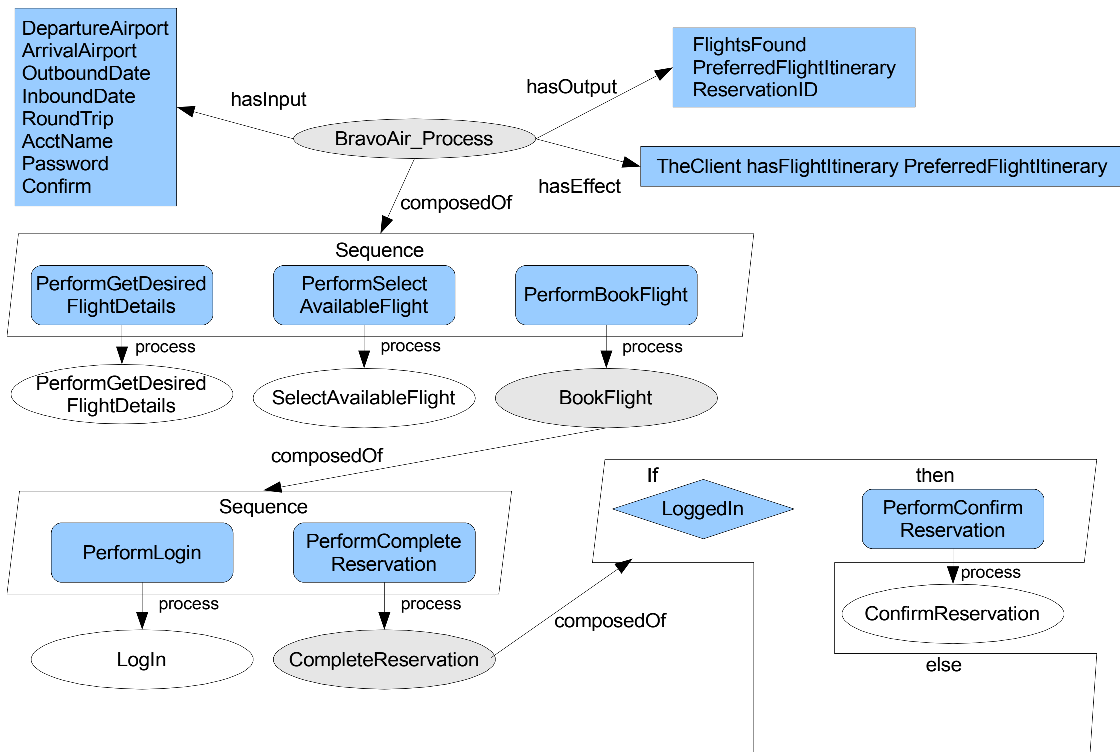


Figure 23: Example composite process reconstructed from http://www.ai.sri.com/daml/services/owl-s/1.2/BravoAirProcess.owl (own illustration)

There exist two specialised control constructs to actually assign values to these variables. *Produce* can be used to define a value for a *Link* variable using a *LinkBinding* instance and *Set* can be used to write a value to a *Loc* variable via a *LocBinding*. To identify the source of the data that should be written to *Loc* or *Link*, the binding class provides the same approaches already discussed above.

Figure 23 shows a composite process that describes a flight booking process that consists of atomic and other composite processes. As mentioned earlier composite processes cannot use more than one top-level control construct (sequence or if-then-else in this example). Control constructs in turn can refer to an arbitrary number of other control constructs (e.g. via lists and bags of control constructs) which includes *Perform* elements. It is important to notice, that the process description is not used by any server side process engine but it is an instruction for clients on how to use the various service methods in order to achieve a particular effect.

4.2.3 Service Grounding

The mapping between the semantic description and the actual web service description language document (WSDL) is called grounding. Whereas the service profile and the service model are considered to be abstract descriptions of the service the grounding contains the necessary information that allows an agent to execute the actual operation(s). This section covers the principles and most important OWL-S classes to ground a service. A detailed description on how to ground to a WSDL based web service can be found in [105].

Figure 24: OWL-S classes to model WSDL service grounding (extracted from http://www.ai.sri.com/daml/services/owl-s/1.2/Grounding.owl, own illustration)

Figure 24 provides an overview of OWL-S' classes and their relations in order to describe a service grounding. The central entry point is the `WSDLGrounding` class, which is a specialised sub-class of `Grounding`. Whereas `Grounding` constitutes a collection of `AtomicServiceGrounding`, `WSDLGrounding` is restricted to `WSDLAtomicServiceGrounding` instances only. From the various process types that can be modelled using OWL-S only `AtomicProcess`es can be grounded. Basically every atomic process represents one particular WSDL operation together with the necessary input and output messages. To map an `AtomicProcess` to the corresponding operation, `WSDLAtomicGrounding` uses the `owlsProcess` and `wsdlOperation` object properties. To map the atomic process's input and output parameters to the appropriate message parts of the corresponding operation's input and output messages OWL-S uses `WSDLOutputMessageMap` and `WSDLInputMessageMap`. WSDL usually uses XML schema datatypes to model the structure of message parts. OWL-S also allows for the use of OWL classes to describe the type of a message part. Therefore an additional namespace ("http://www.daml.org/services/owl-s/wsdl/") has to be added to the WSDL document that provides attributes to directly refer from message part tags inside WSDL to OWL classes. It is important to notice, however, that this approach is not supported by conventional web service tools. Thus, the automatic generation of web service client stubs will fail. Additionally OWL-S proposes the use of XML style sheets to map from OWL-S parameters to elements of complex XML schema types [105].

4.3 Semantic Web Service Framework (SWSF)

SWSF[106] is a comprehensive framework to define semantic web services that was submitted to the W3C by the Semantic Web Service Initiative[11]. Although it is heavily influenced by OWL-S it does not use OWL but

11 http://www.swsi.org/

defines its own set of language variants. SWSF can therefore be split into two major parts: T he Semantic Web Service Language (SWSL)[107] and the Semantic Web Services Ontology (SWSO)[108]. SWSL itself has two sublanguages: one (SWSL-FOL) that is based on first-order-logic[109] and one (SWSL-Rules) that is based on the rules/logic programming paradigm[110]. Both languages are significantly more expressive than OWL and meet the needs of describing web services in the terms of constraints and transitions. Consequently there are two versions of the ontology: First-Order Logic Ontology for Web Services (FLOWS) and Rules Ontology for Web Services (ROWS). The actual web service ontology clearly emphasises on the description of processes and resembles an extension of ISO 18629, the Process Specification Language (PSL)[111]. PSL is intended to be an exchange format for process definitions that consists of various ontologies expressed in Common Logic Interchange Format[112]. A key concept within PSL is *activity*. Activities can be composed of sub-activities. The simplest type of activity that can not be decomposed is called *primitive*. An *atomic* activity is either a primitive or a set of concurrent activities. PSL distinguishes between activities and activity occurrences, which intuitively reflects a potential activity execution. Every activity can occur several times.

SWSO adopts these principles an introduces the service theory. A *service* represents a semantic web service and can be associated to several descriptive elements. Beside this, every service is associated to a PSL activity that represents the actual process and to an occurrence of this activity. Like OWL-S also SWSO uses the notion of an atomic process that represents an invocable web service operation. Atomic processes therefore cannot be decomposed into further sub-activities and are related to PSL primitives. Domain-specific atomic processes are related to inputs, outputs, preconditions and effects (IOPEs). Beside this type of atomic processes SWSO also has the notion of so called message-specific atomic processes that either produce, read or destroy messages. Messages are represented by so called fluents, which are predicates that might change their values upon activity occurrences. Conditions used in an atomic process' precondition or conditional output definition are arbitrary first-order logic formulae.

Composite processes are represented by complex PSL activities. The actual flow of sub-activity occurrences is defined by control constraints that are functionally similar to those found in OWL-S (e.g . sequence, split, unordered. Please see [108] for a complete reference). In such choreographies domain specific atomic processes exchange messages via *Produce_Message* and *Read_Message* activities.

The grounding of SWSO services into WSDL is relatively straight forward and similar to OWL-S' grounding. Activities are mapped to operations and SWSO messages are mapped to an operation's input and output messages. Obviously SWSF has never left the state of a specification and was not adopted by tool vendors or practicians. This is why this framework is only discussed briefly here, however, the architecture of WSMO (see next section) is based on SWSF recommendations.

4.4 Web Service Modelling Ontology (WSMO)

WSMO[113] is another framework that was specifically designed to model semantic web services. It has adopted some of the core principles of SWSF and is separated into the conceptual framework and a specific language, the Web Service Modelling Language (see section 3.4). Like SWSF, also WSMO provides a description logics and a logic programming variant. As already mentioned in the WSML section, WSMO defines the following top-level elements:

- goal
- ontology
- webservice
- mediator
- capability
- interface

Whereas the underlying ontology was already discussed in detail, this section provides an analysis of the remaining constructs. Technically WSMO/WSML is based on the Meta Object Facility (MOF)[6], consequently WSMO is basically a meta-model for semantic web services.

4.4.1 The WebService Element

This WSMO element captures the model of the actual executable web service. The MOF meta-model defining the structure of this element is given in Listing 31.

Class webService **sub-Class** wsmoElement
  importsOntology **type** ontology
  usesMediator **type** {ooMediator, wwMediator}
  hasNonFunctionalProperties **type** nonFunctionalProperty
  hasCapability **type** capability multiplicity = single-valued
  hasInterface **type** interface

Listing 31: Meta-model definition of the WSMO webService element[114]

Every `webService` instance can import an arbitrary number of ontologies that define concepts, which can be used in the other elements. If there is some need to translate between different ontologies, also mediators can be defined. Non functional properties are used to describe aspects related to quality, security, costs, trust and reliability aspects together with a description of the provider of the service. Generally non functional properties can be used to provide arbitrary additional information but can also contain logical expressions.

Class capability **sub-Class** wsmoElement
  importsOntology **type** ontology
  usesMediator **type** {ooMediator, wgMediator}
  hasNonFunctionalProperties **type** nonFunctionalProperty
  hasSharedVariables **type** sharedVariables
  hasPrecondition **type** axiom
  hasAssumption **type** axiom
  hasPostcondition **type** axiom
  hasEffect **type** axiom

Listing 32: Meta-model definition of the capability element[114]

A central part of a WSMO web service description is the `capability` element (see Listing 32). The `importsOntology` and the `usesMediator` properties have the same purpose as for the `webService` concept. The remaining properties, however, are used to describe the IOPE's of a web service like already discussed in section 4.2.1.

The `hasSharedVariables` property is used to define variables that can be used in the precondition, assumption, post-condition and effect axioms, which are basically logical predicates. Shared variables can be compared to OWL-S' process variables since there is no explicit distinction according to their role in the service, such as input, output or internal values. Variables in WSMO/WSML are named values that start with a questioned mark. Every variable used in any of the subsequent elements has to be defined in this property, thus it can be seen as a scope for variables belonging to a capability description. The general contract for shared variables, preconditions, assumptions, postconditions and effects is the following [113]:

```
forAll ?v1,...,?vn ( preconditions(?v1,...,?vn) and assumptions(?v1,...,?vn)
       implies ( postconditions(?v1,...,?vn) and effects(?v1,...,?vn) ).
```

This formalises the fact that whenever the stated preconditions and assumptions hold for the given values assigned to the shared variables the defined postconditions and effects will hold as well. Whereas OWL-S uses OWL to embed the description of rules, WSMO can directly model these conditions as WSML axioms. WSML reasoners therefore can evaluate these rules.

Preconditions define the necessary input and the required state of the world related to this input. This is also called the information space, since it is only related to concepts and properties that are directly accessible. The actual web service checks the validity of the preconditions and can't be successfully enacted if they are not met. To illustrate the meaning of this element Listing 33 provides an example describing a service from a "Virtual Travel Agency" (VTA)[12]. The variable `reservationRequest` represents the input to the web service but

12 Taken from http://www.wsmo.org/TR/d17/industryTraining/SWS-tutorial-potsdam-20070220.pdf

does not appear as a shared variable. Although it is not explicitly stated as a *sharedVariable*, its existence as input concept is required since the precondition refers to this concept's attributes. The precondition axiom is a logical expression that has to hold true. The first few lines are used to assign values to most of the shared variables. These values are then restricted to particular individuals.

```
capability VTAcapability
   sharedVariables {?item, ?passenger, ?creditCard, ?initialBalance, ?reservationPrice}
   precondition
      definedBy exists ?reservationRequest
        (?reservationRequest[ reservationItem hasValue ?item,
        passenger hasValue ?passenger,
        payment hasValue ?creditcard]
        memberOf tr#reservationRequest and
        (?item memberOf tr#trip or ?item memberOf tr#ticket) and
        ?passenger memberOf pr#person and
        ?creditCard memberOf po#creditCard and
        (?creditCard[type hasValue po#visa] or
        ?creditCard[type hasValue po#mastercard]) ) .
```

Listing 33: Definition of the preconditions for the Virtual Travel Agency example web service

Basically this precondition defines that the value of *?reservationRequest* has to be an instance of a WSML concept called *reservationRequest*. This requires this individual also to meet all general constraints that are defined for this concept in the ontologies (e.g. cardinality restrictions for its properties). Furthermore this individual's *reservationItem* property is only allowed to refer to *trip*s or *ticket*s, reservations are exclusively allowed for *person*s and the only accepted *payment* method are *creditCard*s of type Visa or Mastercard.

```
assumption definedBy
    po#validCreditCard(?creditCard) and
    ?creditCard[balance hasValue ?initialBalance] and
    (?initialBalance >= ?reservationPrice) .
```

Listing 34: Assumption definition of the Virtual Travel Agency example

Listing 34 shows the definition of an assumption taken from the same example. Assumptions are used to express presumptions about the state of the world that are necessary for the service in order to complete correctly. The major difference between assumptions and preconditions is that assumptions are not directly checked by the service. Nevertheless, a service call will fail if the assumptions are not met. That is why the error message of a failed service call should contain a description of all assumptions that did not hold in order to provide the caller with additional information to look up other services that might establish the necessary conditions. In the example above the provided credit card has to be valid and it has to be able to cover the price of the reservation.

```
postcondition definedBy
   exists ?reservation(?reservation[
      reservationItem hasValue ?item,
      price hasValue ?reservationPrice,
      customer hasValue ?passenger,
      payment hasValue ?creditcard]
   memberOf tr#reservation and
   ?reservationPrice memberOf tr#price) .
```

Listing 35: Postconditions of the Virtual Travel Agency example

Postconditions define the guaranteed state of the information space after the service was executed successfully. They also establish a relationship between the input to the web service and its output. An example definition of a postcondition is presented in Listing 35. This example guarantees the existence of an

instance of the type *reservation* that has specific property values set. This includes the actual item that was reserved (either a trip or a ticket according to the precondition), the price, the passenger and the payment method.

Effects are conceptually similar to postconditions. Like with preconditions and assumptions, postconditions define the state of the information space, whereas effects describe how the successful execution of the actual web service will change the state of the world. Listing 37 shows an example of an effect definition,

Class interface **sub-Class** wsmoElement
importsOntology **type** ontology
usesMediator **type** ooMediator
hasNonFunctionalProperties **type** nonFunctionalProperty
hasChoreography **type** choreography
hasOrchestration **type** orchestration

Listing 36: Meta-model definition of the WSMO/WSML interface element

which states the current balance of the credit card used will be reduced by the price of the reservation.

Whereas the capability element of a WSMO web service description defines the service's IOPE's, the description of how to actually interact with the service is captured by the *webService*'s *interface* element. The MOF meta-model definition of the interface element is shown in Listing 36. The first three properties of the *interface* element have the same function and meaning as for the previously discussed elements. Thus the specific properties of the *interface* definition are *hasChoreography* and *hasOrchestration*. Before these elements are presented in more detail, Figure 25 provides an overview of how capability, choreography and orchestration elements are used to semantically describe a web service. The choreography describes the interaction pattern between the agent or user and the actual web service and can therefore be compared to OWL-S' composite process element (see section 4.2.2). The orchestration element in turn describes how the web service uses other web services in order to achieve its goals.

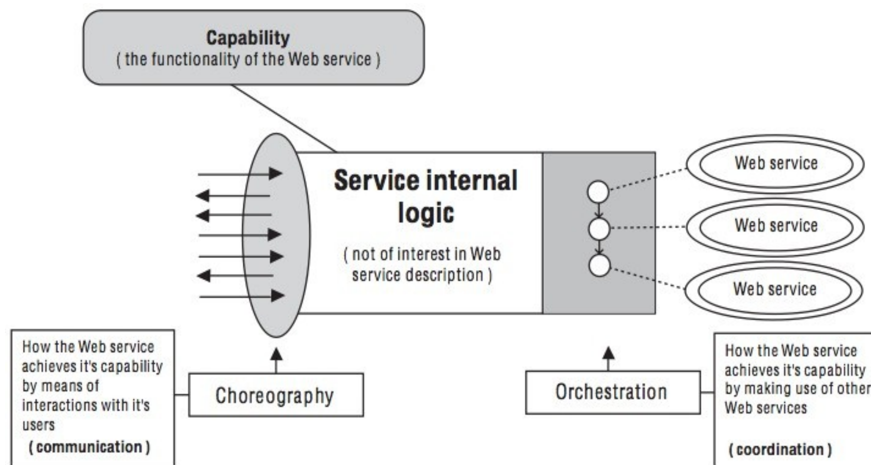


Figure 25: The WSMO approach to describe a web service's functional aspects (Reprinted from [113], page 87 with permission from IOS Press)

WSMO's *orchestration* element has undergone significant changes over the course of the specification process. The most recent available specification[115] dates from February 2007. The latest specification of WSML[116] from August 2008, however, which defines the syntax and semantic of all WSMO constructs, did not incorporate the latest changes of the interface element, but seems to reflect the state of a previous version[117]. The following analysis refers to the latest version of the orchestration specification. Identified inconsistencies are mentioned wherever they occur.

WSMO's choreography description is based on an approach called abstract state machines[118] (ASM).

ASMs are a generalisation of finite state machines in which states are not described by a finite collection of names but by arbitrary mathematical structures. The effective state of an abstract state machine is defined by the current values of its so called locations. A location can be seen as a function that depends on an arbitrary number of parameters and a value of arbitrary type. An ASM consists of a finite number of update rules that can modify the current value of a location. However, not all locations are updatable. Within an ASM locations are separated into the following categories that are adopted in the WSMO choreography as well[119]:

- *Static*: These are locations that are never updated. Thus they can be defined using functions or axioms.
- *In*: Values of these locations can only be updated by the environment (e.g. by external sensors or the user of a system) but can be read by the ASM. These locations are also called *monitored*.
- *Out*: These are locations that are only updated by the ASM but can be read by the environment.
- *Controlled*: These are locations that are only write- and readable by the ASM. The environment has no access to these values at all.
- *Shared*: This describes locations that can be updated and read by the ASM as well as by the environment.

ASM rules – also called guarded update rules – have the following form:

if *Condition* ***then*** *Updates*

Beside the classical conditional rule there exists a selection rule and a universally quantified rule of the following forms:

choose *x* ***with*** *Condition* ***in*** *Updates*

forAll *x* ***with*** *Condition* ***do*** *Updates*

Updates is represented by a finite set of location assignments of the form $f(t_1,...,t_n):=t$, where f is the name of the location and $t_1,...t_n$ are the parameters of the location according to its arity n ≥0. All ASM rules are executed simultaneously, thus, all updates with a satisfied condition happen in parallel.

```
Class choreography
   hasNonFunctionalProperties type nonFunctionalProperties
   hasStateSignature type stateSignature
   hasState type state
   hasTransitionRules type transitionRules
```

Listing 38: Meta-model definition of the WSMO choreography element[115]

Listing 38 presents the definition of the choreography element that is used to model the interaction pattern with the service as an abstract state machine. Consequently it consists of all elements needed by an ASM definition.

The *hasStateSignature* property refers to an element of type *stateSignature* that contains the definitions of the various locations (see Listing 39). The meaning of these properties is the same as for the abstract state machine. Concepts that are used as in, out or shared locations have to be defined with a grounding (see section 4.4.3). The *hasState* property of the *choreography* element defines the possible states of the ASM in terms of ground facts, which in this case are WSMO instances. This element, however, is not part of the WSML syntax definition[116] nor is it formally defined in the syntax or semantics section of the choreography definition document itself[115]. There is also no formal specification of the *state* type that is used in the WSMO meta-model as far as the reference documentation is concerned. Generally the state of an ASM is defined by the set of the current values of its locations. Since locations can be of arbitrary type there is actually no need for an ASM's state to be finite.

Class stateSignature
hasNonFunctionalProperties **type** nonFunctionalProperties
importsOntology **type** ontology
usesMediator **type** ooMediator
hasStatic **type** mode
hasIn **type** mode
hasOut **type** mode
hasShared **type** mode
hasControlled **type** mode

Class mode **sub-Class** {concept, relation}
hasGrounding **type** grounding

Listing 39: Definition of the stateSignature and the mode class[115]

However, there is a variant of ASMs called control state ASMs[119] that maintain the nature of named states like in finite state machines (FSM). These types of machines use a special variable called `ctl_state` that holds the FSM style state information. Update rules of control state ASMs take the current `ctl_state` into account and have the following form:

if ctl_state = i **and** cond **then** rule
ctl_state := j

The `transitionRules` property of a choreography definition defines the update rules of the ASM. The syntax of the transition rules is almost identical to the rules definition in classical ASM:

if *Condition* **then** *Rules* **endIf**
forall Variables **with** Condition **do** Rules **endForall**
choose Variables **with** Condition **do** Rules **endChoose**

The `forall` rule executes all rules that meet the given condition whereas the `choose` rule randomly takes one of the locations that meet the given condition and executes the update. This explicitly models the non-deterministic nature of ASMs. The actual rules are either `add`, `delete` or `update` and can be used to add, delete or update instances or attribute values of instances. Each `update` rule can be decomposed into a sequence of a `delete` and an `add` rule that has the same effect. Add and delete rules are therefore called primitive rules. As a result all rules with a met guard condition form an update set U that turns the current state S into an updated state S^U by the following transitions, where a is a ground atomic WSML formula:

$$S^U = S \setminus \{a \mid \mathbf{delete}(a) \in U\} \cup \{a \mid \mathbf{add}(a) \in U\}$$

To illustrate the application of the choreography element Listing 40 shows an example of the virtual travel agency[13]. Although the use of the choreography's state as well as the use of the `ctl_state` variable are not covered by the WSML syntax reference, this sample indicates that the latest revision of the WSMO choreography favours control state ASMs over general ASMs. The sample ASM in this snippet consists of three states. From the initial state (`htl#start`) there are transitions either to the state `htl#offerMade` or `htl#noAvail`. This is modelled by so a called piped rule, where several alternative rules are separated by the pipe character ("|"). The semantics of piped rules is that one of the available update rules is randomly chosen. Thus the consecutive state is determined in an entirely non-deterministic fashion. This example, however, also contains an error, since the `?name` variable is used in the first `add` rule but is not defined in the variable section of the `forall` rule and also not bound in the `with` clause, as required by the WSMO choreography specification. When reduced to the actual message exchange the meaning of the ASM modelled in Listing 40 is that the service might respond to a `HotelRequest` message either with a `HotelOffer` message (indicating the state `offerMade`) or a `HotelNotAvailable` message (indicating the state `notAvail`). All these messages are appropriately defined as either input or output messages in the choreography's state signature. The current state, represented by the value of the `ctl_state` variable, can be used to reason which messages are expected next by the service. The *with* clause of the transition rules can in turn be used

13 Taken from http://www.wsmo.org/TR/d17/industryTraining/SWS-tutorial-potsdam-20070220.pdf

to model the dataflow between consecutive message exchanges (e.g. by assigning values of previous output concepts to attributes of expected input concepts).

```
interface htl#BookHotelInterface choreography
  stateSignature importsOntology htl#simpleHotelOntology
     in htl#HotelRequest withGrounding _"http://...",
        htl#HotelConfirm withGrounding _"http://...",
        htl#HotelCancel withGrounding _"http://..."
     out htl#HotelNotAvailable withGrounding _"http://...",
        htl#HotelOffer withGrounding _"http://..."
     shared htl#Hotel, htl#HotelAvailable, htl#HotelBooked
     ctl_state {htl#start,htl#offerMade,htl#noAvail,htl#confirmed,htl#cancelled}
  transitionRules
     if (ctl_state = htl#start) then
        forall {?req,?date,?loc,?client} with
           ?req[trv#date hasValue ?date, trv#location hasValue ?loc,
           htl#client hasValue ?client] memberOf htl#HotelRequest
        do
           add(htl#offer(?req)[trv#date hasValue ?date,
              trv#hotelName hasValue ?name, trv#location hasValue ?loc,
              htl#client hasValue ?client] memberOf htl#HotelOffer)
           ctl_state := htl#offerMade
           |
           add(htl#notAvailable(?req)[trv#date hasValue ?date,
              trv#location hasValue ?loc] memberOf htl#HotelNotAvailable)
           ctl_state := htl#noAvail
        endForall
     endIf
```

Listing 40: Example choreography definition of the Virtual Travel Agency

An additional example containing a choreography can be found in [120]. It demonstrates the use of the Amazon web service but reflects the previous version of the choreography specification only. Thus, there is no state variable and dependencies between consecutive operations have to be modelled by making the existence of output concepts of previous operations part of the if-condition of the corresponding transition rules.

4.4.2 The Goal Element

WSMO goals represent the objectives of a client that should be met by the execution of appropriate web services. Thus, goals are the base elements for identifying available web services that can achieve the desired intention. As shown in Listing 41 a goal in WSMO is syntactically almost identical to WSMO's web service description element (see Listing 31). The *requestsCapabilty* element can be used to define the required functionality whereas the *requestsInterface* element can be used to define the expected communication behaviour of suitable web services[121].

```
Class goal sub-Class wsmoElement
   importsOntology type ontology
   usesMediator type {ooMediator, ggMediator}
   hasNonFunctionalProperties type nonFunctionalProperty
   requestsCapability type capability multiplicity = single-valued
   requestsInterface type interface
```

Listing 41: WSMO meta-model definition of the goal element [114]

This, however, requires the user to express the requested functionality or behaviour in significant formal detail that depends on in-depth WSMO knowledge and skills. To facilitate the usage of WSMO goals new

recommendations[122] suggest pre-defined goals that are defined by ontology designers and should be stored in a goal repository. These pre-defined goals are called *goal templates*, whereas goals that have concrete values assigned to their defined input variables are called *goal instances*.

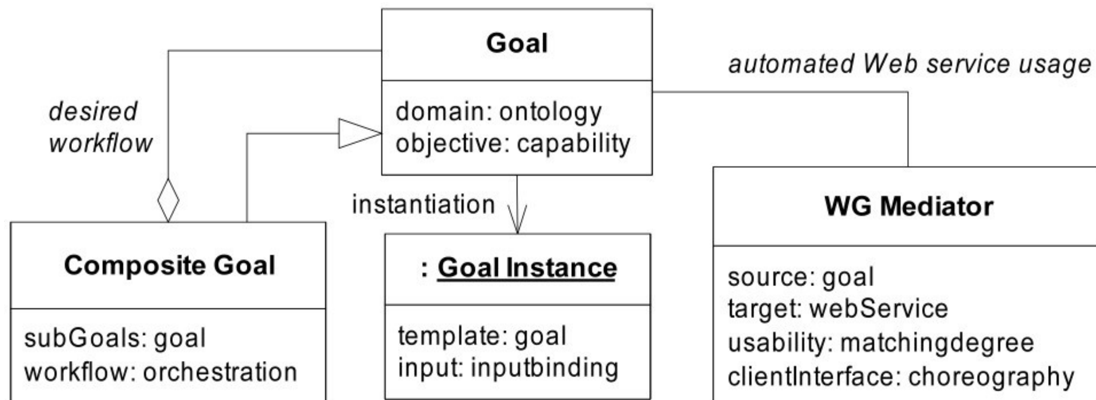


Figure 26: WSMO Goal Model Overview[122]

The entire recommended goal model is shown in Figure 26. Besides goal templates and goal instances also so called composite goals are proposed that can be used to model more complex goals as an orchestration of sub-goals. A *WG mediator* that links goals to web service descriptions is used to identify appropriate web services. The formal semantics of this approach is also provided by a model theory that makes use of so called Abstract State Spaces (ASS), which basically represent all valid states of the world of discourse [123]. A web service is considered a sequence of state transitions $\tau=\{s_0,...,s_m\}$ that transfer the world from an initial state s_0 into a final state s_m. Since web service and goal descriptions are made up of the same elements (i.e. capabilities and interfaces) the same basic semantics applies to them:

> **Definition 3** [122]: A capability is a 9-tuple $C=(O,\Sigma_A,IN,OUT,\Phi^{pre},\Phi^{ass},\Phi^{post},\Phi^{eff},NFP)$ with:
> O ... set of imported Ontologies
> Σ_A ... the state signature consisting of all dynamic symbols Σ_D and all static symbols Σ_S
> $IN=(i_1,...,i_n)$ the set of all input variables
> $OUT=(o_1,...,o_m)$ the set of all output variables
> $\Phi^{pre},\Phi^{ass},\Phi^{post},\Phi^{eff}$ ... preconditions, assumptions, post-conditions and effects
> NFP … non functional properties
>
> Let $\tau=\{s_0,...,s_m\}$ be a sequence of state transitions in an Abstract State Space A.
> The meaning of C is that if $s_0 \models_{wsml} \Phi^{pre}$ then $s_m \models_{wsml} \Phi^{post}$ and $s_m \models_{wsml} \phi^{eff}$ if for all $s \in \tau$ holds $s \models_{wsml} \phi^{ass}$. If this holds, we say that τ satisfies C, denoted by $\tau \models_A C$.

Thus, every web service W can be interpreted as set of valid state sequences $\{\tau\}_W$ that hold for the given capability C_W. Therefore $W \models_A C_W$ if for all $\tau \in \{\tau\}_W$ holds $\tau \models_A C_W$. Conforming with this every goal G can be described by a capability C_G such that $\{\tau\}_G$ is the set of all state transitions which are solutions to G and for all $\tau \in \{\tau\}_G$ holds $\tau \models_A C_G$.

A goal instance $GI(G)$ is created by assigning concrete values to the input variables of a capability C_G. This is achieved by a so called input binding β that maps IN-variables to elements of the universe U_A:

$$\beta:\{i_1,...,i_n\} \rightarrow U_A$$

An input binding is considered to be valid if the current state of the world s_c that is indicated by β meets the required preconditions of the capability C_G:

$$s_c,\beta \models_{wsml} \Phi^{pre}$$

By substituting all occurrences of the IN-variables in all formulae ϕ of C_G, all possible end-states for a

particular goal instance $GI(G)$ can be determined. Since concrete values limit the number of potential solutions it holds that $\{\tau\}_{GI(G)} \subset \{\tau\}_G$. This view on web services and goals allows for formal match-making between goals and goal instances on the one side and web services on the other side, enabling the discovery of appropriate services that meet the user's desires[124]:

Definition 4 [124]: "*Let W be a Web service, G a goal template, and $GI(G)$ a goal instance that instantiates G with an input binding β. Let $\tau=(s_0, \dots, s_m)$ be a sequence of states in an Abstract State Space A. We define the following sets:*

$\{\tau\}_G$	*:= possible solutions for G*
$\{\tau\}_W$	*:= possible executions of W*
$\{\tau\}_{GI(G)} \subset \{\tau\}_G$	*:= possible solutions for $GI(G)$ that defines β*
$\{\tau\}_{W(\beta)} \subset \{\tau\}_W$	*:= possible executions of W when invoked with β*

We define the usability of a Web service for solving a goal as:

(i) $match(G,W) \quad : \exists \tau . \tau \in (\{\tau\}_G \cap \{\tau\}_W)$

(ii)$match(GI(G),W) \quad : \exists \tau . \tau \in (\{\tau\}_{GI(G)} \cap \{\tau\}_{W(\beta)})$"

Clauses (i) and (ii) in Definition 4 define the matching criteria between web services and goal templates or goal instances respectively. If there is at least one possible solution part of the web service's executions th en this service can solve the given problem. Since $\{\tau\}_{GI(G)} \subset \{\tau\}_G$ a match for a goal instance also implies a match for the corresponding goal template ($match(GI(G),W) \quad match(G,W)$). On the other side, if there does not exist a match for the goal template there cannot exist a match for any of its instances: $match(G,W) \quad match(GI(G),W)$. This justifies the so called two-phase web service discovery approach[124] in which suitable web services are linked to corresponding goal templates at design time. Thus, only these web services have to be considered during run-time when looking up an appropriate web service for the given goal instance. This significantly accelerates the discovery process but limits the possible results to those services known during design time. Whereas Definition 4 provides the basic requirements for a web service that can solve a given goal, the extent to which a web service's functionality matches the required functionality of a goal can be further classified. WSMO distinguishes the following degrees of matches between goal templates and web services:

- ***exact****:* The set of transitions sequences of the goal template and the web service are identical (if and only if $\tau \in \{\tau\}_G$ then $\tau \in \{\tau\}_W$)
- ***plugin****:* The set of transition sequences defined by the goal template is sub-set of the web service's transitions (if $\tau \in \{\tau\}_G$ then $\tau \in \{\tau\}_W$)
- ***subsume***: The set of the web service's possible transition sequence is a sub-set of the goal templates possible transition sequences (if $\tau \in \{\tau\}_W$ then $\tau \in \{\tau\}_G$)
- ***intersect***: There exists at least one transition sequence that is part of both sets (there is a τ such that $\tau \in \{\tau\}_G$ and $\tau \in \{\tau\}_W$). This is equivalent to clause (I) of Definition 3.
- ***disjoint***: There is no common transition sequence, thus, the web service cannot solve the given goal template (there is no τ such that $\tau \in \{\tau\}_G$ and $\tau \in \{\tau\}_W$).

From the first two matching degrees directly infers that the web service can also solve any instance of the goal template, whereas for the latter two additional tests have to be conducted to see whether the given goal instance contains transition sequences that are also part of the web service's set of transition sequences.

To relate web services to goal templates so called WG (web services to goal) mediators are used (compare Figure 26). According to [122] a WG mediator should consist of the following elements:

- A *source* attribute that refers to a goal template
- A *target* attribute referring to the web service
- date and process level mediation facilities

- the matching degree between goal template and web service like described above
- a client interface in oder to utilise the web service

This, however, is a recommendation for the implementation of the two-phase web service discovery approach. The current specification of the `wgMediator` element in WSML[116] only contains the first three elements and the `sources` attribute refers to web services whereas the `target` attribute links to a goal.

4.4.3 WSMO Grounding

WSMO terminology refers to the WSMO part of a web service as the semantic description whereas the WSDL description of a web service is called the syntactic description. The mapping between corresponding elements in the semantic and the syntactic description which is necessary to actually invoke the web service is called grounding[125]. Like described in the previous sections, the semantic description of a web service in WSMO consists of its capabilities and its interfaces including its choreography. Accordingly WSMO distinguishes between the mapping of concepts that are used as messages, which is called data grounding and the mapping of the interaction patterns, which is called behaviour grounding.

WSMO proposes three different approaches for data grounding:

1. Create a mapping at the meta-model level from XML schema to WSMO concepts and axioms. This allows for automatic creation of an ontology based on XML schema datatypes.
2. Transfer between the XML serialisation of WSML and the web service's messages using technologies like XML style sheets.
3. Use a specialised mapping language to map directly between XML messages and the WSMO ontology.

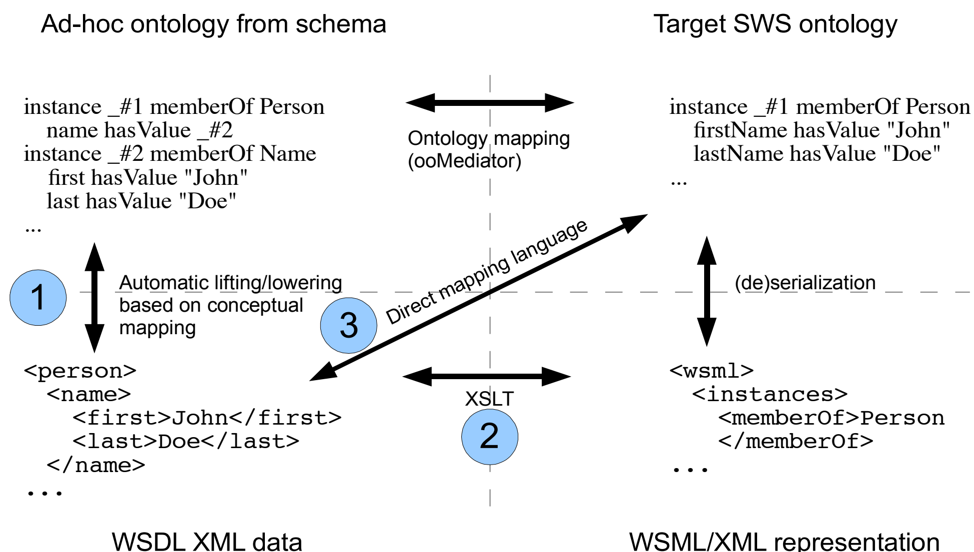


Figure 27: WSMO data grounding approaches[125]

Figure 27 provides an overview of the approach mentioned. At the upper right corner of this illustration a sample snippet of an instance modelled in the so called target ontology is shown. This is the ontology that is used to provide the semantic description of the web service. In the lower left corner a part of an actual XML message as required by the web service is shown. The first data grounding approach is based on the creation of a so called ad-hoc ontology. This is an ontology that is automatically created by direct translation of the corresponding XML schema (XSD) into WSMO elements. A mapping between XSD elements and

WSMO/WSML elements enables this transformation. XSD elements are mapped to WSML according to their type. Since WSML uses the XSD type schema, simple types can be used on an as-is basis. Complex XSD types are represented by concepts, where each element is either represented by a corresponding simple type property or an object property. Restrictions are converted into concepts together with constraining axioms. XML attributes are also mapped to concepts representing them. The complete mapping can be found in [126]. This approach supports automatic conversion from XML data to corresponding WSMO instances (lifting). Since every automatically created WSMO concept holds a mapping to its source XSD element in its non-functional properties, also conversion from WSMO instances to XML data (lowering) is possible. Due to the low-level mapping between XSD and WSMO, ad-hoc ontologies are relatively extensive and reflect XSD artefacts (e.g. every XML attribute is represented by a separate concept). This is also indicated in Figure 40 where instance representation in the ad-hoc ontology is longer than in the original target ontology. To bridge the differences between these two ontologies a mediator is used. However, it is also possible to directly use the ad-hoc ontology for the semantic description. In this case the ad-hoc and the target repository are identical which eliminates the need for mediation.

An alternative approach to perform data grounding is to use the XML serialisation of WSML and to convert between this XML version and the one required by the web service using standard technologies like XML style sheets (XST). This requires the definition of an appropriate style sheet and the invocation of the transformation using a suitable style sheet processor. The drawback of this approach, however, is that there is no semantically interpretable mapping, since the actual transformation is simple text conversation. The third recommended approach is the use of a direct mapping between WSMO/WSML elements and XML, although there are no implementations available yet. A special mapping language could facilitate lifting and lowering between the semantic and the syntactic description without any of the disadvantages of the other to approaches.

Behaviour grounding deals with the mapping between a service's choreography and the appropriate WSDL elements. A choreography's state signature defines concepts that are used as *in*, *out* or *shared* messages in the transition rules. When grounding the choreography to an existing web services one has to be aware that there can be significant differences in the granularity of the messages used in the semantic and the syntactic description. According to [126] WSDL messages are generally more course grained to reduce network roundtrips. Semantic descriptions, however, are typically fine grained and also split information over several concepts to facilitate re-use of individual concepts. This leads to structural heterogeneity that has to be considered during the design of the mapping. Generally the following cases can occur:

- Single rule per request/response operation: In this case there is a perfect match between one web service operation and one transition rule. Thus, the *in* concept used in the transition rule is mapped to the input message of the web service operation and the response message is represented by the concept that is created by the transition rule.
- Multiple rules for an aggregate operation: This reflects the situation where a web service's operation constitutes the interface of a sub-flow (i.e. several operations that are executed sequentially to create the same result) that is explicitly modelled in the choreography spanning multiple transitions. Consequently all *in* concepts of all the individual transitions that make up the web service's aggregate operation are grounded to the single input message of this operation. Respectively, all *out* concepts of the involved transitions are mapped to this operation's *out* message.
- Grounding one concept to multiple operations: This is caused by the different granularity of messages in the semantic and syntactic description rather than by the different granularity of operations and transitions and can occur in both situations mentioned above. *In* concepts can be used as the input to several operation. Thus every *in* concept can be grounded to multiple input messages. It is also possible that different operations have output messages of the same type, corresponding to the same concept in the ontology.

Beside this cases a general constraint is that *in* concepts can only be grounded to input messages, *out* concepts to output messages and *shared* concepts to input and output messages. This includes fault messages as well. It is important to realise that the actual URIs that are used to ground these concepts are not referring to XSD message type definitions but to the operations' input and output messages.

```
...
<interface name="BookTicketInterface">
    <operation name="queryPrice" pattern="http://www.w3.org/ns/wsdl/in-out">
        <input element="tns:TripSpecification"/>
        <output element="tns:PriceQuote"/>
        <outfault ref="tns:TripNotPossible"/>
    </operation>
    <operation name="bookTicket" pattern="http://www.w3.org/ns/wsdl/in-out">
        <input element="tns:BookingRequest"/>
        <output element="tns:Reservation"/>
        <outfault ref="tns:CreditCardNotValid"/>
        <outfault ref="tns:TripNotPossible"/>
    </operation>
    <fault name="TripNotPossible" element="tns:TripFailureDetail" />
    <fault name="CreditCardNotValid" element="tns:CreditCardInvalidityDetail" />
</interface>
...
```

Listing 42: Snippet from an example WSDL file[126]

URIs used as values of the withGrounding attribute of the state signature (see Listing 40) have the following form:

```
namespace#wsdl.interfaceMessageReference(interface/operation/message)
```

or

```
namespace#wsdl.interfaceFaultReference(interface/operation/message/fault)
```

Interface is the name of the interface in the WSDL file that contains the operation. The next value is the local name of the operation followed by the role of the message (in or out). In case of a fault message the local name of the fault message has to be provided. Having given the example shown in Listing 42, the following grounding URIs would be possible:

http://example.com/#wsdl.interfaceMessageReference(BookTicketInterface/bookTicket/In)

http://example.com/#wsdl.interfaceMessageReference(BookTicketInterface/bookTicket/Out)

http://example.com/#wsdl.interfaceFaultReference(BookTicketInterface/bookTicket/Out/CreditCardNotValid)

As a result every transition rule that makes use of any of these messages is mapped to WSDL operations as well. When a web service should be consumed based on a WSMO choreography, the client needs a choreography engine that is capable of executing the ASM represented by the semantic description. The goal of the choreography is to allow a client to use a web service that was previously unknown and therefore to provide the necessary information on how to use the different web service operations in order to achieve a specific goal. Assuming that the client has already evaluated the web service's capabilities holding the description of its IOPEs, some instances of *in* and or *shared* concepts that are required by some transition rules will be present in the information space. Thus, one transition rule will fire when the ASM is evaluated. The client now marks all *in* and *shared* concepts used by this rule as those that have to be sent next. Based on the grounding of these concepts the corresponding web service operation together with its input message has to be identified. Since, as discussed previously, each concept can be grounded to multiple input message, the client has to uniquely select the appropriate operation by sorting out those mappings that do not apply. The detailed algorithm on how to perform this can be found in [126]. To figure out the next operation that needs to be performed, these steps are repeated until the desired state is reached.

Besides grounding a semantic web service description to an already existing web service, WSMO also proposes a way to generate a WSDL file based on the semantic description. This allows for a forward engineering approach that starts with a semantic model where the actual web service endpoint description is automatically generated. One prerequisite for this approach is the existence of some default grounding that is used by the generation process. Data grounding in this scenario is implemented by wrapping each WSMO concept that is accessible by the client (which is true for all *in*, *out* and *shared* concepts) in an XML schema element declaration. This approach is shown in Listing 43 where “concept name” is replaced by each concept's local name.

```
<xs:element name="concept name">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="wsml:instance" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Listing 43: XML schema template for the default grounding of WSMO concepts in generated WSDL files

Thus the transformation process between semantic instances and XML messages at the syntactic level is straight forward. The default behavioural grounding of a WSMO choreography is based on the following considerations. Every *in* concept in the state signature can only be created by the client and is sent to the service endpoint. This is considered a *write* operation. Every *out* concept is created or updated by the service and is sent to the client. This leads to the semantics of a *read* operation. A *shared* concept, which can be created or modified by both communication peers, will be part of a *read* and a *write* operation. Consequently, the generated WSDL operations are either based on the in-only or out-only message exchange pattern. Thus, there are no *in-out* operations in the generated WSDL, although this is usually the most common message exchange pattern. Another shortcoming of this approach is that there is no distinction between ordinary responses and fault messages. Since a fault message, however, can be seen as a web service message with special semantics (i.e. indicating an erroneous condition), the loss of this meaning at the syntactic level can be compensated by adequate meaning at the semantic level. Nevertheless, more severe is the loss of synchronously produced fault messages since *in-only* operations do not allow for any responses at all. The separation into in-only and out-only operations is argued by the lack of information that is required to figure out which concepts belong together as input and output of a single operation[126]. Interestingly, these considerations do not take the transition rules into account, which would easily enable the automatic creation of web service operations that use the ordinary in-out message exchange pattern. This, however, would require a different data grounding approach, since in this case all input concepts used by a transition rule had to be aggregated into one input message for the corresponding operation. The same is true for all concepts that are created or updated by the transition rules and had to be aggregated to single output messages. Thus, an approach that translates every transition rule into one corresponding web service operation would probably improve the overall usability of this generation approach.

So far only approaches to map from WSMO to WSDL have been discussed. Nevertheless, there also exist ways to map from WSDL to WSMO. One possible approach is the use of SAWSDL (Semantic Annotations for WSDL and XML Schema)[127]. SAWSDL provides additional XML attributes that can be used to map WSDL or XML schema elements to a corresponding semantic description. This approach does not make any assumptions about the framework used to model the semantic descriptions thus it can be used together with WSMO or OWL or any other semantic framework. The only prerequisite is that semantic model elements can be unambiguously referenced via URIs. Since all SAWSDL attributes are multivalued it is even possible to map a WSDL web service to several different semantic models simultaneously. The core attribute is `sawsdl:modelReference` that directly maps the element it appears within to a semantic element. If additional translation between the WSDL or XML element and the corresponding semantic element is needed it is also possible to add additional processing instructions. Therefore the `sawsdl:liftingSchemaMapping` and the `sawsdl:loweringSchemaMapping` attributes can be used to refer to XML stylesheets that describe the transformation from XML to semantic models (i.e. lifting) and vice versa (i.e. lowering).

4.5 Comparison

When analysing semantic web service frameworks that could be candidates for the implementation of the envisaged approach to ontology-driven E-Government it is important to particularly focus on those frameworks that have already left the conceptual state and become available with implementations and probably tool support. This is true for OWL-S and WSMO but not for SWSF. That is why SWSF will not be explicitly regarded, although WSMO can be seen as an adoption of key SWSF concepts.

There already exists some research on the comparison of WSMO and OWL-S [128][129] but all these studies were published even before some key features of WSMO (e.g. service orchestration and grounding) had been specified. Thus, their results lack important facts when it comes to a complete evaluation of these approaches. When directly comparing semantic web service frameworks, especially when they should be assessed according to their suitability for a given usage scenario, it is important to point out how these frameworks support common semantic web service design goals. Like already pointed out in section 2, the semantic web should allow automated software agents to accomplish non-trivial tasks to achieve a user's goal [16]. Semantic web services are considered to be the core elements of the realisation of the semantic web, which is for example reflected by the OWL-S design goals [96]:

- Automatic Web service discovery
- Automatic Web service invocation
- Automatic Web service composition and interoperation

Besides these goals, the basic process to actually use semantic web services should consist of the following phases[130]:

- Goal discovery and refinement
- Service discovery
- Service contracting

Integrating these design goals with the proposed utilisation process a semantic web service has to support the identification of candidate web services based on a user's or agent's goal, has to describe how to actually make use of these services and should additionally provide information that allows to reason about combinations of different web service operations to get more complex goals accomplished. This section will therefore compare the different frameworks in question along these requirements and will point out their respective strengths and weaknesses.

4.5.1 Goal based discovery

Supporting goal based discovery of web services requires a notion of a goal that reflects something a user wants to get done or achieved. In the case of OWL-S, there does not exist an explicit notion of goals whereas WSMO provides a built-in component that reflects user goals. However, regardless of the existence of an explicit formal goal as part of a framework it seems intuitive that a web service description that unambiguously states what a service does and which output it produces can be used to figure out whether this service contributes to the given goal or not. To provide this type of description OWL-S as well as WSMO use so called IOPE's (inputs, outputs, preconditions and effects).

In general OWL-S proposes two approaches to implement goal based service discovery [12]. One family of algorithms utilises the classification aspect of OWL-S service profiles. OWL-S recommends that every profile should not only be a pure instance of OWL-S' *Profile* class but should also subclass other classes adding extra semantics to facilitate service discovery. This approach is paradigmatically demonstrated in the Congo Book Shop reference example. The profile of this service is a subclass of *Profile* but also a subclass of *BookSelling*. This class is in turn a specialised subclass of *E_Commerce* that has its *merchandise* property restricted to instances of type *Book* only. Thus, it can easily be subsumed that the Congo Book Shop is an e-commerce service selling books. By extending the class hierarchy and adding another super-class like *PayPalEnabledService* it would be possible to reason that one could use PayPal[14] in order to pay purchases.

However, it is also possible to model these facts in the IOPEs or capabilities of the service. Whereas inputs and outputs can be natively expressed in OWL, OWL-S relies on external rule languages to model preconditions and results/effects. Thus, the second family of goal based service discovery algorithms takes the inputs and outputs of available services into account. According to [12] these algorithms interpret a service description as a functional transition of input state into output state. To illustrate the advantages of

14 http://www.paypal.com

this approach, [12] mentions a language translation service that accepts *n* different languages as input and provides *n* languages as output. Expressing this service in the previously explained service class hierarchy approach would require up to n^2 service classes to describe all possible translations. This can be avoided by modelling the input and the output of the service as classes that represent the supported languages. There are several algorithms[131][132] that consider a service's input and output. Some are already integrated into runtime and development environments[133]. Basically, all these approaches use subsumption on input and output classes. This requires a goal to be expressed as a service profile that contains the required output and desired input. Since this type of a goal notation is very similar to an OWL-S service profile (see section 4.2.1), these algorithms compare the actual service's profile, the advertised profile and the goal description to the requested profile. This, however, is almost identical to WSMO's notion of a goal (see section 4.4.2). To identify appropriate services, this kind of discovery algorithms take a requested service profile as input and reason about the requested profile's outputs (inputs) and all advertised profiles' outputs (inputs). It is checked whether a requested output (input) is identical to an advertised service's output (input) or if one subsumes the other. This is done for all inputs and outputs of the requested service. Based on the result of this step a rank can be defined for every input/output element that leads to a total rank for every advertised service with respect to the requested profile. The resulting rank depends on the subsumption relationship between a requested and an advertised element and is almost identical to WSMO's goal matching states (see section 4.4.2)[132]:

- ***exact****:* The advertised element A is equivalent to the requested element R ($A \equiv B$)
- ***plugin****:* Requested element R is a sub-concept of the advertised elements A ($R \sqsubseteq A$)
- ***subsume***: Requested element R is a super-concept of the advertised elements A ($A \sqsubseteq R$)
- ***intersection***: The intersection of requested element R and advertised element A is satisfiable ($(A \sqcap R \sqsubseteq \bot)$)
- ***disjoint***: Requested element R and advertised element A are disjoint ($A \sqcap R \sqsubseteq \bot$)

Although this schema uses the same notation as WSMO, the outcome is different. When considering the language translation service again, the service discovery algorithm above only works correctly if the service translates any of the supported input languages into every of the supported output languages. If, however, the number of resulting output languages depends on the input language, the algorithm will lead to inappropriate results. Instead of treading the service profile as a transaction that describes how some input state is translated into some output state, the algorithm uses simple subsumption on input and output concepts. Thus, the fact that not every input language can be translated into every output language is not covered. This would require a model like it is used in WSMO, that describes the service interaction in terms of a so called abstract state machine that transforms a state described by the available input into a resulting state described by specific output. In the case of a WSMO based discovery approach the state machine would receive the requested input language and would result in a set of so called runs containing all possible output languages. If the desired output language is part of the output of at least one run, the service can be used to achieve the goal. This shows that there are certain limits in OWL-S based discovery algorithms compared to WSMO. Although OWL-S can model more sophisticated facts by embedding rule descriptions for preconditions and effects, evaluation of them requires heterogeneous tool environments and therefore more complex procedures. That is why OWL-S matchmaking is solely based on subsumption of service profiles. Whereas especially reasoning over service class hierarchies is relatively fast and can be applied to a large number of advertised services, WSMO's approach to model a service's capabilities is more expressive but also more demanding in terms on computational complexity when applied to a large number of advertised services. That is why WSMO proposes a two step matchmaking procedure, where candidate services are related to so called goal templates at design times, limiting the number of services that have to the analysed during runtime. This, however, limits the number of possible results to those already known at design time.

Generally, every approach that takes service capabilities into account requires a formal description of a goal that is similar to the actual service description and includes at least input and output concepts. Since the creation of these descriptions is not very intuitive for non-expert users, this approach requires some additional support. Again WSMO goal templates can be associated to natural language goal descriptions

and therefore can be used to support users in expressing their goals.

As pointed out above, both approaches have some advantages but also disadvantages. Whereas OWL-S' approaches are efficient but rather limited in expressiveness and might potentially lead to incorrect results, WSMO's approach is of high complexity and therefore virtually limited to pre-modelled candidate services. The following three-phase discovery algorithm that is based on the advantages of both frameworks seems to be an optimal solution:

1. Look-up all candidate service using subsumption within a comprehensive service ontology (e.g. looking for *LanguageTranslationService*s).
2. In the resulting set of candidate services apply the subsumption algorithm on inputs and outputs
3. Verify the correctness of the results from phase two by using WSMO's abstract state machine approach.

This would allow for very efficient service identification in phase one. Phase two is used to further reduce the number of candidate services that are verified in phase three. Since this approach requires WSMO's state machine approach it could be implemented by either extending WSMO's element hierarchy or by modelling a meta-service-model-ontology within WSMO that refers to the appropriate WSMO elements when needed.

4.5.2 Service Choreography

The choreography of a semantic web service in this context describes how a web service is used in terms of its interaction protocol, i.e. which messages have to be sent in which order to achieve a specific goal.

To describe how to interact with a web service, OWL-S provides a so called service model (see section 4.2.2). The basic building block of the service model is the process element. Basically there are atomic processes and composite processes. Whereas an atomic process represents one web service operation, a composite process represents a flow of composite and/or atomic processes. The flow of operations is described via so called control constructs. The entire flow definition provides the client with information about how to invoke particular web service operations in terms of their sequence. Control structures used by OWL-S basically represent standard control structures as they are used in virtually any programming language. This results in an intuitive workflow description. Along with the flow of control also the flow of data has to be defined. This describes how information is passed along the entire flow of operations. Since OWL is not a frame-based system and therefore the existence of a class does not allow for expecting the existence of any particular property values, the data flow has to be described at rather fine granularity referring to individual properties. Consequently data propagation has to be defined at low level as well using so called binding classes. Generally this leads to extremely verbose description.

Whereas OWL-S uses a conventional, intuitive flow description, WSMO's choreography element (see section 4.4.1) represents an abstract state machine (ASM). The state signature defines variables , which are used as the state machines locations. The current values of these locations define the state of the machine. To figure out which operations to call, the abstract state machine has to be initialised by assigning the current input values to the appropriate in and/or shared variables of the ASM. Like already described in section 4.4.3 variables in guard conditions of rules that are enabled to fire identify the operations that have to be invoked. Due to potential differences in the granularity of transition rules and web service operations there might be need for additional steps to figure out when and which operation to call. By setting up design guidelines that require ontology experts to model transition rules as web service operation equivalents, this potential additional complexity could be avoided. Given such constraints, a transition rule would be logically identical to an atomic process in OWL-S.

Both approaches to model possible interaction patterns are equivalent according to their expressiveness. Whereas OWL-S describes the flow of control, WSMO uses an ASM with transition and update rules written in WSML. Thus, the WSMO ASM can be directly executed. Due to its frame-based nature, WSMO can map concepts to entire messages, whereas OWL-S has to map individual massage parts. This eliminates the need for separately modelling the data flow in WSMO as it is required in OWL-S. As a matter of fact, however, WSMO's choreography suffers from different specification versions that are neither already

adopted by the modelling language used (WSML) nor by tools supporting WSMO services. Thus the latest revision of the specification cannot be used in practice yet.

4.5.3 Service Execution

In this section the different approaches to ground the semantic descriptions are compared.

Since every atomic process in OWL-S represents exactly one web service operation grounding of OWL-S services is rather straight forward (compare Figure 24). The grounding element for an atomic process refers to the WSDL file of the web service. Via additional properties the process is connected to the appropriate port type and operation. The input and output parameters are mapped to message parts, which are elements of the operation's input and output messages.

WSMO's grounding is entirely based on message exchange patterns [134], i.e. a web service is rather seen as an endpoint that receives and returns messages than as a set of operations available over the web. Thus, there is no explicit equivalent for operations in WSMO, although transition rules represent state changes that can be achieved via the invocation of web service operations. The actual mapping between WSMO's semantic description and the operations of a web service endpoint happens via the messages used in a transition rules guard condition. Concepts are not mapped to message types but directly to the input and output messages of particular operations. Thus, the grounding of a message also refers to the operation it is used with. Potential heterogeneities in the granularity between transition rules and operations might introduce additional complexity as already pointed out in section 4.4.3.

Again the grounding mechanism of OWL-S must be considered to be more intuitive, although both approaches unambiguously map semantic descriptions to the syntactic descriptions of WSDL documents. As already mentioned in the previous section, the possibility of a more coarse grained mapping between WSMO concepts and WSDL messages instead of individual message parts minimises markup effort and generally eases mapping. Both frameworks support the use of XSL transformations to perform more sophisticated mappings between concepts and messages. To support bidirectional mapping also from WSDL to the semantic model OWL-S provides a separate namespace with proprietary markup. WSMO promotes the use SAWSDL, which - due to its framework independent nature - can also be used with OWL-S.

4.5.4 Summary

Taking all core features of semantic web services together, WSMO shows some advantages, however, at the price of higher complexity. There are some suggestions to minimise this complexity, e.g. by using predefined goal templates. OWL-S potential shortcomings are caused by its description logics basis that is not suited to natively express rules. Some of this shortcomings might be overcome be adopting OWL 2. WSMO shows some inconsistencies between the specification of the choreography and orchestration in its adoption in the actual modelling language. The available implementation, however, incorporates all core features, although some new suggestions (e.g. the use of control state ASMs) are still missing.

5 Model Driven Architecture

This chapter presents an approach called Model Driven Architecture [5] that was initially proposed by the Object Management Group[15] (OMG) in 2001. Since MDA provides a conceptual framework for Model Driven Development (MDD)[135], the goal is to identify principles an ideas that can be incorporated into the envisaged approach to Ontology-Driven E-Government.

15 http://www.omg.org

Definition 5: "The MDA defines an approach to IT system specification that separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform. To this end, the MDA defines an architecture for models that provides a set of guidelines for structuring specifications expressed as models." [5]

5.1 Idea/Motivation

The general motivation behind MDA is to emphasise the importance of creating a comprehensive model of a system before it is actually programmed. Therefore MDA wants to add additional value to models, by making them machine readable, re-usable at different levels and making them the basis for automatic code generation:

> *"This is the promise of Model Driven Architecture: to allow definition of machine-readable application and data models which allow long-term flexibility of:*
>
> *implementation: new implementation infrastructure (the "hot new technology" effect) can be integrated or targeted by existing designs*
>
> *integration: since not only the implementation but the design exists at time of integration, we can automate the production of data integration bridges and the connection to new integration infrastructures*
>
> *maintenance: the availability of the design in a machine-readable form gives developers direct access to the specification of the system, making maintenance much simpler*
>
> *testing and simulation: since the developed models can be used to generate code, they can equally be validated against requirements, tested against various infrastructures and can used to directly simulate the behavior of the system being designed." ([136] page 1-2)*

Thus, MDA promotes the use of models not only to improve the design of a system but also to gain significant advantages in later phases of the development cycle.

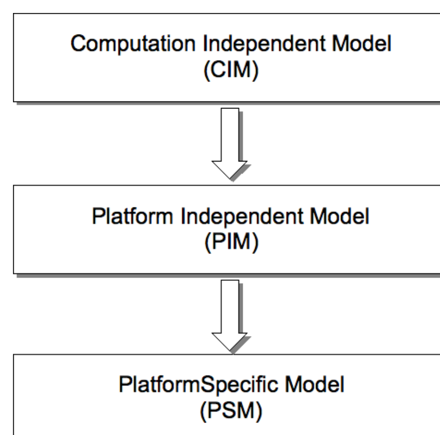


Figure 28: MDA's different model levels (own illustration)

MDA is not an isolated technology but is based on a set of other OMG standards like the Uniform Modeling Language (UML)[137], the Meta Object Facility (MOF)[6], the XML Metadata interchange (XMI)[138] and the Common Warehouse Metamodel (CWM)[139]. The complete model of a system actually consists of up to

three models that represent different viewpoints and levels of abstractions (see Figure 28). The transformation between these different models, especially between the Platform Independent Model (PIM) and the Platform Specific Model (PSM) is a major focus of MDA.

Generally MDA defines a model as follows:

> **Definition 6:** “In the MDA, a model is a representation of a part of the function, structure and/or behavior of a system” ([5] page 3)

In the context of MDA it is important that a model is formal. This is true if the model has a well-defined form (syntax) and every element that makes up the model has some associated meaning (semantic). This might also include the existence of rules that allow for analysis and checks of the model. The form of the representation of the model (e.g. whether it is text or a graphic) is not important as long as it is formal. One way to establish formal models and therefore to setup syntactic rules and semantics, is to formally define possible model elements and their relations by constructing a so called meta-model. This can be done by using the MOF (see section 5.2). Most models used in MDA are expressed in UML, which in turn is based on the MOF.

Each of three different models shown in Figure 28 has a different level of abstraction. Abstraction according to ISO 10746-1 is “The process of suppressing irrelevant detail to establish a simplified model, or the result of that process”[140]. Consequently the model at the highest level of abstraction contains least “irrelevant” detail. The following sub-sections will briefly describe the different models used within MDA.

5.1.1 Computational Independent Model

Before describing the details of all the different models involved in MDA, taking a look a the model taxonomy[141] presented in Figure 29 will help to understand the differences between those models and

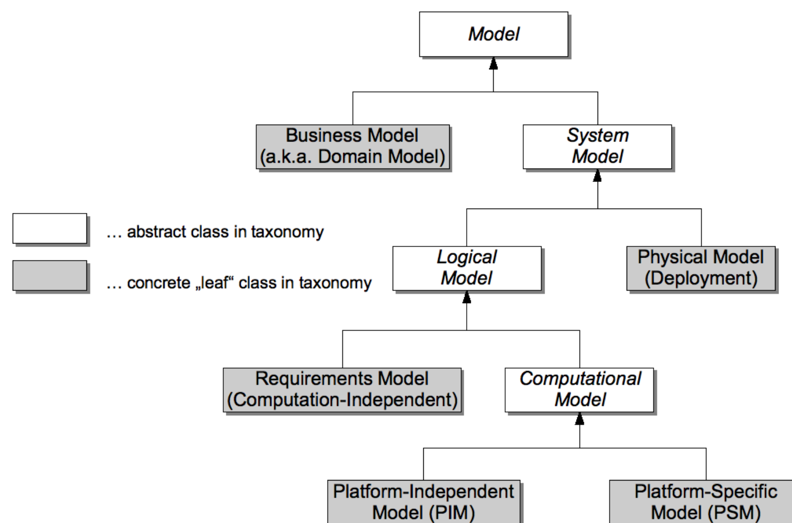


Figure 29: MDA model taxonomy ([141] page 193)

how they contribute to a comprehensive picture of a system.
This taxonomy classifies models along several categories. The first separation happens along the dimension business or domain versus system models. The first category of models clearly focuses on the business aspects omitting all facets that are related to computer systems. This does not mean that the model is incomplete but it simply does not worry about which elements of the business are subject to automation and

which are not. Consequently, system models, which focus on automation aspects typically have a smaller scope than business/domain models. The next categorisation happens along logical models and physical models. Physical models represent the run-time infrastructure of the system and how the different artefacts of the system make use of the available resources. The logical model focuses on the functional aspects and quality aspects of the system.
Another separation follows along the dimension computational and computation independence. This leads to a requirement model that should not be influenced by any constraints imposed by the nature of the technical solution. Thus, technical factors should be left out when establishing the requirements model. On the other side computational models already contain specifics of the eventual technical solution. Computational models are then split into two categories called platform independent and platform specific models. The first category represents models that describe a computer-based solution but do not pay account to artefacts and requirements imposed be the selection of a particular platform (see next sections). Platform specific models are already optimised for the use with a particular platform.
To capture the content that typically goes into the business/domain and requirements models, MDA proposes a so called Computation Independent Model (CIM). Thus, this model contains a business view that is already influenced by the intention to automate certain aspects of the business or domain. Nevertheless, it is not strictly limited to these parts of the system. Its general intention within the MDA process is to bridge the gap between business/domain experts and system design experts to capture the system's requirements in formal models. It does not contain any information about the structure or the parts of the system. Typically, a CIM is made up of a set of UML diagrams like activity, class, interaction, collaboration and use-case diagrams, although it is not explicitly needed by MDA.
Beside a few recommendations MDA does not make any assumptions about the computational independent model since it focuses much more on the transformation process between the PIM and PSM. But there exists significant literature on methods, tools and strategies how to create a CIM [142][143][144][145].

5.1.2 Platform Independent Model

The Platform Independent Model (PIM) is a computational model that further refines the requirements model (CIM) but does not contain any specifics of the eventual run-time environment. To clarify what platform independence in the context of MDA means, a definition of the term platform would be helpful. The MDA literature, however, provides several explanations of this term:

> *“In the MDA, the term platform is used to refer to technological and engineering details that are irrelevant to the fundamental functionality of a software component.”([5], page 5)*

> *“A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented.”([136],page 2-3)*

> *“A platform is an execution environment for models”([146],page 51)*

Taking all these definitions together, a platform constitutes the run-time environment for a system and provides functionality and services for the application, which in turn, however, are not relevant to the fundamental functionality of the software system. Platforms typically form platform stacks [146] since one platform at a higher level makes use of a lower level platform. The platform e.g. consisting of Java API's makes use of the Java virtual machine, which in turn makes use of services provided by the operating system.
A platform independent model ignores the influence of the selected platform on the logic and functionality of the system. It therefore abstracts away this kind of technical detail but captures the entire functionality of the software.

5.1.3 Platform Specific Model

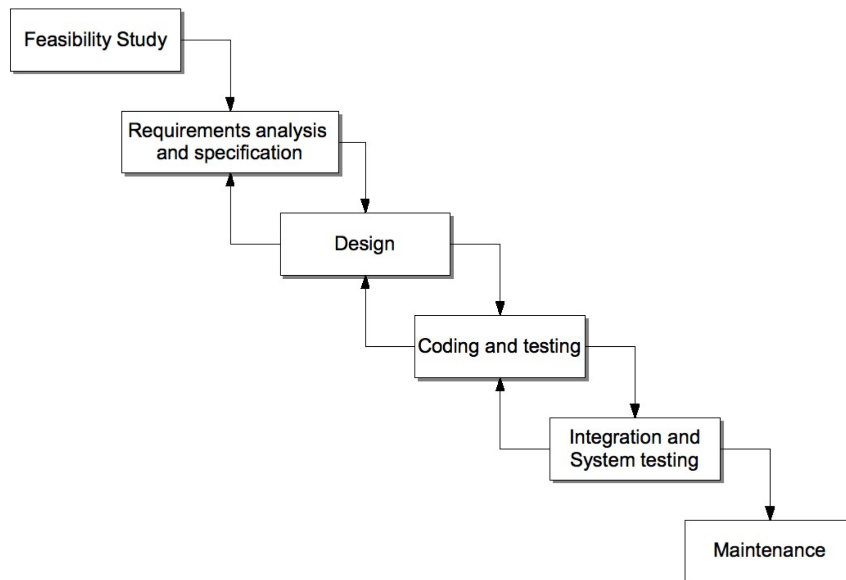


Figure 30: The Waterfall Model in Software Engineering (own illustration based on [147])

The platform specific model (PSM) applies the specification of a PIM to the specific requirements of a particular framework. Therefore it extends the PIM with concepts representing the functionality, services and artefacts provided or needed by the platform. Platform-specific facts like information-formatting, programming language, distributed component or messaging middleware influence the PSM. The PSM therefore is the least abstract model, which holds in turn most technical detail and can thus be used to generate an executable application based on the information contained in the model.

5.1.4 Model Transformation

All classical software development methods that are based on the waterfall model [147] contain a software design phase (see Figure 30). The goal of this phase is to determine the layout of the major system components and to plan the implementation of the system. The result of the design phase is a design document the serves as the input to the implementation phase. If errors are detected in a development phase that have their roots in a previous phase, the waterfall model provides a backtracking mechanism to fix the error in the appropriate phase. Since it is not always clear whether a problem that occurs for example in the testing phase is caused by a design flaw or an inappropriate implementation decision, updates might be done to the code base that are not reflected by the design documentation although these changes influence the design as described there. That is why the design documentation is likely to lose accuracy and actuality over time. This trend is even intensified by the fact that in classical software engineering the design document holds a description of the system but otherwise has no direct influence on the code base. Thus updating the design model is a documentation task rather than an implementation task. The MDA approach changes this significantly since here a model is no longer a pure design artefact but a representation of the system that can be used to automatically generate source code. That is why MDA has a clear emphasise on model transformation means.

MDA proposes three different main levels of abstraction. The computational independent model (CIM), the platform independent model (PIM) and the platform specific model (PSM). Although model transformation is a key factor in MDA, the specification does not deal with how to transform a CIM into a PIM, but generally sees the CIM as an input to the (manual) creation of a PIM.
To overcome this lack of the specification there exist several recommendations aiming at the automatic transformation of computational independent to platform independent models [148][149].

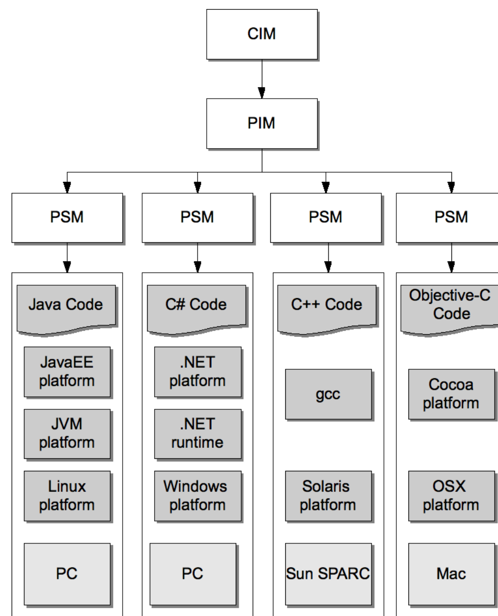


Figure 31: MDA Transformation Paths (own illustration)

The transformation of PIM into PSM, however, is thoroughly covered by the MDA specification. Figure 31 Illustrates the advantage of a comprehensive PIM since it can be transformed to several PSMs representing different technological platforms. This enables reuse at a high level of abstraction and allows for rapid adoption of new technologies.

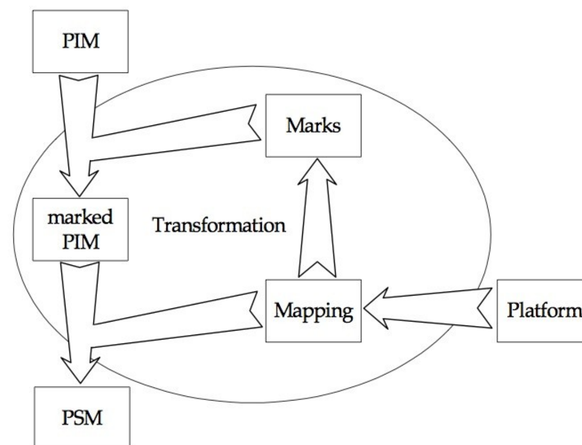


Figure 32: MDA Transformation Process based on Marking ([136], p. 3-8)

While abstraction means suppressing irrelevant detail, the transformation of a platform independent into a platform specific model means to leave a level of higher abstraction and to add more (technical) detail. Thus every transformation relies on additional information that somehow has to be added to the model in order to create a complete model at a more specific layer. MDA basically provides two mechanisms to map models at different levels of abstraction. One approach is called model instance mapping or marking and the other approach is called model type mapping or metamodel mapping. Marking augments the platform independent model with concepts of the PSM. These so called marks add platform specific transformation information to the PIM turning it in a so called “marked PIM” as shown in Figure 32. The marked PIM is then turned into a

PSM.

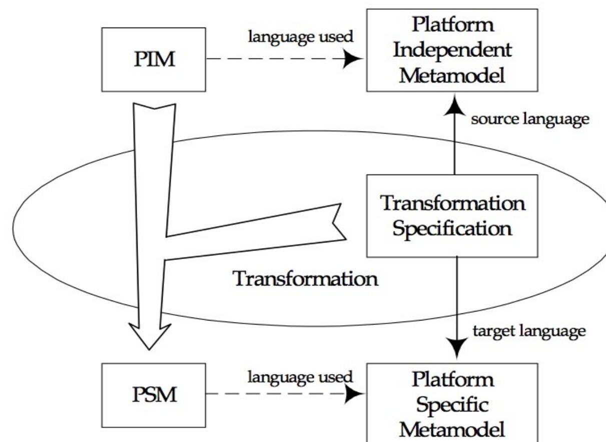


Figure 33: Mapping of metamodels allows for direct model-to-model transformation ([136],p. 3-9)

While marking requires and intermediate step to enrich the PIM with platform specific information and therefore dilutes the strict separation of these two layers, metamodel mapping uses a more general approach. One prerequisite for this approach is that the models used for the PIM and the PSM are based on metamodels. These metamodels define the syntax of the models and therefore contain definitions of all elements that can be used in the actual model. Thus, a mapping between corresponding model elements can be created at metamodel-level like shown in Figure 33. This mapping is used by the transformation process to generate a model at the next concrete level.

Besides these two approaches to automatically create a PSM based on PIM MDA also allows for direct transformation into source code, which makes the use of a PSM obsolete.

An important characteristic of all model transformations in MDA is the fact that transformations have to be traceable. This allows to identify the corresponding source element in models of higher abstraction for every model element that is a result of a transformation. When applied consequently over all levels of abstraction this allows to trace every piece of code back to its related requirements.

In an MDA process a platform independent model is eventually transformed into executable source code. Since MDA models are typically expressed in UML one can think of it as compiling UML into code or even further of UML that is executable[150]. Having comprehensive models at higher level of abstraction allows for re-use at model level and rapid adoption of new technologies and frameworks since platform specific models can be created automatically once an appropriate metamodel mapping was established. MDA considers models as valuable resources and their value rises according to their degree of abstraction.

5.2 Meta Object Facility (MOF)

The Meta Object Facility (MOF)[6][151] is an OMG standard for the definition of models and metamodels. It therefore is one of the key elements of the MDA stack. MOF recommends four layers – so called metalevels - to create new models as illustrated in Figure 34.

The lowest level called M0 contains the elements that are actually modelled. Thus they are instances of the elements in the model, whereas the model itself is located at level M1. In the case of an UML class diagram at level M1 the instances of these classes (concrete objects) were part of M0. More specifically the elements at level M0 are called run-time instances to point out the differences between the actual data and their equivalent model elements.

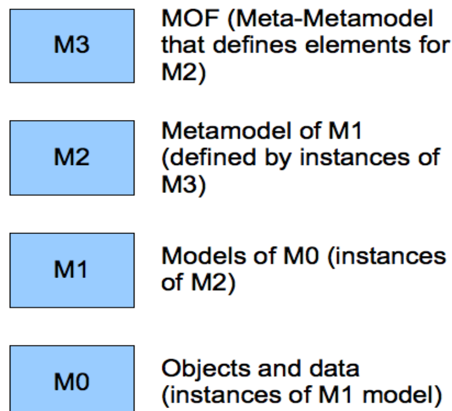


Figure 34: The MOF metalevels (own illustration)

The need for this more specific distinction becomes obvious when thinking about an UML object diagram. By definition, elements of this type of diagram are already objects (instances of *class*) thus these elements cannot be further instantiated at level M0. Corresponding elements at level M0 are therefore called run-time instances, whereas their model equivalents are also called snapshots. The UML metamodel defines an element called *InstanceSpecification,* which is the meta-element of snapshots.

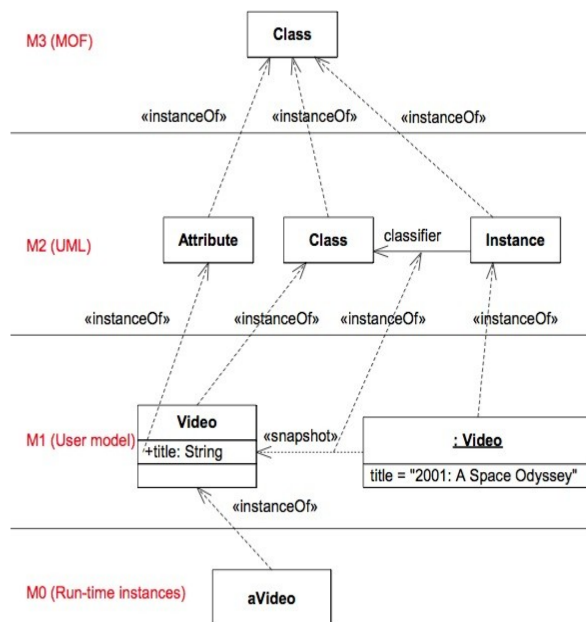


Figure 35: Example of four-layer metamodel hierarchy ([152],page 19)

Elements that can be used at level M1 to create an actual model are instances of elements defined at level M2. This level therefore defines how models can be built and of which elements they might consist of. Hence, this makes models at level M2 classical metamodels since they describe models. The way in which metamodels can be created is in turn defined by models at level M3. Thus, M3 models must be considered metamodels of metamodels, which makes them meta-metamodels. This level is actually the MOF, which makes the MOF a language to describe metamodels. There does not exist any additional level of abstraction beyond level M3 that is used to describe the MOF. In fact MOF is itself described by MOF. Consequently MOF is a self-described or reflective language.

Metamodels are typically more compact than the models they describe. Thus the number of elements found in a metamodel is often much smaller than in the actual models that are made up of instances of the metamodel. To illustrate how all the different levels of abstraction and (meta-)models are linked together Figure 35 shows a simple example of an UML class diagram.

The MOF uses the UML class modelling notation to describe meta models that are MOF compliant. Thus the MOF provides pretty much the same elements that are used in UML class models. As can be seen in Figure 35 the MOF contains an element called *class*. The snippet from the UML metamodel at level M2, however, also contains a *class* element, which is an instance the MOF *class*. Both of these class elements are very similar and share most of their structural and behavioural characteristics.

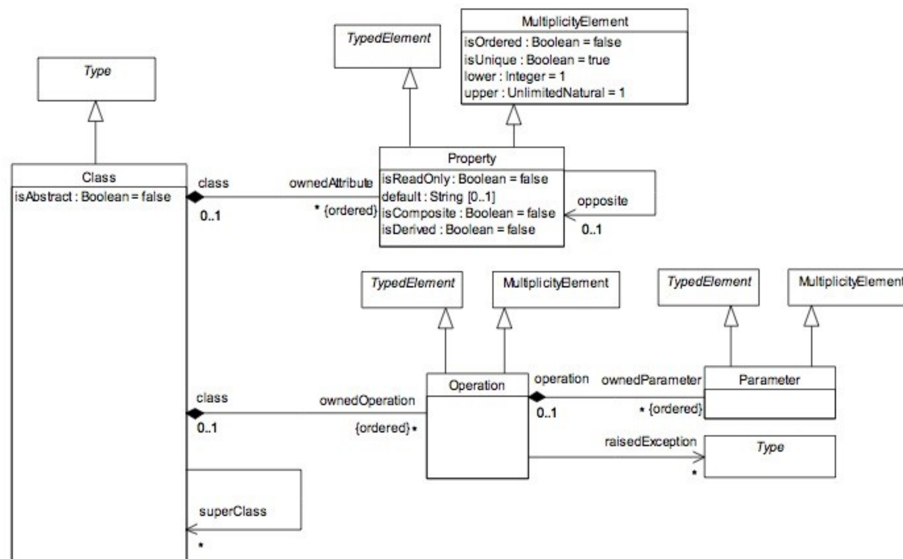


Figure 36: UML Infrastructure Library: Class definition ([152], page 93)

In fact major parts of UML 2 and the MOF share the same meta model, which is the common UML 2 Infrastructure library[152]. Figure 36 shows the definition of a class element as it defined in the infrastructure library. The elements used there are almost identically used in the MOF as well as in UML, thus this diagram can be used to model both classes. MOF uses the model shown in Figure 36 to merge it together with some additional capabilities (e.g. to introduce reflection, which allows to navigate from every element to its describing meta-element) into the MOF meta-model. UML re-uses the same model to describe its UML class element but extends it with additional features. It is worth to mention that the class diagram presented in Figure 36 that defines the capabilities of a class is in turn made up of classes, i.e. every rectangle used in the diagram is an instance of MOF class, which in turn is defined by this model. This is exactly the self-descriptive nature of the MOF. Thus the MOF is made up of MOF instances.

To facilitate the adoption of the MOF it was split into two packages. The Essential MOF (EMOF) represents a kernel for metamodelling, whereas the more sophisticated Complete MOF (CMOF) provides the full expressive power of the MOF. Tool developers can decide to only support EMOF, which simplifies their products. EMOF is a subset of MOF that contains all capabilities that are typically found in object oriented programming languages and XML.

Due to its close relation to UML, the default representation format of MOF and the (meta-)models based on it is graphical. But there are ways to create MOF based models that are not diagrams but text. This approach is called Human-Usable Textual Notation (HUTN)[153]. An example result of using this text based way of metamodelling is WSML (see section 3.4).

One important part of the MOF specification is QVT (Query/View/Transformation), which specifies languages for model-to-model transformations[154].

5.3 Object Constraint Language (OCL)

One central point of the MDA is to generate applications based on models and therefore to minimise the actual coding effort. This requires models to capture all the aspects of running applications. UML, which is the preferred modelling technique in MDA, can represent a broad variety of system aspects with its numerous diagrams (UML 2.0 defines thirteen standard diagram types), but falls short when it comes to represent logical constraints of functionality. This is why the OMG introduced the Object Constraint Language (OCL)[155] that should have all the strengths of a formal language but is also simple enough to be easily used by business and system modellers who do not necessarily have a strong mathematical background.

OCL was not defined to substitute a programming language but is a pure specification language. The evaluation of an OCL expression does not change the state of a model, although an expression can be used to specify the state of a model. To demonstrate the expressiveness of OCL, some of its typical use-cases are briefly presented in the following subsections. The different types of expressions will be compared to the semantic technologies discussed in section 3 in order to identify similarities and differences between UML/OCL based models and pure semantic models.

5.3.1 Invariants

Invariants are rules that can be used to impose restrictions on the attributes of a class that have to hold for all instances of these classes at any time. Thus, invariants, like all other OCL expressions, apply to a particular class, which is called the context of the expression. Assuming that there exists a class *Student* with a property *numberOfStudies* of type integer. The OCL invariant show in Listing 44 is used to define that the *numberOfStudies* has to be at least one for every single student:

```
context Student inv validStudent:
   self.numberOfStudies > 0
```

Listing 44: A simple OCL invariant declaration

The expression first has to define its context, which in this case is the class it applies to. This is done by using the keyword *context* followed by the name of the class. The keyword *inv* specifies the type of this OCL expression, which is invariant. In this case the OCL expression has a name, which is *validStudent*. Naming of expressions in OCL is optional. The keyword *self* refers to the current context, thus *numberOfStudies* is an attribute of the class Student. When comparing the semantics of this type of constraints it can easily be shown that there exist equivalent constructs in OWL and WSML.

```
DataProperty: numberOfStudies
   Range:
      nonNegativeInteger

Class: Student
   EquivalentTo:
      numberOfStudies some nonNegativeInteger[>= 1]
```

Listing 45: OWL ontology represented in the Manchester syntax that is equivalent to the OCL expression in Listing 44.

Listing 45 presents an OWL snippet that has effectively the same semantics as the OCL invariant expression. It uses a data type restriction, which is one of the new features in OWL 2 (see section 3.3.2), on the *numberOfStudies* property to define the members of the class *Student*. WSML, however, uses a constraint axiom that is basically very similar to the OCL invariant notation (see Listing 46). If the body of the axiom evaluates to true the ontology becomes inconsistent.

```
concept Student
    numberOfStudies ofType (1 1) _integer

axiom validStudent
 definedBy
  !- ?x[numberOfStudies hasValue ?y] memberOf Student and ?y < 1 .
```

Listing 46: WSML constraint axiom with equivalent semantics to the OCL expression in Listing 44.

Invariants can also be used to define cardinality restrictions, which are multiplicity restrictions in this context. Assuming that instead of an attribute called *numberOfStudies* there would exist a one-to-many association between the class *Student* and a class *Study* that is represented by an attribute called *studies* within the *Student* class. The number of studies assigned to any student can be restricted to at least one by using the OCL invariant given in Listing 47. OCL treats all attributes with multiplicity higher than one as a Set, which is a pre-defined datatype. A set's *size()* operation returns the number of elements in the set. The same constraints can be established in OWL as well as in WSML by the use of cardinality constraints.

```
context Student inv:
    self.studies->size() >= 1
```

Listing 47: OCL invariant to restrict the number of elements that are part of an association

Invariants significantly extend the semantic capabilities of UML models and bring them one step closer to the expressiveness of semantic models.

5.3.2 Pre- and Postconditions

OCL can be used to define the pre- and postconditions of a class' operations and methods. More precisely pre- and postconditions can be modelled for all instances of an operation's meta-class which is *BehavioralFeature* (see [152], section 9.1.1).

```
context Typename::operationName(param1 : Type1, ... ): ReturnType
    pre : param1 > ...
    post: result = …
```

Listing 48: OCL syntax for defining an operations pre- and postconditions ([155], page 8)

The OCL expression again starts with the *context* keyword to define the operation that should be further specified. The identifier of the operation has to start with the name of the class that owns the operation. Besides the name of the operation, also the parameter list and the return type have to be part of the context. In this case the keyword *self*, if used in any of the conditions, refers to the owning class. Pre- and postconditions start with the keywords *pre* and *post* respectively. This OCL statement can be used to specify conditions that have to be true whenever an operation is invoked. The postcondition defines the state that is reached whenever the preconditions were met. The special keyword *result* can be used to define the return value of the operation.

When looking for equivalent constructs in semantic frameworks like OWL or WSML some of the differences between UML/OCL and these frameworks become obvious. Semantic frameworks have their roots in the knowledge engineering domain and therefore are used to model knowledge bases. Knowledge bases represent ground facts and rules that can be used to derive even more facts. There is no need for something like a behavioural feature. This explains why representing operations in semantic framework is not very intuitive. Nevertheless, as pointed out in sections 4.2 and 4.4 there exist ways to express the pre- and postconditions of (web service) operations, although they can hardly be compared with the compact notation of OCL.

5.3.3 Initial and Derived Values

OCL can be used to define the initial values of newly created instances. The *init* keyword can be used to assign an initial value to an object's attribute, whereas the *derive* keyword is used to specify a conditional assignment of a value (see Listing 49).

```
context Person::income : Integer
  init:   parents.income->sum() * 1% -- pocket allowance
  derive: if underAge
    then parents.income->sum() * 1% -- pocket allowance
    else job.salary    -- income from regular job
    endif
```

Listing 49: Example of an OCL init and a derive clause ([155],page 10)

There does not exist a similar construct in the world of semantic frameworks since they are based on a completely different paradigm. Semantic frameworks allow to classify existing data according to class axioms. In object oriented programming, which is the conceptual basis of UML and OCL, every object is an instance of at least one class. Thus, there are no instances without predetermined types/classes. Subsumption is also predetermined by a static hierarchy of classes. In ontologies, however, new classes can be introduced at any time and their members can be defined by axioms. In fact, the creation process of instances is beyond the scope of semantic frameworks. Consequently there is no support for restricting the creation of instances.

More specifically in semantic frameworks there exists no functionally equivalent way to express a constraint over a set of instances using an aggregate function like done in the example shown in Listing 49. Here the sum of the *parents' income* is calculated, where *parents* is a self reference of the *Person* class with multiplicity two and *income* is a property of the *Person* class. In this context sum is an operation defined for the OCL set datatype. Logical frameworks that are the basis of the discussed semantic languages do not provide means for mathematical functions over sets of properties.

5.3.4 Operation Body Expressions

```
context Person::getCurrentSpouse() : Person
  pre:  self.isMarried = true
  body: self.mariages->select( m | m.ended = false ).spouse
```

Listing 50: Example of a body expression that defines the functionality of an operation ([155], page 9)

Another feature that has no direct equivalent in semantic frameworks is the capability to define the body of a query operation as shown in Listing 50.

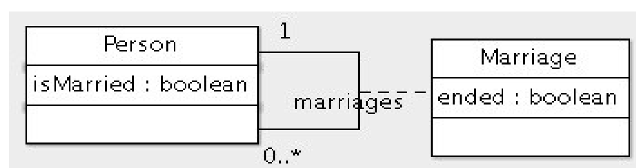


Figure 37: UML model of the sample scenario that is the basis for Listing 50 (own illustration created with ArgoUML)

The class *Person* possesses a multivalued relationship to other persons called marriages (see Figure 37). In OCL multivalued properties are represented by the *Set* datatype, which has some operations defined (e.g. *select*). In this example an instance of *Person* is selected that is related to the actual instance via the *marriages* relation where the *ended* property of this relation is set to false, indicating an active marriage.

```
ObjectProperty: hasMarriage
   Domain:
      Person
   Range:
      Person
   InverseOf:
      hasSpouse

ObjectProperty: hasSpouse
   InverseOf:
      hasMarriage

DataProperty: hasName
   Domain:
      Person
   Range:
      string

DataProperty: isEnded
   Domain:
      Marriage
   Range:
      boolean

Class: Person

Class: Marriage
   EquivalentTo:
      hasSpouse exactly 2 Person

Individual: John
   Facts:
      hasName  "John"
   DifferentFrom:
      Bill,
      Mary

Individual: BillAndMary
   Facts:
      isEnded  "true"^^xsd:boolean

Individual: Mary
   Facts:
      hasMarriage  BillAndMary,
      hasName  "Mary"
   DifferentFrom:
      Bill,
      John

Individual: Bill
   Facts:
      hasMarriage  BillAndMary,
      hasName  "Bill"
   DifferentFrom:
      John,
      Mary

Individual: JohnAndMary
   Facts:
      hasSpouse  John,
      hasSpouse  Mary,
      isEnded  "false"^^xsd:boolean
```

Listing 51: OWL2 example of the marriages scenario in Manchester syntax

As already mentioned above there is no equivalent feature in any of the investigated semantic frameworks since they do not posses the concept of methods or procedures anyway. Nevertheless there exist expressive ways to query the knowledge represented by a semantic model, although query systems are not an integrated part of these semantic frameworks. To point out the expressive and functional similarities and differences between this particular OCL construct and OWL as well as WSML, this scenario will be modelled using any of the two semantic languages.

An OWL 2 model of the sample scenario is given in Listing 51. Like in the UML diagram shown in Figure 37 there are two classes. One representing a *Person* and one representing a *Marriage*. These two classes are linked via the object properties *hasMarriage* pointing from *Person* to *Marriage* and *hasSpouse* defined as the inverse property. The class *Marriage* is further restricted to allow for exactly two *hasSouse* property values only. Beside the definition of classes and properties Listing 51 also contains some sample individuals to demonstrate a few use cases. According to the facts modelled in this example Mary was married to Bill but is currently married to John. Two find out which Person, if any, Mary is married to, the following Manchester syntax query can be used:

```
hasMariage some (Marriage and isEnded value "false"^^boolean and hasSpouse some (hasName value "Mary"))
and not (hasName value "Mary")
```

Semantically this query is the equivalent to the OCL body expression used in Listing 50, although the literal “Mary” needs to be replaced by some appropriate variable markup.

The WSML model that captures the same information is presented in Listing 52. In contrast to the UML model (see Figure 37) and the OWL 2 ontology (see Listing 51) this ontology uses only one class, the *Person* concept. A marriage is modelled by the relation isMarried, which in this case is a three-tuple that relates together two instances of the concept *Person* and a boolean value that indicates whether the marriage is ended or not. The ontology furthermore contains three instances of Person (John, Bill and Mary) as well a two instances of the isMarried relation, which - like in the OWL example - state that Mary was married to Bill and is currently married to John.

```
concept Person
    hasName ofType (1 *) _string

relation isMarried(ofType Person, ofType Person, impliesType _boolean)

instance John memberOf Person
    hasName hasValue "John"

instance Bill memberOf Person
    hasName hasValue "Bill"

instance Mary memberOf Person
    hasName hasValue "Mary"

relationInstance isMarried(Mary,John,false)
relationInstance isMarried(Bill,Mary,true)
```

Listing 52: WSML model of the marriage sample scenario

Two find out who is the current husband of Mary, the following query is needed:

```
isMarried(Mary,?x,true) or isMarried(?x,Mary,true)
```

Since the order in which spouses have to appear in the relation is not defined both options have to be checked here. Again, when replacing the instance literal Mary by a variable this query can be considered an equivalent alternative to the OCL body expression used in Listing 50. Thus, OWL as well as WSML can capture the same facts as the example in Listing 50 and provide equivalent query support as UML/OCL, although OCL and semantic frameworks are based on different paradigms. This equivalence, however, cannot be generalised, since OCL query expressions might also contain constructs that are simply not supported neither by OWL nor by WSML (e.g. using the sum of a multivalued number property within a query expression).

Generally OCL can be used to define constraints on the instances of a class, which restrict the set of possible instances. Semantic frameworks, however, use constraints that define whether a given individual is a member of a particular class or not.

5.4 Ontology Definition Metamodel (ODM)

In the previous section key features provided by the object constraint language where presented and compared to features provided by OWL and WSML in order to point out functional similarities and differences between these frameworks. This section presents an additional approach to bridge the world of semantic frameworks and model driven architecture, which is called ontology definition metamodel (ODM) [156]. The overall goal of this specification is to provide

> *"... the foundation for an extremely important set of enabling capabilities for Model Driven Architecture (MDA) based software engineering, namely the formal grounding for representation, management, interoperability, and application of business semantics."([156], page 1)*

ODM adds models to the MDA that allow for the presentation of ontologies, which are based on description or first order logics, have formal model theoretic semantics and can be used by automatic reasoners. Different profiles and mappings allow for the exchange of heterogeneous models as well as validation and consistency checks like they are already commonly used in semantic frameworks. Due to the different underlying paradigms of semantic models and UML, ODM does not extend UML to capture ontologies but provides its own MOF based metamodel. In fact the specification requires some modifications to the MOF in order to model all required aspects. According to the authors of the ODM specification the required changes will be addressed by one of the next releases of the MOF (see [156], pages 6-7 for a detailed description of the necessary modifications).

ODM is made up of five different metamodels, where some of them consist of different sub-package. The five metamodels are description logics, common logic[157], RDF, OWL and topic maps[158]. Beside these metamodels there exist mappings from UML to OWL, topic maps to OWL and RDFS/OWL to common logic. Just to get an idea about what these metamodels look like, two important elements of RDF and OWL will be presented.

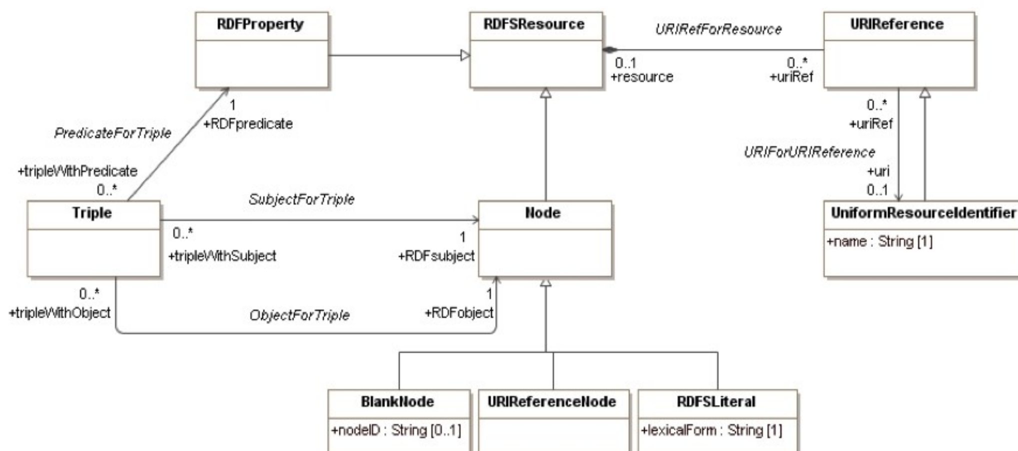


Figure 38: ODM Metamodel of an RDF triple ([156],page 35)

Figure 38 shows a part from ODM's RDF metamodel that captures an RDF triple. The base class of most elements in this diagram is *RDFSResource* (compare section 3.1.3.1). An RDF triple consists of one subject of type *Node*, one predicate of type *RDFProperty* (compare section 3.1.3.2) and one object of type *Node*. The class Node is further refined to the more specific types *BlankNode*, *URIReferenceNode* and *RDFSLiteral*. All RDF elements, since they are subclasses of *RDFSResource*, might have an *uriRef* property that eventually refers to an *UniformResourceIndentifier*. As already pointed out in section 3.1.4 not every type of node might appear at every position within a triple (e.g. Literal values must not be used as subject).

To cover restrictions like these one ODM uses OCL wherever possible as shown Listing 53.

```
context Triple SubjectNotALiteral inv:
    not self.RDFsubject.oclIsKindOf(RDFSLiteral)

context Triple PredicateNotALiteral inv:
    not self.RDFpredicate.oclIsKindOf(RDFSLiteral)
```

Listing 53: OCL statements to constrain the use of certain node types within RDF triples ([156],page 39)

Another example of how to capture important semantic artefacts within ODM is shown in Figure 39. *OWLClass* is modelled as a subclass of *RDFSClass*. Simple class assertions like equivalentClass or disjointClass are modelled as self references. OWL offers various axioms to define classes (see section 3.2.5.1). These different types of axioms are represented by separate subclasses of *OWLClass*. ODM does not provide any OCL constraints for this metamodel since the set semantics of OWL is much richer then is in OCL. Thus, these definitions are outside the expressiveness of OCL. Basically ODM models can be seen as an additional syntax for OWL knowledge bases whereas consistency checks and model validations are done via the use of standard DL reasoners.

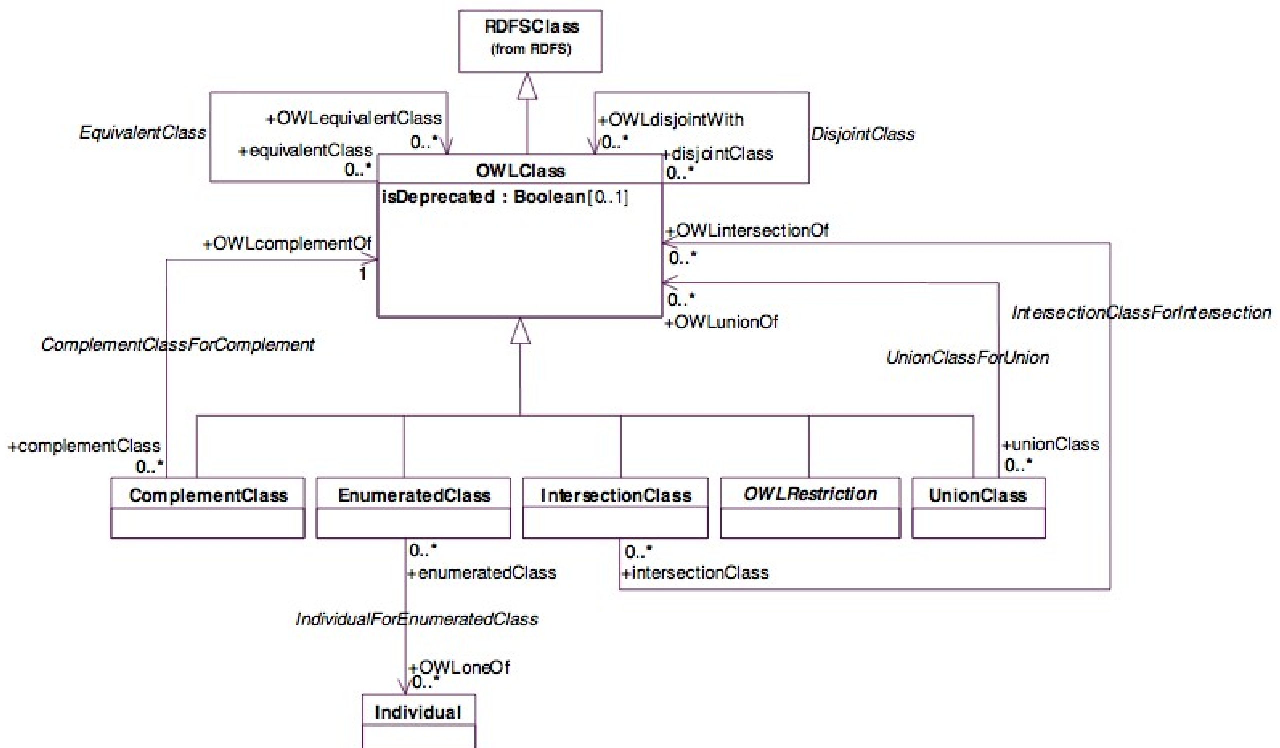


Figure 39: ODM metamodel of the OWL class ([156],page 69)

5.5 Discussion

The biggest benefit proposed by MDA is its possibility to reuse information at a higher level of abstraction and to use automatic model transformation in order to adapt a system to various platforms, which should also allow for rapid adoption of new technologies and platforms. On the other side, since abstraction means the suppression of irrelevant detail, models at higher level of abstraction lack information needed by models at more concrete levels, since at the consecutive layer this information cannot be considered to be irrelevant any longer. This gap of information between models at different layers becomes most evident at the lowest

level transformation resulting in actual source code. Almost all MDA approaches require additional manual coding after the last transformation step[146]. However, to preserve the value of a model, it needs to be updated during the development cycle with every change request, which requires more concrete models being re-generated. One important aspect of this roundtrip engineering is that manually added code does not get lost when source code is re-generated. This is most often achieved by the introduction of so called protected areas in source code files or separate files that contain manually added code, which will not be overwritten by code generators. There also exist recommendations to use specific model transformation languages enabling models at different levels to be kept in synch automatically [159].

Generally the problem of bridging the gap between models representing the problem domain and artefacts of the software implementation domain is perceived as the so called problem-implementation gap [160]. Current trends in practice as well as research indicate that the use of more specifically tailored domain specific modelling approaches helps to bridge or at least to narrow this cap [161][162]. Domain specific modelling (DSM) and domain specific languages (DSL) directly map model concepts to domain concepts and therefore include most of the additional information required for generating executable code. A study comparing the efficiency of DSM and UML models in terms of maintainability comes to the result that DSM models are significantly easier to maintain resulting in less errors [163]. One of the reasons seems to be that DSM models are typically much smaller than more generic UML models.

The integration of MDA and semantic technologies is especially important in the context of the envisaged framework to ontology-driven E-Government. Here the ODM provides one generic approach to extend the range of models to RDF and OWL knowledge bases. Since they provide MOF based metamodels, these models can be translated like any other MDA model. Formal model theoretic semantics, which is implicitly included via the syntactical equivalence of ODM models and RDF/OWL, allows the use of suitable reasoners, although this requires appropriate serialisation of these models. Thus, ODM provides a basis to integrate semantic technologies into MDA approaches. ODM therefore is merely an alternative notation for OWL, although it can be integrated into the MDA tool stack. WSML on the other hand is already based on a MOF metamodel, which should simplify its usage within MDA. An example of how to support UML visualisation using a custom UML profile can be found in [164]. Besides integrating ontologies into MDA models, there are also ways to extract ontologies from existing UML models [165].

UML includes the object constrained language, which provides additional semantics. Although section 5.3 tried to point out the similarities in expressiveness between OCL and semantic frameworks, OCL represents a programming and rule language paradigm and can therefore hardly be compared to logic families underlying semantic technologies. A discussion of the most important difference between OCL and description logics, which focuses on set semantics can be found in [156], section 8.4. Other studies also point out that there are more differences than similarities and that OCL and semantic language families have to play some complementary roles[166].

Whereas ODM provides a mean to integrate ontologies into the MDA model family there exists a variety of recommendations when it comes to the model driven development of semantic web services. Thus, these solutions are domain specific since they are focussing on particular solutions. One framework recommends UML (especially activity diagrams) to model semantic web services based on OWL-S [167]. UML models are firstly serialised to XML and then transformed into OWL-S profiles using XSLT. Thus, this approach allows for a model driven development of OWL-S based semantic web services. Another highly elaborated and comprehensive approach that focuses on the creation of WSMO based semantic web services is presented in [168]. Besides the utilisation of various methodologies and techniques it also comes with a specialised software process model. This process model consists of eight phases that can be executed iteratively: requirement specification, process design, data design, hypertext design, semantic description, architecture design, implementation and testing/evaluation. After elicitation of the system requirements, a high-level model of the application's underlying processes is created using the Business Process Modeling Notation (BPMN) [169]. In the following data design phase a comprehensive domain model based on extended entity-relationship modelling is created that might incorporate imported ontologies. In the hypertext design phase functional requirements are translated into web services and so called website views, which basically represent the user front-end. Both, the data model as well as the hypertext model are part of a methodology called WebML[170] that is adopted in the development process. What makes this approach unique is the semantic description phase, which adds all the required information to implement WSMO compliant semantic

web services. Concepts are extracted from the entity relationship model and a WSMO process model is extracted from the BPMN model. The extraction process is performed semi-automatically.

There exists another recommendation for a semantic model driven approach to the development of service oriented architectures[171]. In the proposed framework ontologies that follow the Web Service Process Ontology (WSPO)[172] are used for the functional model of the application. Together with so called distribution patterns they are the basis of a generation step producing required artefacts like WSDL and WS-BPEL (Business Process Execution Language for Web Services) files. The interesting aspect is that this approach uses ontologies as the model of the application, which comes close to the envisaged approach to ontology driven e-government. However, the adopted WSPO framework is according to the authors of the this approach a predecessor of SWSF (see section 4.3), which in turn is already overhauled by WSMO.

In order to realise most of the benefits that are expected from model driven architecture the creation of domain specific model sets that easily capture the particularities of the problem domain seems to be a key success factor. Possible ways to adopt standard MDA technologies are provided by UML profiles or by providing customised MOF based model elements. ODM provides a standardised way to incorporate OWL into MDA, however various other recommendations exist.

6 Ontology Modelling

Semantic methodologies and language frameworks offer a wide range of capabilities. Thus the question is whether there are any guidelines or best practices that will lead to their efficient use. This chapter tries to identify general guidelines for ontology modelling as well as best practices of ontologies in the context of the E-Government domain.

6.1 General Ontology Modelling Guidelines

Thomas Gruber recommends the following general design criteria for modelling ontologies [17]:

1. **Clarity**: An ontology should clearly define the intended meaning of its concepts and also include natural language documentation. Wherever possible, axioms should be used to express definitions. The motivation for the definition of a particular concept should have no impact on the definition itself, thus allowing the use of this concept in other contexts as well.
2. **Coherence:** An ontology should only allow for inferences that are consistent with the definition. This also applies to the natural language documentation. Any sentence derived from axioms must not contradict the definition or examples given in the documentation
3. **Extendibility**: An ontology should offer the conceptual foundation for a range of uses beyond the ones it was originally defined for. This should allow for extension and specialisation of this ontology without a need to revise existing definitions.
4. **Minimal encoding bias**: The notation used to define an ontology should have no influence on the resulting definitions. I.e., the convenience of notation or implementation should not drive the design.
5. **Minimal ontological commitment**: An ontology should be based on a minimum number of claims about the world being modelled and only define those terms that are essential for the given domain. This should allow other parties to specialise and instantiate the ontology as needed.

Most of these recommendations can be achieved by using a layered approach to ontology modelling. This means that there are several layers of abstractions allowing for efficient re-use of concepts as well as for the necessary domain specific specialisation by extending, adapting or redefining concepts defined in higher layers. Technically this is accomplished by defining different ontologies identified by different namespaces. More specific ontologies import the more abstract ones and add necessary attributes and concepts as well as additional axioms.

6.2 Governance Enterprise Architecture (GEA)

In section 5.5 it was argued that MDA approaches show best results when they were specialised to specific domains. Thus in this chapter a suggestion for an E-Government domain specific model is presented.

The Governance Enterprise Architecture (GEA)[173] provides reusable top-level models for the overall E-Government domain. GEA is the result of a business driven approach to create a reference ontology for the E-Government domain. Even it suggests the use of semantic web services (SWS) the GEA model itself is technology neutral (although there exists a WSMO implementation of GEA [174]). According to this model the interaction between citizens and public administrations (PA) is split into two major parts: planning/informative and execution/performative part.

The planning part consists of all activities and steps that need to be taken in order to provide citizens with all the information necessary to effectively identify, find and use public administration services. This is to answer the "Why, What, Who, Where and How questions" [175]. The planning part is split into the following three activities:

- **Mapping needs-to-services**
  This step tries to bridge the gap between the different points of view of citizens and public agencies. Whereas citizens are typically driven by a particular need or desire, public organisations concentrate on services. Thus there is an obvious need to map citizens' needs to (a set) of PA services that might serve these needs. This is the basis for allowing citizens to identify services that are most appropriate for their particular situation in a need-centric fashion.
- **Service discovery**
  After a citizen's need was translated into a service that is needed within the previous step, this service can now be located. To facilitate this, GEA proposes a so called Central Public Administration Service Directory (CPASD) that holds necessary information to answer the *What*, *Who* and *Where* questions
- **Service exploration**
  Within this phase citizens are provided with information from the actual service provider about the *When* and the *How*. This includes all necessary preconditions.

Like the planning part the execution part is split into three phases as well:

- **Information gathering**
  All information that is needed as input to the selected service is gathered. GEA refers to this type of information as evidence
- **Information checking**
  Evidence provided is checked against the business rules of the service. This might happen in a single step but could also become relatively complex including conditional checks based on the input provided.
- **Providing Output**
  This step provides proper communication about the consequences and effects of the service used. This includes information to other agencies that be notified about these effects.

6.2.1 GEA Object Model for Service Provisioning

This model is based on in-depth analyses of the E-Government domain and is intended to be a conceptual bases for a reference ontology in the field of PA services. An overview of the key concepts can be seen in Figure 40.

There are actually two different entities participating in the service provision model. *Social Entities* who are for example citizens or companies and *Governance Entities*. According to their assignment *Governance Entities* are split into two different types: *Political Entities* who *define Public Administration Services* (not explicitly shown in Figure 40) and *Public Administration Entities* who might play different *roles* in service

provisioning. These *roles* are:

- *Service Provider*: Offers a *Public Administration Service* to *Social Entities*
- *Consequence Receiver*: A *Public Administration Entity* that needs to be informed about the *outcome* of a public service. E.g.: If a family with children moves into a new community the school authority needs to be informed after registration to make sure that the children will attend school.

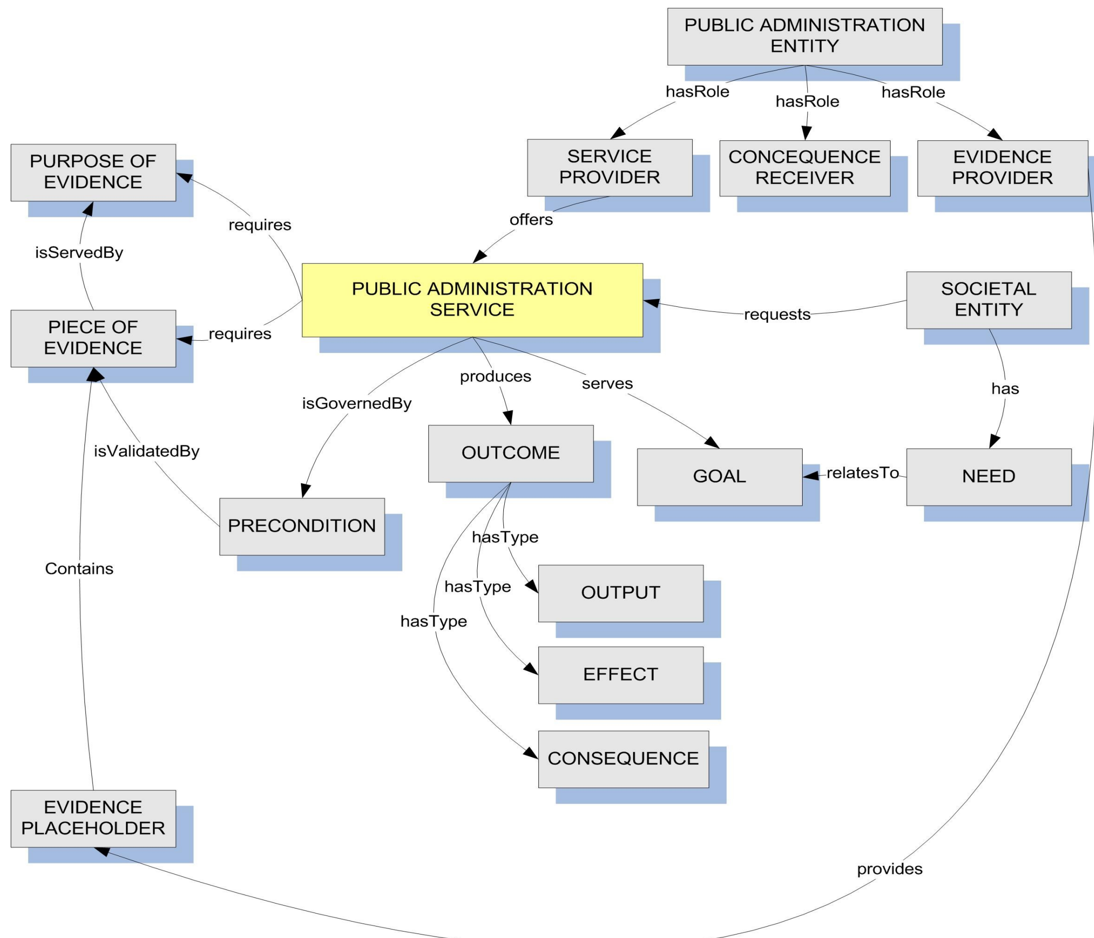


Figure 40: The GEA detailed object model for service provision [175]

- *Evidence Provider:* A *Public Administration Entity* that provides a certain *Piece of Evidence* that is needed as input for the *Public Administration Service*.

Pieces of Evidence are facts and are typically contained in so called *Evidence Placeholders*. An *Evidence Provider* is typically a document that contains information about the fact. In the GEA object model exists a many-to-many relation between these concepts, stating that a *Piece of Evidence* can be contained in several *Evidence Placeholders* and also that an *Evidence Placeholder* can contain several *Pieces of Evidence*. E.g.: A typical *Piece Of Evidence* could be the date of birth of the applicant. This information could be proven by several different *Evidence Providers* like passport, personal identity card, certificate of birth and so on.

Pieces of Evidence are checked against a service's *Preconditions* which represent some of the services business rules. These preconditions have to be met to be eligible for service utilisation. E.g.: To apply for a place in a Kindergarten the date of birth of the child (*Piece of Evidence*) has to be within certain limits (*Precondition*).

Every Public Administration Service results in some kind of output. The output is of one of the following types:

- *Output*: In the GEA object model the output is defined as the documented decision of the *Service Provider.* This information is typically sent to the Social Entity as a administrative document/decision.
- *Effect*: In semantic web services an effect describes the change of the state of the world whenever the service is executed successfully (E.g. an instance of person is transformed into an instance of driver if an application for a driving license was approved). In the GEA object model the *Effect* is the actual right or obligation (permit, punishment, certificate, ...) the Social Entity is entitled with. An *Effect* only exists if the service ends successfully (the *Social Entities* request was not rejected prematurely).
- *Consequence*: This type represents information that is forwarded to other interested parties.

In order to support the needs-to-service mapping step of the planning/informative part, the GEA object model contains two important concepts that allow to link *Social Entities* to *Public Administration Services*. These two concepts are *Need* and *Goal*. Need describes the citizen-centric view of the PA domain. Citizens have certain needs in particular situations (e.g. to build a house). A Goal describes the service-centric view of PA domain, which includes the outcome of PA services that might contribute to serve citizens' needs (e.g. acquiring a building permit). Mapping needs to goals and therewith linking the citizen view to the PA view allows for user-friendly service discovery.

6.3 Discussion

To follow well established guidelines and best practices is particularly important in the field of ontology modelling since the potential solution space is enormous. Good guidelines constrain modelling efforts towards better solutions. Specifically the guidelines and recommendations stated in section 6.1 are aiming at better models and also emphasise on facilitating re-use which adds significant value to the resulting models.

The GEA model was developed as part of an EU sixth framework programme project called SemanticGov [16] (FP6-2004-IST-4-027517) between 2006 and 2009. It therefore is the result of a joint European effort to establish a top-level e-government meta-model using semantic web technologies. It also is the conceptual backbone of the SemanticGov Architecture[176]. Literature research could not find any other reference model with a similar degree of comprehensiveness. Although the authors of the GEA model categorise their model as an initial starting point that should be further developed according to upcoming needs, it is worth to re-use the results of this effort as a starting point for a meta-model. Together with the previously mentioned guidelines the non-invasive adoption of this model to ODEG-specific needs should be possible.

One potential general disadvantage of the GEA-PA model is probably the fact that it reflects the Government domain on an “as-is” basis. This is indicated by its document-centric view when it comes to the description of public services. *`EvidencePlaceHolder`*s are representing documents and certificates, thus the model seems to be influenced by document flows. To support more sophisticated features, that are no longer limited to entire documents, the model needs to be adapted.

7 Ontology Driven E-Government

This chapter will present the implementation of ODEG. The first step in the development of ODEG was a feasibility study that should demonstrate whether semantic technologies are apt to model electronic services at all and that is presented in the next sub-section. The evaluation of the outcome of this study heavily influenced ODEG's design as it will be pointed out in section 7.1.4. One crucial aspect was the selection of the semantic technology used. Although the differences between candidate frameworks have already been discussed in sections 3 and 4 the final decision is motivated in section 7.2. After this different components of ODEG are presented. The goal of ODEG is to support all phases in E-Government service enactment like shown in Figure 41. All components that make up the system are based on the semantic model. This is why the meta-model that is used to define how public services are modelled is presented first in section 7.3. After this the service identification component that allows to identify relevant services based on a citizen's specific

16 http://www.semantic-gov.org/

desire or situation is presented in section 7.4. Once services appropriate to help citizens' are identified they can be directly utilised. This is made possible by the semantic forms component that is presented in section 7.5. This component uses the semantic description of a service to figure out what information is needed in order to access a particular service. Thus the electronic forms rendered by this component are entirely based in the model and are rendered dynamically based on the current situation.

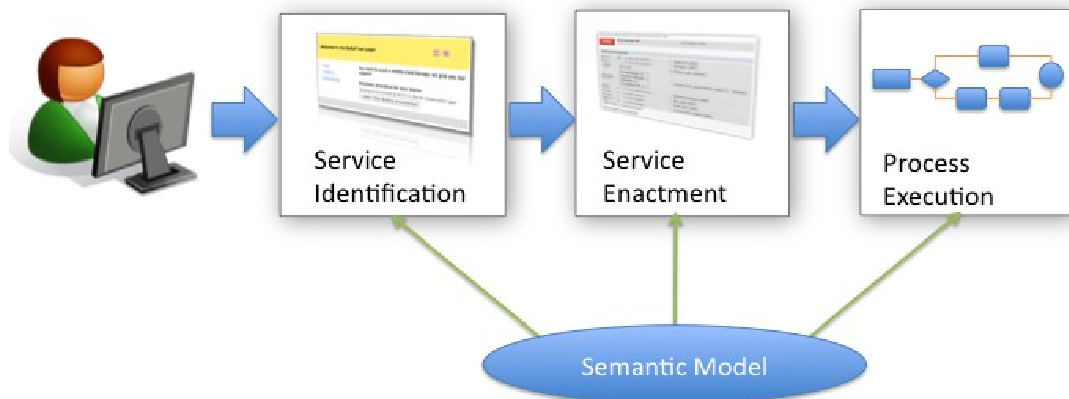


Figure 41: A typical ODEG usage scenario (own illustration)

During the implementation of different public services it became apparent that there is sometimes a need to integrate external services into the forms generation process. These service can be used to validate data (e.g. to validate the existence and correctness of a street address) or to look up values from central databases. These services are called auxiliary services since they can be used add convenient features to the system and that are beyond automatic reasoning. The approach used by ODEG to incorporate such external services into forms creation and validation is described in section 7.6. The last step in an ODEG usage scenario is the execution of the actual service once the necessary data was collected. Basically ODEG here offers different approaches and is open to be tied to almost any type of service implementation as it will become clear in the meta-model sub-section. Nevertheless the preferred implementation type is the usage of standard web service technologies. This is supported by the automatic generation of all necessary web service artefacts. This includes the description of the web service by providing a WSDL document as well as XML schemes that define the types of the messages used by the web service's operations. Creating XML schema based on the content of a semantic knowledge base is a non-trivial task as will be pointed out in section 7.7. After the web service artefacts are available any web service framework could be used to provide an appropriate implementation of the service contract defined in the WSDL file. ODEG, however promotes the use of BPEL since this extends the idea of the MDA, which is to prefer modelling to programming for the actual service implementation phase as well. The detailed explanation why BPEL should be used is therefore given in section 7.8. Finally an overview of the ODEG approach is given in section 7.9.

This structure also reflects the phases in which ODEG was developed. In fact it took several iterations to define the system as it is described here. The overall approach was to define a sound meta-model based on the GEA model already presented in section 6.2 first. After this actual services were implemented based on this model in several iterations. Since the implementation was supported by the City of Graz, these service represent typical procedures offered at municipal level, which is also reflected by the running examples used for illustration purposes in the upcoming sub-sections. Sometimes the requirements of new services exceeded the capabilities of the meta-model. In such cases the lack of functionality was carefully analysed and the most generic way to deal with these requirements was introduced to ODEG's meta-model or interpretation capabilities. This for example led to the model of implementing auxiliary services. After each iteration the number of necessary adaptations of the meta-model declined. Recent application of ODEG to additional domains (e.g. business registration) showed that there was no need at all to extend ODEGs current capabilities to deal with new use-cases. That is why the current state can be considered feature-

complete although extensions to the system can be easily implemented.

7.1 Initial Feasibility Study

To demonstrate that there are suitable ways to use ontologies as models for E-Government applications, a prototypic implementation to create web forms and to validate user input data was created. The basic intention of this prototype that was created as part of a diploma thesis [177] was to provide a proof of concept. Since the outcome of this work provided valuable input for the final implementation it will be briefly presented in the following sub-sections and the most important findings will be discussed.

7.1.1 Prototype Requirements and Example Scenario

The general requirements for the prototype were to create a semantic model of the problem domain that is used to automatically create electronic forms. These forms are used to file an application for the sample procedure. Validation of user input has to happen according to the constraints defined in the model, thus all necessary constraints have to be part of the model. Once the user has completely filled in the form and the provided information has successfully passed validation, the application data has to be provided as XML, which complies to the so called EDIAKT II schema [178]. EDIAKT II is an Austrian national recommendation for a standard data exchange format that is mainly used between different public agencies. This format defines how to exchange either single documents (EDIAKT light) or entire files including various documents (EDIAKT complete) between different peers. To use EDIAKT II as the resulting data format was an essential requirement since it allows processing of the acquired data by any system supporting the recommended standard and therefore allows to use the form creation tool as interface in a broad range of scenarios.

The sample use case chosen for the first prototypic implementation was the application for a permanent parking permit within zones with limited parking in the City of Graz. People who live in Graz and own a car that is also registered in Graz are eligible for this type of permit. Thus, when applying for a permanent parking permit you have to prove that you are a resident of the City of Graz, that you own a car and that this car is registered in Graz as well.

7.1.2 Semantic Service Model and Ontologies

The general model for public services that was developed as part of the prototype is based on the GEA model presented in section 6.2.1. As already pointed out in the discussion of the GEA model, it represents the government domain on an “as-is” basis. Service descriptions rather refer to documents (pieces of evidence and evidence placeholders) than to the actual information that is required by a procedure. Since the focus of the prototype was on the creation of forms, it is essential to model which data is needed to access a particular service at an appropriate level of detail. Therefore the GEA model was modified like indicated in Figure 42.

The major difference to the GEA recommendation is that a service can refer to arbitrary concepts that serve as input to the service. Thus, a service's *required* attribute that lists the required input is not limited to evidence placeholders but can hold any type of concept. This is inspired by the semantic web service frameworks presented in sections 4.2 and 4.4, although the service itself is still an implementation neutral description.

To implement the service model for the prototype OWL was chosen. This was mainly motivated by the fact that OWL was the most widely adopted technology recommended by the W3C and that there was already rich tool support available. The actual domain specific ontology that represents the classes necessary to model the use-case was heavily influenced by the EDIAKT II recommendation. Since one important requirement was to create EDIAKT II compliant XML as the final result, it seemed natural to establish close links between datatypes defined in the schema and classes used in the ontology to facilitate lowering and lifting between the semantic model and the corresponding XML representation. In fact there was a separate OWL ontology created that holds all classes that are also defined as types in the EDIAKT II schema, since it was assumed that the data exchange standard contains a comprehensive set of datatypes that are

intensively used in the E-Government domain. Thus, every complex type used in the EDIAKT II schema was transformed into a corresponding OWL class. This for example included classes that represent different types of persons (natural person, corporate body, …) and addresses.

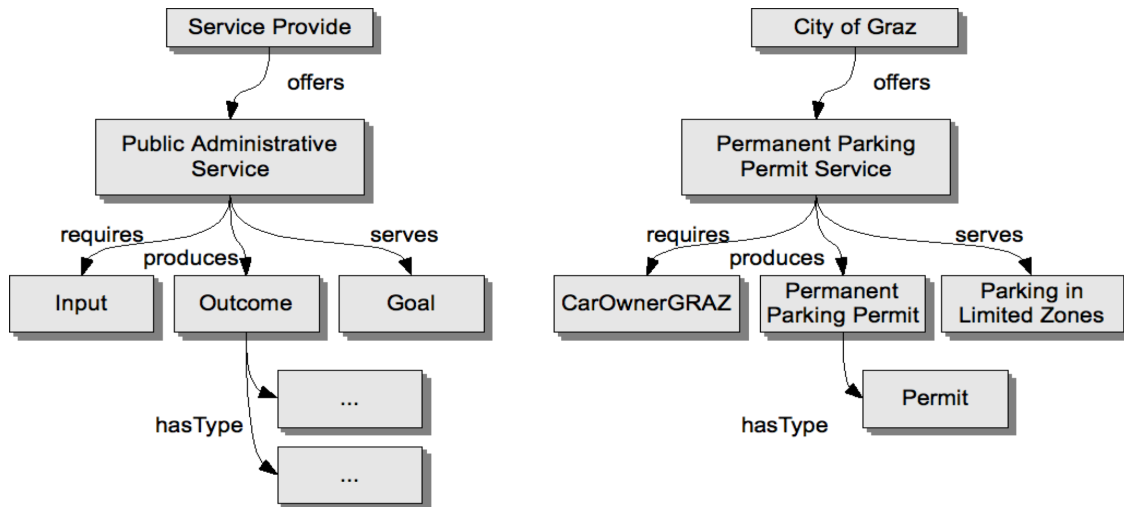


Figure 42: Part of the prototype's service model. The general model is shown on the left and the description of the permanent parking permit service on the right (own illustration based on [177]).

To meet the particular requirements of the prototype usage scenario the resulting concept hierarchy was extended. This led to two additional classes *Car* and *CarOwner* as shown in Figure 43.

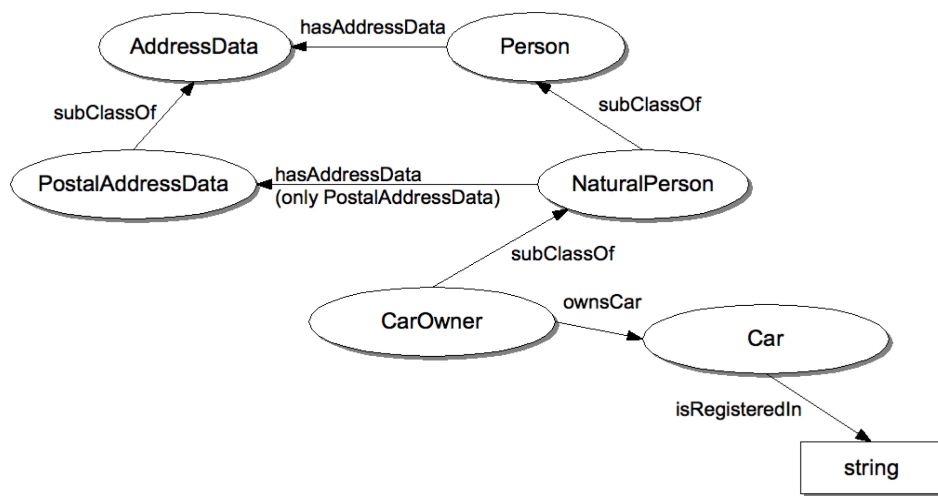


Figure 43: Fragment of the prototype's domain model (own illustration)

The *Car* class reflects the vehicle as it is required to exist by the requirements of the prototype's use-case. Beside other attributes one important property is the place where the car is registered. In this example solution this property is modelled as a datatype property of type string. The *CarOwner* class represents a person who owns a car. Therefore CarOwner is defined as a subclass of NaturalPerson with the additional property *ownsCar*. At this stage the model contains and defines all the data that is needed to apply for a permanent parking permit, although not all properties are shown in Figure 43. What is not covered yet, are the constraints that have to be met in order to being eligible for a permanent parking permit. Therefore the model presented in Figure 43 has to be restricted like shown in Figure 44.

As already mentioned, anyone who lives in Graz and owns a car that is also registered in Graz is eligible for a permanent parking ticket. To express these constraints the relevant classes are subclassed and appropriate restrictions on their properties are defined. This for example leads to a class called *CarGraz*. It is a subclass of *Car* but the value of its *isRegisteredIn* property is restricted to the literal “Graz”. Although the example solution presented here uses a string value to refer to a city this does not restrict the general applicability of the chosen approach to use restricted properties.

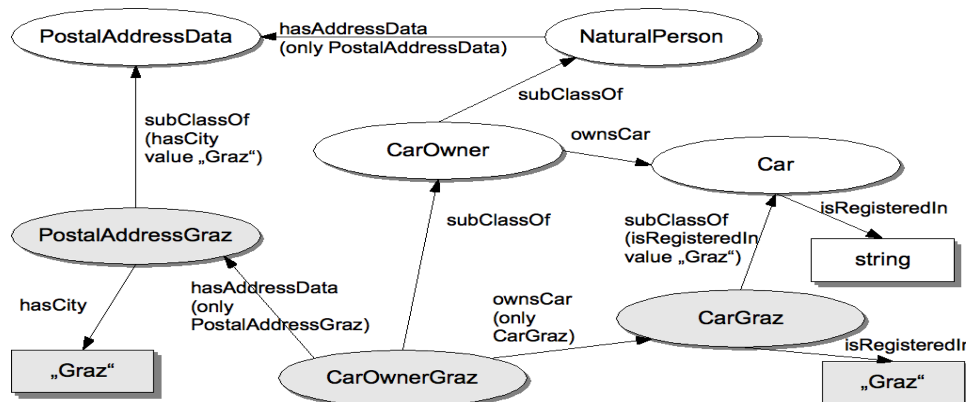


Figure 44: Introducing constraints by subclassing existing class with restricted properties (own illustration)

It could also be applied to a scenario that uses a class to express locations, in which case the restriction of an object property would be almost identical (using a restriction like “*isRegisteredIn* value *Graz*”, where *Graz* is the identifier of the individual representing the City of Graz). Similar to *CarGraz* also a class representing addresses in Graz is defined (*PostalAddressGraz*). Finally a new type of *CarOwner* called *CarOwnerGraz* is defined. Individuals of this class are only allowed to have addresses in Graz (*PostalAddressGraz*) and may only own cars registered in Graz (*CarGraz*). By declaring the class *CarOwnerGraz* as the required input element to the permanent parking permit service (like indicated in Figure 42) the service description now contains all the required information together with the logical constraints that apply.

7.1.3 Generating Forms to Access the Permanent Parking Permit Service

The prototypic implementation uses XForms[179] as presentation technology for electronic forms. XForms is promoted by the W3C[17] and is supposed to be widely adopted as the new standard for electronic forms in the near future. Whereas ordinary HTML based web forms only use the two datatypes string and boolean, input elements in XForms can be bound to types defined in an XML schema which makes them type-safe and allows for XSD constraint checking. Another XForm characteristic is the fact that it uses XML as its data transport format, whereas standard HTML forms are using various text encoding schemes that require further parsing at the server side.

One necessary prerequisite to use XForms is the existence of an XML schema. Therefore the data model definition expressed in the ontology needs to be converted into a schema. Since OWL's standard serialisation format is XML, an XSLT transformation can be used to extract a schema from the OWL classes. The stylesheet used in the prototype analyses all service description classes and selects their input classes for transformation. However, if any of the classes used is a subclass merely defined by property restrictions, then this class is replaced by its direct superclass (e.g. the class *CarOwnerGraz* is replaced by its superclass *CarOwner*). After a class was added to the resulting schema, its properties are added as child elements to the current type. How these properties are treated depends on their type. If a property is of a type that was originally derived from the EDIAKT II schema a reference to the corresponding type is added. If the current

17 http://www.w3.org/MarkUp/Forms/2003/xforms-faq.html

property is a datatype property then it is added as an ordinary XSD datatype. If, however, the current property is an object property then it is recursively added as another class. The resulting schema for the permanent building permit application is shown in Listing 54.

```
<xsd:schema … >
  <xsd:element name="PermanentParkingTicketRequest">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="CarOwner">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="Car">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="LicensePlate" type="xsd:string"/>
                    <xsd:element name="Type" type="xsd:string"/>
                    <xsd:element name="isRegisteredIn" type="xsd:string"/>
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
              <xsd:element name="PhysicalPerson">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element ref="ediaktPersonData:CompactPhysicalPerson"/>
                    <xsd:element ref="ediaktPersonData:CompactPostalAddress"/>
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 54: XML schema generated from the prototype's ontology

After the XML schema was created a corresponding XForm that is embedded inside an XHTML document can be defined. Therefore IBM's XForms Generator [180] was selected. Beside an existing XML schema this tool also requires a sample XML instance of the data that should be gathered. This sample instance has to be created manually. The generated XForm has to be presented to the end user as part of a web application. Within the prototype implementation an XForm processor called Chiba [181] was used to bring up the actual web form. Although XForms are type-safe and can be used to restrict certain fields to a particular data range they do not posses the logical expressiveness of OWL ontologies. Thus, after the form is filled in by a user the data is fed back into a semantic reasoner to check its consistency with the axioms stated in the ontology. This requires the resulting XML data to be translated into OWL first. To get this task accomplished, a tool called JXML2OWL Mapper[18] was used. This software allows to graphically map any XML schema to OWL classes. This mapping results in another XML stylesheet that is used to do the actual transformation. After the user's input is transformed into OWL, the information is loaded into an RDF/OWL reasoner called Jena[19]. The reasoner allows to check whether the loaded data is valid according to the rules of the previously modelled ontology. If so, the data provided by the user is accepted, otherwise an appropriate error message is displayed to the user. The entire process of creating the semantic model, generating an XForm, displaying the form and validating user input is shown in Figure 45.

18 http://jxml2owl.projects.semwebcentral.org/jxml2owlmapper/index.html
19 http://jena.sourceforge.net/index.html

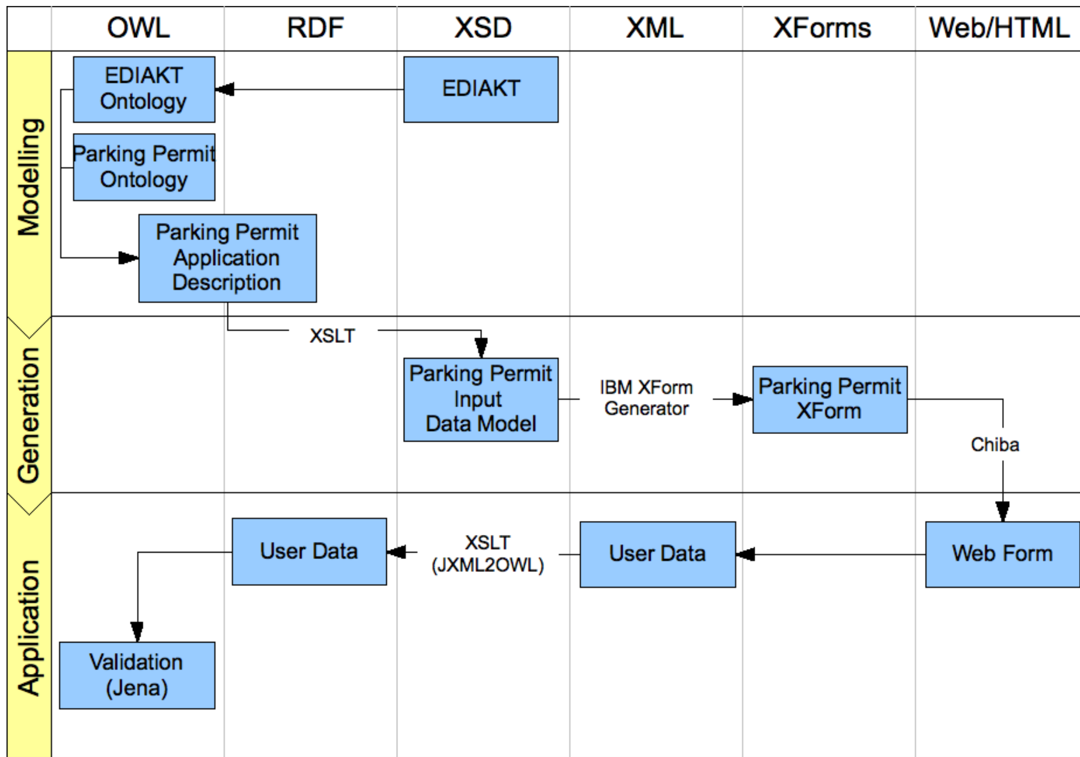


Figure 45: Overview of the prototype's modelling and generation process (adapted from [177], page 63)

7.1.4 Lessons Learned

The intention of the prototypic implementation was to conduct a feasibility study that should demonstrate the possibilities of semantic models as a source for automatically created web forms. It was further intended to use standard technologies, tools and frameworks in order to minimise the amount of necessary custom code. The prototype therefore was successful to demonstrate that there are means to create usable forms based on semantic models, although the creation process is only semi-automatic, since it requires manual intervention. For example, it is necessary to provide a sample XML document of a filled in form in order to create an XForms description. Also the mapping from the XML schema representing the data gathered from the user to OWL classes has to be performed manually. However, since this transformation is based on XML style sheet and is basically the reverse operation of the OWL to XML schema transformation in the first step, further automation of this step is possible.

Whereas the generation of the required XML schema as well as the creation of the sample XML instance happens at design time, the transformation of the resulting XML into OWL instances has to happen during run-time. Thus, the ontology is the model of the application and the creation of the XForms artefacts represents the application generation process according to MDA. Although the numerous transformations and the diversity of technologies used might be seen as disadvantage of the approach used, especially the use of XForms compensates for some of OWL's drawbacks. OWL is hardly suited for expressing constraints in order to enforce consistency checks solely based user provided data. The design of the ontology therefore had to be done carefully, always having the purpose of the model in mind, which directly contradicts the modelling principles presented in section 6.1. The logical constraints of the sample application as discussed in section 7.1.2 and depicted in Figure 44 enforce the applicant to be a citizen of the city of Graz and to own a car that is also registered in Graz. If a person, however, is allowed to live in several places and/or to have more than one car, necessary constraints could still be expressed, but no longer checked with an OWL reasoner. In this case, if the person would fill in an address that is not in Graz (e.g. Vienna) and/or would provide data of a car that was not registered in Graz, the reasoner will try to make the model consistent by assuming that the missing data is simply not known yet, or by inferring that “Vienna” is a synonym for “Graz”.

To prevent this behaviour, the relevant properties had to be restricted to a single possible value by making them *functional*. As a consequence the resulting model does no longer exactly describe the real life situation, which in the best case might be perceived as being less intuitive or simply incorrect in the worst case. Thus, OWL does not seem to be the perfect choice for this kind of application. Nevertheless, as already mentioned before, some facets of constraint checking can be performed at the XML/XForms layer, which at least forces the user to fill in all required fields with values of the valid domain. Having consistency checks spread of several layers and technologies, however, must be considered a general drawback, since it exacerbates quality assurance.

7.2 Technology Selection

Based on the comparison of OWL and WSML/WSMO (see section 3.5) and the findings of the feasibility study (see section 7.1.4) WSML/WSMO was selected as the semantic framework to be used in ODEG. Although WSML/WSMO might be less frequently adopted than OWL, which is also reflected by the fact that OWL is a W3C recommendation, whereas WSMO is a W3C submission only, the functional differences between these two frameworks with respect to the requirements of ODEG clearly favour WSMO. The most important facts that argue for WSML/WSMO are:

- Support of the closed world assumption, which makes constraint checking simple and intuitive.
- Compact frame-based language that can be read even without the use of sophisticated editors.

Both of the aforementioned aspects facilitate the creation of even huge ontologies in a way that is significantly less error-prone than OWL modelling. One shortcoming of the initial prototype was the need to transform data and meta-data back and forth between the semantic notation (OWL) and XML. Especially the use of XForms requires the manual step of creating sample data XML which only allows for a semi-automatic form creation process. On the other side, this XML-based approach was necessary to deploy cardinality restrictions which are otherwise – due to the nature of the open world assumption - not “correctly” checked by reasoners. Since WSML/WSMO supports the closed world assumption, also cardinality restrictions can be intuitively checked eliminating the need for additional representation formats. As a consequence the new solution should be tightly integrated with the semantic reasoner and no additional transformations should be necessary.

7.3 Meta-Model

The aim of ODEG is to offer E-Government services that are almost entirely described by means of semantic models. These models are turned into executable services using MDA principles. One intuitive approach would be to directly model semantic web services. In sections 4.2 and 4.4 two prominent semantic web service frameworks were presented. However, a detailed discussion of these frameworks (see section 4.5) showed that both of them posses some significant disadvantages. Additionally, regardless which framework to choose there are no out of the box approaches to automatically turn these models into executable web services and solutions to provide user-interfaces to these services based on a semantic model do also not exist. Beside these general shortcomings any resulting model would be highly influenced by the semantic web service framework used, since it has to fit the underlying framework specific meta-model. In contrast to this, the GEA-PA model presented in section 6.2 is entirely framework independent. It does not even require the public services modelled to be implemented as web services at all, which also makes it technology neutral. These characteristics comply with the ontology design principles presented at the beginning of section 6. Thus, the adoption of GEA-PA as the basis for an ODEG-specific meta-model complies with ODEG's overall design objectives, since it makes ODEG portable and reusable with other technologies as well. Therefore WSMO-PA[174], an existing WSML/WSMO model of GEA-PA was selected as the starting point for the ODEG meta-model, instead of directly applying the WSMO semantic web service modelling approach.

Figure 46 Provides a schematic overview of how ontologies provided by the framework and service specific ontologies that have to be created as part of the modelling task are used to provide a semantic model an electronic public service.

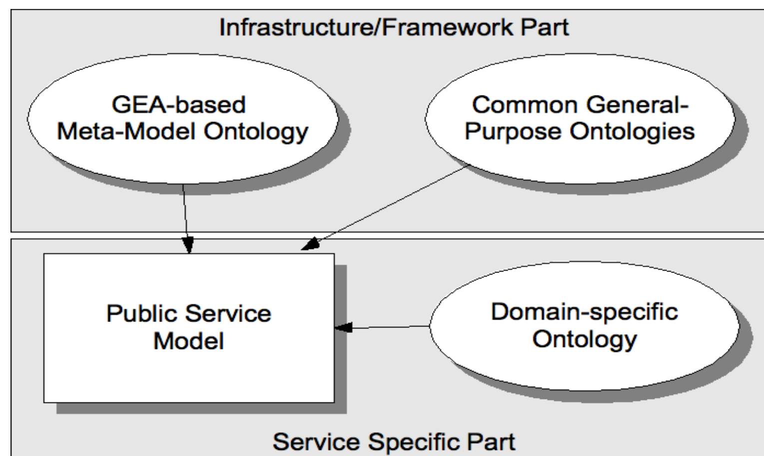


Figure 46: Overview of framework provided ontologies and service specific ontologies (own illustration)

The core ontology is ODEG's meta-model, which is presented in sections 7.3.2 and 7.3.3. Besides this the framework also contains some general purpose ontologies that are likely to be used directly in a service description or that can be used as a basis for domain specific extensions. One of these general purpose ontologies is the `PersonData` ontology presented in section 7.3.3. The service specific part consists of instances of the meta-model ontology and probably some additional ontologies defining concepts that are typically used in the current application domain (e.g. different types of buildings in the case of a building permit procedure).

7.3.1 How to create the ODEG meta-model

As already mentioned, GEA-PA is a good starting point for a meta-model. To find out whether it is really apt to be used together with the ODEG idea, first it has to be evaluated in a variety of sample scenarios. The initial sample scenario selected for an ODEG show-case was the building permit domain. This decision was made together with the City of Graz, the first municipality to adopt ODEG in its E-Government procedures. It was mainly influenced by the fact that this particular problem domain was seen as probably the most complex one at municipal level. Implying the assumption that when the new approach works well within this domain, its results should be easily transferable to other, potentially less complex domains. Due to this decision most of the examples used to illustrate ODEG in the next sections refer to the building permit problem domain.

In the first phase it was important to find solutions for the specific procedures and problems of the building permit domain. To facilitate re-use of identified concepts they were typically layered in several levels of abstraction following the recommendations of section 6.1.

7.3.2 WMSO-PA – An WSMO implementation of GEA-PA

The central element of the ODEG meta-model is the `PublicService` concept (see Listing 55) as it is defined in the WSMO-PA model. This element is a sub-concept of `Service` which is defined in the PROTON[20] Top module ontology. PROTON comprises 300 common domain-independent concepts organised in four modules[182]. By linking to these top-level concepts, very general reasoning about public services is enabled as well. In the terms of the MDA this relationship maps a domain specific model (DSM) to a domain independent model, therefore combines the benefits of general MDA and domain specific approaches (see the discussion in sections 5.5). All definitions of inverse properties were added to the original WSMO-PA concept and therefore are considered minor ODEG specific modifications.

20 http://proton.semanticweb.org/

```
concept PublicService subConceptOf protontop#Service
    annotations
        dc#description hasValue "A public service is a service that a public administration provides to its
clients."
    endAnnotations
    hasClientType inverseOf(requestsPAService) ofType SocietalEntity
    hasPADomain ofType  (0 1) PublicServiceDomain
    hasPASubDomain ofType  (0 1) PublicServiceSubDomain
    hasEffectType ofType  (0 1) PublicServiceEffectType
    hasLocation ofType  (0 *) Location
    hasAdministrationLevel ofType  (0 1) AdministrationLevel
    hasServiceOutcome inverseOf(isServiceOutcomeOf) ofType  (0 *) ServiceOutcome
    isGovernedByLaw inverseOf(governs) ofType  (0 *) Law
    usesServiceInput inverseOf(isServiceInputOf) ofType  (0 *) ServiceInput
    isProvidedBy ofType  (0 *) ServiceProvider
    hasProcess inverseOf(invokesNestedService) ofType (0 *) ServiceProcess
    hasPublicServiceType ofType (0 1) PublicServiceType
```

Listing 55: WSML definition of the PublicService concept

Every service is offered to a particular type of clients represented by the concept `SocietalEntity`. In WSMO-PA exist two sub-concepts representing legal entities and natural persons. This part of the concept hierarchy is shown in Figure 47. Also these concepts specialise the corresponding classes of the PROTON Top module ontology. According to WSMO/GEA-PA every public service is categorised by its service domain (e.g. education, transportation, health, ..) and its service sub-domain (e.g. illness prevention, public health monitoring,...). The effect type describes which consequences are expected from offering and executing this service (e.g. “promote sustainable development”). The `hasLocation` property describes where the service is available. Locations are expressed by sub-concepts of `Location`. WSMO-PA defines two sub-concepts: `PhysicalLocation` (an agency's front office where the service is available) and `ElectronicLocation` (a web-site where the service can be accessed). Since services can be offered conventionally as well as electronically and can also be offered at different locations this property has an unbound cardinality. The administration level describes at which governmental level this service is provided. WSMO-PA defines the levels ministry, region, prefecture and municipality.

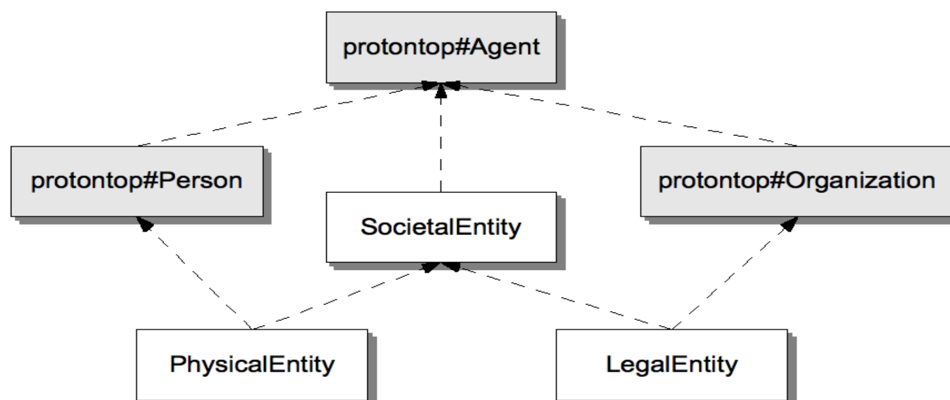


Figure 47: Concept hierarchy of WSMO-PA societal entities and their relations to corresponding PROTON Top module concepts (own illustration)

The `hasServiceOutcome` property can refer to multiple instances of `ServiceOutcome`, which describe what will happen or what is produced when the public service is executed. As shown in Figure 48 the possible values for this property are manifold.

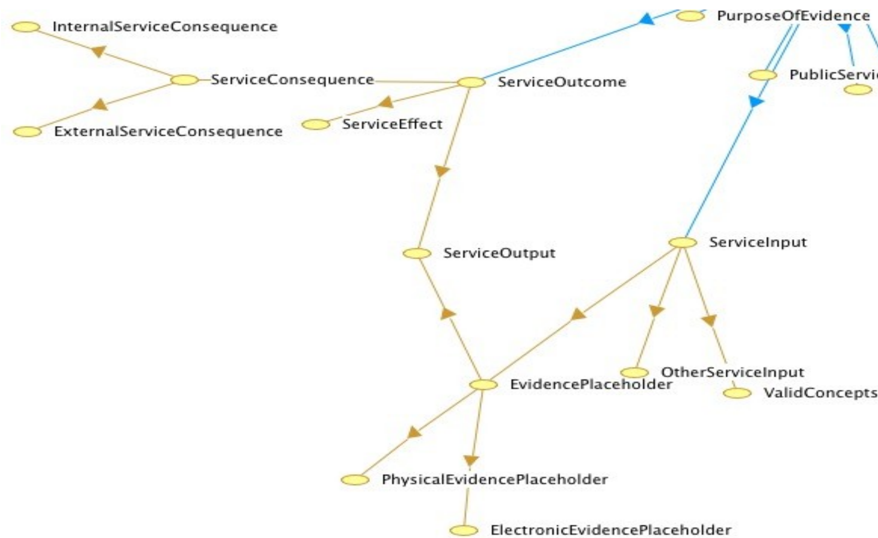


Figure 48: Part of the WSMO-PA ontology representing ServiceOutcome and ServiceInput concepts (own illustration)

A `ServiceOutcome` can be a so called service effect, a service consequence or any instance of type service output. A detailed description of the semantics of these concepts can be found in section 6.2.1.

The `isGovernedByLaw` property refers to potentially several regulations that form the legal basis of this service and therefore might define necessary preconditions. In order to model, which data is needed by a certain `PublicService` GEA provides the `usesServiceInput` property. In the original version of WSMO-PA this property could either refer to an `EvidencePlaceHolder` or an instance of type `OtherServiceInput`. As already pointed out in the discussion of GEA-PA in section 6.3 this reference model rather reflects the (E-)Government domain on an as-is, document-centric basis. In the context of ODEG, however, it is necessary to describe the input of a service in much more detail than simply referring to required documents. WSMO-PA provides a concept called `OtherServiceInput` to extend its capabilities to describe a service's input. This approach, however, is not very helpful, since the only additional semantics introduced by this concept is that it is not an `EvidenceProvider`, which is obvious anyway.

In order to extend its capabilities, the WSMO-PA model was extended by another possible input type called `ValidConcept`. This new concept allows to refer to any concept that is considered to be valid input for a public service. One conceptual problem that became apparent with the introduction of this property is the fact that it crosses the border between concepts and instances. One of the critical features of RDF-S is its possibility to make assertions about triples, which allows for classes that describe classes and instances of classes that are classes themselves. This feature, however, is one of the reasons why RDF-S in undecidable (see section 3.1.5) and eventually led to the introduction of a separate OWL class construct (see section 3.2.5.1). The idea of the newly introduced `ValidConcept` was to allow a service description, which in turn is an *instance* of the `PublicService` concept to refer to a concept and not to another instance. Figure 49 illustrates this situation, where rectangles stand for concepts and ovals indicate instances.

A direct property reference between instances and concepts like the one from `PersonType` to `Person` is simply not possible, since it would break WSML's decidability. Consequently there is no WSML datatype that allows to refer to a class and could be used as the `oneOf` property's type. Versions prior to version 1.0 of WSML provided the datatype `_iri`, which could hold any IRI. Thus, in the initial version of ODEG this datatype was used for the `oneOf` property and allowed to refer to the desired concept's IRI. This approach is conceptually similar to one used by the OWL-S `parameterType` property (see section 4.2.1). Meanwhile, with the introduction of WSML version 1.0 this datatype was removed from the language and wherever `_iri` was used in ODEG ontologies is was replaced by the datatype `_string`. Since every reasoner takes the value of this property as a simple string (or IRI in previous versions) the actual semantics of this description cannot be used for automatic reasoning. Consequently, the interpretation logic responsible for executing the model has

to consider the intended semantics explicitly. Eventually, however, ODEG uses a different approach to describe the input to a service, which will be explained in the next section.

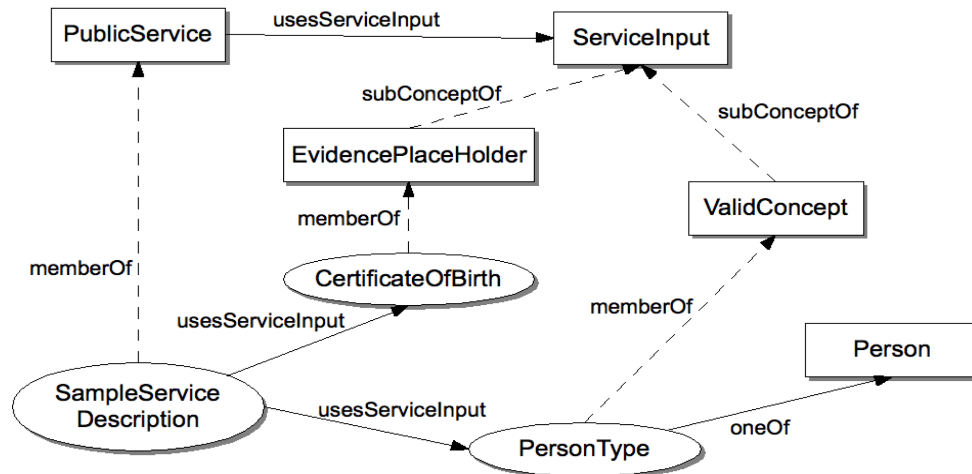


Figure 49: Sample usage scenario of the ValidConcept service input type (own illustration)

The next element of the `PublicService` is its `hasServiceProvider` property, which refers to the `PAEntity` that has the role of the service provider (compare section 6.2.1). The `hasProcess` property of the `PublicService` concept refers to an instance of `ServiceProcess`. This concept is an initial attempt to describe the procedural aspects of a public service. Every `ServiceProcess` is defined by a start date and a property that holds the maximum duration of the process. Besides this, it can hold references to several `ServiceCollaborator`s as well as so called nested services, which are other `PublicServices` that are used by the current service (see Figure 50). A `ServiceCollaborator` is an additional role that a `PAEntity` can have. These properties make a `ServiceProcess` conceptually similar to the WSMO orchestration element (see Figure 25 in section 4.4.1), although it is very basic since it does not describe any control flow aspects and therefore merely represents an enumeration of other public agencies and services involved.

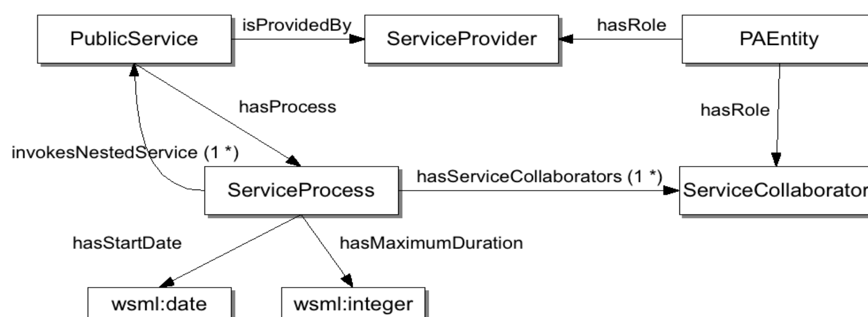


Figure 50: The `ServiceProcess` concept and its relation to other elements (own illustration)

Finally the `hasPublicServiceType` property specifies the type of the service (e.g. control, authorisation, certification, …).

When comparing the recommended GEA-PA object model for service provisioning (see section 6.2.1) to the WSMO-PA model, it becomes obvious that the central concepts `Need` and `Goal` are missing. Since these elements provide the conceptual basis for rather intuitive service identification, WSMO-PA needs to be extended to support such functionality. This is why the so called `Desire` concept was added to the adapted version of WSMO-PA. This concept represents a citizen's desire or need and has only one property with the name `isRelatedToConcept`. The type of this property is string and it holds the IRIs of concepts that are

needed to fully specify a citizen's desire. Thus, the same approach is used to refer from instances to concepts as for the `ValidConcept` mentioned above. A detailed discussion of the idea behind this concept and its internal structure can be found in section 7.4.

This slightly modified and extended WSMO-PA ontology is ODEG's most general top level ontology of the E-Government domain. Following the recommendations for ontology modelling presented in section 6.1 concepts in this ontology are refined by introducing more specific ontologies that specialise WSMO-PA. The next sections present these ontologies.

7.3.3 GEA-SeGoF – Specialising WSMO/GEA-PA

Although, the previous section already described some modifications to the original WSMO-PA ontology as it was presented in [174], these adaptations are not representing a specialisation of WSMO-PA but rather have to be considered fixes of conceptual errors, completion of obviously missing concepts compared to GEA-PA[173] and minor improvements. Thus, the modified version represents a comprehensive WSML specification of GEA-PA. The so called GEA-SeGoF ontology further specialises WSMO-PA and introduces some ODEG specific top-level concepts (see Figure 51). The `ConstrainedPublicService` adds so called `ServiceConstraint`s to a `PublicService` and provides relations to `Desire`s that the service might meet. As already mentioned in the previous section a desire might be further specified by a set of concepts it is related to. Whether a service actually meets a particular desire or not is specified in combination with its service constraints. The `ServiceRequest` concept is the super-concept of all classes that describe input to public services. Specialisations of the service request concept typically contain references to all required concepts and therefore can be compared to application forms in conventional procedures. A `ServiceInputPlaceHolder` is conceptually a more general type of service request. Analysis of some public services showed that facts, which are preconditions can either be proven by adding the appropriate document to the request or alternatively by applying for this document as a sub-procedure. The `ServiceInputPlaceHolder` therefore can represent either a document or the data necessary for the application (in which case the actual concept used with the procedure has to be a sub-concept of `ServiceInputPlaceHolder` as well as `ServiceRequest`). On the other side a `ServiceRequest` is not necessarily a special form of a `ServiceInputPlaceHolder` in every case, this is why there does not exist any sub-concept relationship between these two concepts.

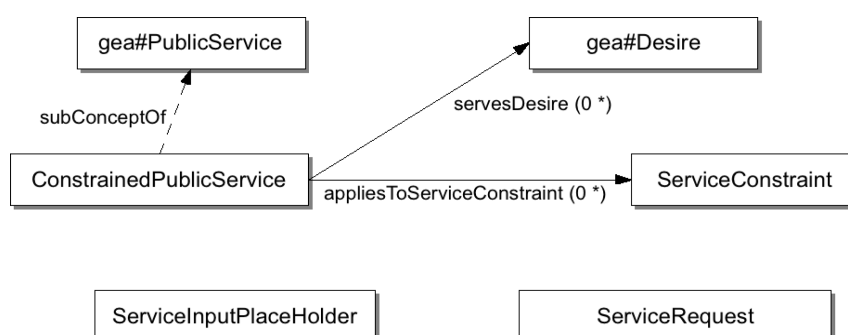


Figure 51: The ODEG specific specialisation of WSMO-PA called GEA-SeGoF ontology (own illustration)

To further explain, how a public service is modelled in ODEG it is helpful to refer to an example. Listing 56 presents the definition of the pull down permit service.

It shows almost all the properties of a public service as presented in section 7.3.2. Additional ODEG specific properties are `servesDesire` and `appliesToServiceConstraint`, which describe the conditions under which this service is apt to meet a citizen's desire or situation. The exact meaning and resulting consequences of these elements are described in all detail in section 7.4.

```
instance GeaPullDownPermitService memberOf geaGraz#ConstrainedPublicService
    annotations
        _"http://www.semantic-gov.org#hasWsmoService" hasValue PullDownPermitService
        dc#description hasValue "Pull down service for constructions that require a pull down permit"
        segofUtil#severityLevel hasValue "1"
    endAnnotations
    gea#isProvidedBy hasValue geaGraz#Graz_Municipality
    gea#hasClientType hasValue construction#ConstructionApplicant
    gea#hasPADomain hasValue construction#BuildingsAndInstallations
    gea#hasPASubDomain hasValue construction#BuildingDomain
    gea#hasEffectType hasValue construction#AllowPullingDownOfConstruction
    gea#hasLocation hasValue geaGraz#Graz
    gea#hasAdministrationLevel hasValue gea#MunicipalityLevel
    gea#hasServiceOutcome hasValue construction#PullDownPermit
    gea#isGovernedByLaw hasValue construction#StyrianConstructionLaw
    gea#hasPublicServiceType hasValue gea#Authorization
    geaGraz#servesDesire hasValue geaGraz#PullDownAConstruction
    geaGraz#appliesToServiceConstraint hasValue construction#PullDownPermitServiceConstraint
    geaGraz#hasContactInformation hasValue geaGraz#ContactInformation_Construction
    geaGraz#implementationType hasValue geaGraz#eGraz
```

Listing 56: The GEA related description of the pull down building permit service that is required whenever particular types of buildings are going to be knocked down

Another additional element that was not part of the original GEA model is the *hasContactInformation* property. Although each service is already related to the service provider it was found that this information is not specific enough to provide citizens with information about responsible departments or persons. Thus, this property allows for fine grained contact information at service level. Also an extension to the original GEA *PublicService* is the *implementationType* property. This property holds information about the service's actual implementation. The GEA model itself is technology neutral, which means that it does not make any assumptions about its own serialisation and representation (whether it is stated in OWL, WSML or any other semantic language) or the implementation of public services described by model instances. Although the *hasLocation* property of GEA's *PublicService* concept can refer to physical and electronic locations (see section 7.3.2), where citizens can find the service, this does not imply any information about how electronic services are implemented. WSMO on the other side, assumes that actual services are implemented by the means of web services. In the context of ODEG it is important to provide information about where to deliver data that was collected by the applying citizen. The ODEG specific extension of the GEA ontology therefore provides the concept *PublicServiceImplementationType* and several implementation specific sub-concepts (see Figure 52).

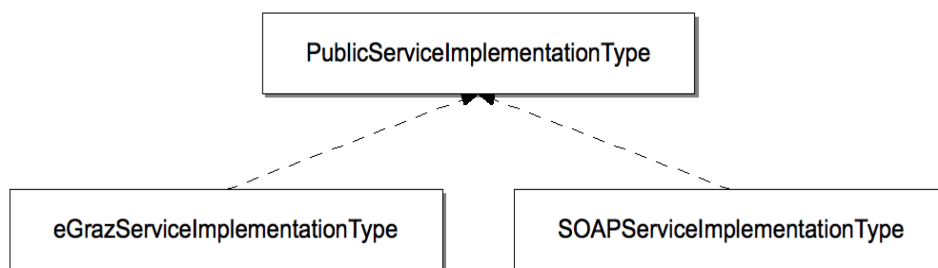


Figure 52: Concept hierarchy describing different possible implementation types of public services (own illustration)

Currently there are two different concrete service implementation types. One is the *eGrazServiceImplementationType*, which stands for a proprietary E-Government solution operated by the City of Graz. Technically this means that collected user data representing the actual application is send to the back-office via a remote procedure call and therefore conducting the public service. An alternative implementation is represented by the *SOAPServiceImplementationType*, which indicates that a service is

implemented by a SOAP based web service. Every instance of *SOAPServiceImplementType* has to have an *endPoint* property which holds the service endpoint of the web service implementation.

When comparing the example public service described in Listing 56 to the GEA *PublicService* concept shown Listing 55 it becomes obvious that the optional *usesServiceInput* property is not used. In fact the information about what data is necessary in order to invoke a public service is key for a framework that wants to interactively gather this information based on a semantic model.

```
webService PullDownPermitService
  nonFunctionalProperties
    wsmostudio#version hasValue "0.7.3"
    _"http://www.semantic-gov.org#geaInstance" hasValue GeaPullDownPermitService
  endNonFunctionalProperties

capability Capability_PullDownPermitService

   sharedVariables {?request}

   precondition
      definedBy
         ?request[construction#pulldownbuilding hasValue ?building] memberOf
                  construction#PullDownPermitApplicationRequest
         and (?building memberOf construction#SmallGarage or
         ?building memberOf construction#MiddleGarage or
         ?building memberOf construction#BigGarage or
         ?building memberOf construction#BigPullDownAdjoiningBuilding or
         ?building memberOf construction#BusinessHouse or
         ?building memberOf construction#MixedHouse or
         ?building memberOf construction#SmallResidentialHouse or
         ?building memberOf construction#BigResidentialHouse or
         ?building memberOf construction#OtherConstruction).

interface Interface_PullDownPermitService
```

Listing 57: The WSMO specific part of the pull down permit service

The approach chosen to model required input is based on some initiatives to further integrate GEA and WSMO[183][184] and is also used by a modelling tool called WSMO Studio[21]. The basic recommendation is to express input to services by the means of appropriate constructs of the WSMO specific *webService* element. This leads to a twofold specification of any public service: a GEA specific part and a WSMO specific part. The WSMO part of the pull down permit service is shown in Listing 57. This specification states that the *PullDownPermitService* requires an input variable (in WSMO represented by the *sharedVariable* element) of type *PullDownPermitApplicationRequest* (which, in turn by ODEG-convention is a sub-concept of *ServiceRequest*). Furthermore, this request element has to contain a property of name *pulldownbuilding* (representing the type of building that is supposed to be knocked down) which has to hold a value of one of the listed types. Thus this service is only eligible for some particular types of buildings.

Since the specification of one public service is scattered over two elements, the question arises how these elements are related. The answer lies in specific annotations that are used with every single specification element. The GEA part of the specification refers to its corresponding WSMO part via an annotation named *http://www.semantic-gov.org#hasWsmoService* where as the WSMO part refers to the GEA element via the annotation *http://www.semantic-gov.org#geaInstance*. Both elements are results of the afore-mentioned integration efforts.

Beside classical transactional services that represent procedures of public agencies ODEG has introduced a special type of constrained services, the so called information services. As the name indicates, these services represent access to information resources like web pages with background information, information about contact persons or electronic brochures. Technically an information service is a sub-concept of a constrained service (see Listing 58). Thus it can be mapped to desires and service constraints just like any other transactional services. This allows for services that provide very specific information for particular situations, like highly customised electronic information brochures that only contain facts relevant for a

21 http://www.wsmostudio.org/

citizens desire. The major difference between an information and a transactional service is the fact, that an information service does not need any back-office processing, hence, its execution does not change the state of the world.

```
concept InformationService subConceptOf geaGraz#ConstrainedPublicService
   nonFunctionalProperties
    dc#description hasValue "super concept for all information services"
   endNonFunctionalProperties
```

Listing 58: Definition of the information service concept.

7.3.4 PersonData Ontology

The `PersonData` ontology is a central general purpose ontology within ODEG. Although there already exist several ontologies that describe persons and their common attributes like the PROTON top module or the friend of a friend (FOAF) ontology[185] this ontology defines different types of persons as they might be needed within the E-Government domain.

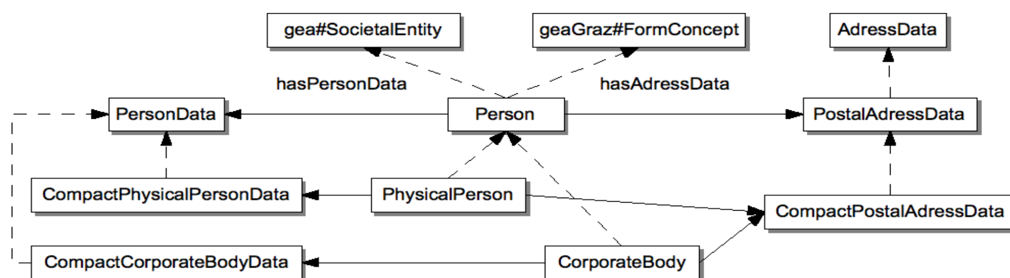


Figure 53: Main concepts and structure of the PersonData ontology (own illustration)

Thus, `PersonData` represents a domain specific specialisation. On the one hand side it was motivated by the idea to facilitate conversion into the EDIAKT II format whereas on the other side EDIAKT II was considered to contain a valid general conceptualisation since it was designed to exchange procedural data between public agencies. Consequently the `PersonData` ontology reflects the structure of the respective part of the EDIAKT II specification, although this might introduce some redundancy.

Figure 53 shows the structure of the top level concepts that make up the `PersonData` ontology. Basically a person is described by a composition of a so called `PersonData` and a `PostalAdressData` concept. The first one describes identifying attributes like a person's name (see Listing 59 for the definition of person data concepts) whereas the latter one holds contact information in form of a postal address (see Listing 60).

```
concept CompactPhysicalPersonData subConceptOf PersonData
    prefix ofType (0 *) AcademicDegreePrefix
    givenName ofType (1 *) _string
    familyName ofType (1 1) _string
    suffix ofType (0 *) AcademicDegreeSuffix
    maritalStatus ofType (0 1) MaritalStatus
    sex ofType (1 1) Gender
    dateOfBirth ofType (1 1) _date
    placeOfBirth ofType (0 1) _string
    iSOCode3 ofType (1 1) _string
    telephoneNumber ofType (0 1) _string
    mobileNumber ofType (0 1) _string
    faxNumber ofType (0 1) _string
    eMailAdress ofType (1 1) _string

concept CompactCorporateBodyData subConceptOf PersonData
    fullName ofType (1 1) _string
    legalForm ofType (0 1) _string
    //organization ofType (0 1) _string
    telephoneNumber ofType (0 1) _string
    mobileNumber ofType (0 1) _string
    faxNumber ofType (0 1) _string
    eMailAdress ofType (1 1) _string
```

Listing 59: Definition of the two concrete PersonData sub-concepts

```
concept CompactPostalAdressData subConceptOf PostalAdressData
    nonFunctionalProperties
        dc#description hasValue "concept for address of houses and societal entities"
    endNonFunctionalProperties
    countryCode ofType (0 1) _string
    countryName ofType (0 1) _string
    postalCode ofType (0 1) _string
    municipality ofType (1 1) _string
    streetName ofType (1 1) _string
    buildingNumber ofType (1 1) _string
    unit ofType (0 1) _string
    doorNumber ofType (0 1) _string
```

Listing 60: Definition of the CompactPostalAdressData concept

Just like in the PROTON top module and WSMO-PA (compare Figure 47) also the PersonData ontology distinguishes between physical persons and corporate bodies, although both types of person share the same `PostalAdressData` concept.

7.4 Service Locator

Analysis of the building permit domain showed that even identifying the correct procedure that is needed in a particular situation is a non-trivial task. For example, to get permission for erecting some new construction the Styrian building law defines three different procedures with different internal complexity and therefore duration:

- **Building development requiring official approval**: In this case you have to apply for approval which will trigger a fairly complex process. If successfully approved, the public agency in charge will issue a building permit at the end of this process.
- **Notifiable building development**: In this case you have to notify the responsible public agency about the project, providing detailed information and blueprints. The agency can prohibit the project within six weeks. Otherwise approval is considered to be granted.
- **Building development not requiring official approval**: In this case you just have to inform the responsible public agency about when construction work will start and provide some basic information about the project.

Which of these procedures is the relevant one mainly depends on the type as well as on the size of the building or facility that is going to be erected. One example that illustrates this is the erection of a garage. Whether an application for a full-blown building permit or a relatively light-weight notification is needed depends on the size of the garage. The size of a garage, however, is not determined directly by its physical extent but implicitly by the type and number of vehicles that can be parked in the garage. Similar rules apply to a variety of other construction types. Due to this inherent complexity it is not easy for citizens to find out the appropriate service. Therefore clients need support by the system to identify those services that are relevant to their specific situation. In this context the `Desire` concept is used to capture information about a citizen's goal at a level detailed enough to decide which services are needed. Typically a citizen's primary intention or desire is not to get a building permit in the first place, which might not be necessary for the given situation anyway, but to erect some particular type of building. Applicants just wants to be sure that they are allowed to build whatever they intend. Thus a typical desire might be "*I want to erect a garage*" rather than "*I want to get a building permit*". This introduces a citizen centric point of view when it comes to goal/desire definitions.
However, as already pointed out at the beginning of this section, a goal like "*I want to erect a garage*" would not contain the necessary details required to identify the relevant service, since this decision depends on the size of the garage as well. To derive a more concrete specification of a citizen's desire, every desire can be related to an arbitrary number of other concepts as shown in Figure 54.

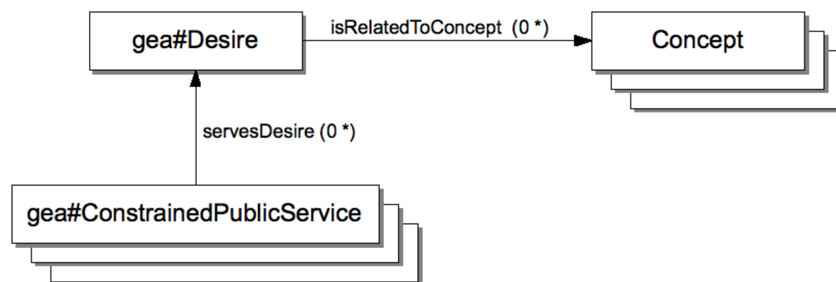


Figure 54: The Desire concept and its related concepts (own illustration)

Every desire, which technically is an instance of the concept `Desire`, can be linked to those types of concepts that are relevant for the decision about the required or appropriate services. In the case of a building permit this could be the type of the building that is going to be erected. Additionally it is important to know where the construction site is located since this determines which public agency is responsible for handling the procedure.

To illustrate how all of these elements fit together lets proceed with the relatively simple example of the pull down permit service that was already used in section 7.3.3. Figure 55 shows how the actual public service is related to a desire and a service constraint. The meta-model defines the concepts `ContraintsPublicService`, `Desire` and `ServiceConstraint` as well as their relations. The shapes with rounded corners in Figure 55 represent instances, whereas rectangles stand for concepts. The `GeaPullDownPermitService` supports the `PullDownAConstruction` desire. This desire is in turn related to the concept `PullDownRelevant` and `PreliminaryConstructionAddress`, which indicates that this goal is only sufficiently specified when it comes together with concepts of these types. `PullDownRelevant` represents things in the knowledge space that can be knocked down. A `PrelimaryConstructionAddress` describes the location of the project. It is different from the `PostalAddress` concept presented in section 7.3.4 and captures the facts that for the location of building projects there sometimes does not exist a street number yet and that a detailed description of the location is not necessary during service discovery phase anyway. In fact in this example it is only necessary to find out in which community the construction or pull down activity takes place and – in case of a larger city like Graz - in which borough.

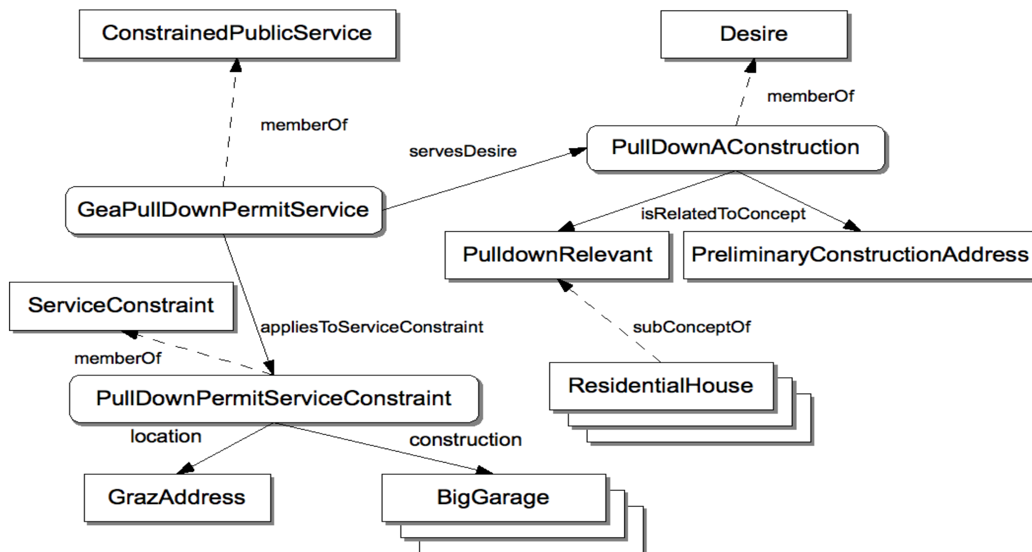


Figure 55: Definition of the pull down permit service and its relations to a desire and its service constraint (own illustration)

7.4.1 Selecting a Desire

From a citizen's point of view, service discovery should start with selecting the appropriate desire. Therefore the start page (see Figure 56) of the service locator provides an overview of all registered desires, i.e. all instances of the `Desire` concept. These instances are found by querying a reasoner that has registered all available ontologies.

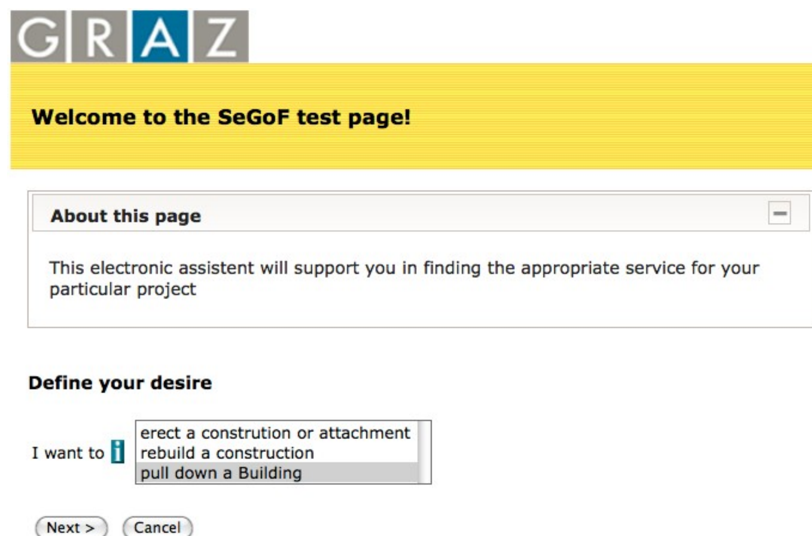


Figure 56: Start page of the service locator showing a selection of available desires (screen shot of the service finder application).

The user can now select the desire that reflects his or her situation best. To translate the semantic model into human readable form, so called resource bundles[22] are used. Basically a resource bundle in this context is nothing but a set of text files that contain key/value pairs.

22 http://java.sun.com/developer/technicalArticles/Intl/ResourceBundles/

Listing 61 shows the resource bundle entries necessary to create the dialog for the selected desire in Figure 56. Resource bundles are typically used for localisation and provide text in different languages. Thus, the usage of resource bundles allows for simple support of internationalization as well.

```
[from English resource bundle]
http.//segof.fh-joanneum.at/GEA.PullDownAConstruction = pull down a {1}
http.//segof.fh-joanneum.at/Construction.PullDownRelevant = Building
http.//segof.fh-joanneum.at/PersonData.PreliminaryConstructionAddress = Construction location

[from German resource bundle]
http.//segof.fh-joanneum.at/GEA.PullDownAConstruction = {0} {1} abbrechen
http.//segof.fh-joanneum.at/Construction.PullDownRelevant = Gebäude
http.//segof.fh-joanneum.at/PersonData.PreliminaryConstructionAddress = Wo werden Sie voraussichtlich
Ihr Bauvorhaben umsetzen?

[from ODEG ontologies]

instance PullDownAConstruction memberOf geaGraz#ConstrainedDesire
   annotations
      dc#description hasValue "Citizen wants to pull down a construction"
      segofUtil#displayPriority hasValue "3"
   endAnnotations
   isRelatedToConcept hasValue {_"http://segof.fh-joanneum.at/Construction#PullDownRelevant",
                  _"http://segof.fh-joanneum.at/PersonData#PreliminaryConstructionAddress"}

concept PullDownRelevant
   annotations
      dc#description hasValue "abstract concept which marks construction that are relevant for pull down
                  application"
      gender hasValue segofUtil#Neuter
   endAnnotations
```

Listing 61: English and German snippets from the resource bundles used together with semantic model elements relevant for the desire selection.

To map elements of the semantic model to the appropriate text in a resource bundle, the element's IRI, which is its globally unique identifier, is used. However, since the colon that is part of each IRI is used as a key/value separator and the hash character is used to indicate comments in resource bundles, some simple character substitutions have to be performed. Thus, the IRI of a model element is equal to its key in the resource bundle, except for colons and hash characters, which are replaced by dots.
To construct the text for the user dialogue, first the text of the corresponding bundle entry, depending on the currently selected language is retrieved ("*pull down a {1}*" in this example). This text might contain placeholders for parameters (indicated by the curly brackets). These placeholders are replaced with the text values of the related concepts. In this example, the run-time system tries to insert the text for `PullDownRelevant` ("*Building*") and `PreliminaryConstructionAddress` ("*Construction location*"). If there are more related concepts than placeholders, these concepts are ignored during the creation of the text. Otherwise, if there are more placeholders than related concepts, an error will be produced. Like shown in Figure 56 the resulting text for the `PullDownAConstruction` desire will be "*pull down a Building*". Generally, for every model element a corresponding entry in the resource bundles is created automatically, which defaults to its local name, i.e. its name without the namespace. This makes sure that there are no missing bundle entries.

In the case of the German version, things are slightly more difficult, since the German language uses different definite and indefinite articles depending on a noun's gender. Thus, the system has to know the gender of a noun in order to determine the required article. This information is contained in a concepts annotation section, wherever it is required. The concept `PullDownRelevant` has a gender annotation, which refers to the value `segofUtils#Neuter`. Thus, in the case of a German version also the text for the desire instance would be retrieved first ("`{0} {1} abbrechen`" in this example). The placeholder with the index zero is always reserved for the article. From the context, the run-time environment knows that it has to use the appropriate indefinite article, which is "`ein`" in this case. Together with the text for the related concept `PullDownRelevant` ("*Gebäude*"), the resulting text for the desire would be "*ein Gebäude abreissen*". This

approach makes sure that always a correct sentence is created for any concept a desire might be related to.

Another aspect that should be mentioned is the *segofUtil#displayPriority* annotation that is used for the *PullDownAConstruction* desire shown in Listing 61. Whenever there is a list of elements a user can choose from, these elements are sorted alphabetically. Sometimes, however, it seems to be more appropriate to use a different order, e.g. determined by the frequency or likelihood certain elements might be needed. In this case the display priority property can be used to explicitly create a different order. This explains why the "*pull down a Building*" desire comes third in the dialogue presented in Figure 56.

7.4.2 Refining a Desire

Desires are typically modelled as abstract and general as possible. In the example of the pull down permit services the corresponding desire is related to a concept called *PullDownRelevant*, which represents all possible types of constructions that can be knocked down and a location. Thus, this desire does not contain any information about the actual building that should be removed. As already pointed out at the beginning of section 7.4 this level of abstraction typically does not allow for selecting the required service. Consequently the desire of a citizen who wants to pull down a particular building has to be refined to the necessary detail in order to determine the appropriate service. In this phase the domain specific ontologies become important. Concepts in these ontologies form graphs along the level of abstraction and therefore define taxonomies.

Figure 57 shows some of the concepts that are part of the domain specific construction ontology. One central concept in this ontology is *Construction*, which stands for all things that can be erected and are therefore covered by the construction law that applies. This ontology, however, also contains more specific types of constructions like fence of garage. The identification of all of these concepts was done by a careful analysis of the Styrian Construction Law, which is the legal basis of all building permit procedures. Whenever the law text was referring to a particular type of construction, it was added to the domain. Later on these terms were re-arranged and classified leading to a taxonomy of buildings and facilities. Although there exist some approaches to automatically extract semantic information from law texts [186][187], this analysis was conducted manually to have full control over the creation process. During interviews with domain experts where the resulting taxonomies were discussed, some of the concepts identified during text analysis were dropped, since – according to the experts – they had no practical relevance for the procedures.

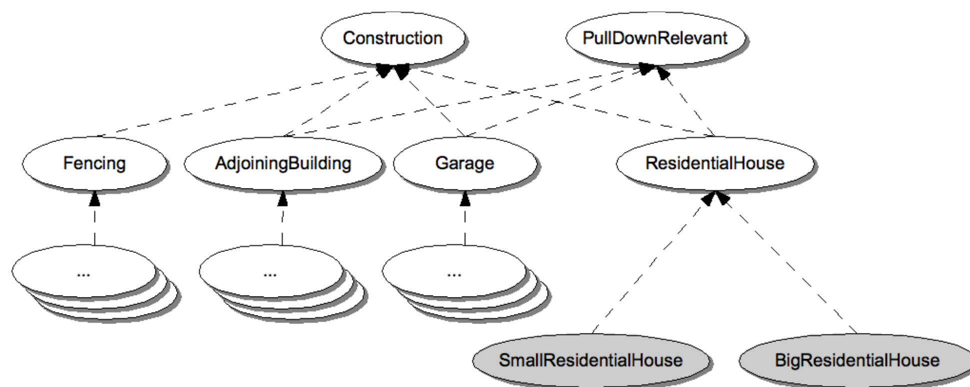


Figure 57: A fragment of the construction ontology showing parts of the construction taxonomy (own illustration)

Thus, the initial version of the construction ontology contained more elements and abstraction layers than it does now. On the other side, concepts like *PullDownRelevant* were not identified in the text analysis phase, but during the analysis of the different procedures. Whereas all concepts shown in Figure 57 are constructions, not all of them are relevant for pull down permit procedures according to the construction law. Although this is not explicitly defined in the law, this became obvious in the review process with the domain

experts. Thus, the additional concept `PullDownRelevant` marks all its instances as relevant for pull down permit services. It therefore defines a subset of all constructions.

ODEG makes one very important assumption about concept hierarchies in general. Every concept that possesses sub-concepts is considered to be abstract, whereas all concepts that are leafs of the type hierarchy graph are considered to be concrete. Furthermore only concrete concepts are allowed to appear in any of the procedures. This intuitively reflects the fact that one has to apply for permission to erect or pull-down a concrete type of building rather than “a building”. Thus refining a desire merely means replacing every related concept which is abstract by any of its concrete sub-concepts.

GRAZ

Welcome to the SeGoF test page!

You want to pull down a Building, we will support you.

Which type of "Building" is it? *

Residential house
Businesshouse
Building for business and residential use
Garage
Adjoining building
Other construction

< Back
Next >
Cancel

Figure 58: Specialisation as one way to refine a desire (screen shot from the service finder application)

One way how this can be accomplished is shown in Figure 58. This dialogue will be shown once you click the next button in the desire selection dialogue (see Figure 56) assuming that the “pull down a Building” option was selected. It presents all direct sub-concepts of the initial `PullDownRelevant` concept that was listed as the first related concept of the `PullDownAConstruction` desire (shown in Listing 61).

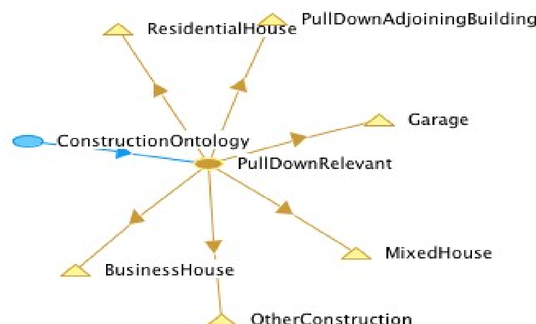


Figure 59: The PullDownRelevent concept and its direct sub-concept (screen shot of the WSMO Visualizer)

These sub-concepts can easily be determined by querying the reasoner (compare Figure 59) and are rendered as radio-button options. Thus the current user has to determine the next more specific type of the building that is about to be torn down. This activity is called specialisation, since the user of the system adds more specific information about the concepts that have to be dealt with.

On the other side, sometimes additional levels of specialisation are necessary due to specific needs of the

underlying regulations. These additional classes are not always intuitive or easily comprehensible for citizens. The Styrian Construction Law for example, explicitly distinguishes between small and other residential houses (see Figure 57). According to this law, a residential house is considered to be small when it does not possess more then three floors and its total floor space is bellow $600m^2$. This distinction is important, since this fact might decide whether a simplified permit procedure is possible or not. The simple straight specialisation approach where the user has to select the appropriate sub-concept type like shown in Figure 58 would lead to usability problems if applied to this type of decision as well. In fact significant amount of additional information had to be shown to the citizen in order to support the decision about whether the house is small or not according to the definition of the regulation. However, semantic reasoners can easily decide, whether a given instance belongs to a specific class of not, by applying an ontology's axioms.

ODEG makes use of these reasoning capabilities here and provides appropriate axioms that exactly reflect the specifications of the law. Listing 62 presents the axiom that specifies whether a given instance is of type `SmallResidentialHouse`. The head of the axiom defines the consequence, which states that the instance represented by the variable `x` is a member (i.e. an instance) of `SmallResidentialHouse`.

```
axiom SmallResidentialHouseDefinition
    definedBy
        ?x memberOf SmallResidentialHouse
:-
?x[effectiveArea hasValue ?effectiveArea, numberFloors hasValue ?numberFloors,
hasNeighbourSignatures hasValue ?hasNeighbourSignatures] memberOf ResidentialHouse
 and wsml#equal(_boolean("true"), ?hasNeighbourSignatures)
 and ?effectiveArea < 600
 and ?numberFloors =< 3.
```

Listing 62: Axiom that defines whether a given instance is of type SmallResidentialHouse or not

The body of the axiom defines the condition that has to hold true for the head to become effective. One important restriction here is that the instance represented by the variable x already has to be a member of the concept `ResidentialHouse` to be further analysed. This improves the performance of the entailment process since far less combinations have to be investigated by the reasoner compared to an unrestricted variable. Besides this condition the effective area of the residential house has to be smaller than $600m^2$ and it must not have more than three floors. There is another property called `hasNeighbourSignatures`, which is of type boolean. This property captures the fact that the small nature of a residential house only leads to simplified procedures if all neighbours explicitly express their approval of the project by signing the blueprints. Thus, if this approval is missing the physically small residential house is treaded like it was bigger. However, to use a reasoner for making the decision about a residential house's concrete class, it has to be fed with an instance holding the relevant information. Therefore the ODED run-time checks for every single refinement step whether there exist axioms that can be used for automatic classification. This search is a two step activity. First all direct sub-concepts of the current concept in question are determined. This information is also needed for manual specialisation. Then the system checks for axioms, which classify instances as a member of any of these previously identified sub-concepts and take variables of the appropriate type as input. The input type is considered appropriate, if the current concept is a member of it. This means that the input variable of the axiom is either the same type as the current concept or one of its super-concepts.

If such axioms where found, the system collects all the properties that are referenced in these axioms and it creates a dialogue in which the user is asked to provide values for these properties. Assuming that the user selects the option “Residential House” in the dialogue shown in Figure 58 the system first would find out that there are two more sub-concepts for the currently selected `ResidentialHouse` concept. Thus refinement has to go on since according to ODEG's assumptions `ResidentialHouse` is abstract and needs to be replaced by one of its concrete sub-concept. In a consecutive step the system now checks for axioms. In this case it would find the one presented in Listing 62 and another one that defines `BigResidentialHouse`s. The variables used in these axioms are extracted and added to a set of properties that are needed for automatic reasoning. This set of properties is used to dynamically render the dialogue shown in Figure 60.

GRAZ

Welcome to the SeGoF test page!

You want to pull down a Residential house, we will support you.

Residential house

Neighbours accepted construction project with signature explicitly: * ☑

Effective area (m2): * 150

Number of floors: * 2

< Back Next > Cancel

Figure 60: Dialogue to further specify the type of a residential house (screenshot of the service finder application)

The input elements used in this dialogue depend on the type of the properties. The dialogue also contains information about the current context, including the currently specified desire ("You want to pull down a residential house ..."). The user now has to specify some of the properties of the house that should be removed. Once the user clicks the next button an instance of a residential house with the specified properties is created and registered with the reasoner. The reasoner now applies all the registered axioms, which will lead to the required classification.

Axioms, however, are also used for enforcing the consistency of the instances (e.g. the number of floors of a house has to be bigger than zero, …), which will be explained more precisely in section 7.5. Generally, ODEG refers to the process of refining the type of a concept by using the reasoner (compare Figure 60) as classification, while refinement that is explicitly performed by the current user (compare Figure 58) is called specialisation. After the reasoner has successfully classified the given instance, the system goes on to the next step (see Figure 61). In our example this will bring us to the specification of the desire's next related concept, the `PreliminaryConstructionAddress` (compare Listing 61).

GRAZ

Welcome to the SeGoF test page!

You want to pull down a small residential house, we will support you.

Construction location

Municipality: * Graz

District - Cadastre Community: * Andritz | Graz Stadt-St.Veit

< Back Next > Cancel

Figure 61: Specifying the location of the pull down activity (screen shot of the service finder application)

This dialogue immediately uses classification, which is caused by the existence of several sub-concepts of `PreliminaryConstructionAddress` that can be inferred using axioms. One of these sub-concepts is the class of all locations within the City of Graz.
The specification of this group of addresses is called `GrazAddress` and is presented in Listing 63. This type is also used in the `PullDownPermitServiceConstraint` shown in Figure 55 on page 117.

```
axiom GrazAddressDefinition
nonFunctionalProperties
    dc#description hasValue "define a location in Graz"
endNonFunctionalProperties
    definedBy
        ?x memberOf GrazAddress
:-
?x[municipality hasValue ?municipality, districtCadastre hasValue ?districtCadastre] memberOf
PreliminaryConstructionAddress
    and (wsml#equal("Graz", ?municipality)
    or wsml#equal("graz", ?municipality)).
```

Listing 63: Axiom to classify all addresses that are located in Graz

The dialogue presented in Figure 61 shows the result of the previously performed automatic classification (“You want to pull down a small residential house ...”). It also contains two peculiarities. The field for the municipality is disabled and already contains the value “Graz”. On the other hand, the field for the district is a pull-down list with predefined values. How such a behaviour can be defined is described in sections 7.5.3 and 7.6 respectively. After values for the location are provided, the reasoner will classify the address.

Now, since all the desire's related concepts are thoroughly specified, the actual service identification takes place. The algorithm used to identify relevant services is rather straight forward. First of all, all services that generally serve the user's desire, i.e. all desires that have the selected desire as a value of their

Figure 62: Result of the service finding process (screen shot of the service finder application)

`servesDesire` property are identified. From these services those are extracted that have matching service constraints associated with them. A service constraint matches the current desire when all of its properties are of the same type as the desire's related concepts. More precisely, there has to be one related concept specified with the desire that is of the same type as the property of a service constraints for all properties of

the constraints. This also matches service constraints that only have a sub-set of the desire's related concepts defined as properties or their properties are of a super-type of the desire's related concept. This provides a great degree of freedom to map services to more or less specific desires. Generally the result of the match can be empty (which would indicate an incomplete model) or can consist of one or more services.

Service identification includes transactional public services as well as information services. The result of our pull down permit example is shown in Figure 62. Based on the data that was filled into the different forms, the system has identified the “Pull down permit request” service as the appropriate one. Additionally an information service that will show the contact information of the responsible contact person was found as well. This information service also serves the `PullDownAConstruction` desire and has a matching service constraint. In this example the responsible person is found, based on the district in which the activity will take place. Next to the bottom of the result dialogue there is a section showing additional services that might be of relevance. Services listed here are all services that serve the same (abstract) desire but do not have matching service constraints associated with them.

```
instance BuildingAConstruction memberOf geaGraz#ConstrainedDesire
    nonFunctionalProperties
        dc#description hasValue "Citizen wants to errect a construction"
        segofUtil#displayPriority hasValue "1"
    endNonFunctionalProperties
    isRelatedToConcept hasValue {
        _"http://segof.fh-joanneum.at/Construction#ConstructionProjectNewbuild",
        _"http://segof.fh-joanneum.at/PersonData#PreliminaryConstructionAddress"}
     relatedConceptCardinality hasValue {"*","1"}
```

Listing 64: Definition of the BuildingAConstruction desire

The example of the pull down permit service used here to illustrate the service identification process is relatively simple. Service identification can be also be a bit more complex as it is the case with the building permit service. The difference is that, although, the type of the required service merely depends on the type of the construction that is about to be erected, citizens can apply for permission to erect several constructions in one single application. A relatively frequent case is that people want to build a residential house but also want to erect a garage. Sometimes the level of the terrain is also changed, which needs approval as well. Thus, several construction types might occur within one desire.

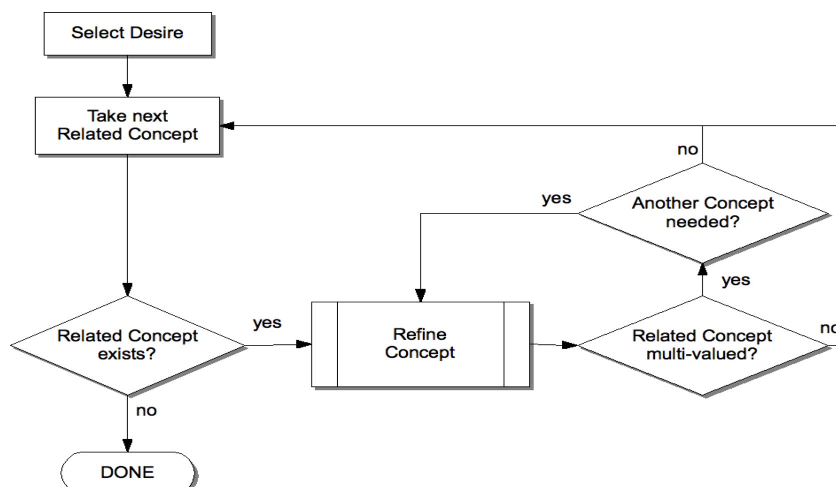


Figure 63: General overview of the desire refinement process (own illustration)

To support a situation like this, ODEG has introduced multi-valued related concepts. However, since the cardinality of the `isRelatedToConcept` property already has an unbound cardinality, this is based on some convention. Technically, if a related concept should by multi-valued, it requires the desire to provide as second property called `relatedConceptCardinality`. Every value of this property represents the corresponding related concept's upper bound. The relation between the values of the cardinality and the

related concept property of a desire is the order of their appearance. An example is given in Listing 64. This desire is related to two different concepts called `ConstructionProjectNewbuild` and `PreliminaryConstructionAddress`. The former, however, is a multi-valued property indicated by the asterisk, which is the first element of the `relatedConceptCardinality` property.
If a related property is multivalued, the user can add additional instances of this concept to further specify the current desire. The complete control flow used in the desire refinement phase is shown in Figure 63.

7.4.3 The Service Finding Algorithm

Although the principle of the service finding algorithm was already presented in the previous section, here all the details and potential scenarios will be discussed.

As already pointed out, appropriate services are found by matching the concrete concepts of a given desire with the types of a service-constraint's properties. Before this can be done, in a first step all the potentially abstract concepts a desire might be related to have to be replaced by their concrete sub-concepts. A concept is considered to be abstract, if there exist any sub-concepts. This refinement process was extensively described in section 7.4.2.

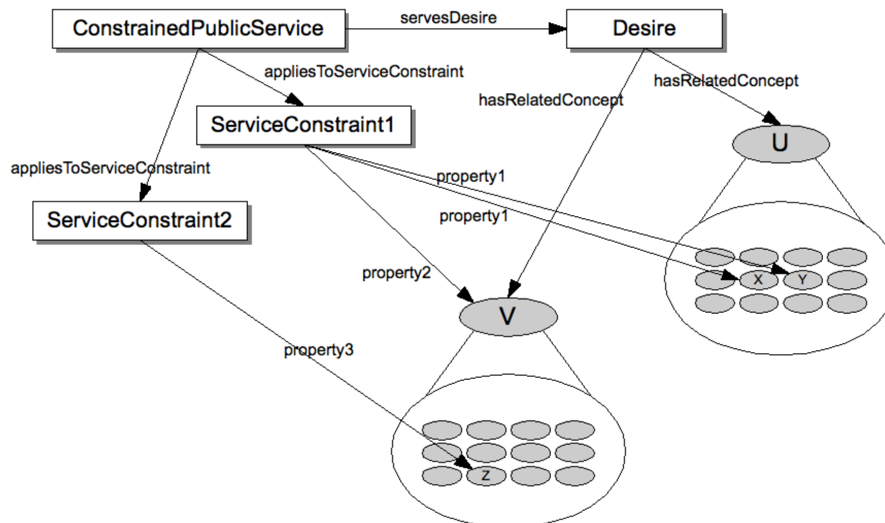


Figure 64: Schematic overview of the service matching step when looking up relevant services (own illustration)

To narrow the number of services that have to checked, in a second step only those that serve the given desire are selected. A schematic overview of how a service, its desires and its service constraints are related to each other in order to find relevant services is shown in Figure 64. Basically every single `ConstrainedPublicService` can serve an arbitrary number of desires but has to be related to at least one. Consequently, different desires can lead to the same service, which is for example the case for the contact person information service found in our pull down permit example presented in the previous section. The same information service will be found in case of a building permit application since it contributes to all desires that are related to the building domain.

Every service that generally serves the selected desire is now further analysed. Each of these services can also be related to an unlimited number of service constraints. If there are several service constraints, they are combined using logical disjunction. Thus if one of these constraints matches the desire then the service is relevant. Consequently every service constraint is checked if all of its properties are of the type of the now concrete related concepts of the desire. This condition is met if there is one concrete concept in the desires set of related concepts that is either exactly of the same type as the constraint's property or a subtype. If a constraint's property contains several values (like `property1` in Figure 64) only one of them has to match. Thus, individual values of a multi-valued property also form a disjunction. On the other side, if a service

constraint possesses several properties (compare *property1* and *property2* of *ServiceConstraint1* in Figure 64), all of them have to match, which represents a conjunction. Taking the example depicted in Figure 64, the service would be relevant in the following cases:

1.) The selected desire has a related concept that was replaced by X **or** Y **and** another related concept that was replaced by any sub-concept of V
or
2.) The selected desire has a related concept that was replaced by Z.

In the first case the shown desire automatically meets this constraint, since V is the type of the original related concept. This situation could also be described by a predicate logic term, where the predicates require a variable to be of a particular type:

$$[\, \exists\, s,t \mid (\, isOfTypeX(s) \vee isOfTypeY(s)\,) \wedge isOfTypeV(t)\,] \vee [\, \exists\, t \mid isOfTypeZ(u)\,] \Rightarrow requiresService(v,s,t)$$

Thus, whether the service *v* is required for the given situation or not depends on the types of the related concepts represented by *s* and *t*. Generally, any combination of dis- and conjunctions is possible, which allows for describing virtually any logical constraint. Listing 65 shows a service constraint that is used together with the pull down notification service, a simplified procedure for obtaining permission to knock down a building. This constraint matches a particular desire if it is related to one of the construction types assigned to its construction property (disjunction) and to a location of type *GrazAddress*.

```
instance PullDownNotificationServiceConstraint memberOf geaGraz#ServiceConstraint
    construction hasValue {_"http://segof.fh-joanneum.at/Construction#SmallToolShed",
    _"http://segof.fh-joanneum.at/Construction#SmallAdjoiningBuildingForAgricultureAndForestry",
    _"http://segof.fh-joanneum.at/Construction#SmallPullDownAdjoiningBuilding"}
    location hasValue _"http://segof.fh-joanneum.at/PersonData#GrazAddress"
```

Listing 65: Service constraint for the pull down notification service

It is important to notice that this algorithm can't be expressed entirely via axioms that are fed to a reasoner. For example, references from instances to concepts are modelled as properties of type string that contain the IRIs of the referenced concepts. This information can only be interpreted correctly by knowing the underlying conventions. Thus, significant parts of this algorithm are implemented in software, whereas the reasoner is used to answer all the queries needed to look up instances and identifying class-hierarchies. When comparing this algorithm to those discussed in section 4.5.1 resulting matching states are either plugin or exact in the case of a service found and disjoint when no service was found.

7.5 Semantic Forms

In the previous section the service identification mechanism was presented that allows for finding appropriate services necessary to fulfil a specific desire. In the upcoming sections the process of the actual service utilisation is described. This comprises elicitation of all the information required by the service that will be used. During this phase citizens are working with electronic forms that are interactively rendered based on the underlying ontologies and the currently selected concepts. Thus, it is important to determine which information is necessary to invoke a particular electronic public service.

7.5.1 Determining Required Service Input

Due to some compatibility considerations pointed out in section 7.3.3 every service description is split into two parts: a GEA inspired instance of a *ConstrainedPublicService* and a corresponding WSMO *webService* element. Whereas the GEA part holds a general description of the service, the desires it might fulfil and the provider of the service, the WSMO part describes the required input and preconditions as shown in Listing 66. In this example – that will be used as a running example to demonstrate the semantic form generation

process - there has to exist an input variable of type *BuildingPermitApplicationRequest* for which one additional constraint has to hold true.

```
webService BuildingPermitService
   annotations
      _"http://www.semantic-gov.org#geaInstance" hasValue GeaBuildingPermitService
   endAnnotations

capability Capability_BuildingPermitService

    sharedVariables {?request}

    precondition
        definedBy
            ?request[construction#constructionProject hasValue ?projectType] memberOf
            construction#BuildingPermitApplicationRequest
            and ?projectType memberOf construction#BuildingPermitConstructionProject.
```

Listing 66: WSMO description of the building permit service.

Thus, once a service is selected, the shared variables of the corresponding WSMO *webService* element are analysed, which form the starting point of the interactive form creation process. The concept definition of the input variable type used in this example is shown in Listing 67. According to this concept, whenever someone wants to apply for a building permit, the following information needs to be provided:

1. At least one or more applicants have to be named
2. A potential delegate who represents the applicant(s) throughout the procedure can be specified
3. A short description of the construction project consisting of all the facilities the will be erected or remodelled
4. The location of the project
5. A proof that the piece of land used for the project is ready to be connected to water and electricity supply networks as well as sanitation. This can be done by either uploading the required confirmation, which is in turn the outcome of a separate procedure, or by applying for this confirmation as part of the building permit procedure
6. An optional set of blueprints and other documents that can be attached to the application

```
concept BuildingPermitApplicationRequest subConceptOf ConstructionServiceRequest
    annotations
        dc#description hasValue "concept representing input to building permit service"
    endAnnotations
    applicant ofType (1 *) personData#Person
    delegate ofType (0 1) personData#Person
    constructionProject ofType (1 *) ConstructionProject
    buildingLocation ofType (1 1) BuildingLocation
    buildingSiteEligibility ofType (0 1) BuildingSiteEligibilityPlaceHolder
    record ofType (0 *) segofUtil#File
```

Listing 67: Definition of the BuildingPermitApplicationRequest concept that describes the required input to the building permit service

Although applying for a building permit is considered to be one of the most complex procedures at municipal level the formal description of the required input is rather short and simple as demonstrated in Listing 67. This stems from the fact, that the semantic model allows the use of abstractions, where as the concrete types (e.g. which type of building is actually erected) are determined during the information elicitation process.

Building permit service

Building permit (§ 19 Styrian construction law)

Attention: ✱ Mandatory field | Hint available | ! Error hint

Building permit service

Applicant:	✱	< no input data available >	Applicant (add)
Delegate:		< no input data available >	Delegate (add)
Project description:	✱	< no input data available >	Project description (add)
Building location:	✱	< no input data available >	Building location (add)
Building site eligibility:		< no input data available >	Building site eligibility (add)
Supplement:		< no input data available >	Supplement (add)

< Back | Next > | Cancel application

Load input data | Save input data

Figure 65: Initial screen of the building permit application without pre-filled instances.

7.5.2 Rendering the Electronic Forms

All forms presented by the semantic forms component are dynamically rendered based on the currently selected concept. When starting a new application the first – and in most cases the only - input concept as defined in the service's `webService` element is the currently selected one. Thus this concept defines the initial form as it is shown in Figure 65. If the form was started after using the service finder, any information that was provided during the preceding phase is automatically added to the information space of the application as shown in Figure 66.

Building announcement

Building announcement (§ 20 Z1, Styrian construction law)

Attention: ✱ Mandatory field | Hint available | ! Error hint

Building announcement

Applicant:	✱	< no input data available >	Applicant (add)
Delegate:		< no input data available >	Delegate (add)
Project description:		Construction specification: ... Type of the project: ...	Project description (edit) (delete)
Project description:		< no input data available >	Project description (add)
Building location:	✱	< no input data available >	Building location (add)
Building site eligibility:		< no input data available >	Building site eligibility (add)
Supplement:		< no input data available >	Supplement (add)

< Back | Next > | Cancel application

Load input data | Save input data

Figure 66: Initial building permit application form with data transferred from the service finder component

The fields of this form directly reflect the structure of the `BuildingPermitApplicationRequest` concept. Any property that is defined with a minimum cardinality greater than zero (compare the properties `applicant`, `constructionProject` and `buildingLocation` from Listing 67) is rendered as a mandatory field. For properties with a cardinality greater than one a series of property instances can be created as long as the upper bound

is not reached. The form's layout follows the recommendation of the Austrian Style-guide for Electronic Forms[188] and is fully style guide compliant.

The form creation algorithm is based on the following simple considerations:

- Internally every concept is seen as a tree.
- Properties that are of a primitive datatype (e.g. string, number and date) are leafs, for which values can be directly provided.
- Every property that is itself a concept is seen as the root of a sub-tree

Besides these assumptions also the general ODEG convention, defining all non-leaf concepts as being abstract is considered by the algorithm. Thus the current concept must not have any sub-concepts before the user can provide values for any of its primitive typed properties. Nevertheless, the entire data-structure defining the information needed by the service is a sub-graph of the underlying ontologies that is not necessarily a tree, but every walk from its initial node – due to continuous refinement either by going from a more general to a more specific concept or by following a concept's attributes – will eventually reach a leaf. These two dimensions of refinement will become clearer with the ongoing example. Since the data-structure behaves like a tree – in fact the only difference to a tree is that any two branches might merge into one node again – it is very well suited for a recursive approach.

Figure 67: Specialisation of the person concept assigned to the application property

Generally the user is free to start with the specification of the current concept's properties in any order. In our example, however, the first property on the form – the applicant – is further specified first. The type of this property is defined as *personData#Person*. As pointed out in section 7.3.4 this type is abstract and subsumes the specific types physical person and corporate body. Thus, when adding a new applicant to the information space, the abstract type person has to be replaced by a more specific type in a first step. ODEG supports the two different approaches called specialisation and classification as already discussed in section 7.4.2. In this case, specialisation is used, which basically means that the user has to select the appropriate type like shown in Figure 67.

Figure 68: Form used to collect information about the applicant

This is an example of the classification/specialisation refinement dimension mentioned earlier. After the use

selected one of the available sub-concepts was, a new instance of this concept – in our example a physical person – is created and the properties of this concept – now the current one – have to defined. At this point the same algorithm that was used for the specification of the initial application concept is applied recursively. Therefore the current implementation of ODEG makes use of Spring Web Flow [189]. This web application framework allows to arrange page and logic sequences in so called flows. In turn, these flows can be recursively called as sub-flows. Thus, one and the same logic is going to be applied over and over again as long as a leaf of a particular branch is reached. But for now, the properties of the applicant, who is a natural person in this example, have to be further specified (see Figure 68).

Building permit service

Building permit (§ 19 Styrian construction law)

Attention: * Mandatory field | Hint available | ! Error hint

Building permit service > Applicant > Person data

Personal data

Academic degree prefix:		Please select	(add)
First name:	*	John	(add)
Surname:	*	Doe	
Academic degree suffix:		BSc	(add)
Sex:	*	male	
Country code:	*	Austria	
Phone:			
Mobile phone:			
Fax:			
E-Mail address:	*	jd@test.com	

< Back | Next > | Cancel application

Figure 69: Input form for specifying the properties of a physical person applying for a building permit

The separation of a person concept into a person and an address data block was caused by the originally close ties between the form solution and the EDIAKT II data exchange standard as already discussed in section 7.3.4. Both of these properties are concept types, thus, in a first step new instances of these properties have to be created by clicking on the respective button. The form now shows the properties of the `CompactPhysicalPersonData` concept. This concept is a leaf in the concept hierarchy and all its properties are of primitive data types, therefore the user can provide values for them as shown in Figure 69. It is important to mention that the set of properties shown in this page also includes all inherited properties that were specified in any of the current concept's super-concepts. Whenever the next button is clicked a concept instance with the values provided by the user is registered with the reasoner. This causes all constraints and axioms in the ontologies to be checked. If this leads to errors, the user is returned to the form and appropriate error messages are rendered like shown in Figure 70.

Building permit service

Building permit (§ 19 Styrian construction law)

Attention: * Mandatory field | Hint available | ! Error hint

Building permit service > Applicant > Person data

• The field "First name" is mandatory.

Personal data

Academic degree prefix: Please select (add)

First name: * ! (add)

Figure 70: Example of an error message caused by a model constraint that was not met

The look and feel of these error messages follows the specification of the Austrian Style-guide for Electronic Forms. Although the example shown here is a very simple one, constraints can get arbitrarily complex as long as they can be captured by F-Logic body clauses.

Some of these check-constraints are also used by the forms solution when rendering input elements. The form shown in Figure 69 contains several pull-down list elements that contain all possible values for the respective fields. There are several ways to provide such ranges of values as will be explained in the upcoming sections. In this example, however, simple constraint axioms like the one shown in Listing 68 are used to populate the pull-down list elements. WSML constraint axioms do not have a head clause. If the body of a constraint axiom is evaluated to true, the ontology is not longer consistent. Thus, the axiom's body is required to be false. This explains why it tests that the value does not contain any of the required values.

```
concept AcademicDegreeSuffix subConceptOf segofUtil#OneOfEnumerationType
    annotations
        dc#description hasValue "concept for suffix academic degree --> create enumeration box in form"
    endAnnotations
    value ofType (1 1) _string

axiom AcademicDegreeSuffixValues
    definedBy
        !- ?x[value hasValue ?value] memberOf AcademicDegreeSuffix
            and ?value != "BA" and
                ?value != "BSc" and
                ?value != "Bakk." and
                ?value != "MA" and
                ?value != "MSc" and
                ?value != "MAS" and
                ?value != "MBA" and
                ?value != "MIB".
```

Listing 68: Definition of academic degrees used in a suffix notation together with a constraint axiom that checks for allowed values.

The form component tries to render a list of values since the concept `AcademicDegreeSuffix`, which is used as a property type in the `PersonData` concept is a sub-concept of the type `OneOfEnumerationType`. This special concept adds the necessary information for the form component. Similar ways to add form specific information to the model are presented in section 7.5.3. The actual values for the list are then extracted from the constraint axiom. If no list of values was created, the ontology would only accept valid values anyway since the axiom is always evaluated. An academic degree is also a well suited example to demonstrate the form components handling of multi-valued properties. Every academic degree property is optional, which means that the lower bound of this property's cardinality is set to zero. On the other hand, the upper bound of this cardinality is not limited, thus, anyone could have an arbitrarily long list of academic titles. If someone therefore wants to add several degrees, it only takes a click on the add-button to create a new instance as illustrated in Figure 71. Conventional electronic forms solution typically offer a variety of possible combinations of degrees. ODEG, however, uses the list of degrees like defined in Listing 68 and allows the

user to combine these degrees in arbitrary order and number.

First name: * John (add)
Surname: * Doe
Academic degree suffix: BSc (delete)
Academic degree suffix: MBA (delete) (add)

Figure 71: Example of registering several degrees

As already explained before, when the user clicks on the next-button the correctness of the current concept instance is checked using a semantic reasoner. If this check is successful the user is sent to the next form, which in our example is the concept possessing the now completed person data concept as an attribute. This brings us back to the applicant. The data that was collected in the previous step is now displayed in this form as well and a little green checkmark next to the person data property label indicates that this block has already passed validation (see Figure 72).

Building permit service
Building permit (§ 19 Styrian construction law)
Attention: * Mandatory field Hint available ! Error hint
Building permit service > Applicant
Physical Person
✓ Person data:
Country code: AUT
First name: John
Surname: Doe
Academic degree suffix: BSc MBA
Sex: male
Phone: 0123456
E-Mail address: jd@test.com
Person data (edit) (delete)
Adress data: * < no input data available > Adress data (add)
< Back Next > Cancel application

Figure 72: Definition of the applicant property after personal data was successfully collected.

The entire path in the refinement process that was taken so far is illustrated in Figure 73. Going from left to right means that the user has decided to specify the value(s) for a selected property. Going down indicates the need for specialisation/classification as already described in section 7.4.2. If there are any axioms that can be used for automatic classification they will be used by the forms component to ask the user for the required property values just like in the case of service identification.

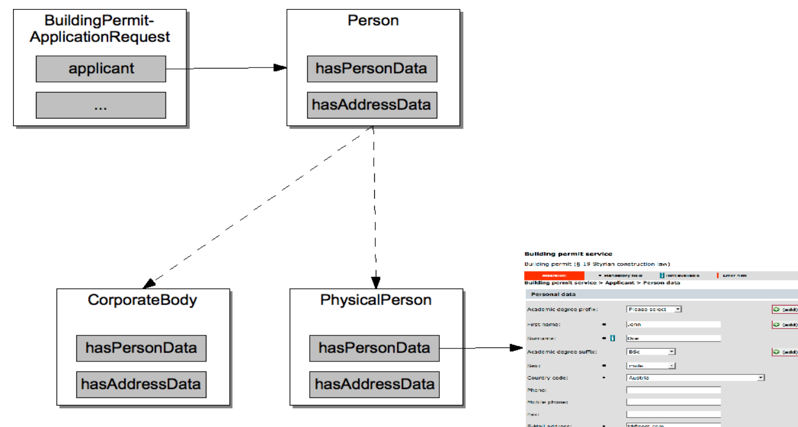


Figure 73: Refinement path consisting of attribute value specification and classification/ specialisation (own illustration)

Once all properties that are spanned by the root concept – *BuildingPermitApplicationRequest* in our example – are filled with valid values, the information collection process is completed. In the case the complete data is successfully validated by the semantic reasoner a final overview of the provided information is presented to the user in a flat a form (see Figure 74). The user can cross-check all the provided data and either go back to make some corrections or go on with submitting the application to the public agency. There is also an option that allows the user to digitally sign the entire application – including all supplements – using the Austrian Citizen Card[190].

What happens to the submitted data depends on the implementation type of the actual public service that is used. As presented in section 7.3.3 there are currently two different implementation types available: *eGrazServiceImplementationType* and *SOAPServiceImplementationType*. The first implementation type indicated that the service is invoked via the so called E-Government platform of the City of Graz [191]. This software system can handle E-Government requests and forwards them to the appropriate department in charge using an electronic file handling system. Furthermore this platform allows citizens to track the current state of their applications is they log on to the system using the Austrian Citizen Card. Technically, since this platform – just like ODEG - is also based on Java, this interface is implemented via remote method invocation (RMI[23]). However, before the data is passed to the interface method all concept instances representing the current application are converted into XML first (see section 7.7.1) and are then packed into a valid EDIAKT II structure. The resulting XML document is passed to the E-Government platform. The method returns an object that contains a digitally signed acknowledgement of receipt or possible error messages. The confirmation is presented to the citizen and can be archived for later reference.

If the service implementation type is an instance of *SOAPServiceImplementationType* this instance contains a reference to a web service endpoint. If this implementation type is used the web service is required to implement the port types that are specified in an automatically created WSDL file (see section 7.7.2 for a detailed description). This allows for a direct web service invocation using the XML serialised instances of the information space.

23 http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html

Building permit service	
Applicant	
Physical Person	
Personal data	
Country code	AUT
First name	John
Surname	Doe
Sex	male
Phone	0316 123456
E-Mail address	john.doe@doe.com
Address data	
Municipality	Graz
Postal code	8010
Street	Inffeldgasse
Building number	16a
Project description	
constrution or attachment	
Type of Building	Residentialhouse
Description	This is the description
Total area (m2)	220
Effective area (m2)	150
Number of floors	2
Floorheight (m)	2,60
Neighbours accepted construction project with signature explicitly	nein
New build or extension	
This project effects the site density	nein
This project will change an existing driveway	nein
Building location	
Building Location	
Piece of land	
District number	15
District	Wetzelsdorf
Cadastre Community	
Cadastre number	63128
Cadastre community	Wetzelsdorf
Realty number	.1750
Protection Area(s)	Historic city protection area AS1

Figure 74: Part of the final overview that allows the user to review the application before it is actually submitted.

7.5.3 Marking the Model

In the previous section one aspect of how to extend a model for application-specific needs was briefly presented when discussing a list of values that can be used to select academic degrees. In this section more possibilities to extend a semantic model in order to add special run-time behaviour will be presented. Most of them have evolved over time to cope with special use-cases and requirements found in various public service domains.

As discussed in section 5, MDA uses several models at different levels of abstraction. The so called Platform Independent Model (PIM) is the root for every MDA driven development approach. This model can be automatically transferred into a more detailed model, the Platform Specific Model (PSM). The advantage of this approach is the simplified reuse of models that are of higher abstraction. On the other side, when turning

a higher level model into a more specific lower level model additional information that is needed by the targeted platform needs to be added. Different approaches on how this information can be added were presented section 5.1.4. Although ODEG directly interprets the PIM, which is represented by a semantic model of the domain, rather than performing any transformation or even source code generation steps, it also relies on additional information that is necessary to cope with more specific or complex use-cases. A simple example to illustrate such a need is the requirement to create different on-screen representations of properties that are actually of the same type. Typically a property of type string is rendered as an ordinary text input field in the resulting web form. This is desirable if the property represents the name of an applicant. What, however, if this property should capture some longer describing text? Thus there has to be some mean that allows the form component to render this field as a text area and not as a text field.

```
concept Construction
    annotations
        dc#description hasValue "Super-concept for all constructions that can be built"
        gender hasValue segofUtil#Neuter
    endAnnotations
    constructionDescription ofType (0 1) _string
    annotations
         formElementType hasValue "textArea"
    endAnnotations
```

Listing 69: Definition of the construction concept containing two marks needed for form creation

One mechanism that is proposed by MDA to achieve this is called marking. Marks are model tokens that add extra information to the PIM, which is needed for a proper transformation into a PSM. This marking approach was also adopted by ODEG. In the case of the text area field ODEG uses WSML's annotation mechanism. As the name suggests, these elements can be used to annotate model items. Thus, they can also be seen as meta-data since they represent "information about information". In fact, annotations add information about the model. Listing 69 shows how an annotation can be used to provide the required information that will cause a property to be rendered as text area rather than as a default text field. Annotations directly have to follow the element they refer to. The `formElementType` is a property known by the form component and its value defines the graphical appearance of the `constructionDescription` property. However, Listing 69 also contains a second mark that is represented by an annotation. The `gender` property is necessary to find the correct German articles for the goal templates and labels as already discussed in section 7.4.1. The value assigned to the `gender` property is defined in the *utility* ontology. This ontology contains the definition of most of the marks that can be expressed as concepts and/or instances. Thus, this ontology will be presented in more detail in the rest of this section.

The gender specific snippet of the utility ontology is shown in Listing 70. The instances defined here can be used to determine the gender of a concept. This information is used to provide correct sentences during the goal definition phase and all classification steps.

```
concept Gender
    annotations
        dc#description hasValue "class for grammatical gender"
    endAnnotations

instance Male memberOf Gender

instance Female memberOf Gender

instance Neuter memberOf Gender
```

Listing 70: Definition of different genders needed to determine correct german articles

In the previous section the list box with different academic degrees was already discussed. This list of values is indicated by the fact that the underlying concept `AcademicDegreeSuffix` is a sub-concept of `OneOfEnumerationType` (compare Listing 68). Thus, this mark is no longer an annotation that adds meta-information to the domain model but is a direct part of the model. This might cause some arguments that the

pure semantic model is now contaminated with implementation specific constructs, which might limit its general use and reusability. In fact when taking a closer look at this example it simply asserts that values assigned to some academic degree attribute indicated by the `AcademicDegreeSuffix` type have to be members of some given set of values, although this approach does not look to be the most intuitive. Thus, also this part of the ontology can used in any other environment and would allow reasoning about consistent instances. On the other hand, no other form of expressing such as restriction would cause the form component to render a list of values. Thus existing models need to be adapted to the particular needs of the form components meta-model.

```
concept OneOfEnumerationType
    annotations
        dc#description hasValue "super concept for all one of enumeration types"
    endAnnotations

concept SomeOfEnumerationType
    annotations
        dc#description hasValue "super concept for all some of enumeration types -> becomes check box
in form"
    endAnnotations
```

Listing 71: Mark concepts that can be used to model lists of values

Besides the `OneOfEnumeration` concept there also exists a `SomeOfEnumerationType` concept as shown in Listing 71. The meaning of this second concept is almost self-explaining. It will cause any attribute of this type to be rendered as set of checkboxes. The values that are actually rendered as checkboxes are extracted from a corresponding constraint-axiom just like shown in Listing 68.

An approach that shows a similar result in the user interface but is conceptually complete different is the use of so called enumeration instances. Whereas in the previously discussed approaches the list of values was limited to elements of type string, this approach allows for entire concept instances to be elements of a list of values. An example is presented in Listing 72, which is taken from the business registration domain. If you

```
concept HotelRestaurantIndustry subConceptOf DLAProfession
    activities ofType (1 *) HotelRestaurantActivity

concept HotelRestaurantActivity subConceptOf Activity

instance Lodging memberOf {HotelRestaurantActivity, segofUtil#EnumerationInstance}
instance RestaurantActivity memberOf {HotelRestaurantActivity, segofUtil#EnumerationInstance}
instance DrinksInBuses memberOf {HotelRestaurantActivity, segofUtil#EnumerationInstance}
instance MountainShelter memberOf {HotelRestaurantActivity, segofUtil#EnumerationInstance}
instance SmallClosedDrinks memberOf {HotelRestaurantActivity, segofUtil#EnumerationInstance}
instance SmallLodging memberOf {HotelRestaurantActivity, segofUtil#EnumerationInstance}
instance Buschenschank memberOf {HotelRestaurantActivity, segofUtil#EnumerationInstance}
instance DrinksInAutomators memberOf {HotelRestaurantActivity, segofUtil#EnumerationInstance}
```

Listing 72: Ontology snippet showing the use of the EnumerationInstance concept to define list of values.

want to register a business you have to state exactly what you plan to do, allowing the system to identify all relevant regulations that will apply to this situation. The conceptual model is based on professions. Every profession enables or allows one to perform certain activities. The `HotelRestaurantIndustry` is a special type of profession that allows for some activities that are of type `HotelRestaurantActivity`. Depending on the set of activities that the owner of the respective business wants to offer, different regulations and therewith different procedures might be relevant. Thus it is important to ask the user about the planed activities. The ontology therefore provides a set of activity instances, which are all members of `HotelRestaurantActivity` as well as `EnumerationInstance`. Whereas the first type allows them to be used as fillers of the `HotelRestaurantIndustry` concept's `activities` property, the latter type tells the form component that these instances can be used to populate a list of values.

Since in this example the cardinality of the *activities* property allows for multiple values, the list of values is actually rendered as a set of checkboxes like in the case of the previously mentioned *SomeOfEnumerationType*. Figure 75 shows the resulting form. The text entries presented there are all taken from a resource bundle as explained in section 7.4.1.

Figure 75: Dialogue asking the current user to select different types of activities that will be offered via the new business that is about to be registered (currently available in German only).

So far there were already two different approaches to create lists of values presented. One that can be applied to string values and one that can be applied to instances as elements of a list of values. Sometime, however, it also necessary to provide different concepts in a list of values. As already discussed in section 7.3.2 it is not possible to refer from an instance to the concept. ODEG, however, uses a convention to use a string property with the IRI of a concept as its value whenever a relation between an instances and concepts should be simulated. This is for example done to create a relation between public service instances and desires that are fulfilled by such a service. Sometimes, however, such situations can occur in application domains as well. In the building permit domain for example it is necessary to specify the type of vehicle that should be parked inside a garage, since this might have a major impact on the procedures that need to be conducted. Therefore the concept *Garage* defines a property called *forVehicleType* that is of type *MotorVehicle*. This type comes with an existing third-party vehicles ontology that was simply imported by the construction specific ontology. Consequently, this property refers to all different types of motor vehicles. The question, however, is how to model these different types. In the context of this use-case it is necessary to find out whether the applicant plans to park cars, motorbikes or any other type of vehicle in the garage. Thus one option would be to model *Car* as an instance of the concept *MotorVehicle*. Although this is possible it is not very intuitive since car is rather seen as a special type of vehicle and an instance would represent a specific existing car described by model, brand, license-plate and so on. In fact the imported motor vehicle ontology introduces car as a sub-concept of motor-vehicles. Thus, the *forVehicleType* attribute has to refer to a concept rather than holding an instance as value.

```
concept Garage subConceptOf {Construction,ClosedBuilding,SignatureRelevant, PullDownRelevant,
RebuildRelevant}
    annotations
        dc#description hasValue "Building for parking vehicles"
        gender hasValue segofUtil#Female
        segofUtil#displayPriority hasValue "4"
    endAnnotations
    forVehicleType ofType  (1 1) vehicle#MotorVehicle
    annotations
        dc#description hasValue "Vehicle type for which garage is built"
        segofUtil#subConceptEnum hasValue "true"
    endAnnotations
    vehicleCapacity ofType  (1 1) _integer
    effectiveArea ofType (1 1) _decimal
```

Listing 73: Definition of the garage concept

The `forVehicleType` property is therefore annotated with a property of type `subConceptEnum`. This is a hint for the form component to render a list of values consisting of all sub-types of `MotorVehicle` as defined in the ontologies. The resulting form can be seen in Figure 76.

Figure 76: Automatically generated form to specify a garage

Since the cardinality of the `forVehicleType` property is set to exactly one the list of values is rendered as radio buttons. Although this looks similar to a specialisation step like presented in section 7.4.2 (compare Figure 58), it is conceptually different since this step is not about determining the actual type of a still abstract attribute concept but about determining a reference to a concept that is held as the value of a property. Actually Figure 76 also demonstrates the effect of the text area hint discussed earlier in this section.

Another requirement that came up with certain use-cases is the possibility to define decent default values for certain properties or to pre-set properties to values in a way that this value can't be changed by the user. One such use case is the initialisation of address concepts. Since most of the services modelled yet are offered by the City of Graz it seems plausible that most of the users will have addresses located in Graz as well. For some procedures it is even necessary to live in Graz. However, defining Graz as the default value for the municipality or city attribute of the address concept would limit the reusability of such a concept in other contexts. Consequently a default value for a concept's property should not be defined as part of this concept but should be defined in the appropriate context in which such a concept is used. The solution to this problem is a way that allows to define default values for a concept's attribute at any pace this concept might occur, for example as part of a public service's request object definition.

```
concept BuildingPermitApplicationRequest subConceptOf ConstructionServiceRequest
    annotations
        dc#description hasValue "concept representing input to building permit service"
        segofUtil#criticalAxiom hasValue "true"
    endAnnotations
    applicant ofType (1 *) personData#Person
    annotations
        segofUtil#defaultValue hasValue "Graz"
        segofUtil#defaultValueAttribute hasValue "applicant.hasAdressData.municipality"
    endAnnotations
    delegate ofType (0 1) personData#Person
    annotations
        segofUtil#defaultValue hasValue "Graz"
        segofUtil#defaultValueAttribute hasValue "delegate.hasAdressData.municipality"
    endAnnotations
    construction ofType (1 1) Construction*/
    constructionProject ofType (1 *) ConstructionProject
    buildingLocation ofType (1 1) BuildingLocation
    annotations
        segofUtil#defaultValue hasValue "Graz"
        segofUtil#defaultValueAttribute hasValue "buildingLocation.onPieceOfLand.address.municipality"
        segofUtil#defaultValueExclusive hasValue "true"
    endAnnotations
    buildingSiteEligibility ofType (0 1) BuildingSiteEligibilityPlaceHolder
    record ofType (0 *) segofUtil#File
```

Listing 74: Complete definition of the BuildingPermitApplicationRequest concept including marks to define default values

Such an example is shown in Listing 74. A default value definition might consist of up to three properties. The `defaultValue` property defines the actual value that should be used whenever a new instance of the concept containing the corresponding attribute is created. The `defaultValueAttribute` property refers to the property to which the default value should be applied. The value of this attribute is a sequence of attribute names that are separated by a dot. The first element of this value refers the a property of the current concept. Taking a closer look at the first occurrence of `defaultValueAttribute` in Listing 74 reveals that it applies to the `applicant` property of the current application request concept. The next part of the value refers to the applicant's `hasAddressData` property and the final part to the address' `municipality` attribute.

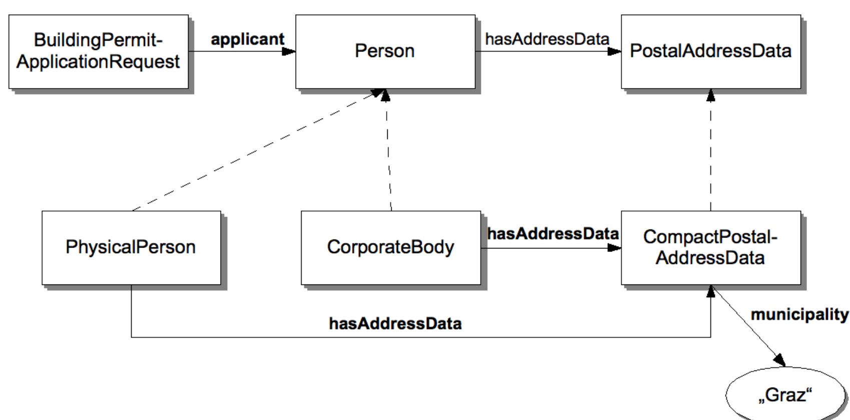


Figure 77: Concept graph representing the path of a default value for the municipality attribute

The entire situation is illustrated in Listing 74. The `applicant` property of the `BuildingPermitApplicationRequest` concept is of type `Person`, which is considered to be abstract. Thus it has to be replaced by an instance of any of its concrete sub-concepts. The advantage of the notation used to specify the attribute that should hold a default value is the fact that it spans all possible paths that might lead to the resulting municipality property. The notation is therefore independent of the actual types that are used in the current situation. On the other hand, if the path cannot be found in the existing instance hierarchy this

does not lead to any error. In such a case, that might for example be caused by the fact the not all potentially selectable concepts share the same attribute, the default value is simply ignored.

The third attribute that might be part of a default value definition is called `defaultValueExclusive`. Whenever this attribute is set to true the provided default value cannot be overridden by the user. This attribute is used in the last default value definition in Listing 74. The location of a building project has to be in Graz since otherwise the service offered by the City of Graz can't be used to get permission for this project. Therefore the form component takes the default value and applies it to the appropriate field, but the on-screen representation of this value is set to read-only.

Another feature that is supported by the run-time environment is the possibility to add help and information messages to arbitrary model elements. This is not done by extending or marking the actual model but by providing additional resource-bundle keys that follow a simple convention. The mechanism used to turn model elements into human-readable form was already explained in section 7.4.1. The IRI of a model element is – after some simple character substitutions - used as a key in the resource-bundle. If such a key with the suffix ".help" exists, this entry is used as a help message that is rendered right next to the corresponding model element.

An example of such a help text definition is show in Listing 75. The form component indicates the existence of a help text by rendering an information icon. This is in compliance with the Austrian Style Guide for Electronic Forms[188]. The actual help text becomes visible when the user either clicks on the information icon or hovers over this icon with the mouse like shown in Figure 78.

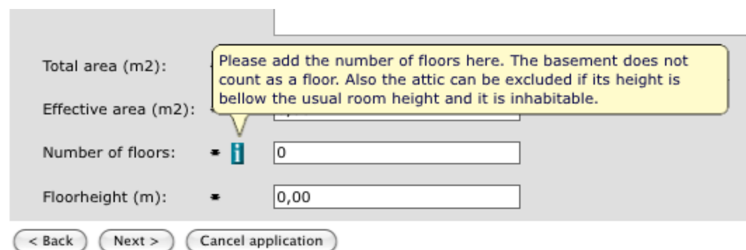


Figure 78: Screenshot showing the effect of a help text

Listing 75: Snippet from the resource bundle defining a help text for the attribute "numberFloors"

7.6 Auxiliary Service Modelling

The ODEG approach to interpret a semantic model and using a semantic reasoner to evaluate this model at run-time is very powerful. The possibility to make intensive use of abstraction, specialisation and classification allows for simple yet enormously expressive models. Axioms can be used to enforce complex plausibility constraints on the ontologies used. Besides this the previous section showed how the formal model of a domain can be extended by application specific information that influences the look and feel as well as functionality of a system. Nevertheless, not everything can be done by using semantic technologies exclusively. At some point it is necessary and/or reasonable to integrate external services as well. Let's take the following scenario to motivate the integration of such external services. Austria as well as many other European countries operates a central register of citizens where all people living in Austria are stored together with their current address. Information like this could be used to check whether an address that was provided by the user is correct. It is simply impossible to do checks like this within a reasoner since the sheer size of the required data would exceed the capabilities of these tools. As pointed out in sections 3.2.4 and 3.4 reasoning about semantic models can easily have an exponential complexity. That is why the number of elements registered with a reasoner should always be kept as small as possible. Thus, alternative ways have to be provided to enable plausibility checks outside the reasoner. For this purpose ODEG provides a so called auxiliary service ontology to integrate the use of external services into a semantic domain model. Some of the elements defined in this ontology can be used just like the marks that were introduced in the

previous section.

7.6.1 The Auxiliary Service Ontology

The auxiliary service ontology (see Figure 79) defines a meta-model for external services that can mainly be used either to check the consistency, correctness and plausibility of some data in the information space or to provide values which are valid fillers for certain properties.

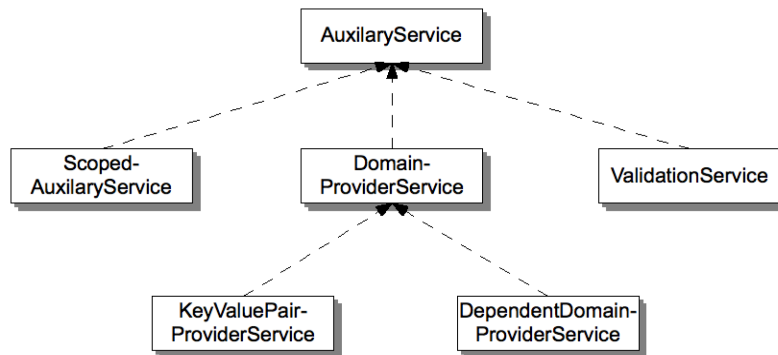


Figure 79: Different auxiliary services as defined in the auxiliary service ontology

All external services that could be used inside a semantic domain model are sub-concepts of `AuxiliaryService`. The three direct sub-concepts of this type fall into two categories. The types `DomainProviderService` and `ValidationService` indicate different functionality of an external service, whereas the type `ScopedAuxiliaryService` provides information about when an auxiliary service can be used. The definition of `ScopedAuxiliaryService` and its super-concept is given in Error: Reference source not found.

```
concept AuxiliaryService
    annotations
    dc#description hasValue "super concept for all auxiliary services"
  endAnnotations
  providedBy ofType (0 1) gea#ServiceProvider

concept ScopedAuxiliaryService subConceptOf AuxiliaryService
    annotations
    dc#description hasValue "super concept for all scoped auxiliary services which means that the service
can only be applied to certain administrative levels and locations"
    endAnnotations
    appliesTo ofType (1 *) gea#AdministrationLevel
    validWithin ofType (1 *) geaSeGoF#GovernmentalEntity

concept GovernmentalEntity subConceptOf gea#ServiceProvider
    hasGovernmentalLevel ofType (1 1) gea#AdministrationLevel
    hasParentAdministrativeUnit ofType (0 1) GovernmentalEntity
```

Listing 76: Definition of AuxiliaryService and ScopedAuxiliaryService together with GovernmentalEntity.

Every auxiliary service holds information about its service provider. Additionally each scoped service defines to which governmental level the service applies (via its `appliesTo` property) and by which governmental entities it can be effectively used (via its `validWithin` property). By combining these two attributes the scope of a service can be described. To illustrate the idea behind this concept the structure of governmental entities is shown in Figure 80. All oval-shaped forms are instances, whereas all rectangular shapes are concepts. The background colour of the shapes indicates which instance is a member of which concept.

The structure starts with the Austrian national government. At least in this model this element has no parent

administrative unit, although one might model the European commission as a parent unit for example. Austria is structured into provinces reflecting different regions at the next lower administrative level. Every province is split into prefectures, which in turn contain communities.

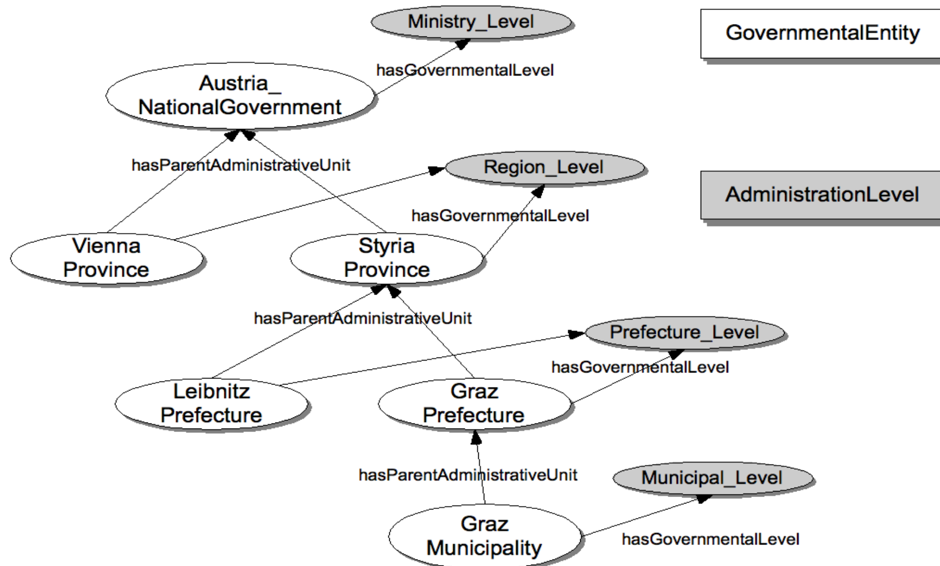


Figure 80: A small part of the Austrian administration hierarchy according to the GEA meta-model

Based on this model every possible combination of where an auxiliary service could be used can be modelled. Assuming that there exists an auxiliary service that can be used by all public services offered at municipal level in Austria. In this case the `appliesTo` property of the auxiliary service has to contain the instance `Municipal_Level` and the `validWithin` property has to have the value `Austrian_NationalGovernment`. This indicates that all services at `Municipal_Level` that have the `Austrian_NationalGovernment` as their `hasParentAdministrativeUnit` can use this auxiliary service. In another scenario there should exists a service that can be used by all prefectures within Styria. To achieve this, the `appliesTo` property of the auxiliary service must contain the value `Prefecture_Level` and the `validWithin` property the value `Styria_Province`. Therefore the concept `ScopedAuxiliaryService` can be used to determine in which situation an auxiliary service can be used but it does contain any hint about what a service can be actually used for.

To describe what an auxiliary service can be used for the remaining two sub-concepts of `AuxiliaryService` are used. Listing 77 shows the definition of the concept `ValidationService`. It adds one single property called `validatesValuesFor` that refers to the IRIs of those concepts that can be validated. More precisely instances of these concepts can be validated according to the internal implementation of the auxiliary service.

```
concept ValidationService subConceptOf AuxiliaryService
    annotations
        dc#description hasValue "validates values for a given concept"
    endAnnotations
    validatesValuesFor ofType(1 *) _string

instance BuildingLocationGrazValidationService memberOf {ScopedAuxiliaryService, ValidationService,
                                                          service#SpringBeanService}
    providedBy hasValue geaSeGoF#Graz_Municipality
    appliesTo hasValue gea#MunicipalityLevel
    validWithin hasValue geaSeGoF#Graz_Municipality
    validatesValuesFor hasValue _"http://segof.fh-joanneum.at/Construction#PieceOfLand"
    service#beanName hasValue "districtValidationGrazService"
```

Listing 77: Definition of the ValidationService concept together with a sample instance

Listing 77 also contains an instance of a validation service to illustrate the use of `ValidationService` together

with the previously presented concept *ScopedAuxiliaryService*. The service shown in the listing is used to verify the correctness of building project locations which are represented by instances of the type *PieceOfLand*. Thus the service is able to validate this type of instances, expressed by its *validatesValuesFor* property. Besides this, this validation service is also a scoped service. According to the rules of interpreting the *appliesTo* and *validWithin* properties it can only be used by services offered by the City of Graz since its implementation has only access to locations within the city. Additionally the service instance is a member of *SpringBeanService*. This implementation specific concept will be explained in the next section. Whereas validation of a given instance is the primary task of a validation service, these services can also be used to set missing values on instances as well as we will see later.

```
concept DomainProviderService subConceptOf AuxiliaryService
    annotations
    dc#description hasValue "provides values for given attributes or concepts"
   endAnnotations
   providesValuesFor ofType (1 *) _string
   usesAjax ofType (1 1) _boolean

concept KeyValuePairProviderService subConceptOf DomainProviderService

concept DependentDomainProviderService subConceptOf DomainProviderService
    annotations
    dc#description hasValue "provides a domain for an attribute that is restricted by the value of another
                              attribute"
   endAnnotations
   dependsOn ofType (0 *) _string
```

Listing 78: Definition of the different types of domain provider services

Listing 78 presents the definitions of the three different types of so called domain provider services. These services are used to retrieve lists of values for certain properties in different ways. How to define simple lists of values was already discussed in section 7.5.3. In contrast to these approaches domain provider services are used to access external datasources like databases or other web-services. The root of this branch of auxiliary services (see Figure 80) is the concept *DomainProviderService*. It has two additional properties. The *providesValuesFor* property holds the IRI's of those concepts and properties the service is able to provide instances or values for. The *usesAjax* property defines how the list of values is rendered. If this property is set to false then the list of values will be rendered as pull-down list. For a large number of entries, however, this approach is unsuitable. Therefore, in cases like this the *usesAjax* property should be set to true, which will cause the form component to render an Ajax[192] based text field with dynamic autocompletion.

The *KeyValuePairProviderService* is a specialised sub-concept. Sometimes codes are used to ease automatic processing of data, which, however, are typically not very well suited to be used in user interfaces. For example public electronic services operated by the City of Graz prefer to use the number of an inner-city district or borough where citizens typically refer to them by their names rather than their numbers. In such a scenario a key-value-pair provider service can be used that takes the district number as the key and the district name as the value. The user of the system can select the name of the appropriate district, but the system uses the districts number internally. Thus, this type of provider service is used whenever some encoded information should be looked up in a user-friendly way. There are additional attributes that allow to decide which of these two elements should be displayed in the final overview dialogue (compare Figure 74).

Another specialised auxiliary service is represented by the concept *DependentDomainProviderService*. Just like the other domain provider services it provides values for concepts or property instances but to do so, this type of service requires additional information. One example of such dependent service is shown in Listing 79.

Figure 81: Screenshot of a form that uses the street name provider service

This service provides a list of street names, however, to know which street names are relevant, the service needs to know the current city. To indicate this, the `dependsOn` property refers the IRI of the concept `municipality`. To deal with the huge amount of data that might be retrieved by this service the `usesAjax` property is set to true. Figure 81 shows the effect of using the `StreetnameProviderService`.

7.6.2 Implementing Auxiliary Services

In the previous section the part of the ODEG semantic meta-model necessary to specify auxiliary services was presented. This section will present some implementation details.

The approach chosen to implement auxiliary services is in fact very similar to the one chosen for actual public services as presented in section 7.3.3. The implementation of a public service is determined by its `PublicServiceImplementationType`. There are currently to sub-types of this concept, which are called `eGrazServiceImplementationType` and `SOAPServiceImplementationType`. The implementation of an auxiliary service is defined by its membership to a sub-concept of `ServiceImplementation`. The currently available types are shown in Listing 80.

```
concept ServiceImplementation
    annotations
        dc#description hasValue "super concept for all service implementations"
    endAnnotations

concept SpringBeanService subConceptOf ServiceImplementation
    beanName ofType _string

concept WSMOService subConceptOf ServiceImplementation
    hasWSMOService ofType _string
```

Listing 80: Concepts representing different implementation alternatives for auxiliary services

In fact all currently available auxiliary services are of type `SpringBeanService`. This means that the service implementation is represented by an ordinary JavaBean[191] that is managed by the Spring framework[193]. Basically this allows to refer to an instance of the service implementation class by a simple name. Since the run-time environment has to tread all auxiliary services uniformly, every implementation class has to implement the same interface, which is shown in Listing 81.

```
public interface SpringBeanService {

    /**
     * central execute method for every springbean service
     */
    public Object execute(Object params);

}
```

Listing 81: Interface for all auxiliary service implementations

This is a typical application of the so called command-pattern [194], which is used when different activities should be triggered in a uniform way. The actual types of the argument and the return value of a specific service implementation depend on the type of the auxiliary service. Validation services for example receive an array that contains the current instance as it is registered with the reasoner and a set of attributes that were collected by the user. This allows validation services not only to check the provided values but also to modify or to complete values of the current instance depending on the user's input. This possibility can explicitly be used in the model as presented in the next section. The return value of validation services is a set of possible error messages. Thus, if this set is empty, validation was successful. Otherwise these messages are presented to the user. Beside the service specific arguments, there is always an object passed along that holds information about the current user's selected language, which allows for localised return values.

In the case of a domain provider service there are typically no service specific arguments required except for those provider services that depend on other input like the *StreetnameProviderService*. Thus instances of *DependentDomainProviderService* are passed the values of those properties that are listed in the *dependsOn* attribute of their definition in the order they appear. Again, in the case of the *StreetnameProviderService* the name of the current municipality is passed as the only argument besides the afore-mentioned localisation information. The return type is a list containing the appropriate values found if any. Thus, this list might be empty as well.

Every instance of a *KeyValuePairProviderService* returns a map instead of a list. This map contains the keys together with the associated values.

7.6.3 Enabling Auxiliary Services

```
concept PieceOfLand
    annotations
        dc#description hasValue "concept for preliminary address of houses and societal entities in Graz"
        segofUtil#valuesValidatedByService hasValue "true"
    endAnnotations
    propertyOwner ofType (0 1) personData#Person
    districtCadastre ofType (1 1) _string
    annotations
        segofUtil#writeOnlyField hasValue "true"
        segofUtil#valuesProvidedByService hasValue "true"
    endAnnotations
    district ofType (0 1) District
    annotations
        segofUtil#valueDerivedByOtherField hasValue "true"
    endAnnotations
    cadastreCommunity ofType (0 1) CadastreCommunity
    annotations
        segofUtil#valueDerivedByOtherField hasValue "true"
    endAnnotations
    realtyNumber ofType (1 1) _string
    ez ofType (0 1) _string
    postalCode ofType (1 1) _string
    annotations
        segofUtil#valuesProvidedByService hasValue "true"
        segofUtil#minQueryLength hasValue "1"
    endAnnotations
    securityArea ofType (0 *) _string
    annotations
        segofUtil#valueDerivedByOtherField hasValue "true"
        segofUtil#lineBreak hasValue "true"
        segofUtil#containsTranslation hasValue "true"
    endAnnotations
    securityAreaInfo ofType (0 *) _string
    annotations
        segofUtil#defaultValue hasValue "noSecurityArea"
        segofUtil#valueDerivedByOtherField hasValue "true"
        segofUtil#containsTranslation hasValue "true"
        segofUtil#hasHTML hasValue "true"
        segofUtil#lineBreak hasValue "true"
    endAnnotations
    municipality ofType (1 1) _string
    annotations
        segofUtil#defaultValue hasValue "Graz"
        segofUtil#defaultValueAttribute hasValue "municipality"
        segofUtil#defaultValueExclusive hasValue "true"
    endAnnotations
    streetName ofType (1 1) _string
    annotations
        segofUtil#valuesProvidedByService hasValue "true"
        segofUtil#minQueryLength hasValue "3"
    endAnnotations
    buildingNumber ofType (0 1) _string
    annotations
        segofUtil#valuesProvidedByService hasValue "true"
        segofUtil#minQueryLength hasValue "1"
    endAnnotations
```

Listing 82: Definition of the PieceOfLand concept used to capture locations of building projects

In the previous sub-sections it was shown which auxiliary services are available, how they can be defined in the semantic model and also how they have to be implemented. In this section it will be shown, how these services are actually mapped to those concepts and attributes they will be eventually applied to and how the discovery process for auxiliary services works.

Basically the semantic description of every auxiliary service is precise enough to figure out when and where it can be used by the run-time environment. However, to optimise the performance of the form generation

component it does not look for appropriate validation or value provider services for every single concept or attribute. A design decision was made to trigger the search for appropriate auxiliary services only if this is indicated by the existence of corresponding marks in the model. Listing 82 shows the definition of a concept that makes intensive use of different types of auxiliary services. This concept, called `PieceOfLand` land is used to capture information about the location of a building project.

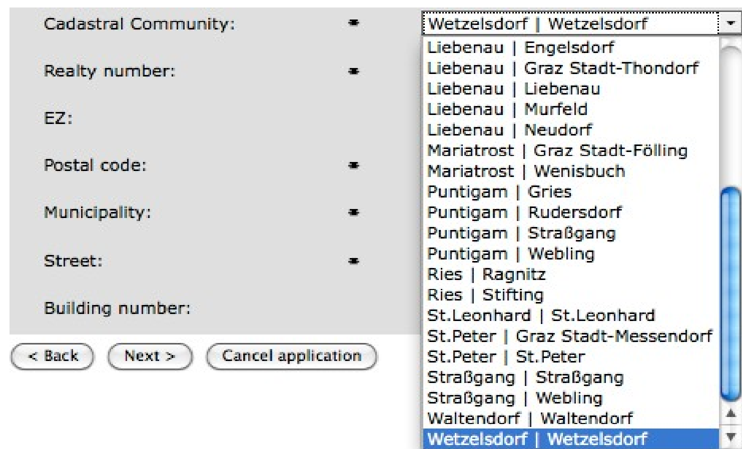


Figure 82: The list of values created for the disctrictCadastre property of the PieceOfLand concept.

The concept definition is followed by an annotation containing the property `valuesValidatedByService`, which indicates that the validation of instances of this type is not exclusively done by the reasoner but also by validation services. This causes a multi-step validation process. Whenever an instance of this concept with values provided by the user becomes available, it is first checked using those validation services that are found by querying the semantic model using the reasoner. Thus, the system will look up all instances of `ValidationService` that are able to validate instances of `PieceOfLand`. After calling the validation services the instance is checked by the reasoner whether it is consistent with the rules imposed by the ontologies. However, if no validation services are found in the first step, service based validation is skipped and no error messages will be produced. This behaviour allows for dynamically adding and removing validation services to and from the model without causing any errors. By design, validation using auxiliary services is rather seen as an optional step. Thus, when modelling a concept and adding the `valuesValidatedByService` annotation, the semantics is more like “if there exists an appropriate validation service, then use it”. Later, by adding more and more validation services to the model, the quality of data can be improved but the initial model can also be used without any validation services available yet. If any errors occur during the validation phase, the user is sent back to input page containing the error messages.

The next attribute that contains annotations is called `districtCadastre`. This property refers to the name of the city district as it occurs in the cadastre. In fact, most city districts are subdivided into more specific cadastral communities in the cadastre. There are two annotations used with this property. The first one, called `writeOnlyField`, means that this property only has to be rendered in the input form but should not be included in any read-only form that is presented to the user in oder to review the provided information. The second one, called `valuesProvidedByService` actually indicates the use of a value provider service. This annotation causes the form generation component to look for an appropriate service registered with the reasoner. Again, if no such service is found, this annotation is ignored and an ordinary input field is rendered instead, otherwise, a list of values is rendered as shown in Figure 82.

The next two attributes called `district` and `cadastreCommunity` are annotated with the `valueDerivedByOtherField` property. This indicates, that the values for these fields do not have to be filled in by the user, but there exists an auxiliary service that will add appropriate fillers for these properties. This is why those properties do not appear in the input form shown in Figure 83. However, if no validation service is found, or the service does not add the missing values to the instance, this could result in validation errors if such derived properties were mandatory, which is not the case in the current example. The `postalCode`

property is also annotated with `valuesProvidedByService`. Additionally there is the annotation property called `minQueryLength`. If the domain provider service for the postal code is configured to use AJAX, then this property defines the required minimum length of user input to trigger an AJAX request. In this example every key typed will cause the list of values to be updated.

The next two properties reflect some peculiarities that only apply to the City of Graz. These two attributes are called `securityArea` and `securityAreaInfo`. A security area is some part of the city where special regulations apply to building projects. For example, if you plan to erect or modify a building in the historic city district

Figure 83: Screenshot of the input form used to get values for an instance of type PieceOfLand.

there apply rather rigid rules in order to maintain the overall appearance of the townscape. Another example of such zones are those areas of the city that bear a certain risk of floodwaters. Both of these properties are annotated with `valueDerivedByOtherField`, thus these fields do not occur in the input form shown in Figure 83. Besides this there exist various hints for the form generation component. The `containsTranslation` property indicates that the values set by the validation service are actually keys and that the text presented in the user interface should be looked up in the current resource bundle based on these keys. The `lineBreak` attribute tells the form generation component to add HTML line break tags between consecutive values of every such property and the `hasHTML` attribute prevents HTML escaping and therefore allows for the use of HTML in (translated) property values.

Once the form was submitted and has successfully passed validation the resulting information is shown on the next page (see Figure 84). The property `districtCadastre` that is annotated with `writeOnlyField` is not shown on this page. Instead all the other properties that are annotated with `valueDerivedByOtherField` are now shown with the values set by the validation service for instances of `PieceOfLand`.

In the general the use of auxiliary services requires the existence of such a service that is represented by an actual service implementation and a semantic description. On the other side, the need for such a service has to be indicated by appropriate annotations at the concepts and properties that should be validated or provided with values. It is important to point out that there is no direct reference between a particular auxiliary service and a concept that would like to utilise such a service. In fact by querying the reasoner a service is sought that can be applied in the current situation. If no such service is found, this does not necessarily indicate an error and the system proceeds as if there was no auxiliary annotation at all. This allows for dynamic extension of the model with additional auxiliary services at any time. Appropriate, available services are automatically found based to their semantic description and applied wherever they are needed. In fact, the auxiliary service framework as implemented by ODEG can be seen as a lightweight

semantic (web) service framework in its own.

Building Location

District: 15 Wetzelsdorf
Cadastre community: 63128 Wetzelsdorf
Realty number: .1750
Postal code: 8020

✓ on Property:

Protection Area(s): Historic city protection area AS1
Development area Bebauungsrichtlinie im Anlassfall im Grüngürtel
Well protection area BS1
Space heating restriction area DPL2
Memorial protection area DM
Groundwater protection area GS2
Mineral spring protection area HS
Flood threat area HQ30 HQ30
Flood threat area HQ100 HQ100
Dedication / additional Dedication / min. site density / max. site density KG+WA/[EZ]/,3/2,5
Nature preserve GLT
Spring protection area QS

Protection Area Information: Piece of land is within a zone of protection of historic buildings and monuments. Please contact the committee for the protection of historic buildings and monuments!
Piece of land is in Protection Area HQ 30 or HQ 100. Please contact the municipal administration offices regarding further information
Piece of land lies within a preserve area, please contact the nature conservation department of the City of Graz!

Municipality: Graz
Street: Wetzelsdorfer Straße
Building number: 33

on Property (edit) | (delete)

on Property: < no input data available > | additional on Property (add)

< Back | Next > | Cancel application

Figure 84: An instance of PieceOfLand was successfully added to the application

7.7 WSDL and XSD Generation

As described in the previous chapters, ODEG, through its service finder and electronic form components, offers extremely rich and flexible support for the front-end used by citizens. Nevertheless the basic idea of this approach as well as the technologies used were influenced by frameworks for implementing semantic web services (compare section 4). Thanks to the adoption of GEA principles, however, the description of public services is implementation agnostic. Thus, the model makes no assumption about whether a service is implemented as a web service or not. Nevertheless, there exists the `implementationType` property (see section 7.3.3) to indicate the actual nature of the service's implementation.

ODEG has chosen to keep all its information at the semantic model level in order to use the expressive power of ontologies that can be utilised via semantic reasoners. Thus, all the information that was entered by citizens using the service finder and/or the semantic forms component is held in the ontologies by the means of concept instances. In order to provide this information to other systems that are not ontology based, it has to be transformed into some data exchange format. The most popular technology used to exchange information between different systems is XML. In fact also web-services are using XML as their message format. To describe the possible structure of these messages XML schema is used. Consequently also ODEG provides a way to export its information state to XML.

7.7.1 Converting Ontologies to XML Schema

Whereas there already exist ways to serialise ontologies into XML, the requirements for exchanging user data are different. The default XML serialisation is used to write or store ontologies that can for example be loaded into reasoners for further processing. Thus, these formats are used to exchange ontologies rather than the data that is kept in a model as concept instances. At this point however, we need to extract instances and have to represent them in XML as intuitively as possible, which means that the resulting XML

should not be biased towards specific semantic model requirements. Actually this is the same problem as producing a service grounding, where instances have to be mapped to messages of a web service, which are instances of XML schema types as well. Possible solutions to the grounding problem together with their drawbacks have already been discussed in section 4.4.3.

Thus, the procedure presented here to turn ontologies used with ODEG into XML schema and instances into corresponding XML is effectively a default grounding. However, before turning ontologies into XML schema the differences between these two technologies have to be pointed out.

There exist recommendations for an algorithm to create XML schema out of OIL ontologies [195]. Although OIL is quite different from WSML (e.g. datatype support is limited), the core problems of translating ontology definitions into XML schema are essentially the same. Even though, these problems are pointed out in the referenced paper, it does not provide any solutions but stays with a simple example that can easily be translated. Fensel argues that the problems related to turn ontologies into (database) schemes derive from the obviously different goals associated to the underlying methodologies:

> *"An ontology provides a domain theory and not the structure of a data container." [196]*

As a matter of fact it is virtually impossible to translate ontologies into corresponding XML schemes without compromising. This is mainly due to the different means that are available to express class hierarchies and therefore inheritance. Whereas WSML supports multiple-inheritance, which can either be expressed explicitly using the `subConceptOf` construct or implicitly via axioms there is fairly limited support for inheritance in XML schema. XML schema's extensions mechanism (see [197] for a detailed explanation) allows for types to be based on other types. These new types can either extend the elements of their super-type by adding additional elements or can restrict elements of their super-type to certain values. In contrast to ontologies, the extension method cannot be used to redefine the type of an existing element to a more specific one.

Besides the limitations imposed by the conceptual differences between ontologies and XML schema there are ODEG specific conventions that have to be considered as well when designing a procedure to create XML schema elements for ontology instances. ODEG considers all concepts that are refined by sub-

```
concept CompactPostalAdressData subConceptOf PostalAdressData
    annotations
        dc#description hasValue "concept for address of houses and societal entities"
    endAnnotations
    countryCode ofType (0 1) _string"
    countryName ofType (0 1) _string
    postalCode ofType (0 1) _string
    municipality ofType (1 1) _string
    streetName ofType (1 1) _string
    buildingNumber ofType (1 1) _string
    unit ofType (0 1) _string
    doorNumber ofType (0 1) _string
```

Listing 83: A concrete concept representing an address with attributes of primitive data types only

concepts to be abstract. Abstract concepts are not supposed to have any instance in the context of a problem domain but provide generalised abstractions of their concrete sub-concepts. Therefore, once the elicitation process is completed only instances of concrete concepts will be part of the information space. Thus, since the resulting XML will be used to exchange the data gathered by the user, only XML elements of these concrete types will be used. On the other side, however, it is important that at least major parts of the concept hierarchy will be preserved in the XML schema type hierarchy as well. Abstraction allows to create simple and generic, yet formally correct service descriptions. Like shown in Listing 67 in section 7.5.1, the required input to a relatively complex service like a building permit application can be captured by a few concepts, covering all possible types of construction scenarios. Maintaining a similar abstraction hierarchy in XML schema will allow for a similar degree of abstraction and therefore compact and comprehensive message types in web service descriptions as well. Also in contrast to the conceptional differences, WSML

and XML schema share a common type system when it comes to primitive data-types like strings, numbers or dates. This allows for a direct mapping between those types in the semantic model as well as in the XML schema definitions.

To point out the characteristics of the algorithm that was designed to create the corresponding XML schema lets start with some simple examples. Listing 83 shows the `CompactPostalAddressData` concept, which represents an address. This concept is a leaf in the concept hierarchy, which according to the ODEG convention makes it a concrete concept. All its attributes are data value attributes, thus, they do not refer to other concepts. Besides this, it has one direct super-concept of type `PostalAddressData`. Due to this relatively simple nature, it can be directly translated into a corresponding XML type like shown in Listing 84. The resulting schema data type has the same name as the concept in the ontology. In fact every concept of an ontology is translated into a complex type in the XML schemes. Every attribute of the concept eventually ends up as an element with the same name. Also the cardinality restrictions stated for a property are translated into the corresponding boundaries in the element definition. The WSMO type *_string* that was used for the properties is replaced by its corresponding XML schema type. Like the concept is a sub-type of `PostalAddressData` its XML schema representation extends the `PostalAddressData` type.

```
<complexType name="CompactPostalAdressData">
    <!--http://purl.org/dc/elements/1.1#description : [concept for address
        of houses and societal entities] -->
    <complexContent>
        <extension base="tns:PostalAdressData">
            <sequence>
                <element maxOccurs="1" minOccurs="0" name="unit" type="string" />
                <element maxOccurs="1" minOccurs="0" name="postalCode" type="string">
                </element>
                <element maxOccurs="1" minOccurs="1" name="municipality"
                    type="string">
                </element>
                <element maxOccurs="1" minOccurs="1" name="streetName"
                    type="string">
                </element>
                <element maxOccurs="1" minOccurs="0" name="doorNumber"
                    type="string">
                </element>
                <element maxOccurs="1" minOccurs="1" name="buildingNumber"
                    type="string">
                </element>
            </sequence>
        </extension>
    </complexContent>
</complexType>
```

Listing 84: XML schema type representing the concept shown in Listing 83.

The XSD extension mechanism has almost the same semantics as in the ontology although there are much more restrictions. It creates an explicit type hierarchy, thus, a sub-type can be used everywhere a super-type is expected. All the elements that are defined in any super-type along the type hierarchy are also elements of the extended type. As shown in this example, annotations are translated into comments. This should improve the readability of the resulting schema.

Another difference between WSMO ontologies and XML schema is the fact that WSMO ontologies that are defined in separate files can share the same namespace. In XSD, however, every namespace can only refer to elements defined in a single file. The translation algorithm therefore stores all elements that are defined in the same namespace in a single file. Thus, the content of several WSML files might end-up in a single schema file if they share the same namespace. Generally schema files have the same namespace as the ontologies that hold the concepts from which the schema elements were derived.

In the simple example used above all attributes of the concept were data properties and therefore used primitive data types. In case of object properties – i.e. properties referring to other concepts – the transformation algorithm works similarly, since every single concept is translated into an XSD complex type.

```
concept Person subConceptOf {FormConcept}
    hasPersonData ofType (0 1) PersonData
    hasAdressData ofType (0 1) PostalAdressData
```

```
<complexType abstract="true" name="Person">
    <complexContent>
        <extension base="tns:FormConcept">
            <sequence>
                <element maxOccurs="1" minOccurs="0" name="hasAdressData"
                    type="tns:PostalAdressData" />
                <element maxOccurs="1" minOccurs="0" name="hasPersonData"
                    type="tns:PersonData" />
            </sequence>
        </extension>
    </complexContent>
</complexType>
```

Listing 85: The Person concept and its translation into XSD schema

Thus, the element has to refer to the corresponding complex type. An example of such a translation is shown in Listing 85. Here the concept `Person` is turned into its XSD representation. Like already mentioned in section 7.3.4 this concept is an abstraction and has further sub-concepts like physical person or corporate body. Accordingly the resulting XML type is declared *abstract* as well. Both properties of the person concept refer to other concepts. As a consequence the elements of the resulting complex type refer to those complex types representing the person's property types. One of these elements is of type `PostalAddressData`, which is the abstract super-type of the `CompactPostalAddress` used in the previous example. Thus an element of the type `CompactPostalAddress` can be used as a value for the `hasPostalAddressData` element. This example therefore demonstrates that the concept hierarchy of the ontology is maintained as a type hierarchy in the XML representation as well.

```
concept FiringInstallation subConceptOf Installation
     annotations
        dc#description hasValue "super concept for all firing installations"
        gender hasValue segofUtil#Female
    endAnnotations
    heatOutput ofType (1 1) _decimal

concept GasInstallation subConceptOf Installation
     annotations
        dc#description hasValue "super concept for all gas installations"
        gender hasValue segofUtil#Female
    endAnnotations
    gasTankType ofType (1 1) GasTankType
    gasTankVolume ofType (1 1) _decimal

concept GasFiringInstallation subConceptOf { FiringInstallation, GasInstallation}
```

Listing 86: Snippet of the construction ontology showing the definition of the concept GasFiringInstallation.

However, as already mentioned one severe problem for the transformation is the fact that ontologies support multiple inheritance and axiomatic classification. XSD requires every inheritance relation to be expressed explicitly and only supports a single super-type that can be extended. Although this is briefly discussed in [195], the algorithm presented there does not take care of multiple inheritance. The procedure presented here tries to overcome the lack of multiple-inheritance support within XML schema by extending the first super-type with all the inherited elements from the other super-types. Thus the super-concept that occurs first in the list of super-concepts is extended and is therefore the only one that shows up in the explicit type hierarchy of the resulting schemes.

An example of such a situation where a concept has several super-concepts is shown in Listing 86. The concept called `GasFiringInstallation` is merely a combination of the concepts `FiringInstallation` and

`GasInstallation`. It does not define any properties but inherits the properties of both super-concepts. The strategy in transforming this concept is to select the first super-concept – `FiringInstallation` in this case – as the super-type of the resulting XSD complex type. To make all other properties, which are added to the concept via multiple inheritance available as well, these properties are added to the extended type. The resulting type is shown in Listing 87. The transformation algorithm takes care that all inherited properties become elements of the XSD types. This is not limited to direct super-concepts but includes all properties along the concept hierarchy. The rational behind this approach is that instances of concrete concepts will find corresponding elements for all their properties in the relevant XSD types.

```
<complexType abstract="true" name="GasFiringInstallation">
    <complexContent>
        <extension base="tns:FiringInstallation">
            <sequence>
                <element maxOccurs="1" minOccurs="1" name="gasTankType"
                    type="tns:GasTankType" />
                <element maxOccurs="1" minOccurs="1" name="gasTankVolume"
                    type="decimal" />
            </sequence>
        </extension>
    </complexContent>
</complexType>
```

Listing 87: XML schema data-type for the GasFiringInstallation concept

A schematic overview of the situation is provided in Figure 85. It illustrates that the properties of the `FiringInstallation` concept are inherited whereas the properties of the `GasInstallation` concept are directly added to the extended type. Only the type-hierarchy for the `FiringInstallation` will be available in XSD types whereas there will be no relation to `GasInstallation` any longer .

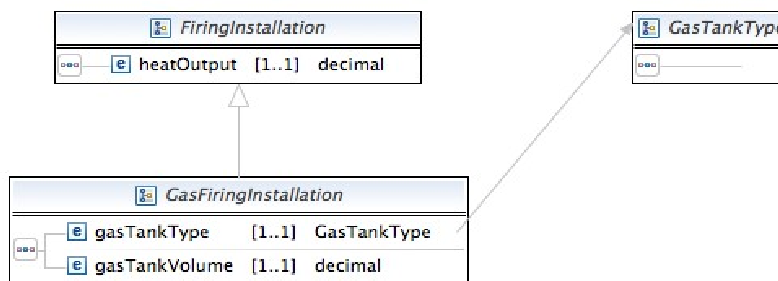


Figure 85: XSD type hierarchy of the GasFiringInstallation type.

An analysis of concepts with multiple super-concepts in the various ODEG ontologies has shown that the lack of multiple inheritance is not too problematic anyway. In fact the overwhelming majority of concepts that fall into this category have additional super-concepts that work as extra classifiers like the `PullDownRelevant` concept. These concepts merely work as markers and are used to support efficient semantic queries rather than being part of structural concept hierarchies in the terms of generalisation or specialisation. However, during ontology modelling it is important to use the most relevant super-concept as the first one in the list of super-concepts.

Another aspect that has to be dealt with is the possibility to re-define concept attributes within WSML. To illustrate this scenario Listing 88 provides a sample ontology where the property `attribute1` of the concept `RootA` is defined to be of type `TypeA`. In the concept `SubConceptOfRootA`, however, this attribute is re-defined to be of type `TypeB`. Thus, `RootA` as well as its sub-concept `SubConceptOfRootA` define the same property but with a different type. Generally concepts in WSMO ontologies can re-define properties that are already defined in any of their super-concepts with an arbitrary new type. However, if there exists an instance of such a concept that provides a value for such a re-defined property, then this value has to be a member of all the types along the concept hierarchy that were used to define it. Otherwise an instance of the sub-concept would not be a

valid member of the super-concept(s), which breaks the concept hierarchy. This of course can only be true, if any type that is used in a re-definition is a sub-concept of the type used in the previous definition(s) of the property. This means that property re-definitions like the one shown Listing 88 are basically a refinement or restriction, since the type of the re-defined property (`TypeB`) is a sub-concept of the previously already defined property type (`TypeA`). The value provided for this property in `Instance2` fulfils all the constraints of the concept hierarchy since it is a member of `TypeA` as well.

```
concept TypeA
    theValue ofType _string

concept TypeB subConceptOf TypeA
    anotherValue ofType _string

concept RootA
    attribute1 ofType (0 *) TypeA

concept SubConceptOfRootA subConceptOf RootA
    attribute1 ofType (0 *) TypeB

instance ValueA memberOf TypeA
    theValue hasValue "Value A"

instance ValueB memberOf TypeB
    theValue hasValue "Value B"
    anotherValue hasValue "Another Value"

instance Instance1 memberOf RootA
    attribute1 hasValue ValueA

instance Instance2 memberOf SubConceptOfRootA
    attribute1 hasValue ValueB
```

Listing 88: A sample ontology where an attribute (attribute1) is re-defined with a more specific type in a sub-concept

Nevertheless, it is technically also possible to re-define properties with arbitrary types like shown in Listing 89. Any reasoner would register this ontology without complaining. Yet, there is no way of creating an instance of the concept `SubConceptOfA` that holds a value for property `attribute1` in this case. Like mentioned before any value of a re-defined property has to be a member of all the types of this property in any concept along the hierarchy. In this case however, the set of values that are of type string as well as of type integer is empty! Thus it is quite questionable whether models that contain problematic re-definitions like the one presented in Listing 89 should be considered correct or consistent.

```
concept RootA
    attribute1 ofType _string

concept SubConceptOfA subConceptOf RootA
    attribute1 ofType _int
```

Listing 89: Inconsitent re-definition of a property

One also has to be aware to the fact that the afore-mentioned effects are not only caused by re-definitions of properties within a single branch in the hierarchy graph, but could also be triggered by multiple inheritance. Listing 90 shows a situation where multiple inheritance causes the same effect as the definition in Listing 89.

```
concept RootA
    attribute1 ofType _string

concept RootB
    attribute1 ofType _int

concept SubConceptOfAandB subConceptOf {RootA,RootB}
```

Listing 90: Inconsistent property type caused be multiple inheritance

In this scenario the concept `SubConceptOfAandB` inherits the property `attribute1` of both super-concepts. Thus, again any value of this property has to be a member of the two different types used in the property definitions. In this case this leads to an empty set of possible values. Being aware of this problematic, the question remains how to deal with this in the XSD transformation process.

As pointed out before, the re-definition of a property does not impose any problems if the type used in the re-definition is a sub-concept of the types this property was already defined with in all super-concepts. This, however, means that the set of possible fillers for such a property is restricted to a particular and more specific sub-type of the previously used super-type(s). XML schema also provides a mechanism to derive new types based on restrictions of a so called base type (see section 4.1.2.1 in [198]). This construct, however, can only be used to restrict the domain of a type e.g. to certain exhaustively stated values, string values that mach a particular regular expression or numbers within a given range. Therefore this approach cannot capture the semantics of a re-defined property in the ontology. Generally the type extension mechanism of XSD is quite limited. As demonstrated at the beginning of this section it is only possible to add new elements. Even repeating an existing element with the same type leads to an error. Replacing an element with a new type other than restricting it in the afore-mentioned ways is simply not possible.

The algorithm designed for the transformation process checks for every concept property that is locally defined whether there exists a property with the same name in any of the current concept's super-types. If such a property is found, the type of this previous definition is compared with the local type. If these two types are compatible, which means that they are either identical or the current concept's property type is a sub-concept of the previously defined one, then the locally defined property is removed from the transformation. The rational behind this is the following: According to the ODEG convention every concept that is further refined via sub-concepts is considered to be abstract. Hence, there should never exist any instances of theses concepts in the information space. The intention of the XML schema, however, is to provide a mean for exchanging instances. In any relevant situation the abstract type is represented by an instance of one of its non-abstract sub-concepts. By removing a re-defined property from the transformation in this case, the information about the actual type restriction to a particular sub-type gets lost, and the resulting XML schema type will accept any instance of the super-type. Consequently the resulting schema will be more relaxed than the ontology but the type hierarchy remains correct, which allows for the same simplification via abstraction as in the ontology. The semantic difference in describing the correctness of valid types is considered to be less important than the retention of the concept hierarchy, especially since validation of consistency and plausibility is performed at the semantic level before the instances are transformed into XML.

The second form of property re-definitions that leads to type conflicts should probably be considered a modelling error. Thus the algorithm produces an error message for every such type conflict that is encountered. Nevertheless, the algorithm creates valid schema types for concepts with conflicting properties as well. There are two possible solutions. One approach is to exclude the conflicting properties from the resulting XML like in the case of a valid re-definition. As a consequence the re-definition is simply ignored, which allows for XML instances that may contain values for a conflicting property. Although, there is no chance for a corresponding WSMO instance to exist – at least in a consistent ontology – this creates a valid schema type.

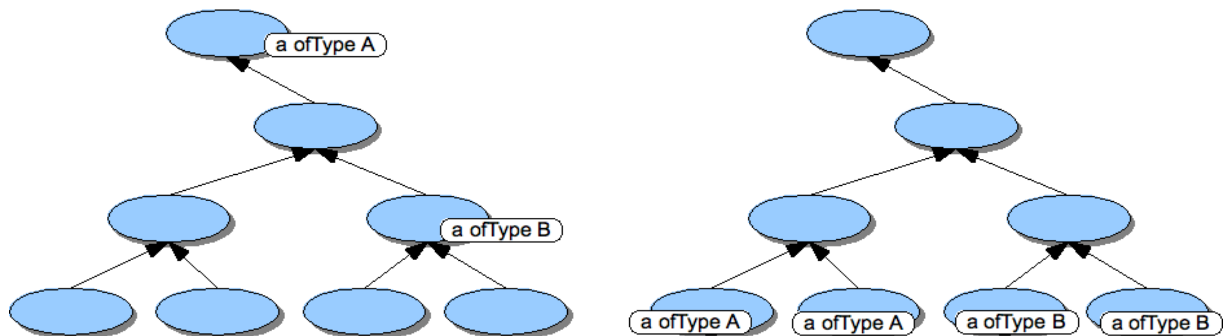


Figure 86: Resolving type conflicts be re-arranging of properties (own illustration)

An alternative approach is to re-arrange conflicting properties in the concept hierarchy. Therefore the property definitions causing the conflicts are removed from their original definition location in the hierarchy and pushed into the leafs of the concept graph. Since ODEG only allows for instances of leaf concepts, this does not change the elements of these instances. In contrast to the first approach, member representations of the concept that caused the conflict by re-defining a property with an incompatible type will inherit the modified type while all other instances will have a property with the same name but of the original type. Once again, a consistent ontology would not allow for instances that contain values for such a conflicting property. Still the XML schema tries to capture the model in a way that might have been intended by the designer. That's why the current implementation of the transformation algorithm uses the latter approach when dealing with conflicting properties.

7.7.2 Generation of WSDL Files

The creation of XML schema types that represent and hold those concept instances that have to be fed to public procedures is a prerequisite for the automatic generation of WSDL files, which act as an entry-point for a web service implementation of the semantically modelled services. Thus, these types can be used to model messages that are exchanged by web service methods. Before the creation process is discussed in more detail, lets recap how a public service is modelled in ODEG.

Conceptually, in the context of ODEG every public service is seen as a single (web service) operation, which takes some input and produces some result. In section 4 semantic web service frameworks have been presented and discussed. Comparing the ODEG approach of relating semantic service descriptions to physical (web) services, it has to be noted that this corresponds to the atomic process elements used in OWL-S. As also mentioned in the discussion in section 4.5 the advantage of this approach is its very intuitive and simple way to provide a grounding for semantic web services. Besides these simple atomic processes, ODEG does not offer any composite constructs, as provided by OWL-S and WSMO. Therefore, there is also no need for describing a service choreography. Although this might be seen as an undue simplification it is nevertheless justifiable for the special requirements of the E-Government domain specifically when focusing on a Citizen-to-Government scenario. A simple atomic process perfectly represents the classical approach of submitting an application and receiving an official notification as a result. Thus, in a typical E-Government scenario where processes are offered to end customers – whether they are citizens or business – there is simply no need for composite process constructs. On the other side, however, every semantic web service that offers several operations that depend on particular call sequences could also be modelled as individual services made up of one atomic process each and the interaction between them is defined by their respective IOPE's. Thus a service the produces and effect or some output that is needed as precondition or input for another service has to be called first. Consequently ODEG's capabilities are not necessarily restricted by focusing on this simple atomic process model.

Since every service represents one operation that takes the application as input and produces for example some notification as output, this type of interaction corresponds to the in-out message exchange pattern presented in section 4.1.2. Due to the processing time of an application, (electronic) public services are typically asynchronous. After an application was submitted it usually takes days or weeks (e.g. in the case of

a building permit application) until the result becomes available.

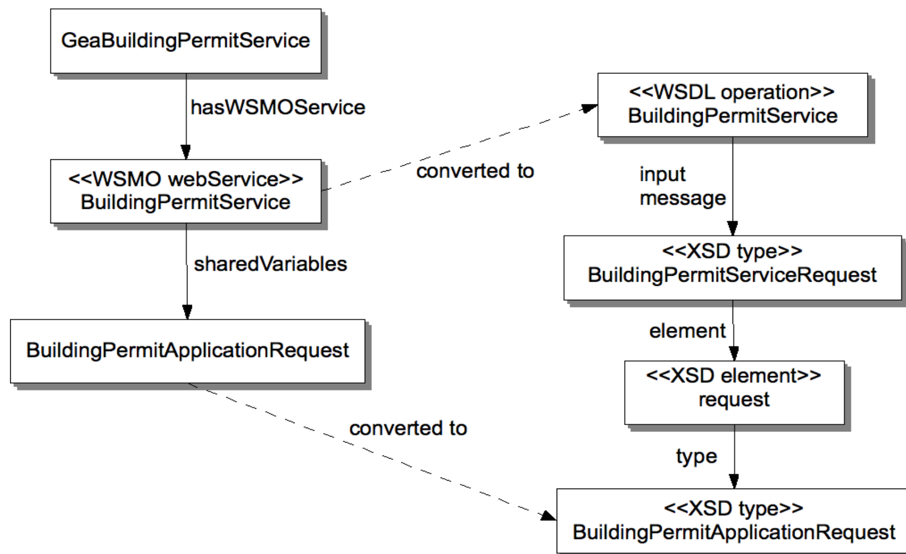


Figure 87: Schematic overview of the WSDL creation process based on the building permit application example (own illustration)

Thus, resulting notifications are often sent via conventional mail or electronic delivery services [199] to the applicant after a significant period of time has elapsed since the actual application was submitted. By default, however, every web service that represents a public service in the ontologies is modelled as an in-out operation and therefore is a synchronous web service.

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://segof.fh-joanneum.at/ConstructionServices"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    name="BuildingPermitService" targetNamespace="http://segof.fh-joanneum.at/ConstructionServices">
    <types>
        <xsd:schema>
            <xsd:import namespace="http://segof.fh-joanneum.at/ConstructionServices"
                schemaLocation="./BuildingPermitService_2.0-SNAPSHOT.xsd" />
        </xsd:schema>
    </types>
    <message name="Capability_BuildingPermitServiceRequest">
        <part element="tns:Capability_BuildingPermitServiceRequest" name="parameters" />
    </message>
    <message name="Capability_BuildingPermitServiceResponse">
        <part element="tns:Capability_BuildingPermitServiceResponse"
            name="parameters" />
    </message>
    <message name="Capability_BuildingPermitServiceFault">
        <part element="tns:Capability_BuildingPermitServiceFault" name="parameters" />
    </message>
    <portType name="Capability_BuildingPermitService">
        <operation name="Capability_BuildingPermitService">
            <input message="tns:Capability_BuildingPermitServiceRequest"
                name="Capability_BuildingPermitServiceRequest" />
            <output message="tns:Capability_BuildingPermitServiceResponse"
                name="Capability_BuildingPermitServiceResponse" />
            <fault message="tns:Capability_BuildingPermitServiceFault"
                name="Capability_BuildingPermitServiceFault" />
        </operation>
    </portType>
</definitions>
```

Listing 91: The automatically created WSDL document

If the nature of the underlying procedure that is triggered via a call to a web service is asynchronous, then the resulting synchronous response of the web service represents an acknowledgement of receipt instead of the actual process result. In the case of a synchronous service, the return type represents the final official notification or whichever result is expected.

The overall idea of how to turn a semantic service description into a WSDL document is presented in Figure 87. Within the semantic model, the service description consists of the GEA based service instance and a corresponding WSMO web-service element. The web-service's capability element refers to one or more concepts of the application domain as so called shared variables. These shared variables are the actual service input as already presented in section 7.5.1. In the case of the building permit service the service input is represented by the `BuildPermitApplicationRequest` concept. During the XML generation process this concept is first turned into a corresponding XML schema type as described in the previous section. Than, based on the definition of the building permit service, a WSDL document is created. This WSDL document contains exactly one operation, which has the same name as the WSMO `webService` element. Therefore in this example the name of the operation will be “BuildingPermitService”. Since - by convention – every public service is mapped to an in-out operation, the created operation is defined with input and output messages as well as with a fault message that can capture error messages produced during service invocation (see Listing 91). The input message to the service operation is an XSD type called `Capability_BuildingPermitServiceRequest`. This type is not inferred from the model but is based on a convention of the transformation process. Its name is the name of the service operation extended by the prefix “Capability_” and the suffix “Request”. As shown in Listing 92 this input data type contains one element called “request”. The type of this element is `BuildingPermitApplicationRequest`, which is the XML representation of the input concept as defined in the semantic service model. Thus the actual input type is wrapped into the request element of a service's input message type. This allows for a unique top level description of all generated web service and therefore simplifies the invocation process of such services.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:segofMessage="http://segof.fh-joanneum.at/schema/messages"
    xmlns:segof_fh-joanneum_at_Construction="http://segof.fh-joanneum.at/Construction"
    xmlns:tns="http://segof.fh-joanneum.at/ConstructionServices"
    targetNamespace="http://segof.fh-joanneum.at/ConstructionServices"
    version="1.0">
    <import namespace="http://segof.fh-joanneum.at/Construction"
        schemaLocation="./segof_fh-joanneum_at_Construction_2.0-SNAPSHOT.xsd" />
    <import namespace="http://segof.fh-joanneum.at/schema/messages"
        schemaLocation="./default/messages_2.0-SNAPSHOT.xsd" />
    <element name="Capability_BuildingPermitServiceResponse" type="segofMessage:return" />
    <element name="Capability_BuildingPermitServiceFault" type="segofMessage:exception" />
    <element name="Capability_BuildingPermitServiceRequest"
              type="tns:Capability_BuildingPermitServiceRequest" />
    <complexType name="Capability_BuildingPermitServiceRequest">
        <sequence>
            <element name="request"
                type="segof_fh-joanneum_at_Construction:BuildingPermitApplicationRequest" />
        </sequence>
    </complexType>
</schema>
```

Listing 92: Default XML schema that is generated for every WSDL document. This example shows the input type created for the building permit service operation.

Currently ODEG's meta-model does not provide any means to define specific return values for public services. Although this could be easily achieved by extending the meta-model, up to now the approach currently used by the frameworks has proven to be sufficient. In this approach the return type is defined in a separate schema file that is automatically attached to the WSDL definition. In this file the type `segofMessage:return` (compare Listing 92) needs to be defined. The default type definitions used for output and fault messages are presented in Error: Reference source not found. Thus, the output element of a service might have some id, which for example could hold a result code, a `messageHeader` to describe the outcome and some arbitrary XML called `messageBody` that represents the actual return value.

Probably the biggest benefit of the automatic generation of XML schema datatypes and WSDL files is the fact that this allows for taking advantage of abstraction almost to the same extend as within the semantic model. As presented in section 7.5.1 abstraction allows for a compact description of the required data. The building permit service for example requires the applicant to describe the construction that is about to be built. The service input concept (see Listing 67 on page 127) therefore can refer to a construction concept that is the super-concept of all things that can be built. In the data elicitation phase based on semantic forms this abstract concept has to be replaced by any concrete subclass. Thus, the construction concept can be seen as a container or placeholder for all possible construction types that have not to be mentioned explicitly.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://segof.fh-
joanneum.at/schema/messages"
    targetNamespace="http://segof.fh-joanneum.at/schema/messages" version="1.0">
    <complexType name="return">
        <sequence>
            <element name="id" type="int" />
            <element name="messageHeader" type="string" />
            <element name="messageBody" type="anyType" />
        </sequence>
    </complexType>
    <complexType name="exception">
        <sequence>
            <element name="id" type="int" />
            <element name="messageHeader" type="string" />
            <element name="messageBody" type="string" />
            <element name="source" type="string" />
        </sequence>
    </complexType>
</schema>
```

Listing 93: Default schema file use to define the output and fault messages of a service operation

On the other side, instances of the concrete sub-concepts will be used, which hold their specific attributes so that no situation-specific information will get lost. Due to the XSD generation process that tries to maintain the type hierarchy this feature can be used on the web service side as well.

```
<complexType name="BuildingPermitApplicationRequest">
        <!--http://purl.org/dc/elements/1.1#description : [concept representing
            input to building permit service] -->
        <complexContent>
            <extension base="tns:ConstructionServiceRequest">
                <sequence>
                    <element maxOccurs="unbounded" minOccurs="0" name="record"
                        type="segofUtil:File" />
                    <element maxOccurs="unbounded" minOccurs="1"
                        name="constructionProject" type="tns:ConstructionProject" />
                    <element maxOccurs="1" minOccurs="0" name="buildingSiteEligibility"
                        type="tns:BuildingSiteEligibilityPlaceHolder" />
                    <element maxOccurs="unbounded" minOccurs="1" name="applicant"
                        type="personData:Person">
                            </element>
                    <element maxOccurs="1" minOccurs="1" name="buildingLocation"
                        type="tns:BuildingLocation">
                    </element>
                    <element maxOccurs="1" minOccurs="0" name="delegate"
                        type="personData:Person">
                    </element>
                </sequence>
            </extension>
        </complexContent>
    </complexType>
```

Listing 94: The XML schema type used as input message to the building permit service

Listing 94 shows the XSD representation of the building permit service's input type. Its structure is very

similar to the corresponding WSML concept and makes use of abstraction. The `ConstructionProject` type for example uses the same abstract types for its attributes as its WSML concept equivalent. Also the `Person` type used here to describe applicants and potential delegates is declared abstract and needs to be replaced by one of its non-abstract sub-types. Since abstract types comprehend all their concrete sub-types, messages can be declared extremely compact and nevertheless their actual values will consist of sub-type instances with all their specific elements, covering a huge variety of different scenarios.

To illustrate this, a snippet of a building permit service SOAP request that was automatically created is shown in Listing 95. While the type of the `applicant` element is defined as `Person` in the schema presented in Listing 94 the actual XML message part is of type `PhysicalPerson`. The same is true for the construction that is of type `BigResidentialHouse` and comes with specific elements like `floorHeight`.

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:segof_fh-joanneum_at_Construction="http://segof.fh-joanneum.at/Construction"
    xmlns:segof_fh-joanneum_at_PersonData="http://segof.fh-joanneum.at/PersonData"
    xmlns:segof_fh-joanneum_at_UtilConcepts="http://segof.fh-joanneum.at/UtilConcepts"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    ns="http://segof.fh-joanneum.at/ConstructionServices"
    xsi:schemaLocation="...">
    <soapenv:Body>
        <ns:Capability_BuildingPermitServiceRequest
            xmlns:ns="http://segof.fh-joanneum.at/ConstructionServices">
            <request>
                <applicant xsi:type="segof_fh-joanneum_at_PersonData:PhysicalPerson">
                    <hasPersonData
                        xsi:type="segof_fh-joanneum_at_PersonData:CompactPhysicalPersonData">
                        <givenName>John</givenName>
                        <familyName>Doe</familyName>
                        <sex xsi:type="segof_fh-joanneum_at_PersonData:Gender">
                            <value>male</value>
                        </sex>
                        <telephoneNumber>0316 123456</telephoneNumber>
                        <eMailAdress>john.doe@doe.com</eMailAdress>
                        <iSOCode3>AUT</iSOCode3>
                    </hasPersonData>
                    <hasAdressData
                        xsi:type="segof_fh-joanneum_at_PersonData:CompactPostalAdressData">
                        <municipality>Graz</municipality>
                        <postalCode>8010</postalCode>
                        <streetName>Inffeldgasse</streetName>
                        <buildingNumber>16a</buildingNumber>
                    </hasAdressData>
                </applicant>
                <constructionProject
                    xsi:type="segof_fh-joanneum_at_Construction:BuildingPermitConstructionProject">
                    <construction
                        xsi:type="segof_fh-joanneum_at_Construction:BigResidentialHouse">
                        <constructionDescription>This is the description
                        </constructionDescription>
                        <totalArea>220</totalArea>
                        <effectiveArea>150</effectiveArea>
                        <numberFloors>2</numberFloors>
                        <floorHeight>2.6</floorHeight>
                        <hasNeighbourSignatures>false</hasNeighbourSignatures>
                    </construction>
                    <constructionProjectType xsi:type=
                            "segof_fh-joanneum_at_Construction:NewBuildOrExtensionWithoutChange">
                        <hasChangeOfSiteDensity>false</hasChangeOfSiteDensity>
                        <hasChangeOfDriveway>false</hasChangeOfDriveway>
                    </constructionProjectType>
                </constructionProject>
                ...
```

Listing 95: Part of an automatically created SOAP call of the building permit service

Generally, the resulting XML schema and WSDL files are a starting point for implementing a web service that

provides services described in the semantic model. Although there exist numerous frameworks for web service creation, the next chapter describes an approach that will probably add the most value to ODEG based service implementation.

7.8 Implementing ODEG web services

As pointed out in the previous sections, ODEG proposes a simple view on public E-Government services. Every service represents an electronic interface to the process providing the actual service as illustrated in Figure 88. The left side shows the situation of a synchronous service, whereas the right side sketches the situation of an asynchronous service. Every electronic service therefore triggers a process that is either entirely automated or involves some human interaction. Every process typically utilises other processes to accomplish its task. These utilised processes are then called sub-processes in the context of the consuming process and are indicated by the grey arrows in Figure 88.

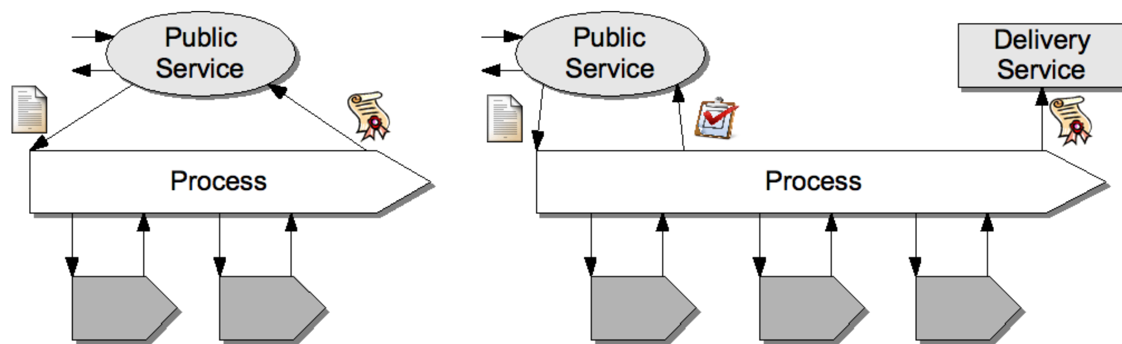


Figure 88: Schematic view on services and underlying processes (own illustration).

On the other side, every process is exposed via its service interface. From a client's perspective, the process itself is seen as a black box since no details about the process' structure will be revealed via its interface. In the context of ODEG this service interface is the WSDL file that was automatically generated based on the semantic model of a service. Consequently, implementing a web-service that fulfils this WSDL interface means to implement the process executed underneath. As discussed in section 4.5.2 OWL-S as well as WSMO provide some means to describe processes, however, both approaches are suffering from significant shortcomings. OWL-S uses rule languages to describe the control flow. Although syntactically embedded into OWL, these rules require additional tooling for interpretation. The WSMO specification on the other side uses abstract state machines to model processes, but this part of the specification is not adopted by the latest available WSML implementation. Generally semantic frameworks are great in describing the state of a problem domain in the terms of ground facts and axioms that can be used to subsume and entail additional facts, but are not very apt to model dynamic aspects like processes. Thus is seems natural to adopt a technology that is better suited for the definition and execution of processes. Again taking the situation shown in Figure 88, a process that makes intensive use of sub-process is reaching its goal by combining these sub-processes according to the actual business needs. In this case the choreography of this process is describing the orchestration of its sub-process in order to reach some goal. Being able to deal with such a scenario is one of the design goals of an approach called Business Process Execution Language for Web Services (BPEL4WS):

> *"BPEL4WS should define business processes that interact with external entities through Web service operations defined using WSDL 1.1 and that manifest themselves as Web services defined using WSDL 1.1..." ([200], page 1)*

BPEL[201] processes communicate to sub-processes via WSDL based interfaces. On the other side, the interface of every BPEL process is also a WSDL operation. Thus BPEL processes are essentially web

services. Therefore it is completely transparent to the caller of such a service whether the implementation is a BPEL process or any other type of web service implementation. Consequently also the WSDL based services used by a BPEL process could be ordinary web services or other BPEL processes as well. This perfectly matches the situation illustrated in Figure 88, where a process is represented by a WSDL operation and makes use of other services to get its task accomplished. Since an excellent introduction to BPEL concepts and its main building blocks can be found in [202] the rest of this section will focus on pointing out the application of BPEL in the ODEG context and the advantages of doing so.

To define the dependencies of a BPEL process and the external services used by it, BPEL defines so called partner links. Each partner link element refers to a service's WSDL document and assigns a logical name to this link. Relevant attributes of a partner link definition are *partnerRole* and *myRole*. One of these two attributes has to be used. If the *myRole* attribute is used, then this link refers to the WSDL file that represents the actual functionality of the current BPEL process. Otherwise the link refers to some service that is needed by this process. Thus, the set of partner links defines the complete external interface of a BPEL process. Figure 89 shows the partner links of the business registration BPEL process.

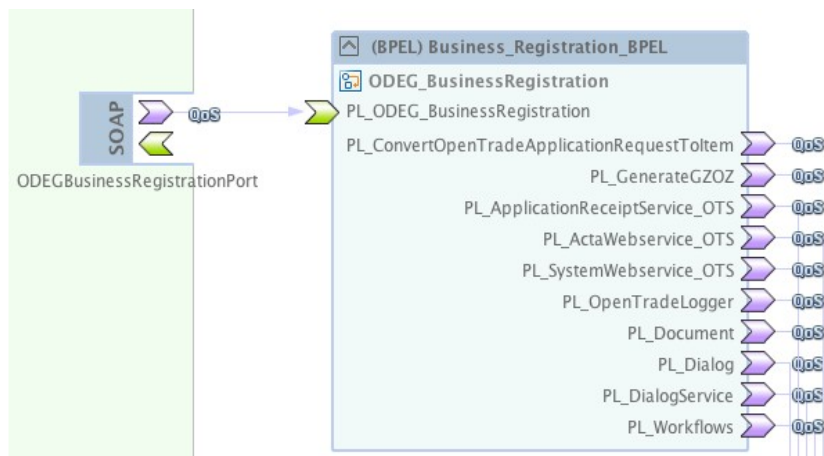


Figure 89: Outside view on a BPEL process and its partner links (Screenshot of the Business Registration Process opened in the Netbeans SOA Module's CASA viewer)

This process was defined based on an automatically generated WSDL file. It represents the process that is invoked when someone wants to register a new business. It can be accessed using the semantic form component and the collected information is sent to the web service endpoint via a SOAP request. The WSDL port and therewith the web service operation that is implemented by the process is shown on the left lane (*ODEGBusinessRegistrationPort*). This is the interface that is exposed to potential clients. The link between the actual process and the business registration port is established via the *PL_ODEG_BusinessRegistration* partner link (indicated by the green ingoing arrow in Figure 89). Generally ingoing arrows on the left side of the BPEL process indicate services that are offered by the process, whereas outgoing arrows on the right side represent those services that are consumed by the BPEL process. These needed services have to be accessible via WSDL defined interfaces. Thus, this approach perfectly fits into a service oriented environment. Clients of the BPEL process only see the provided WSDL file and do not have to take care of all the other partner links.

An overview of the example business registration process used in this section is provided in Figure 90. It interacts with several back-office services that are shown on the right side. The first activity within the process stores the entire application as it comes with the web service request to a database. This task provides a backup of all incoming data so that it could be used in case of a system crash. Records older than 72 hours are automatically deleted. In the next step the received XML request is transformed into another XML format, which represents the required input structures of some of the external services. To perform this transformation, the process uses an external transformation engine. After this transformation the process

logs on to the back-office infrastructure. The security mechanism used here is Kerberos [203]. Token negotiation is invoked transparently at HTTP protocol level using SPNEGO [204]. The BPEL process uses a predefined service account as its identity when interacting with other protected service. Upon connection, the identity service returns a session id that has to be used as part of all other messages. After this session id was successfully retrieved, the process calls the reference number service to get new ids for the file and for every attachment that might be included in the request. In the next step, a new electronic file is created in the document management system. This includes the transmission of all structured data, represented by the XML serialised concept instances that come from the semantic form component. Then all the attachments are added to the document management system and are logically linked to the file. Since all the application data is now available in the back-office, a new workflow is created an started. This will cause a new task to appear on the electronic work-desk of the civil-servant in charge. Thus, the rest of the business registration will be processed manually and controlled by the workflow system in place. After the workflow was triggered, an acknowledgement of receipt is created, which contains the reference number of the new file. The receipt is then digitally signed using the so called MOA-SPSS [205] service.

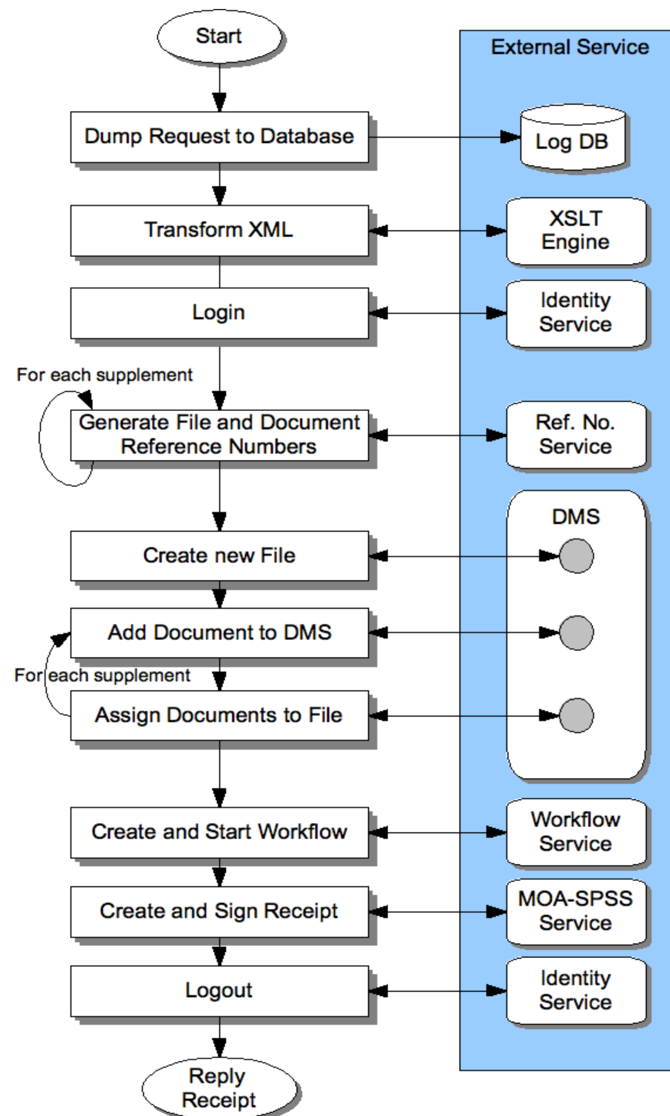


Figure 90: Schematic overview of the business registration BPEL process (own illustration)

MOA stands for modules for online applications and is a set of services that support the creation, use and verification of digital signatures provided by the office of the Austrian CIO. After the receipt was created and signed the current session id is invalidated and the receipt is returned as the response of the BPEL web

service call.

Now that the overall functionality of the process is clear, lets take a closer look at some details of the BPEL process in order to get some insight in the complexity or simplicity of creating such a flow. Figure 91 Shows a fragment of the BPEL process that is responsible for creating a new file in the document management system. The box labelled `scope_Acta` is the equivalent of the “Create new File” activity in Figure 90. The external service is called by the BPEL `invoke` element called `invoke_CreateActa`. An `invoke` element is typically preceded by one or more `assign` elements that set up the input messages that are sent by the invoke element. Using an appropriate editor, these mappings can be done by drag-and-drop operations. Figure 92 shows the mapping that is represented by the assign`_ActaMetaData` assign element in Figure 91.

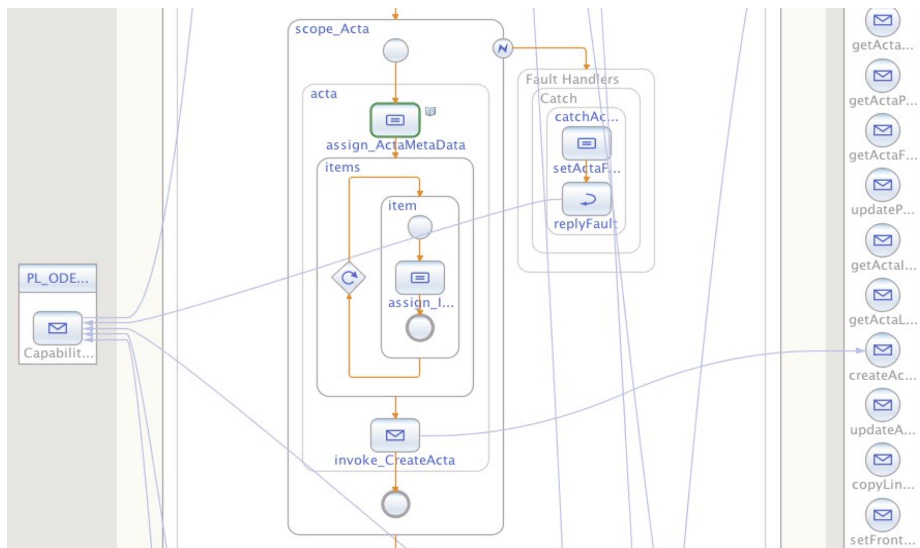


Figure 91: Snippet of the business registration BPEL process showing the creation of a new file (screenshot from Netbeans BPEL Designer)

All variables that are defined within the BPEL process are shown on the left and the right side of the mapping editor. Simple mappings can be created by dragging an element from the left side to one of the right side. In

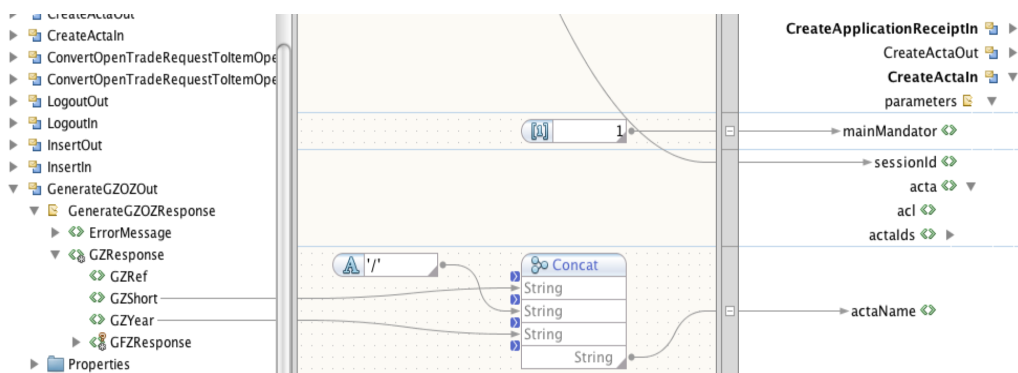


Figure 92: Mapping of part of the message necessary to create a new file

our example the name of the new file will consist of one part of the reference number and the current year separated by a slash character. This can be achieved by adding a string concatenation element that has to be connected with the appropriate input and output elements. Thus, creating a BPEL process is largely a modelling task that can be performed by a skilled business and therefore ideally complements the ODEG approach.

To execute a BPEL process an appropriate run-time infrastructure is needed. ODEG currently uses a

framework called OpenESB/GlassfishESB[24].

"An Enterprise Service Bus (ESB) is a standards-based integration platform that combines messaging, web services, data transformation, and intelligent routing to reliably connect and coordinate the interaction of significant numbers of diverse applications across extended enterprises with transactional integrity"([206], page 1)

OpenESB is an implementation of the Java Business Integration (JBI) specification [207]. A schematic overview of some of its major components is shown in Figure 93.

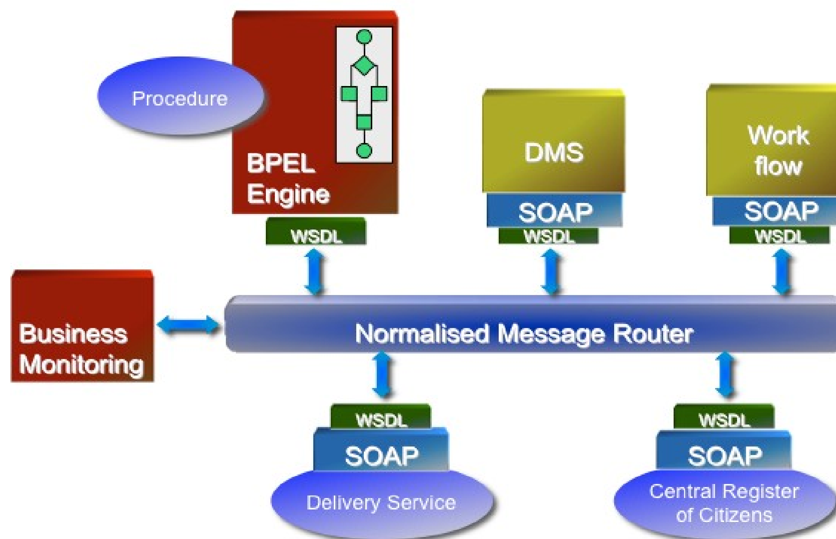


Figure 93: Sample JBI components configuration (own illustration)

The core element of a JBI based ESB is the normalised message router. It uses a unified message protocol that can intelligently route messages, for example based on their content. Besides this, JBI specifies so called service engines. These service engines – like the BPEL or the XSLT engine – are closely tied to the message router and provide services to other components. External services or systems are connected to the message router via so called binding components (BC). These are intelligent interfaces that translate messages between the internal and the application specific protocol. They also expose the functionality of their connected external systems to the rest of the ESB via WSDL documents. JBI implementations typically ship with various standard binding components. There are BCs that can be used to connect to web services, files, databases or other Java programs. Additionally OpenESB provides some monitoring components that can be used to observe the activities on the enterprise service bus.

ODEG is not specifically bound to this framework since it only proposes the use of BPEL for its convenient approach to implement web services as a modelling task. Thus, arbitrary other BPEL or ESB implementations could be used as well. There exist several commercial products but also some additional open source projects like Apache ServiceMix[25], Fuse ESB[26] or Swordfish[27]. Generally the use of BPEL to describe business processes that are in turn exposed as electronic public service seems to be much more practical than trying to capture this information in the semantic model. As found out as a result of the analysis of existing semantic web services (see section 4.5), they show some significant shortcomings when

24 http://www.logicoy.com/OpenESB
25 http://servicemix.apache.org
26 http://fusesource.com/
27 http://www.eclipse.org/swordfish/

modelling the choreography of a web service. Due to the simplifications in the business protocol that are possible for E-Government services (compare Figure 88) there is no need to expose any choreography to the service client, since every public service is eventually represented by one web service operation. BPEL completes ODEG since it is a perfect mean for defining the internal choreography of the service implementation, which is in turn an orchestration of the external services consumed by a BPEL process.

7.9 The Big Picture

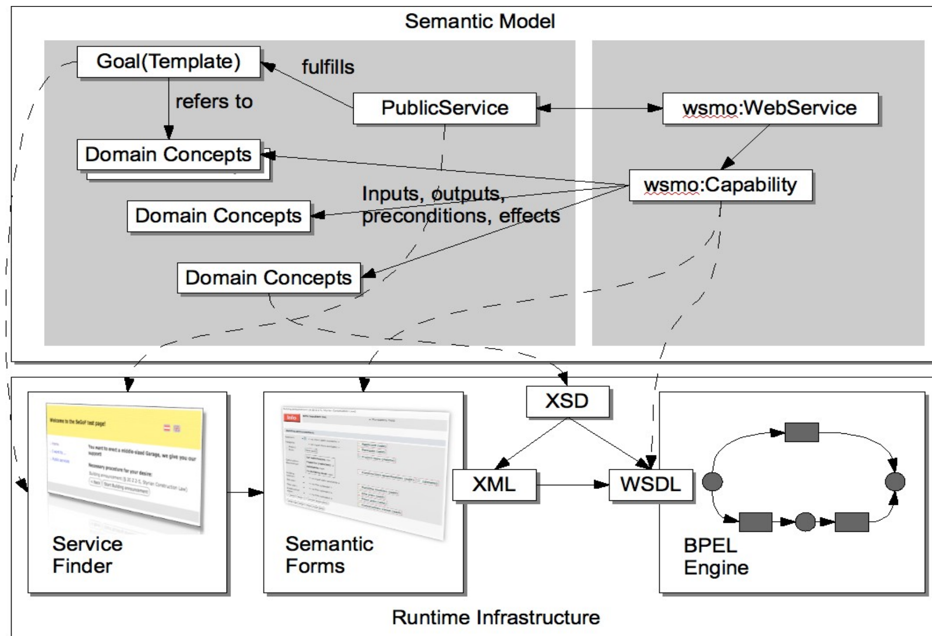


Figure 94: ODEG structural overview (own illustration)

As pointed out at beginning of section 7 the goal of ODEG is to provide support for all important phases during the utilisation of public services. This includes support for identifying relevant services, accessing a service by collecting all the required information needed by the service and the execution of a service by submitting collected and valid data to the service end-point. Figure 94 provides an overview of how all this is achieved. The core of the system is the semantic model that holds several service descriptions. Every service description is split into a GEA and a WSMO part. The GEA part of the service description holds references to all desires/goals that a service might possibly fulfil. Whether a service actually contributes to a desire/goal or not, depends on the concrete desire. A desire is considered to be concrete if all abstract concepts its template is related to are replaced by concrete ones. The service finder or service identification component supports citizens in specifying desires. It therefore makes use of specialisation and classification. Once a service is identified, citizens can access them using the semantic forms component. This component takes the services capability element to figure out the required input. Also here only instances of concrete concepts are allowed as valid input. Thus, if any of the specified input concepts is abstract, it also has to be replaced by a concrete sub-concept first.

To support the implementation of service end-points, ODEG automatically generates WSDL and XML schema documents. Therefore every single service represented by a WSMO capability element is translated into a WSDL file and the entire concept hierarchy that is used to define the types of the messages is translated into XML schema. This allows to take these files as the interface of a BPEL process that can be modelled using appropriate third-party tooling. The run-time component automatically translates collected data into schema compliant XML structures and sends them as messages to the service end-point.

8 Related Work

Semantic technologies have been intensively investigated over the last couple of years and there exist several projects that try to apply these technologies to the field of E-Government. This chapter will present some of these approaches and will compare them to ODEG.

8.1 Goal Oriented Discovery for Semantic Web Service

One very interesting approach that, however, focuses on the service identification phase only is called GODO, which stands for Goal Oriented Discovery for Semantic Web Service [208]. Although GODO is not specifically adapted to E-Government, it uses a similar goal based approach to look-up relevant services, which is why it is discussed here. The ODEG service identification component presented in section 7.4 uses a structured approach to specify desires/goals based on templates and concept taxonomies. This is similar to GODO's approach since it also hosts a goal template repository. These templates, however, play a different role, since they represent certain solution scenarios that might match a user-entered goal description. To get the user's actual goal, GODO offers two different approaches. One is called ontology-guided input. This technique produces structured goal descriptions that are built by the user by creating sentences consisting of predefined terms. These terms are presented to the user, who can select the appropriate one reflecting the current intention. This approach is equivalent to the one currently used by ODEG's service identification components, except that ODEG uses different input elements to select relevant terms. Like ODEG also GODO uses terms that are extracted from ontologies. Alternatively GODO provides an interface to formulate desires in natural language. Such a desire could be a sentence like "I want to buy an airline ticket from London to New York". These goals are processed by a language analyser called KAText[209]. The analyser tries to extract concepts (like "airline ticket") and instances (like "London" and "Heathrow") as well as properties (like "to", "from", "want to buy") and therewith relations. To successfully perform this analysis, the component needs to be intensively trained by domain experts, so that all possible goal formulations are known by the system. This has to be performed for every domain that should be supported by GODO. On the other side, experience with GODO shows that people who start using the system prefer the guided approach since their first results with natural language are relatively poor, whereas more experienced users prefer the natural language approach. This leads to the conclusion that people first learn the terms that are understood by the system. In the case of E-Government, however, where the system is not used that frequently by an individual user, this learning effect will most likely not occur. Thus the natural language approach does not seem to be very promising, when used in the E-Government domain.

8.2 Domain Knowledge-Based Automatic Workflow Generation

Domain Knowledge-Based Automatic Workflow Generation [210] is an approach that was developed in the USA by a team from Rutgers University. It focuses on the creation of workflows that consist of different public services that all have to be invoked in order to achieve a specific goal. In the context of service orientation this kind of workflow is equivalent to a choreography of several services. Although the authors use ontologies as a knowledge base, they do not make use of semantic frameworks but designed their own domain specific language and interpretation for their models. In fact there exist two ontologies. The Service or Task Ontology that contains all services known to the system and the Rules Ontology that contains various rules that are derived from applicable regulations. An example service ontology for the business registration domain is shown in Figure 95. Although this looks similar to an ODEG service hierarchy it is semantically different since the relations between the individual nodes do not represent specialisation (i.e. *subconceptOf*) but indicate that a child-node is a *componentOf* its parent node. Thus the image does not represent a taxonomy but a de-composition of a root element. Consequently the different branches do not indicate available alternatives but are potential parts of a workflow. However, not all components of a node are always needed in order to achieve some particular goal. In order to determine when and how specific nodes or services are actually needed, the rule or regulation ontology is needed.

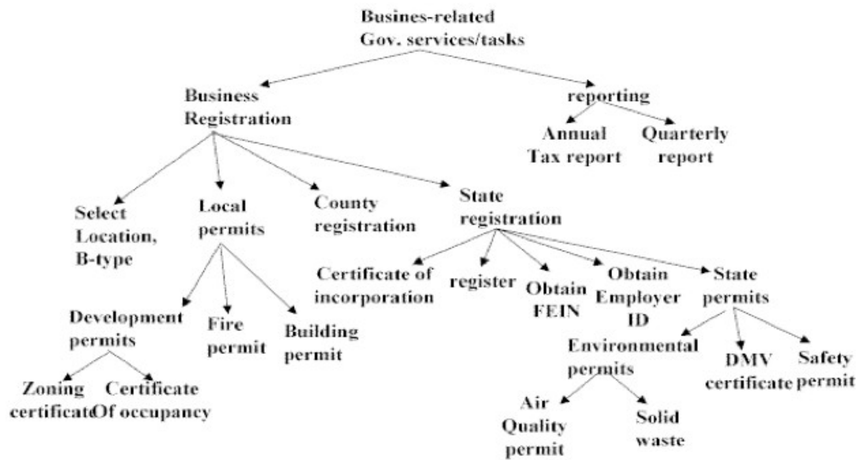


Figure 95: Example service component hierarchy used by the automatic workflow generation approach ([210], page 4)

Figure 96 shows such a rule ontology. The nodes here are organised by a topic/sub-topic relation and every leaf is associated with a rule. Rules consist of conditions and some action that is executed in case the condition evaluates to true. Besides this every rule refers to the text of the actual regulation that applies and one or more services of the service ontology. Actions are used to add a service to the current workflow or changes services that are part of the current workflow.

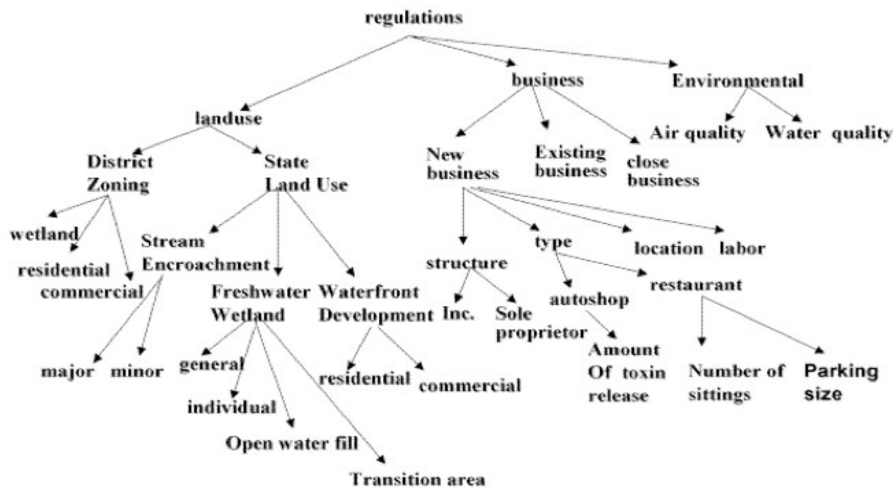


Figure 96: Regulation ontology used for business registration process ([210], page 5)

The actual goal of a specific citizen is captured by a so called user profile. A user profile is a set of attributes that are hierarchically organised in a tree (see Figure 97). Every leaf represents an attribute that holds a value. Determining the goal of a user means to traverse the tree and to associate values to some of the leafs of the user profile. Whether all child-nodes of a node have to be visited (conjunction) or only one of them (disjunction) is indicated by an arc. The attribute values gathered by the user are then used to determine the required services by applying the rules stored in the regulations ontology (e.g. when the structure of the new business is "*incorporated*" then add the services "register business' name" and "file original business certificate" to the current workflow).

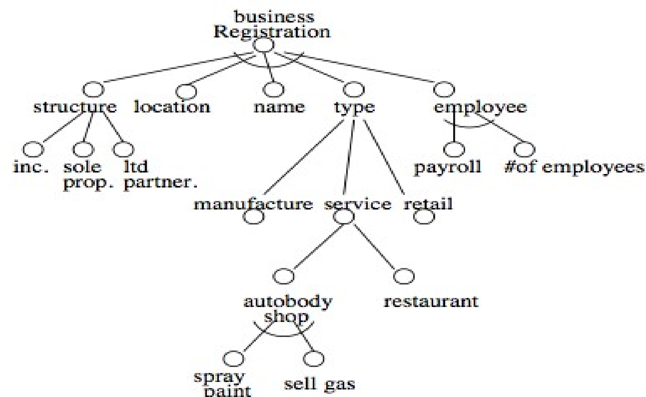


Figure 97: A blank user profile ([210], page 9)

From the users point of view, this approach is almost identical to one provided by ODEG, since also the ODEG service identification component uses a tree to determine the user's desire. This tree is spanned be the current goal template and its related concepts, which are forming the child-nodes of the tree root. The height of the tree is determined by the height of the concept hierarchy ranging from the probably abstract related concept to one of its concrete sub-concepts. Although ODEG does not explicitly add any attributes to a goal template the user might be asked for attribute values as well in case the system can automatically infer a more specific sub-concept based on axioms (classification). In the example used above, there exists a rule that says if the business (*autobody shop*) uses more than half a gallon of paint per hour, the applicant has to apply for a so called air quality permit as well. This is why this attribute occurs in the user profile shown in Figure 97. To achieve equivalent behaviour ODEG would introduce a concept called "*paint emission relevant*" with an attribute "*spray paint (per hour)*" and all businesses that might probably use spray paint will be modelled as sub-concepts of this class. Additionally a classification axiom would classify every "*paint emission relevant*" business as an "a*ir polluting business*" if it uses more than half a gallon of spray paint per hour and as a "n*on-air polluting business*" otherwise. Every "a*ir polluting business*" that is about to be registered will need the "*air quality permit service*" as well (This is achieved by linking the "air quality service" to the "Register business" goal template as well but constraining it to concepts of type "air polluting business"). Thus, if the currently selected business is a sub-concept of "*paint emission relevant*", the service finder component will automatically bring up a dialogue asking for the amount of spray paint used by the business and will call the reasoner to perform the classification.

Although there are significant paradigmatic differences in this approach compared to the one used by ODEG, the functionality is almost equivalent, when it comes to the service identification phase. The workflow generation approach creates complete flows explicitly including the correct order in which the identified services have to be used, whereas ODEG currently only identifies the set of relevant services. On the other side, however, the user profile shown in Figure 97 only contains two concrete businesses (autobody shop and restaurant). A profile that contains all possible options would be tremendously bigger and hard to maintain. Since the presented approach does not support inheritance, all relevant attributes would have to be modelled for every leaf node in which they are needed (e.g. the "spray paint" attribute has to be included to every single business that might use spray paint). Besides the service and the rules ontology, there is no domain ontology describing the elements used in the current application domain. This workflow centred approach could probably be used to provide basic electronic forms (since every service can be mapped to a set of input attributes), but these forms can not be compared to the adaptive forms created by ODEG due to its use of specific domain relevant concepts as service input elements.

8.3 SemanticGov

The SemanticGov[28] project represents a major European effort to incorporate semantic web services in the

28 http://www.semantic-gov.org/

E-Government domain. It also contributed to the creation of the WSMO framework and uses this framework to model ontologies and to implement semantic web services. Another important outcome of this project is the GEA-PA reference model, which was also used by ODEG as an initial starting point for its meta-model. SemanticGov focuses on the integration of electronic public services and therefore distinguishes between so called PEGS (Pan European Governmental Service) and NEGS (National European Governmental Service) [211]. Due to its strong connections with WSMO, it uses state machines (compare section 4.4.1) to describe a sequence of semantic web service calls whenever orchestration of existing services is needed.

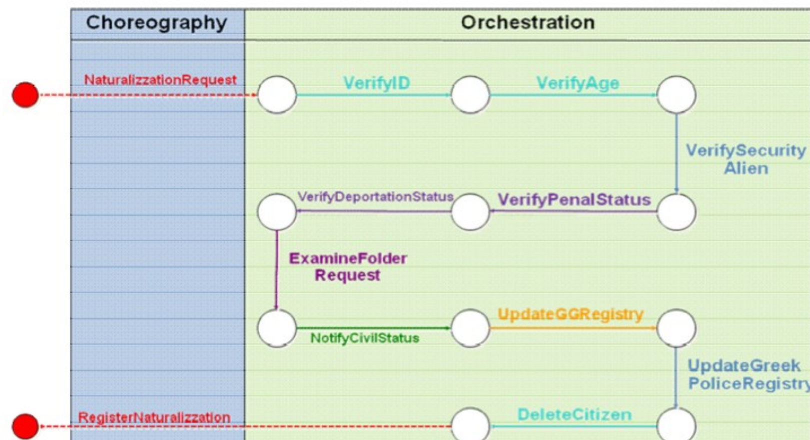


Figure 98: Composing public services based on existing service operations ([211],page 37)

Figure 98 shows an example, where several services are combined to provide two different web service operations. When comparing this scenario to ODEG, the two operations would be implemented by two independent services and the orchestration would be modelled as BPEL processes. BPEL processes are definitely more intuitively to model then state machines, especially when state machines require the strict WSMO formalism as described in section 4.4.1. Taking this example it is also quite questionable whether it is meaningful to expose process internal tasks like *VerifyAge* as an externally accessible semantic web service, since an operation like this is rather integral part of other processes than a public service on its own.

Since the goal notation of WSMO services basically describes a service interface it is not very well suited to express a user's need or desire. Consequently, also SemanticGov introduced its own approach to support citizens in looking-up services that are appropriate for their specific situation. This approach is based on so called goal trees, which are explicitly modelled data structures [212]. SemanticGov therefore introduced the concept *Node*. Every instance of *Node* has a description, holds a reference to its parent node and has some question text and a condition associated to it. To model an actual goal tree, two sub-concepts of *Node* are used: *InternalNode*, which holds references to all its child-nodes and *LeafNode*, which holds a so called post-condition.

Figure 99 shows an example of a goal tree. The tree starts with a root node, representing the overall goal. All internal nodes are grey and all leaf nodes are white. Similar to ODEG, also SemanticGov considers internal nodes to be abstract and only leaf nodes are related to actual services. Thus the tree needs to be traversed till one of the leafs is reached. Thus, every leaf in this tree represents a specific situation a citizen is currently faced with. To decide where to go next while traversing the tree, the user has to answer a question that is explicitly modelled and assigned to every node in the goal tree.

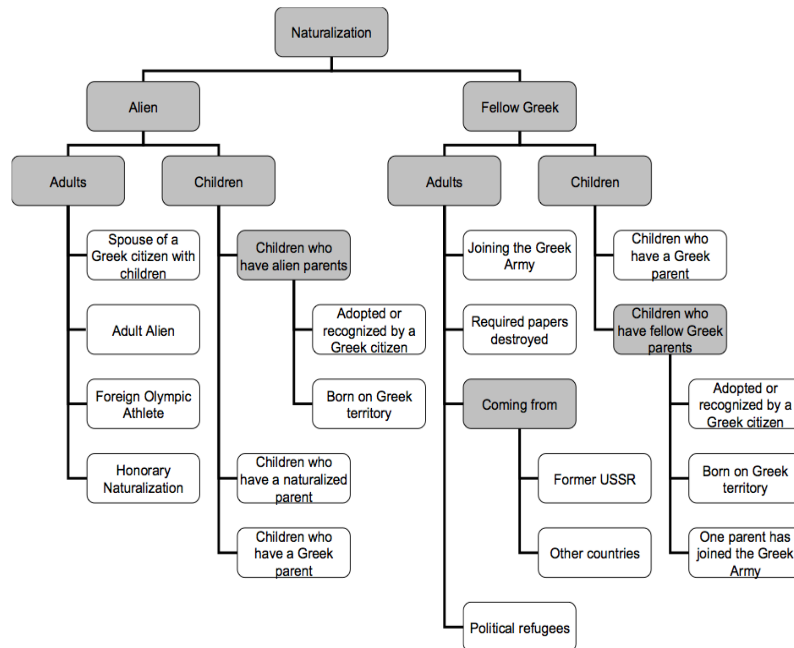


Figure 99: A sample goal tree used for the so called Greek Naturalization Service ([212],page 4)

The actual definition of two of the nodes used in the example goal tree is shown in Listing 96. It is the definition of the “Alien” node used in the left branch of the goal tree and its child-node labelled “Adult Alien”.

```
instance AdultAlien memberOf prtl#InternalNode
    hasDescription hasValue "Alien Service node"
    hasChildNode hasValue AlienCitizen
    hasChildNode hasValue SpouseofaGreekCitizen
    hasChildNode hasValue OlympicAthlete
    hasChildNode hasValue Honorary
    hasParentNode hasValue Alien
    hasQuestion hasValue "Please select the case that best fits your profile:
        (a) Foreign citizen
        (b) Foreign citizen married with a Greek
        (c) Foreign Olympic Athlete
        (d) Honorary Naturalization"
    hasCondition hasValue "IsAdult"

instance AlienCitizen memberOf prtl#LeafNode
    hasDescription hasValue "Alien that wants to get the greek citizenship"
    hasParentNode hasValue AdultAlien
    hasQuestion hasValue "In which region will you execute the service?"
    hasCondition hasValue "IsAlienCitizen"
    hasPostcondition hasValue "?x memberOf co#Citizen and ?x[co#hasCitizenship hasValue co#Greek]"
```

Listing 96: Ontology snippet showing two nodes of the goal tree shown in Figure 98 ([212],page 4)

The question text associated to the internal node called `AdultAlien` is obviously used to directly select one of its child-nodes, whereas the question text of the leaf node is intended to collect necessary information for selecting the actual service provider. The answer to every question is fed to a reasoner and the response leads to the next node. However, besides the definition of custom question texts this approach also requires the creation of custom axioms that can be used to determine the next node, since the reasoner has to know which of the current child-nodes has to be selected when the answer to the current question is “b”. On the other side, it is also possible to register with the SemanticGov platform and to create a so called user profile. If some required information can be inferred from this user profile (e.g. your current nationality is not Greek),

it is used and the question is skipped. Generally, if any plausibility axiom that might be part of the underlying domain ontology fails to validate the provided information, the current user is told not to be eligible for any of the available services.

When comparing the goal tree approach used by SemanticGov to ODEG's desires some important differences can be identified. Since ODED's desires typically refer to abstract super-concepts, the detailed specification of a desire also requires to traverse one or more trees (considering every related concept a root of a type-hierarchy tree). The set of reachable concrete concepts or the combination of such sets in the common case that a desire refers to more than one concept, determines the number of concrete desires. While by linking a few concepts to an ODEG desire template, a relatively large number of concrete desires is covered, the goal tree approach used in SemanticGov requires to model every possible result as an explicit leaf-node. This leads to no less then 18 leaf nodes in the example used above. On the other side, there is no need to explicitly model any questions in ODEG. For example the question of the `AdultAlien` node shown in Listing 96 is almost the same as ODEG's automatically rendered specialisation dialogue. To achieve the same functionality, ODEG would need no more than two concepts that are linked to a *NaturalizationDesire*. One necessary to capture the location of the service and one to determine the state of the applicant. Most of the necessary classification could be done automatically. For example to decide whether the current application is adult or not, one axiom based on the age of the applicant could be used. This would result in a question like “What is your current age?”. In a similar way all the other decisions could be made.

8.4 TerreGov

TerreGov[29] is another EU funded project that tries to adopt semantic technologies in the E-Government domain. Its goals are to provide integrated public procedures that are made up of different, locally distributed available public services and to provide easy access to these services. TerreGov, however, does not focus on citizens as end-users but on civil servants that should be enabled to find the best available services for their clients. The core of TerreGov is the so called eGovernment Interoperability Centre (EGIC) Platform[213], which is shown in Figure 100.

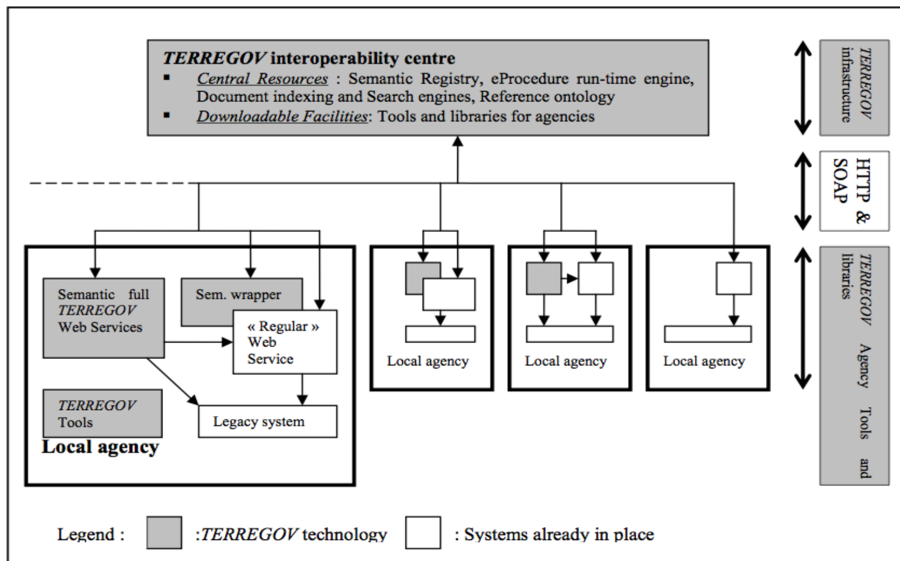


Figure 100: The TerreGov eGovernment Interoperability Centre Platform(EGIC, [213], page 5)

The basic idea is that local agencies expose their services via semantic web service technology. The implementation of these semantic services could be semantically enriched conventional web services or so called “Semantic full TerreGov Web Services”. TerreGov adopts OWL-S (see section 4.2) as its semantic

29 http://www.terregov.eupm.net

web service framework. All of these locally distributed services are registered with the EGIC. The EGIC thus holds a repository of semantic web services that can be searched by civil servant. EGIC also allows domain experts to create new composite services that are based on the already registered ones. Although the initial plan of the project was to create its own composition and modelling tool-stack, TerreGov uses BPEL to define these composite procedures and a BPEL engine to execute them [214].

What distinguishes TerreGov from ODEG is first of all the fact that TerreGov does not aim at directly supporting citizens but civil servants. This stems from its initial application domain which was arranged around social services and welfare. Although TerreGov – like ODEG – uses BPEL to implement semantic web services, there are significant differences between the usage scenarios. While ODEG uses BPEL to define the internal choreography, TerreGov uses BPEL for orchestrating existing semantic web service in order to create new ones.

8.5 SUPER - Semantics Utilized for Process management within and between Enterprises

Although the SUPER[30] project is not directly focusing on E-Government it is also discussed here since it uses a similar technology stack as ODEG. The overall goal of SUPER is to provide a framework that would allow enterprises to easily and quickly adapt their business processes to changing requirements, regulations or business opportunities:

> *"The major objective of SUPER is to raise business process management from the IT level to the business level" ([215],page 43)*

This should be achieved by integrating semantic web services and business processing modelling techniques. Therefore, a new graphical business process modelling tool will allow to compose business processes out of semantically annotated artefacts (see Figure 101). The SUPER BPM (Business Process Management) modelling approach introduces the following phases [215]:

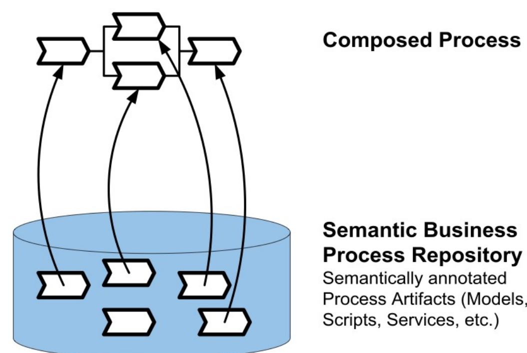


Figure 101: Business process composition based on semantic annotations of services and processes ([215], page 43)

- Semantic Business Process Discovery: This should facilitate the re-use of existing elements by querying the semantic business process repository.
- Semantic Business Process Composition: This will automatically derive executable business processes from conceptual models. This step should allow business experts to model processes without having to take care of technical details.
- Semantic Business Process Mediation: This will enable the integration of heterogeneous artefacts offered by various providers.

30 http://www.ip-super.org/

SUPER calls this approach to business process modelling sBPM (Semantic Business Process Management) [216]. Executable business processes in SUPER are based on a modified version of BPEL called sBPEL (semantic BPEL)[217]. This type of BPEL allows to directly integrate (WSMO) semantic web services in BPEL processes. SBPEL is a WSML ontology consisting of concepts that in turn represent BPEL elements. These concepts, that are based on BPEL's XML schema types were enriched with additional hierarchies, attributes and axioms. This allows that every BPEL process can be represented by sBPEL as well. Processes described in sBPEL are serialised to BPEL4SWS (BPEL for Semantic Web Services) [218], which are then deployed to a so called Semantic Service Bus[219] (see Figure 102).

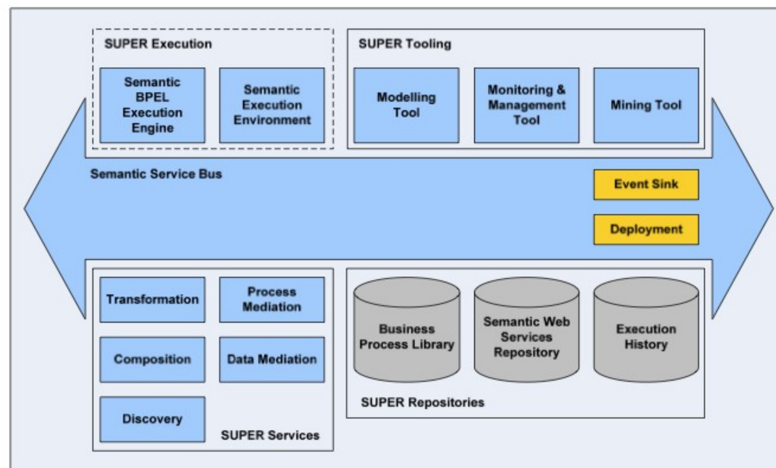


Figure 102: The SUPER architecture ([219], page 7)

The semantic BPEL execution engine basically consists of Apache ODE[31] with some extensions.

Although SUPER as well ODEG try to integrate BPEL and semantic technologies, both approaches have a different focus. While ODEG is aiming at the outside view of processes (by facilitating their identification and their utilisation) SUPER concentrates on the internal structure of processes. In contrast to ODEG, SUPER makes modifications to standards like BPEL in order to achieve its goal. The overall problem that arises here, is the fact that this might hamper the adoption of future releases of systems and BPEL standards and might therefore lead to a lock-in situation. ODEG, however, uses custom models and tools only for those aspects that are not sufficiently covered by existing standards.

8.6 Access-eGov

The Access-eGov[32] project is another EU-funded project that applies semantic technologies to the E-Government domain. Its focus lies on front-office integration which is about supporting citizens in identifying those services that are relevant based on a given situation. The framework does not make any assumptions about the online-availability of public services and can also provide information about conventional face-to-face services that are not electronically accessible at all. From a users point of view the functionality and the results of the underlying system are very similar to the services provided by the workflow centred approach presented in section 8.2. Also this project has decided to use WSMO as its underlying semantic framework.

Access-eGov's approach is centred on extended life events that are structured into goals and sub-goals[220]. When analysing a particular scenario, the ontology modelling approach is not primarily driven by the requirements and constraints of the public services available but by the needs of potential service consumers who are citizens or businesses. Access-eGov therefore calls its modelling methodology *requirements-driven*, which is a seven step procedure for ontology modelling[221]:

1. Identify Informational needs: Prior knowledge of citizens and the diversity of informational needs of

31 http://ode.apache.org/
32 http://www.accessegov.org

different groups of citizens are analysed.

2. Identify required Information Quality (IQ): Depending on the particular needs of the different user groups IQ properties like scope, relevance etc. are determined
3. Create glossary of topics and terms: All topics and terms that are needed to describe the services are added to a glossary
4. Create controlled vocabulary: Based on a glossary a controlled vocabulary is created
5. Group and relate items: The items defined in the controlled vocabulary are grouped and related by predefined relations like “is-input”, “is-output”, “is-reference-to-law” etc.
6. Design an ontology: The previously created items are transformed into a formal WSML ontology
7. Implement semantics: Concepts of the ontologies are used to model WSMO services (webservice capability elements, see section 4.4.1)

This seven step model makes clear that the Access-eGov approach particularly considers the information needs of different types of users when modelling a life event. This is one of the points that distinguishes this project from the other ones discussed so far.

Although Access-eGov adopts WSMO, is also had to modify the service model in oder to keep up with its project goals. The major modifications and extensions are[222]:

- Life Events: These are additional top level elements that represent a specific situation (e.g. building a house) that requires interaction with public agencies. In the notion of Access-eGov a public service is used to achieve a goal (e.g. obtaining a building permit) that is part of the current life event.
- Services: WSMO – as a framework used to describe semantic web services – has its own *webservice* element to model services. However, since Access-eGov also wants to model services that are not electronically available it requires its own implementation neutral notion of a service. This is represented by the service element. Every service has functional properties that describe preconditions and postconditions and non-functional properties that hold information about the service (like service name, service provider, office hours, ...)
- Goals: These elements reflect the requirements of a user when invoking a service. This includes requested output, effects and functionality.

Besides these additional elements Access-eGov has replaced WSMO's abstract state machines by a workflow model to describe a service's choreography. More precisely it still uses WSMO's state signature but instead of transition rules workflow constructs similar to those used within OWL-S are used. Technically every life event in Access-eGov is a *goal* element that specifies WSMO *interface* element (compare section 4.4.1). To illustrate this, Listing 97 shows the definition of the so called “getting married” life event. The goal of this scenario[223] is to assist citizens who want to get married in the German province Schleswig-Holstein and was one of various case studies that were conducted as part of the Access-eGov project.

The actual life event is modelled as a *goal* element that consists of an *interface* specification. The central part of this definition is the *workflow* element. To achieve the overall goal three sub-goals (*ApplyForMarriageGoal*, *WeddingPlaceReservationGoal* and *WeddingCeremonyGoal*) need to achieved. The first step of the workflow, however, requires some input as indicated by the *receive* workflow construct. The type of this input has to be *Q1*. Actually *Q1* is a concept that models various questions (see Listing 98) the user has to answer as the initial part of the *MarriageLifeEvent* workflow. Thus in this example the variable *?q1* will hold the answers to all four questions defined as properties of concept *Q1*. Beside the *workflow* node that describes the various building blocks there are two additional nodes that describe different aspects of the execution of the workflow. The *controlFlow* node describes the sequence and – if necessary – conditions of the execution. In the case of the *MarriageLifeEvent* all four workflow elements are executed strictly sequentially. The *dataFlow* node defines how data is passed along the various workflow nodes. In the example shown in Listing 97 the answers of the user are passed to the first two sub-goals. Additionally the answers are also passed as input *?p1* to a node called *n2_1d*. This node is a decision node within the *ApplyForMarriageGoal*. The ontology snipped shown in Listing 99 illustrates the use of a decision node

together with WSMO's guard conditions. In this example there are three different scenarios.

```
goal MarriageLifeEvent
 nfp
  dc#title hasValue "Marriage"
 endnfp
 interface MarriageLifeEventInterface
  orchestration
   workflow
    perform n1_q1 receive ?q1 memberOf Q1.
     nfp
      aeg#configuration hasValue _boolean("true")
     endnfp
    perform n1_1g achieveGoal ApplyForMarriageGoal
    perform n1_2g achieveGoal WeddingPlaceReservationGoal
    perform n1_3g achieveGoal WeddingCeremonyGoal

   controlFlow
    source n1_q1 target n1_1g
    source n1_1g target n1_2g
    source n1_2g target n1_3g

   dataFlow
    source n1_q1{?q1} target n1_1g{?q1}
    source n1_q1{?q1} target n1_2g{?q1}
    source n1_q1{?q1} target n2_1d{?q1}
    source n2_1o{?a1} target n1_1g{?a1}
```

Listing 97: The MarriageLifeEvent (taken from the file shg/MarriageLifeEvent.wsml available as part of the public deliverable D 7.1 from http://www.accessegov.org)

If the answer to question `q2` of `Q1` (“What is your nationality?”) is “German” then the workflow proceeds with node `n2_q3`.

```
concept Q1
 q1 ofType (1 1) _boolean
  nfp
   dc#title hasValue "Are you 18 years or older?"
  endnfp
 q2 ofType (1 1) Nationality
  nfp
   dc#title hasValue "What is your nationality?"
   dc#description hasvalue  "If you are not German, the system can only provide very limited information
for your case. If you have more than one citizenship and one of them is German, please select German."
   aeg#enumType hasValue _iri("http://www.accessegov.org/ontologies/shg#Nationality")
  endnfp
 q3 ofType (1 1) Municipality
  nfp
   dc#title hasValue "Where is your place of residence in Germany?"
   dc#description hasvalue  "Please enter the place of residence where you would preferably want to get
married. This can be either your primary or your secondary place of residence."
   aeg#enumType hasValue _iri("http://www.accessegov.org/ontologies/shg#Location")
  endnfp
 q4 ofType (1 *) Region
  nfp
   dc#title hasValue "Where do you like to have your wedding ceremony?"
   dc#description hasvalue  "You can choose any location in Germany for this, independently of where
you live."
   aeg#enumType hasValue _iri("http://www.accessegov.org/ontologies/shg#Region")
  endnfp
```

Listing 98: Definition of questions used in the MarriageLifeEvent (taken from the file shg/Concepts.wsml available as part of the public deliverable D 7.1 from http://www.accessegov.org)

If the answer was “Slovakian” then the next node is `n2_Xd` otherwise the workflow will go on with node `n2_2e`.

```
workflow
      …
      perform n2_1d decision
      ...

controlFlow
      source n2_q2 target n2_1d

      source n2_1d target n2_q3 guard ?q1[q2 hasValue iso_3166_deu].
      source n2_1d target n2_Xd guard ?q1[q2 hasValue iso_3166_svk].
      source n2_1d target n2_2e guard neg (?q1[q2 hasValue iso_3166_deu] or ?q1[q2 hasValue
iso_3166_svk]).
      ...
```

Listing 99: Fragment of the ApplyForMarriageGoal showing a decision state (taken from the file shg/ApplyForMarriageGoal.wsml available as part of the public deliverable D 7.1 from http://www.accessegov.org)

It is worth to mention that every goal might consist of sub-goals. Thus, the hierarchy of goals can become arbitrarily deep. As can be seen from the `MarriageLifeEvent` example, especially modelling the data flow is not very intuitive and might easily lead to errors due to the naming convention that has to be used along the goal and sub-goals hierarchy. Whereas the ontologies presented so far are making up the semantic core of Access-eGov the overall system architecture[224] is shown in Figure 103.

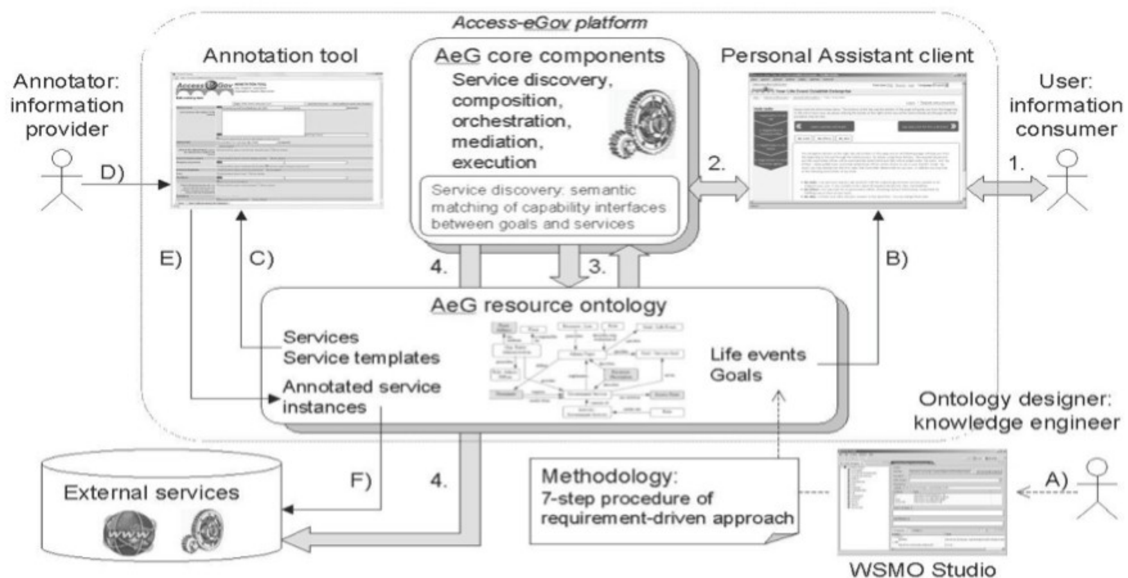


Figure 103: Architecture of the Access-eGov platform ([224], page 3)

The platform offers tools for three different user groups. Ontology designers create the core concepts that are needed to support life events following the requirements-driven modelling approach. The results are ontologies like the ones that have been discussed above. Service providers can use the so called annotation tool to provide semantic annotations for their services regardless whether they are electronically accessible or not. The so called personal assistant client (PAC) is the actual front end that can be used by citizens and businesses. Starting with a life event, users have to answer the questions that are modelled for a life event's workflow. Based in these answers sub-goals are selected that might require answers to additional questions. As a final result the user will be provided with a detailed description of the individual services that are required and the order in which they have to be accessed.

When comparing Access-eGov to ODEG the most obvious difference is the fact that Access-eGov focuses

on the service identification phase whereas ODEG is a comprehensive approach to offer E-Government services. ODEG, however cannot determine an explicit order in which services that match a specific concrete desire have to be invoked. It simply identifies a set of services. On the other side there are also some similarities. The concept of a life event as it is used within Access-eGov is very similar to a desire or goal template as used with ODEG. Although sub-goals that are part of a life event and organised via workflow constructs are conceptually different to a desire template's related concepts, they have a similar consequence from a users point of view, since they might trigger additional questions that have to answered in order to fully specify the user's situation. Whereas these questions are created automatically by ODEG, they have to be explicitly modelled as part of the Access-eGov ontologies. The modelling approach is also different. Access-eGov focuses on life events and identifies flows of services that meet these life events. ODEG's focus is on public services. Based on a service, one or various desire templates that represent goals from a user's point of view are created and mapped to a service. Services can also be mapped to already existing desire templates. Than the selection process is constraint to a specific combination of concrete concepts that are related to the desire templates. Access-eGov is not limited to electronic service but can also provide information about face-to-face services. ODEG can achieve the same functionality by means of so called information services.

9 Conclusion & Outlook

The initial research question of this work was to show whether there exists an approach to combine MDA principles with semantic technologies in order to provide a framework that allows to efficiently create and maintain electronic public services. Therefore several candidate technologies and frameworks were analysed and compared. The capabilities as well as the limits of these frameworks were identified. After this the principles of MDA were examined and discussed. This allowed to identify some key success-factors for any such approach. One of these factors is that the need for any type of manual coding should be eliminated or at least reduced to a minimum, since model generated and manual code always lead to problems in round-trip engineering. Additionally experience with MDA has shown, that approaches, which are limited to well known and clearly defined domains tend to be more successful than more general ones. A third success-factor is the point that the generated functionality has to keep up with features provided by other, manually created applications. Thus, there must not be any trade-off in functionality. In the case of web-based E-Government applications this means that modern web-functionality like auto-completion or sophisticated plausibility checks have to be available. In order to justify the effort for creating a formal model, the functionality created based on this model should cover as much of the entire application as possible.

The presented approach meets all of these success-factors. The coding effort was reduced to the creation of optional auxiliary services. These implementations do not interfere with any other components, thus, there are no update anomalies. As pointed out in section 7.5 the functionality of auxiliary services is integrated dynamically based on existing marks in the model and the availability of these services. Any update of the model will show immediate effect when deployed to the server. There is no need for any programming at all in case of such modifications.

Due to the approach's close alignment to the E-Government domain and its focus on Citizen-to-Government (C2G) scenarios several simplifications compared to general purpose semantic web service frameworks could be achieved. A clear and lean reference model mainly consisting of a *service* and a *desire* concept together with some application domain specific concepts allows for rapid and easy development of new services.

One outstanding feature of ODEG is its service identification component. In section 8 several approaches to look-up (public) services based on a user's goal were presented. All of them focus on a user-centric view when capturing a goal or desire since this is one point that is not sufficiently supported by existing semantic web service frameworks. ODEG tries to decouple the user's point-of-view from the public agency's point-of-view. Thus, it does not include desire's like “I would like to get a building permit” but uses desires like “I want to build *something*”, which better reflects the actual goal of the user. Getting a permit might be just one sub-goal but as we saw in the examples its not even necessary to get a permit in any case. ODEG takes basic intentions and sets them in relation to concepts of the application domain, which forms desire templates. These related concepts are typically abstract according to ODEG's convention of abstract and concrete

concepts in a taxonomy. Thus desire templates like “I want to build a *construction*”, require the user to substitute the abstract concept *construction* with one of its concrete sub-concepts. Basically this requires to traverse a tree like in some of the approaches discussed in section 8. One thing that distinguishes ODEG from these other approaches is that the tree does not need to be modelled explicitly since it is represented by concept hierarchies as they naturally occur in the application domain. This refinement of concepts allows for the unique feature of automatic classification where the user has to answer some automatically generated questions based on the variables of classification axioms. We have seen, how this feature provides graceful alternatives for complex goal trees as they are for example used inside SemanticGov (see section 8.3).

As it was shown in sections 7.4 and 8 ODEG's approach to support service identification is very well suited to assist citizens in finding relevant services even in complex situations. However, what was not covered at all yet, is how a larger number of desire templates could be presented to the end-user in an E-Government portal. Desire templates are bound to particular application domains like building, health care and so on. Currently E-Government portals use so called life-events or business episodes to structure their services in a way that should make them easy to find by citizens [225]. The life-event approach, however, falls short in more complex scenarios when there exist several services that serve the same or at least very similar life-events (like in our building permit example, where depending on the type and size of the building one of three different services might be relevant). To overcome this shortcoming [226] was also one goal of the automatic workflow generation approach and has led to the tree based approach presented in section 8.2. Consequently a combination of life-events and desire templates, where the selection of a life-event leads to the related desire templates, seems an to be an optimal solution. This approach is actually used by one site that makes already use of ODEG.

Another feature, which makes ODEG unique when compared to other approaches is its capability to interactively access its modelled services. As shown in section 7.5 the run-time infrastructure manages to render interactive electronic forms that are automatically and dynamically created based on a service's required input. In fact, there is no longer anything like a form, which is needed as a separate central artefact. The forms used are simply a mean to collect information that is needed in order to invoke a public service. This is a major paradigmatic shift in E-Government, since this approach no longer reflects an electronic version of paper driven procedures but is focusing on services and their prerequisites. The specialisation and classification mechanism used by the semantic forms component to specify the current user's situation furthermore introduces a whole new paradigm of adaptive forms, where the system selectively adapts to the user. Since the system knows whether you are about to build a residential house or a garden wall, it always only collects information – indicated by the attributes of the currently defined concepts – that is relevant. Thus, there is no longer any need for generic general-purpose description fields like they were used on paper forms and probably are still used in most conventional E-Government applications.

The overall design goal of ODEG was to provide a framework for the creation of new electronic public services. There are basically two scopes in doing so. One is to create and provide all elements that are needed to find and access a public service and binding these elements to an already existing service end-point. The other possible scope is to create the implementation of the service end-point as well. This means that an electronic version of a public procedure has to be created. ODEG supports both scopes. Depending on the filler of the *implementationType* attribute of the corresponding *PublicService* instance the nature of the actual service end-point can be defined. The analysis of candidate SWS frameworks (see section 4) revealed that these frameworks show significant shortcomings in modelling business processes. This is based on the general nature of ontologies that are good knowledge bases but fall short in modelling procedural knowledge. This, however, directly effects ODEG's capabilities to keep this procedural knowledge inside the application model. Therefore the decision was made not to include the description of processes inside the semantic model but to delegate this use-case to techniques that more apt to capture processes. The technology that is recommended to be used together with ODEG is BPEL. The same choice was also made by some of the approaches presented in section 8. BPEL fits nicely into ODEG since it is basically representing executable models of processes just like the semantic service model used by ODEG. Unlike the SemanticGov project, ODEG did not make any modifications to BPEL but defines WSDL and XML schemes as the freeze-point between the semantic model and the process model. This allows to use standard BPEL tools and execution engines together with ODEG.

While one focus of MDA is the automatic generation of running applications based on PIMs and PSMs, ODEG has decided to directly interpret the annotated semantic model. This allows to use the capabilities of semantic reasoners during run-time without any data transformation. As it was pointed out in sections 3.1.4 and 3.4.2 algorithms for entailment and stratification show a poor computational complexity. Thus, integrating such algorithms in interactive systems is critical. A crucial factor for acceptable performance is the number instances that are registered with the reasoner, since they are basically representing additional facts that have to be used in substitution steps. Therefore the number of instances has be kept as small as possible. ODEG regularly removes instances that are no longer needed, which improves the overall performance of the system. Another aspect is the number and the form of the axioms available to the reasoner. Axioms including negation-as-failure conditions (indicated by the naf-operator) show significantly worse performance than other axioms. Critical steps are typically those, were entire instance trees have to be validated, which happens when a user completes a semantic dialogue. Thus, ODEG selectively adds axioms to the reasoner whenever they are needed. A general problem with using reasoners is the fact that changes to the information space cannot be performed incrementally. There is no way to simply add a new instance to the reasoner. Instead all ontologies with all concepts, axioms and instances have to be re-registered, which causes the re-evaluation of the information space. Despite these facts, ODEG achieves reasonable response times which are bellow two seconds in the aforementioned critical worst case steps and significantly bellow one second otherwise. Nevertheless, for intensively used web-sites standard up-scaling mechanisms like load-balancing are necessary.

Although the current ODEG framework could prove the research hypothesis, there are still some issues that need further research and improvement. What is still a major issue is the strong mathematical formalism of ontologies. Although the skills needed to create an ODEG compliant model are minimised due to the clear meta-model, ontology modelling requires some significant methodological background. Thus, it would be highly desirable to see some future research with the goal to make ontology modelling more intuitive.

Another aspect that could be improved in follow-up work is usability of the web interface. Data that is collected by the semantic forms component represents a tree structure. Thus a conventional form or wizard paradigm can hardly be used here. Although the current version has done its best to provide an intuitive user interface that is compliant with the Austrian style guide for electronic forms, usability experts could probably come up with new paradigms that are better suited to navigate through tree structures more intuitively.

ODEG has already left the laboratory stage and is currently deployed at two different governmental levels. The Austrian Federal Chancellery uses ODEG's service identification component to support enterprises from other EU countries in offering their services in Austria according to the EU service directive. At municipal level the City of Graz uses all of ODEG's components to run several of their E-Government services. Additional new public services offered by the City of Graz shall be based on ODEG and BPEL.

Bibliography

[1] von Lucke J., Reinemann H., *Speyerer Definition von Electronic Government (german)*, Deutsche Hochschule für Verwaltungswissenschaften, foev.dhv-speyer.de/ruvii/Sp-EGov.pdf, 2000, p1

[2] Layne, K and Lee, J., *Developing Full Functional E-government: A Four Stage Model* in Government Information Quarterly,18 (2) , 2001, pp122-136

[3] Wauters, P. and Van Durne, P, *eGovernment Benchmarking 2005*, European Commission, http://europa.eu.int/information_society/soccul/egov/egov_benchmarking_2005.pdf, 2005, p7

[4] Irani Z., Al-Sebie M., Elliman T., *Transaction Stage of e-Government Systems: Identification of its Location & Importance* in Proceedings of the 39th Hawaii International Conference on System Sciences - 2006, HICSS, 2006, pp1-9

[5] Miller J., Mukerji J. (Ed.), *Model Driven Architecture (MDA)*, OMG, http://www.omg.org/cgi-bin/doc?ormsc/01-07-01.pdf, 2001

[6] OMG, *Meta Object Facility (MOF) Specification*, OMG, http://www.omg.org/docs/formal/02-04-03.pdf, 2002

[7] Alonso G., Casati F., Kuno H., Machiraju V., *Web Services - Concepts, Architectures and Applications*, Springer, 2004

[8] Christensen E., Curbera F., Meredith G., Weerawarana S., *Web Services Description Language (WSDL) 1.1*, W3C, http://www.w3.org/TR/2001/NOTE-wsdl-20010315s, 2001

[9] Clement L., Hately A., von Riegen C., Rogers T. (Ed.), *UDDI Version 3.0.2*, W3C, http://uddi.org/pubs/uddi_v3.htm#_Toc85907968, 2004

[10] McIlraith, Sheila A. and Son, Tran Cao and Zeng, Honglei, *Semantic Web Services* in IEEE Intelligent Systems,2 16, 2001, pp 46-53

[11] James Hendler, *Agents and the Semantic Web* in IEEE Intelligent Systems,2 16, 2001, pp. 30-37

[12] Martin, D., Burstein, M., Mcdermott, D., Mcilraith, S., Paolucci, M., Sycara, K., Mcguinness, D. L., Sirin, E., and Srinivasan, N., *Bringing Semantics to Web Services with OWL-S* in World Wide Web,10 10, 2007, pp 243-277

[13] W3C, *OWL Web Ontology Language - Overview*, W3C, http://www.w3.org/TR/owl-features/, 2004

[14] Jos de Bruijn, Holger Lausen, Axel Polleres and Dieter Fensel, *The Web Service Modeling Language WSML: An Overview*in Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen (Ed.), The Semantic Web: Research and Applications, Lecture Notes in Computer Science, vol. 4011, Springer,2006, pp590-604

[15] Tim Berners-Lee, *Semantic Web Road map*, W3C, http://www.w3.org/DesignIssues/Semantic.html, 1998

[16] Tim Berners-Lee, James Hendler and Ora Lassila, *The Semantic Web - A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities* in Scientific American,5 284, 2001, pp. 34-43.s

[17] Thomas R. Gruber, *Toward Principles for the Design of Ontologies Used for Knowledge Sharing* in International Journal Human-Computer Studies,43 , 1993, p.907-928

[18] Genesereth, M. R., & Nilsson, N. J., *Logical Foundations of Artificial Intelligence*, Morgan Kaufmann Publishers, 1993

[19] Thomas R. Gruber, *Ontology*, , http://tomgruber.org/writing/ontology-definition-2007.htm, 2007

[20] M. Ehrig and P. Haase and N. Stojanovic, *Similarity for ontologies - a comprehensive framework* in In Workshop Enterprise Modelling and Ontology: Ingredients for Interoperability, at PAKM 2004, DEC 2004, , 2004,

[21] Drummond, N. and Shearer, R., *The Open World Assumption*, The University of Manchester, http://www.cs.man.ac.uk/~drummond/presentations/OWA.pdf, 2006

[22] Clark, K. L. 1987, *Negation as failure*in M. L. Ginsberg (Ed.), Readings in Nonmonotonic Reasoning, Morgan Kaufmann Publishers, San Francisco,1987, pp 311-325

[23] Tim Berners-Lee, *Semantic Web - XML2000*, W3C, http://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html, 2000

[24] Berners-Lee T., Fielding R. and Masinter L., *RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax*, IETF, http://www.isi.edu/in-notes/rfc2396.txt, 1998

[25] Allen, Julie D. and Becker, Joe (Ed.), *The Unicode Standard, Version 5.0*, Addison-Wesley Longman, Amsterdam, 2006

[26] Bray T., Paoli J., Sperberg-McQueen C.M. and Maler E., *Extensible Markup Language (XML) 1.0, Second Edition*, World Wide Web Consortium, http://www.w3.org/TR/REC-xml, 2000

[27] Lasila, Ora and Ralph R. Slick (Ed), *Resource Description Framework (RDF) Model and Syntax*, W3C, http://www.w3.org/TR/WD-rdf-syntax-971002/, 2007

[28] Klyne G. and Carroll J.J. (Ed.), *Resource Description Framework (RDF):Concepts and Abstract Syntax*, W3C, http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/#section-data-model, 2004

[29] Brickley D. and Guha R.V., *RDF Vocabulary Description Language 1.0: RDF Schema*, W3Cs, http://www.w3.org/TR/2004/REC-rdf-schema-20040210/, 2004

[30] Manola F. and Miller E. (Ed), *RDF Primer*, W3C, http://www.w3.org/TR/2004/REC-rdf-primer-20040210/s, 2004

[31] Grant J. and Beckett D., *RDF Test Cases*, W3C, http://www.w3.org/TR/2004/REC-rdf-testcases-20040210/#ntriples, 2004

[32] Beckett D. (Ed.), *RDF/XML Syntax Specification (Revised)*, W3C, http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/#section-Data-Model, 2004

[33] Hayes P., *RDF Semantics*, W3C, http://www.w3.org/TR/rdf-mt/#glossInterpretation, 2004

[34] Cahng C.C and Keisler H.J., *Model Theory*, North Holland, 1990

[35] Seaborne, Andy, *RDQL - A Query Language for RDF*, W3C, http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/, 2004

[36] Horroks, I. and Patel-Schneider, P. F. and van Harmelen F., *From SHIQ and RDF to OWL: The Making of a Web Ontology Language* in Journal of Web Semantics,1 1, 2003, 7-26

[37] Dan Connolly, *Web Ontology (WebONT) Working Group Charter*, W3C, http://www.w3.org/2002/11/swv2/charters/WebOntologyCharter, 2004

[38] Mike Dean and Guus Schreiber (Ed), *OWL Web Ontology Language Reference*, W3C, http://www.w3.org/TR/owl-ref/s, 2004

[39] McGuinnes, D. and van Harmelen, F (Ed), *OWL Web Ontology Language Overview*, W3C, http://www.w3.org/TR/owl-features/, 2004

[40] Luke, S., Spector, L., Rager, D., and Hendler, J., *Ontology-based Web agents* in Proceedings of the First international Conference on Autonomous Agents (Marina del Rey, California, United States, February 05 - 08, 1997), , 1997, pp. 59-66

[41] Heflin, J., Hendler, J., and Luke, S, *SHOE: A Knowledge Representation Language for Internet Applications. Technical Report CS-TR-4078 (UMIACS TR-99-71)*, University of Maryland at College Park,1999,pp 1-30

[42] Chaudhri, V.K. and Farquhar, A. and Fikes, R. and Karp, P.D. and Rice, J.P., *Open knowledge base connectivity 2.0. Technical Report KSL-98-06*, Knowledge Systems Laboratory, Stanford,1997,

[43] Horroks, I. et al, *The Ontology Inference Layer OIL*, OIL, http://xml.coverpages.org/OIL-inference.pdf, 2000

[44] Horrocks I., Sattler U., and Tobies S., *Practical reasoning for expressive description logics* in , Springer, 1999, pp 161-180

[45] Baader, F. and Horrocks, I. and Sattler, U., *Description Logics*in Staab, S. and Studer, R. (Ed.), Handbook on Ontologies, Springer Verlag,2004, pp 3-28

[46] Fensel, D. and van Harmelen, F. and Horrocks, I. and McGuinness, D. L. and Patel-Schneider, P. F.,

OIL: An ontology infrastructure for the semantic web in IEEE Intelligent Systems,16 (2) 16, 2001,

[47] Horrocks, I., Fensel, D., Harmelen, F., Decker, S., Erdmann, M, Klein, M., *OIL in a Nutshell*, , http://www.cs.vu.nl/~ontoknow/oil/downl/oilnutshell.pdf, 2000

[48] Bechhofer, S. et al., *An informal description of Standard OIL and Instance OIL*, ontoknowlegde.org, http://www.ontoknowledge.org/oil/downl/oil-whitepaper.pdf, 2000

[49] Hendler, J. and McGuinness, D., *The DARPA Agent Markup Language* in IEEE Intelligent Systems,6 15, 2000, pp 67-73

[50] McGuinness, D., Fikes, R., Stein L.A., Hendler J., *DAML-ONT: An Ontology Language for the Semantic Web*in Fensel, D., Hendler, J., Lieberman, H., Wahlster (Ed.), Spinning the Semantic Web, MIT Press,2003, pp 65-94

[51] Horroks, I. and Patel-Schneider, P. F. and van Harmelen F., *From SHIQ and RDF to OWL: The Making of a Web Ontology Language* in Journal of Web Semantics,1 1, 2003, 7-26

[52] Horrocks, I., *DAML+OIL: a Description Logic for the Semantic Web* in IEEE Data Engineering Bulletin, 25, 2002, pp 4-9

[53] Heflin, J. (Ed), *OWL Web Ontology LanguageUse Cases and Requirements*, W3C, http://www.w3.org/TR/2004/REC-webont-req-20040210/, 2004

[54] Horrocks, I. and Patel-Schneider, P. F., *Reducing OWL Entailment to Description Logic Satisfiability*in Springer (Ed.), Lecture Notes in Computer Science, Volume 2870/2003, Springer Berlin / Heidelberg,2003, 17-29

[55] Motik, B., *On the Properties of Metamodeling in OWL* in Journal of Logic and Computation,4 17, 2007, pp 617-637

[56] Patel-Schneider, P. F. and Horrocks, I., *OWL Web Ontology Language Semantics and Abstract Syntax Section 3. Direct Model-Theoretic Semantics*, W3C, http://www.w3.org/TR/2004/REC-owl-semantics-20040210/direct.html, 2004

[57] Smith, K. and Welty C. and McGuinnes,D.L. (Ed.), *OWL Web Ontology Language Guide*, W3C, http://www.w3.org/TR/2004/REC-owl-guide-20040210/#ComplexClasses, 2004

[58] Motik, B. and Horrocks, I., *Problems with OWL Syntax.*, CEUR-WS.org, http://ceur-ws.org/Vol-216/submission_13.pdf, 2006

[59] W3C OWL Working Group, *OWL 2 Web Ontology Language Document Overview*, W3C, http://www.w3.org/TR/owl2-overview, 2009

[60] Ian Horrocks, Oliver Kutz, and Uli Sattler, *The Even More Irresistible SROIQ* in Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2006), AAAI Press, 2006,

[61] B. Motik, P.F. Schneider-Patel and B. Cuenca Grau (Ed.), *OWL 2 Web Ontology Language Direct Semantics*, W3C, http://www.w3.org/TR/2009/REC-owl2-direct-semantics-20091027/, 2009

[62] B. Motik and P.F. Patel-Schneider and B. Parsia (Ed.), *OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax*, W3C, http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/, 2009

[63] B. Motik, B. Parsia and P.F. Patel-Schneider (Ed.), *OWL 2 Web Ontology Language XML Serialization*, W3C, http://www.w3.org/TR/2009/REC-owl2-xml-serialization-20091027/, 2009

[64] S. Gao, C.M. Sperberg-McQueen, H.S. Thompson (Ed.), *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*, W3C, http://www.w3.org/TR/2009/CR-xmlschema11-1-20090430/, 2009

[65] M. Horridge and P.F. Patel-Schneider, *OWL 2 Web Ontology Language Manchester Syntax*, W3C, http://www.w3.org/TR/2009/NOTE-owl2-manchester-syntax-20091027/, 2009

[66] D. Beckett and T. Berners-Lee, *Turtle - Terse RDF Triple Language*, W3C, http://www.w3.org/TeamSubmission/2008/SUBM-turtle-20080114/, 2008

[67] Christine Golbreich and Evan K. Wallace (Ed.), *OWL 2 Web Ontology Language New Features and Rationale*, W3C, http://www.w3.org/TR/2009/REC-owl2-new-features-20091027/, 2009

[68] B. Motik, B. Cuenca Grau, I. Horrocks, Z. Wu, A. Fokoue and C. Lutz (Ed.), *OWL 2 Web Ontology Language Profiles*, W3C, http://www.w3.org/TR/2009/REC-owl2-profiles-20091027/, 2009

[69] Franz Baader, Sebastian Brandt and Carsten Lutz, *Pushing the EL Envelope* in Proc. of the 19th Joint Int. Conf. on Artificial Intelligence (IJCAI 2005), IJCAI, 2005, pp 346-352

[70] IHTSDO, *Systematized Nomenclature of Medicine - Clinical Terms*, Int. Health Terminology Standards Development Org., http://www.ihtsdo.org/snomed-ct/, 2009

[71] Jos de Bruijn (Ed.), *The Web Service Modeling Language WSML*, wsmo.org, http://www.wsmo.org/TR/d16/d16.1/v0.21/20051005/, 2005

[72] Roman D., Lausen H., Keller U. (Ed.), *Web Service Modeling Ontology (WSMO)*, wsmo.org, http://www.wsmo.org/TR/d2/v1.3/20061021/, 2006

[73] Roman D., de Bruijn J., Mocan A., Toma I., Lausen H., Kopecky J., Bussler Ch., Fensel D., Domingue J., Galizia S. and Cabral L., *Semantic Web Services - Approaches and Perspectives*in Davies J., Studer R., Warren P. (Ed.), Semantic Web Technologies: Trends and Research in Ontology-based Systems, Wiley,2006, pp 191-236

[74] Grosof B.N., Horrocks I., Volz R., Decker S., *Description Logic Programs: Combining Logic Programs with Description Logic* in Proceedings of International Conference on the World Wide Web (WWW-2003), , 2003, 1-10

[75] Steinmetz N., de Bruijn J., *WSML/OWL Mapping*, DERI, http://www.wsmo.org/TR/d37/v0.1/20080125/, 2008

[76] Michael Kifer and Georg Lausen and James Wu, *Logical foundations of object-oriented and frame-based languages* in Journal of the ACM (JACM), Volume 42 , Issue 4 (July 1995) , 1995,

[77] de Bruijn, J., *WSML Abstract Syntax and Semantics*, WSML Working Group, http://www.wsmo.org/TR/d16/d16.3/v1.0/d16.3v1.0_20080808.pdf, 2008

[78] Steinmetz N., Toma I., *WSML Language Reference*, WSML Working Group, http://www.wsmo.org/TR/d16/d16.1/v1.0/20080728/, 2008

[79] de Bruijn J., Lausen H., Polleres A., Fensel D., *The WSML rule languages for the Semantic Web*, W3C, http://www.w3.org/2004/12/rules-ws/paper/128/, 2005

[80] Duerst M. and Suignard M., *Internationalized Resource Identifiers (IRIs)*, IETF RFC3987, http://www.ietf.org/rfc/rfc3987.txt, 2005

[81] Gelfonf M. and Lifschitz V., *The stable model semantics for logic programming* in Proceedings of the Fifth International Conference on Logic Programming, The MIT Press, Cambridge, Massachusetts, 1988, 1070-1080

[82] Austin D. and Barbir A. and Ferris Ch. and Garg S., *Web Services Architecture Requirements*, W3C, http://www.w3.org/TR/2002/WD-wsa-reqs-20020819, 2002

[83] Roberto Chinnica, Jean-Jacques Moreau, Arthur Ryman and Sanjiva Weerawarana (Ed.), *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*, W3C, http://www.w3.org/TR/2007/REC-wsdl20-20070626/, 2007

[84] White, James E., *A high-level framework for network-based resource sharing* in AFIPS '76: Proceedings of the June 7-10, 1976, national computer conference and exposition, ACM, 1976, pp 561-570

[85] Object Management Group, *Common Object Request Broker Architecture: Core Specification*, OMG, http://www.omg.org/cgi-bin/doc?formal/04-03-12.pdf, 2004

[86] A. Gokhale, B. Kumar, A. Sahuguet, *Reinventing the Wheel? CORBA vs. Web Services* in WWW2002 Conference Proceedings, University of Hawaii, 2002, 1

[87] Burner, Mike, *The Deliberate Revolution* in Queue,1 1, 2003, pp 28-37

[88] Mitra, Nilo and Lafon, Yves, *SOAP Version 1.2 Part 0: Primer (Second Edition)*, W3C, http://www.w3.org/TR/2007/REC-soap12-part0-20070427/, 2007

[89] Fielding, Roy T.; Gettys, James; Mogul, Jeffrey C.; Nielsen, Henrik Frystyk; Masinter, Larry; Leach, Paul J.; Berners-Lee, *Hypertext Transfer Protocol -- HTTP/1.1*, IETF, http://tools.ietf.org/html/rfc2616, 1999

[90] Postel, Jonathan B., *Simple Mail Transfer Protocol*, IETF, http://tools.ietf.org/html/rfc821, 1982

[91] Chinnici R., Haas H., Lewis A., Moreau J., Orchard D. and Weerawarana S. (Eds.), *Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts*, W3C, http://www.w3.org/TR/2007/REC-wsdl20-adjuncts-20070626/#patterns, 2007

[92] Lewis, Amelia A. (Ed.), *Web Services Description Language (WSDL) Version 2.0: Additional MEPs*, W3C, http://www.w3.org/TR/2007/WD-wsdl20-additional-meps-20070326/, 2007

[93] Booth, David and Lui, Canyang Kevin (Eds.), *Web Services Description Language (WSDL) Version 2.0 Part 0: Primer*, W3C, http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626/, 2007

[94] Nitzsche, Jörg and Lessen, Tammo van and Leymann, Frank, *WSDL 2.0 Message Exchange Patterns: Limitations and Opportunities* in ICIW '08: Proceedings of the 2008 Third International Conference on Internet and Web Applications and Services, IEEE Computer Society, 2008, pp 168-173

[95] Padmanabhuni S., Chaudhari A.P., Bharti S. and Kumar S., *WSDL 2.0: A Pragmatic Analysis and an Interoperation Framework*, wldj, http://weblogic.sys-con.com/node/219029?page=0,0, 2007

[96] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, K. Sycara, *OWL-S: Semantic Markup for Web Services*, W3C, http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/, 2004

[97] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, K. Sycara, *OWL-S: Semantic Markup for Web Services*, DAML.org, http://www.ai.sri.com/daml/services/owl-s/1.2/overview/, 2008

[98] Horrocks I., Patel-Schneider P., Boley H., Tabet S., Grosof B. and Dean M., *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*, W3C, http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/, 2004

[99] Hirtle D., Boley H., Grosof B., Kifer M., Sintek M., Tabet S. and Wagner G., *Schema Specification of RuleML 0.91*, RoleML.org, http://ruleml.org/0.91/, 2006

[100] Patel-Schneider Peter F., *A Proposal for a SWRL Extensionto First-Order Logic*, DAML.org, http://www.daml.org/2004/11/fol/proposal, 2004

[101] McCermott, Drew, *DRS: A Set of Conventions for Representing Logical Languages in RDF*, DAML.org, http://cs-www.cs.yale.edu/homes/dvm/daml/DRSguide.pdf, 2004

[102] Genesereth, Michael R., *Knowledge Interchange Format - draft proposed American National Standard (dpANS) NCITS.T2/98-004*, Stanford University, http://logic.stanford.edu/kif/dpans.html, 1998

[103] Prud'hommeaux, Eric and Seaborne Andy, *SPARQL Query Language for RDF*, W3C, http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/, 2008

[104] Pèrez, Jorge and Arenas, Marcelo and Gutierrez, Claudio, *Semantics and complexity of SPARQL* in ACM Trans. Database Syst.,3 34, 2009, pp 1-45

[105] Martin D., Burstein M, Lassile O., Paolucci M., Payne T. and McIlraith S., *Describing Web Services using OWL-S and WSDL*, daml.org, http://www.ai.sri.com/daml/services/owl-s/1.2/owl-s-wsdl.html, 2008

[106] Battle S., Bernstein A., Boley H., Grosof B., Gruninger M., Hull R, Kifer M., Martin D., McIlraith S., McGuiness D., Su J. and Tabet S., *Semantic Web Services Framework (SWSF) Overview*, W3C, http://www.w3.org/Submission/2005/SUBM-SWSF-20050909/, 2005

[107] Battle S., Bernstein A., Boley H., Grosof B., Gruninger M., Hull R, Kifer M., Martin D., McIlraith S., McGuiness D., Su J. and Tabet S., *Semantic Web Services Language (SWSL)*, W3C, http://www.w3.org/Submission/2005/SUBM-SWSF-SWSL-20050909/, 2005

[108] Battle S., Bernstein A., Boley H., Grosof B., Gruninger M., Hull R, Kifer M., Martin D., McIlraith S., McGuiness D., Su J. and Tabet S., *Semantic Web Services Ontology (SWSO)*, W3C, http://www.w3.org/Submission/2005/SUBM-SWSF-SWSO-20050909/, 2005

[109] Weidong Chen and Michael Kifer and David S. Warren, *HiLog: A foundation for higher-order logic programming* in Journal of Logic Programming,3 15, 1993, pp 187-230

[110] Yang, Guizhen and Kifer, Michael, *Well-Founded Optimism: Inheritance in Frame-Based Knowledge Bases*in Meersmann R. and Zahir T. et al. (Eds.) (Ed.), On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002, Springer-Verlag,2002, pp 1013-1032

[111] Michael Grüninger, *A guide to the ontology of the process specification language* in Staab, Steffen and Studer, Rudi (Ed.), Handbook on Ontologies, Springer-Verlag,2003, pp 575-592

[112] ISO/IEC, *ISO/IEC 24707 Information technology - Common Logic (CL): a framework for a family of*

logic- based languages, ISO/IEC, http://standards.iso.org/ittf/PubliclyAvailableStandards/c039175_ISO_IEC_24707_2007(E).zip, 2007

[113] Dumitru Roman, Uwe Keller, Holger Lausen, Jos de Bruijn, Rubén Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Christoph Bussler, and Dieter Fensel, *Web Service Modeling Ontology* in Applied Ontology,1 1, 2005, pp 77-106

[114] de Bruijn J., Bussler Ch., Domingue J., Fensel D., Hepp M., Kifer M., König-Ries B., Kopecky J., Lara R., Oren E., Polleres A., Scicluna J. and Stollberg M., *D2v1.4. Web Service Modeling Ontology (WSMO)*, wsmo.org, http://www.wsmo.org/TR/d2/v1.4/20070216/, 2007

[115] Dieter Fensel and Axel Polleres and Joerg Nitzsche, *D14v1.0. Ontology-based Choreography*, wsmo.org, http://www.wsmo.org/TR/d14/v1.0/, 2007

[116] Nathalie Steinmetz and Ioan Toma, *WSML Language Reference*, WSML Working Group, http://www.wsmo.org/TR/d16/d16.1/v1.0/, 2008

[117] James Scicluna (Ed.) and Alex Polleres (Ed.) and Dumitru Roman (Ed.) and Dieter Fensel, *D14v0.2. Ontology-based Choreography and Orchestration of WSMO Services*, wsmo.org, http://www.wsmo.org/TR/d14/v0.2/20060203/, 2006

[118] Egon Börger and Robert Stärk, *Abstract State Machines: A Method for High-Level System Design and Analysis*, Springer Verlag Berlin Heidelberg, 2003

[119] Egon Börger, *High Level System Design and Analysis Using Abstract State Machines* in , Springer, 1998, pp 1-43

[120] Jacek Kopecky and Dumitru Roman and James Scicluna, *D3.4v0.2. WSMO Use Case: Amazon E-commerce Service*, wsmo.org, http://www.wsmo.org/TR/d3/d3.4/v0.2/20060113/, 2006

[121] U. Keller, R. Lara, H. Lausen, and D. Fensel, *Semantic Web Service Discovery in the WSMO Framework*in Jorge Cardoso (Ed.), Semantic Web Services: Theory, Tools and Applications, Idea Publishing Group,2006, pp 281-316

[122] Stollberg, M. and Norton, B., *A Refined Goal Model for Semantic Web Services* in Proceedings of the Second international Conference on internet and Web Applications and Services (May 13 - 19, 2007). ICIW, IEEE Computer Society, Washington, DC, 2007, 17

[123] Uwe Keller, Holger Lausen, and Michael Stollberg, *On the Semantics of Functional Descriptions of Web Services* in Lecture Notes in Computer Science, 4011/2006, 2006, pp 605-619

[124] Stollberg, M., Keller, U., Lausen, H., and Heymans, S, *Two-Phase Web Service Discovery Based on Rich Functional Descriptions*in E. Franconi, M. Kifer, and W. May (Ed.), Lecture Notes In Computer Science vol. 4519, Springer-Verlag, Berlin, Heidelberg,,2007, pp 99-113

[125] Jacek Kopecky and Dumitru Roman and Matthew Moran and Dieter Fensel, *Semantic Web Services Grounding* in Proceedings of the Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services, IEEE Computer Society, Washington, DC, 2006, 127

[126] Kopecky J., Moran M., Vitvar T., Roman D. and Mocan A., *D24.2v0.1. WSMO Grounding*, WSMO, http://www.wsmo.org/TR/d24/d24.2/v0.1/20070427/, 2007

[127] Farell J. and Lausen H., *Semantic Annotations for WSDL and XML Schema*, W3C, http://www.w3.org/TR/2007/REC-sawsdl-20070828/, 2007

[128] Lara R., Roman D., Polleres A. and Fensel D., *A Conceptual Comparison of WSMO and OWL-S*in Springer (Ed.), Lecturer Notes in Computer Science - Volume 3250/2004, Springer Berlin / Heidelberg,2004, pp 254-269

[129] Axel Polleres, Ruben Lara and Dumitru Roman, *D4.2v01 Formal Comparison WSMO/OWL-S*, DERI, http://www.wsmo.org/2004/d4/d4.2/v0.1/20040315/, 2004

[130] Keller U., Lara R., Lausen H., Polleres A. and Fensel D., *Automatic Location of Services*in Springer (Ed.), The Semantic Web: Research and Applications, Springer Berlin / Heidelberg,2005, pp 1-16

[131] Paolucci, Massimo and Kawamura, Takahiro and Payne, Terry R. and Sycara, Katia P., *Semantic Matching of Web Services Capabilities* in ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web, Springer-Verlag, 2002, pp 333-347

[132]Li, Lei and Horrocks, Ian, *A Software Framework for Matchmaking Based on Semantic Web Technology* in Int. J. Electron. Commerce,4 8, 2003, pp 39-60

[133]Naveen Srinivasan and Massimo Paolucci and Katia Sycara, *Semantic Web Service Discovery in the OWL-S IDE* in Proceedings of Hawaii International Conference on System Sciences, IEEE Computer Society, 2006, 109b

[134]Gudgin M. and Lewis A. and Schlimmer J., *Web Services Description Language (WSDL) Version 2.0 Part 2: Message Patterns*, W3C, http://www.w3.org/TR/2003/WD-wsdl20-patterns-20031110/, 2003

[135]Selic, Bran, *The Pragmatics of Model-Driven Development* in IEEE Software,5 20, 2003, pp 19-25

[136]Miller, Joaquin and Mukerji, Jishnu (Eds.), *MDA Guide Version 1.0.1*, OMG, http://www.omg.org/docs/omg/03-06-01.pdf, 2003

[137]OMG, *OMG Unified Modeling LanguageTM (OMG UML), Infrastructure Version 2.2*, OMG, http://www.omg.org/spec/UML/2.2/Infrastructure/PDF/, 2009

[138]OMG, *MOF 2.0/XMI Mapping, Version 2.1.1*, OMG, http://www.omg.org/spec/XMI/2.1.1/PDF/index.htm, 2007

[139]OMG, *Common Warehouse Metamodel (CWM) Specification*, , http://www.omg.org/spec/CWM/1.1/PDF/, 2003

[140]ISO, *ISO/IEC 10746-2 Information technology -- Open Distributed Processing -- Reference Model: Foundations*, ISO, http://www.iso.org/iso/catalogue_detail.htm?csnumber=18836, 1996

[141]Frankel, David S., *Model Driven Architecture - Applying MDA to Enterprise Computing*, Wiley Publishing Inc., 2003

[142]Debnath, N., Leonardi, M. C., Mauco, M. V., Montejano, G., and Riesco, D., *Improving Model Driven Architecture with Requirements Models* in Proceedings of the Fifth international Conference on information Technology: New Generations (April 07 - 09, 2008). ITNG., IEEE Computer Society, Washington, DC, 2008, 21-26

[143]Meertens, L. O., Iacob, M. E., and Nieuwenhuis, L. J., *Goal and model driven design of an architecture for a care service platform* in Proceedings of the 2010 ACM Symposium on Applied Computing (Sierre, Switzerland, March 22 - 26, 2010). SAC '10, ACM, New York, NY, 2010, 158-164

[144]Garrido, J. L., Noguera, M., González, M., Hurtado, M. V., and Rodríguez, M. L., *Definition and use of Computation Independent Models in an MDA-based groupware development process* in Sci. Comput. Program.,1 66, 2007, 25-43

[145]Poernomo, I. and Tsaramirsis, G. and Zuna V., *A methodology for requirements analysis at CIM level* in Proceedings of the 1st International Workshop on Business Support for MDA, CEUR-WS.org, 2008, 1-7

[146]Arlow, Jim and Neustadt Ila, *Enterprise Patterns and MDA*, Addison-Wesley, 2004

147: Mall, Rajib, Fundamentals of Software Engineering, 2004

[148]Rodríguez, A. and Fernández-Medina, E. and Piattin, M.i, *CIM to PIM Transformation: A Reality*in Springer (Ed.), Research and Practical Issues of Enterprise Information Systems II, Springer Boston,, pp. 1239-1249

[149]Kherraf, S., Lefebvre, É., and Suryn, W., in Proceedings of the 19th Australian Conference on Software Engineering (March 26 - 28, 2008). ASWEC, IEEE Computer Society, Washington, DC, 2008, 338-346

[150]Mellor, S.J. and Balcer M.J., *Executable UML - A Foundation for Model-Driven Architecture*, Addison-Wesley Longman, Amsterdam, 2002

[151]OMG, *Meta Object Facility (MOF) 2.0 Core Specification*, OMG, http://www.omg.org/cgi-bin/doc?ptc/03-10-04.pdf, 2003

[152]OMG, *Unified Modeling Language: Infrastructure, v2.0*, OMG, http://www.omg.org/spec/UML/2.0/Infrastructure/PDF/, 2005

[153]OMG, *Human-Usable Textual Notation (HUTN) Specification*, Object Management Group, http://www.omg.org/spec/HUTN/1.0/PDF, 2004

[154]Object Management Group, *Meta Object Facility (MOF) 2.0 Query/View/ Transformation Specification*, OMG, http://www.omg.org/spec/QVT/1.0/PDF/, 2008

[155]OMG, *Object Constraint Language*, Object Management Group, http://www.omg.org/cgi-bin/doc?formal/06-05-01.pdf, 2006

[156]OMG, *Ontology Definition Metamodel Version 1.0*, OMG, http://www.omg.org/spec/ODM/1.0/, 2009

[157]ISO/IEC, *ISO/IEC 24707 Information technology - Common Logic (CL): a framework for a family of logic- based languages*, ISO/IEC, Geneve, Switzerland, 2007

[158]Lars Marius Garshol and Graham Moore, *Topic Maps - Data Model*, ISO/IEC, http://www.isotopicmaps.org/sam/sam-model/2008-06-03/, 2008

[159]Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., and Mei, H., *Towards automatic model synchronization from model transformations* in Proceedings of the Twenty-Second IEEE/ACM international Conference on Automated Software Engineering (Atlanta, Georgia, USA, November 05 - 09, 2007). ASE '07, ACM, New York, NY, , pp 164-173

[160]France, R. and Rumpe, B., *Model-driven Development of Complex Software: A Research Roadmap* in 2007 Future of Software Engineering (May 23 - 25, 2007). International Conference on Software Engineering, IEEE Computer Society, Washington, DC, 2007, 37-54

[161]Portier B. and Ackerman L., *Model Driven Development Misperceptions and Challenges*, InfoQ, http://www.infoq.com/articles/mdd-misperceptions-challenges, 2009

[162]Kovari, Peter, *Explore model-driven development (MDD) and related approaches: Applying domain-specific modeling to Model-Driven Architecture*, IBM Technical Library, http://www.ibm.com/developerworks/library/ar-mdd4/, 2007

[163]Cao, Lan and Ramesh, Balasubramaniam and Rossi, Matti, *Are Domain-Specific Models Easier to Maintain Than UML Models?* in IEEE Softw.,4 26, , 19-21

[164]Tanler, Martin, *Visualizing WSML using an UML Profile*, Univ. of Innsbruck, Digital Enterprise Res. Inst., http://www.sti-innsbruck.at/fileadmin/documents/thesis/Wsml2Uml_Final.pdf, 2006

[165]Na, H., Choi, O., and Lim, J., *A Method for Building Domain Ontologies based on the Transformation of UML Models* in Proceedings of the Fourth international Conference on Software Engineering Research, Management and Applications (August 09 - 11, 2006), IEEE Computer Society, Washington, DC, 2006, 332-338

[166]Yuxiao Z., Assmann U. and Sandahl K., *OWL and OCL for Semantic Integration*, Linkoeping University, Dep. of Comp. and Inf. Sc., http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.2.7683&rep=rep1&type=pdf, 2004

[167]Timm, J. T. and Gannod, G. C., *A Model-Driven Approach for Specifying Semantic Web Services* in In Proceedings of the IEEE international Conference on Web Services, IEEE Computer Society, Washington, DC, 2005,

[168]Brambilla, M., Ceri, S., Facca, F. M., Celino, I., Cerizza, D., and Valle, E. D., *Model-driven design and development of semantic Web service applications* in ACM Trans. Internet Technol.,1 8, 2007, 3

[169]White, Stephen and Miers, Derek, *BPMN Modeling and Reference Guide*, Future Strategies Inc., Lighthouse Pt, FL, 2008

[170]Ceri, S., Fraternali, P., Bongio, A., Brambilla, M., Comai, S., and Matera, M., *Designing Data-Intensive Web Applications*, Morgan Kaufmann Publishers Inc., 2002

[171]Pahl C. and Barrett R., *Semantic Model-Driven Development of Service-centric Software Architectures* in Proceedings of the SEMSOA Workshop 2007 onSoftware Engineering Methods for Service-Oriented Architecture, Leibniz University Hannover, Germany, 2007, 31-45

[172]Pahl, Claus, *An ontology for software component matching* in Int. J. Softw. Tools Technol. Transf.,2 9, 2007, 169-178

[173]V. Peristeras and K. Tarabanis, *Reengineering the public administration modus operandi through the use of reference domain models and Semantic Web Service technologies* in 2006 AAAI Spring Symposium, The Semantic Web meets eGovernment (SWEG), Stanford University, California, USA, , 2006,

[174]X.Wang et al, *WSMO-PA: Formal Specification of Public AdministrationService Model on Semantic Web Service Ontology* in Proceedings of the 40th Hawaii International Conference on System Sciences

- 2007, HICSS, 2007, 1-10

[175]V. Peristeras and K. Tarabanis, *Reengineering the pub- lic administration modus operandi through the use of reference domain models and Semantic Web Service technologies* in 2006 AAAI Spring Symposium, The Semantic Web meets eGovernment (SWEG), Stanford University, California, USA, , 2006,

[176]Vitvar T. and Nazit S. (Ed.), *D3.2 - SemanticGov Architecture v2.0*, SemanticGov.org, http://www.semantic-gov.org/index.php?name=UpDownload&req=getit&lid=373, 2007

177: Bernd Stadlhofer, Semantische E-Government Formulare, 2007

[178]Freitter M., Gradwohl N., Denner R., *XML-Schema EDIAKT II*, E-Government Bund-Laender-Gemeinden, http://reference.e-government.gv.at/XML-Schema_zu_Ediakt_II__ediak.739.0.html, 2005

[179]J. M. Boyer, D. Landwehr, R. Merrick, T. V. Raman, M. Dubinko, L. L. Klotz Jr., *XForms 1.0 (Second Edition)*, W3C, http://www.w3.org/TR/xforms/, 2006

[180]J. J. Kratky, K. E. Kelly, K. Wells, S. Speicher, *XML Forms Generator*, IBM, http://www.alphaworks.ibm.com/tech/xfg,, 2006

[181]J. Turner, *Chiba UserGuide*, Chiba Project, http://chiba.sourceforge.net/ChibaUserGuide.pdf, 2008

[182]Terziev I. and Kiryakov A. and Manov D., *D1.8.1 Base upper-level ontology (BULO) Guidance*, SEKT Project, http://proton.semanticweb.org/D1_8_1.pdf, 2005

[183]Marin Dimitrov, Alex Simov, Vassil Momtchev, and Mihail Konstantinov, *WSMO Studio --- A Semantic Web Services Modelling Environment for WSMO* in 4th European conference on The Semantic Web: Research and Applications (ESWC '07), Enrico Franconi, Michael Kifer, and Wolfgang May (Eds.), Springer-Verlag, 2007, 749-758

[184]Vassilios Peristeras, Adrian Mocan, Tomas Vitvar, Sanullah Nazir, Sotirios Goudos and Konstantinos Tarabanis, *Towards Semantic Web Services for Public Administration based on the Web Service Modeling Ontology (WSMO) and the Governance Enterprise Architecture*, DERI, http://library.deri.ie/resource/YhlntUkT, 2006

[185]Brickley, D. and Miller, L., *FOAF Vocabulary Specification 0.98*, foaf-project.org, http://xmlns.com/foaf/spec/20100809.html, 2010

[186]Erich Schweighofer and Andreas Rauber and Michael Dittenbach, *International Conference on Artificial Intelligence and Law* in Proceedings of the 8th international conference on Artificial intelligence and law, ACM, 2001, pp78-87

[187]Biagioli, C., Francesconi, E., Passerini, A., Montemagni, S., and Soria, *Automatic semantics extraction in law documents* in Proceedings of the 10th international Conference on Artificial intelligence and Law, ACM, 2005, pp133-140

[188]Heike Wagner-Leimbach and Gerhard Kainz, *E-Government Styleguide für E-Fomulare*, reference.e-government.gv.at, http://reference.e-government.gv.at/uploads/media/sg-stg_2_1_1_2010-06-24_01.pdf, 2010

[189]Keith Donald and Erwin Vervaet and Jeremy Grelle and Scott Andrews and Rossen Stoyanchey, *Spring Web Flow Reference Guide*, Springsource, http://static.springsource.org/spring-webflow/docs/2.3.0.RELEASE/spring-webflow-reference/pdf/spring-webflow-reference.pdf, 2010

[190]Herbert Leitold, Arno Hollosi, and Reinhard Posch, *Security Architecture of the Austrian Citizen Card Concept* in Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC '02), IEEE Computer Society, Washington, DC, USA, 2002, 391-

[191]Peter Salhofer, David Ferbas, *A Business Process Engine Based E-Government Platform* in Second International Conference on Internet and Web Applications and Services (ICIW'07), IEEE Computer Society, Washington, DC, USA, 2007, 54ff

[192]Jesse James Garrett, *Ajax: A New Approach to Web Applications*, adaptive path, http://www.robertspahr.com/teaching/nmp/ajax_web_applications.pdf, 2005

[193]Rod Johnson et al., *Spring Framework 3.0 Reference Documentation*, Springsource.org, http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/pdf/spring-framework-reference.pdf, 2010

[194] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Softwre*, Addison Wesley, 1994

[195] Michel Klein and Dieter Fensel and Frank van Harmelen and Ian Horrocks, *The relation between ontologies and XML schemas*, Linköping, http://www.comlab.ox.ac.uk/people/ian.horrocks/Publications/download/2001/etai01.pdf, 2001

[196] Dieter Fensel, *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce*, Springer, 2000

[197] David C. Fallside and Priscilla Walmsley (Eds.), *XML Schema Part 0: Primer Second Edition*, W3C, http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/, 2004

[198] Paul V. Biron and Ashok Malhotra, *XML Schema Part 2: Datatypes Second Edition*, W3C, http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/, 2004

[199] Larissa, Naber and Peter, Reichstaedter and Arne, Tauber and Thomas Roessler, *Elektronische Zustellung - Technische Spezifikation 1.2.0 [german]*, e-government.gv.at, http://reference.e-government.gv.at/uploads/media/zusespec_1-2-0_20070425.pdf, 2007

[200] Frank Leymann, Dieter Roller, and Satish Thatte, *Goals of the BPEL4WS Specification*, OASIS, http://xml.coverpages.org/BPEL4WS-DesignGoals.pdf, 2003

[201] Alves et al., *Web Services Business Process Execution Language Version 2.0*, OASIS, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf, 2007

[202] Charlton Barreto et at., *Web Services Business Process Execution Language Version 2.0 Primer*, OASIS, http://docs.oasis-open.org/wsbpel/2.0/Primer/wsbpel-v2.0-Primer.pdf, 2007

[203] J. Kohl and C. Neuman, *The Kerberos Network Authentication Service (V5)*, IETF, http://www.ietf.org/rfc/rfc1510.txt, 1993

[204] L. Zhu, P. Leach, K. Jaganathan and W. Ingersoll, *The Simple and Protected Generic Security Service Application Program Interface (GSS-API) Negotiation Mechanism*, IETF, http://tools.ietf.org/html/rfc4178, 2005

[205] Stabsstelle IKT-Strategie, *Spezifikation Module für Online Anwendungen - SP und SS*, cio.gv.at, http://egovlabs.gv.at/docman/view.php/6/20/MOA-SPSS-1.3.pdf, 2005

[206] David Chappell, *Enterprise Service Bus: Theory in Practice*, O'Reilly Media, 2004

[207] Ron Ten-Hove and Peter Walker, *JavaTM Business Integration (JBI) 1.0*, JCP, http://jcp.org/aboutJava/communityprocess/final/jsr208/index.html, 2005

[208] Juan Miguel Gomez, Mariano Rico and Francisco Garc a-Sanchez, *GODO: Goal Oriented Discovery for Semantic WebServices*, GODO, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.4476&rep=rep1&type=pdf, 2006

[209] Valencia-García R., Ruiz-Sánchez J.M., Vivancos-Vicente P.J., Fernández-Breis J.T., and Martínez-Béjar R., *An incremental approach for discovering medical knowledgefrom texts* in Expert Systems with Applications 26,3 26, 2004, 291-299

[210] Soon Ae Chun, Vijayalakshmi Atluri, and Nabil R. Adam, *Domain Knowledge-Based Automatic Workflow Generation* in Proceedings of the 13th International Conference on Database and Expert Systems Applications (DEXA '02), Springer-Verlag, London, UK, 2002, 81-92

[211] S. Bonomi, M. Mecella, D. Pozzi, V. Colaianni, N. Loutas, *D6.2 Methodology to Design and Develop NEGS and PEGS with SWS Technologies*, SemanticGov, http://www.semantic-gov.org/index.php?name=UpDownload&req=getit&lid=549, 2009

[212] Loutas Nikolaos, Peristeras Vassilios, Goudos Sotirios K., Tarabanis Konstantinos, *Facilitating the Semantic Discovery of eGovernment Services: The SemanticGov Portal* in EDOC Conference Workshop, 2007. EDOC apos;07. Eleventh International IEEEVolume, , 2007, pp. 157-164

[213] Maria Perez and Saúl Labajo and Tom Lyons, *Citizen meets the new semantic eGovernment revolution* in Proceedings of the eChallenges e-2007 Conference & Exhibition, 24-26 October 2007, The Hague, Netherlands, echallenges.org, 2007, 1-8

[214] Terregov Project Team, *D4.6- Revised Definition of TERREGOV Prototype*, terregov.eupm.net, http://80.14.185.155/egovinterop/www.egovinterop.net/Res/8/D4.6%20-%20Revised%20Definition

%20of%20TERREGOV%20Prototype.pdf, 2006

[215] Matthias Born, Christian Drumm, Ivan Markovic, Ingo Weber, *SUPER - Raising Business Process Management Back to the Business Level* in ERCIM News,1 70, 2007, 43-44

[216] J. Frankowski, H. Kupidura, P. Rubach, E. Szczekocka, *Business Process Management for Convergent Services Provisioning Using the SUPER Platform* in International Conference on Intelligence in service delivery Networks (ICIN), Bordeaux, France, ICIN Events Ltd, 2008, 1-6

[217] Jörg Nitzsche, Daniel Wutke, Tammo van Lessen, *An Ontology for Executable Business Processes* in SBPM 2007Semantic Business Process and Product Lifecycle Management, CEUR, 2007, 1-12

[218] Jörg Nitsche and Tammo van Lessen, *BPEL for Semantic Web Services (BPEL4SWS) - final Version*, SUPER, http://www.ip-super.org/res/Deliverables/M24/D1.10.pdf, 2008

[219] Alessio Carenini and Jörg Nitzsche and Tammo van Lessen, *D 4.7 sBPEL to BPEL4SWS Lifting and Lowering*, ip-super, http://www.ip-super.org/res/Deliverables/M24/D4.7.pdf, 2008

[220] P. Bednár, K. Furdík, M. Paralič, T. Sabol, M. Skokan, *Semantic integration of government services - the Access-eGov approach*in P. Cunningham, M. Cunningham (Ed.), Collaboration and the Knowledge Economy: Issues, Applications, Case Studies. Proc. of conference eChallenges 2008, Stockholm, Sweden, IOS Press, Amsterdam,2008, 22 - 24

[221] R. Klischewski and S. Ukena, *Designing semantic e-Government services driven by user requirements* in Electronic Government, 6th International EGOV Conference. Proceedings of ongoing research, project contributions and workshops (September 3-6, 2007, Regensburg, Germany), Trauner Verlag, Linz, Austria, 2007, 133-140

[222] Ralf Klischewski, Stefan Ukena, Karol Furdik, Andrzej Marciniak, Jan Hreno and Marek Skokan, *D7.1: Public administration resource ontologies*, Access-eGov, http://www.accessegov.org/acegov/uploadedFiles/webfiles/cffile_2_20_08_5_50_43_PM.zip, 2007

[223] Bednar, P. et al, *Semantic Integration of eGovernment Services in Schleswig-Holstein* in Electronic Government, 7th International EGOV Conference, LNCS 5184, Springer, 2008, 315-327

[224] K. Furdík, R. Klischewski, M. Paralič, T. Sabol, M. Skokan, *E-Government Service Integration and Provision Using Semantic Technologies* in Electronic Government. Proceedings of Ongoing Research, General Development Issues and Projects of EGOV 09, 8th International Conference, Linz, Austria, August 31 - September 3, 2009, Trauner Verlag, Linz, 2009,

[225] Anamarija Leben, Mateja Kunstelj and Marko Bohanec, *Evaluation of Life-Event Portals: Trends in Developing E-Services Based on Life-Events* in Proceeding of the 4th European Conference on e-Government, 17/18 June 2004, ACI, 2004, 1-15

[226] Efthimios Tambouris, Mirko Vintar and Konstantinos Tarabanis, *A life-event oriented framework and platform for one-stop government: The OneStopGov project* in Proceedings of Eastern European eGov days conference. (19-21 April, Prague), OCG, 2006,