Enabling Semantic Web Services for Desktop Environment

Bachelor-Thesis von Markus Schröder
November 2012

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
The Knowledge Engineering group

Enabling Semantic Web Services for Desktop Environment

Vorgelegte Bachelor-Thesis von Markus Schröder

1. Gutachten: Prof. Johannes Fürnkranz
2. Gutachten: Dr. Heiko Paulheim

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den November 5, 2012

(Markus Schröder)

Abstract

Semantic Web Services is a technology to markup Web Services with semantic annotations therewith machines can read, process and invoke them. This technology can be transferred to the desktop environment. Each console program in a operating system environment can be described as a service. Existing Semantic Web Service technologies can process a declarative goal and invoke console programs automatically to solve this goal. The approach of using normal programs in a desktop environment as Semantic Web Services is called in this research “Semantic Desktop Services”.

Contents

List of Figures

Listings

1 Introduction

Service-oriented approaches gained considerable interest in the World Wide Web. This thesis is about the possibility to acquire these approaches for personal computers. With the help of new Semantic Web Services technologies it is possible to transfer design ideas to the desktop environment. Step by step we will see how the Web Service technology evolves with the Semantic Web to Semantic Web Services and how we can reuse this technology. We will discover that the functionality of Linux is still available but only usable by humans. With the help of Semantic Web Services technology we make the programs machine processable. This enables automated creation of valid command trees that solve a specific goal defined by the user. At the same time we need to extract for this purpose machine processable knowledge. This is done by using Ontologies as a knowledge representation language.

The following is a short summary of the sections:

In section 2 we will cover the definition of Web Services and the basic technologies. At the end of the section we will see how Web Services interact with each other.

Section 3 introduces the Semantic Web. Particularly, this section shows the Layer Cake of the Semantic Web and explains the term Ontology.

Section 4 defines the difference between Web Services and Semantic Web Services. We will go thru the life cycle of Semantic Web Services and introduce WSMO, the Web Service Modeling Ontology, with many WSML code examples. At the end we will investigate how WSMO implements discovery and choreography.

In section 5 we will cover the operating system Linux and why this thesis uses this system as a testbed. We discover a lot of advantages that support the selection.

The last but one section 6 introduces the own approach called “Semantic Desktop Services”. We will see how we transform knowledge and functionality to enable the usage of Semantic Web Services technology. At the closure this thesis shows the programmed prototype called “Engine”, its inner workings and the graphical user interface.

In the conclusion section 7 we will finally sum up all results and make an outlook what can be done in the future.

1.1 Related Work

There are other related works which use semantic annotations.

1.1.1 Generating RDF description of Debian package sources with ADMS.SW

On August this year Oliver Berger posted on his Blog [8] an idea of generating RDF description of Debian package sources. For this purpose he uses ADMS.SW [22] that allows to describe software with meta data vocabulary. As an information source he uses the Debian PTS (Package Tracking System). In [8] we can see an example of a transformation of the apache2 package in Debian.

The goal of this project [8] is to publish RDF data about package information so that a machine can consume and process it. Berger mentioned that more informations could be added, for example binary packages [7].

1.1.2 KOntoR: An Ontology-enabled Approach to Software Reuse

KOntoR [25] is concerned with the reuse of software. Because of representation and retrieval issues KOntoR presents an ontology based approach. The architecture has two key elements: A meta data repository component describes software artifacts with the help of a metaschema and XML. The knowledge component uses an ontology and imports the metaschema and information about software. With SPARQL interesting queries can be performed on the knowledge base: What artifacts dealing with a specific concept, license issues and developer searching. So it is possible to get more implicit knowledge about software. This makes it easier to reuse software in a proper way.

1.1.3 Semantic Desktop

The Semantic Desktop [38] transfers the Semantic Web to desktop computers. The aim is to store all digital information as Semantic Web resources. They are identified by URIs and queryable as RDF graphs. Further more applications use ontologies to communicate. The goal is to have a own Semantic Web for a single user on the computer.

2 Web Services

This thesis will use Semantic Web Service technology, but before we will go into detail we will look at Web Services without semantic annotations to understand the main concepts.

2.1 Definition

"A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network."[1]

To give an example we will take a look at the american multinational electronic commerce Amazon[2]. In Amazon you can search for products, see details to a product and add it to a cart. Later you can buy the products collected in the cart. You, the buyer, must go online and access the site `http://www.amazon.com/` . On web pages you will get the information you want and interact with the websites. This human-machine interaction is called B2C[3] (business-to-consumer). The human uses a Internet Browser to download HTML (Hypertext Markup Language) pages. For Amazon it is important that the user likes the layout of the pages, because the customer should browse a long time and buy a lot. Amazon don't transfer the meaning of a product to the user, because it is absolutely no problem to understand the content.

But Amazon offers not only a Web Interface for customers. The Product Advertising API "provides programmatic access to Amazon's product selection and discovery functionality" [3]. It is possible to access Amazons information with machines by Web Service Technology. We can see a machine-to-machine interaction that is called B2B[4] (business-to-business). To make this real Web Services use standardized and open web technology. A machine can't understand the meaning of something. Amazons Web Service must transfer XML[5] (instead of HTML) as a basic description language.

In the subsections we will see how

- Web Services will be described,
- Web Services will be discovered and
- messages between Web Services will be sent.

2.2 WSDL

WSDL (Web Service Description Language) is a IDL (Interface Definition Language) to "provide an exact and machine readable definition of service interfaces"[6].

The Amazon E-Commerce Service (ECS) is described with WSDL[7]. To understand this description this thesis will explain the five main sections[8]: documentation, types, interfaces, binding and service.

The *documentation* section is thought for humans. Here we will find textual explanations how to use this Web Service and the meaning of different types and operations.

In the *types* section all simple and complex types will be defined with XML Schema [19]. With XML Schema simple data types like integers, strings etc. can be used. These Type definitions will be used to send or receive messages.

Each Web Service provides operations. In the *interface* section all operations are listed with correspond inputs and outputs. Inputs and outputs have types defined in the types section.

1 [41] p. 19
2 `http://www.amazon.com/`
3 [41] p. 16
4 [41] p. 17
5 see section 3.3.2
6 [41] p. 36
7 `http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl`
8 [41] p. 38

At this time we only described a abstract interface with types and operations. In the *binding* section we bind this to a concrete service. Moreover we determine the message format and the message transport protocol.

In the *service* section we define the endpoint. An endpoint is the place where we can access the real functionality of the Web Service.

2.3 UDDI

UDDI (Universal Description, Discovery and Integration) is used to register Web Services. It's a directory of Web Services that can be used by Web Services to find specific Web Services. It's a centralized approach and can be compared with a phone book:[9]

- White Pages contain basic information about a business.
- Yellow Pages are grouped to an industrial categorization.
- Green Pages contain technical information about a service provided by a business.

The aim is to find Web Services with a Web Service. This is important when the mass of Web Services enlarge. UDDI is designed to find a Web Service by keywords and categorizations.

2.4 SOAP

SOAP (Simple Object Access Protocol) is an XML language used for exchanging structured information.[10] Its format contains Header and Body.

Header information is used for ongoing interaction. An example will be a username password authentication or a transaction ID.

In the Body you can specify a message. The format of the message is not defined by SOAP. You can send in the message block whatever you want. Mostly the message contains an XML document that uses a defined XML Schema description.

```
<soap:Envelope (...)>
    <soap:Header>
    (...)
    </soap:Header>
    <soap:Body>
    (...)
    </soap:Body>
</soap:Envelope>
```

Listing 1: Short SOAP example from http://www.w3schools.com/soap/soap_syntax.asp

[9] [41] p. 41
[10] [41] p. 31

2.5 Interaction

In the WSDL section we have seen how Web Services can be described. The UDDI service administrates Web Service descriptions and can be used to discover Web Services. With the help of SOAP we have a structure for messages exchanged by Web Services. Now everything must work together. This interaction-model is called “Web Service Role Model” or the “SOA Triangle” (see fig. 1).[11]

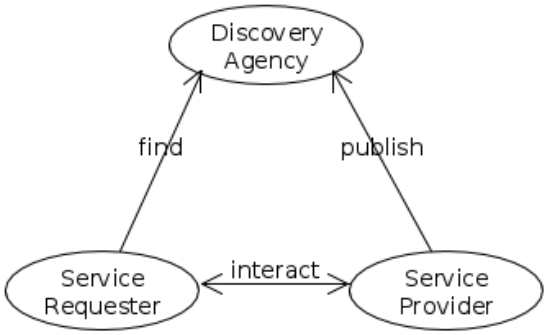

Figure 1: The Web Service Role Model

A service provider describes the service it provides. Amazon is a service provider and we know now that it describes the service with WSDL. A service requester wants to use a service. But first of all the service requester must find a suitable Web Service. With a Discovery Agency (realized by UDDI) a requester downloads WSDL documents describing Web Services. On the other side the service provider used this Discovery Agency to publish its WSDL document.

After the requester has interpret the WSDL document it will interact over SOAP with the service provider.

[11] [41] p. 45

3 Semantic Web

Before we add "Semantic" to Web Services we will cover in this section the Semantic Web. While the Semantic Web holds data in a machine-readable way, the Semantic Web Services process these data and interact in an automated way (machine-to-machine).

3.1 The Web

The World Wide Web is a huge collection of information. Everybody can nowadays create a own website and post information on many sites. From all over the world people can download these information. With keyword search engines like Google[12] we can search for websites with localization of text strings. But because of the overflow of information you frequently can't find what you are looking for. Sometimes information are spread over different sites and you must think about the meaning and link these information.

3.2 The Semantic Web

To solve these problems we have two ways:

With Data Mining [43] we extract information from sites with technologies from the artificial intelligence domain. A machine can interpret information and find patterns in data. With this approach Data Mining uses existing information and makes it accessible for machines.

Another way is the Semantic Web [27]. With a standard for knowledge representation people can create information in a standardized way. We add meaning that can be interpreted by machines. To share "a formal explicit specification of a shared conceptualisation of a domain of interest" [23] the Semantic Web uses Ontologies. The World Wide Web Consortium[13] defines and holds Ontology languages and other Semantic Web specifications. Because Ontologies are a key concept in the Semantic Web domain and later used by Semantic Web Services as a knowledge exchange model we will go deeper into the Semantic Web area.

3.3 The Layer Cake

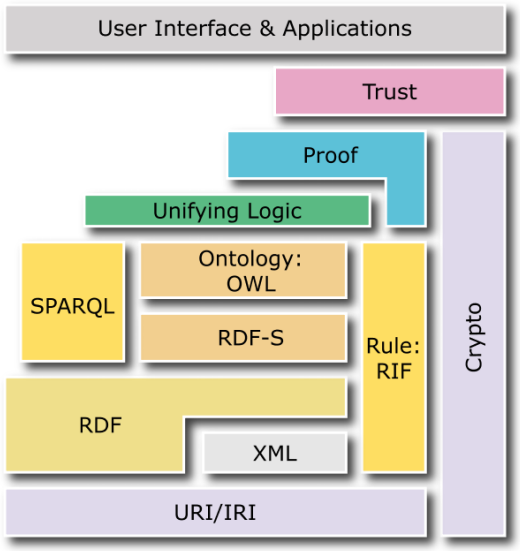

Figure 2: The Layer Cake of the Semantic Web

[12] http://www.google.com/
[13] http://www.w3.org

The Layer Cake of the Semantic Web (fig. 2[14]) expresses how the technologies depend on each other. This thesis will give a short introduction to URI/IRI, XML, RDF and OWL.

3.3.1 URI/IRI

A URI (Uniform Resource Identifier) is a textual string that identify resources in the World Wide Web.[15] Because the World consists of people from different nationalities the resource identification is internationalized: IRI (Internationalized Resource Identifier) "is a sequence of characters from the Universal Character Set (Unicode/ISO 10646)"[16]. Because Semantic Web Services is a new research field it uses the newest identification protocol IRI. But this thesis will not need the internationalization because the scope will be limited to the desktop environment.

We need URIs/IRIs later to identify ontologies, concepts, relations, web services etc.

3.3.2 XML

XML (eXtensible Markup Language) as the name says is extensible. That's why XML can be reused and is a key technology at the bottom of the Layer Cake (fig. 2). Its annotations with tags can add meta data to textual data.[17] The structure enables machines to process the data in an easy way. Nowadays it is the standard for interchange of structured data.

```
<Metadata>Data</Metadata>
```

Listing 2: Simplified XML document that adds meta data to data with surrounded tags

3.3.3 RDF

With RDF (Resource Description Framework) [31] we can express complex directed graph and reuse the representation of XML. Therefore it's easy to describe a semantic network[18] with annotated nodes. Now it is possible to define relations between things identified with URIs/IRIs.[19] We can express triples that contains of subject, predicate and object to create a relationship between subject and object with a relation.[20] Another advantage is the possibility of composition of decentralized information because documents can link to each other. The set of defined and linked individuals is called A-Box (assertion box).

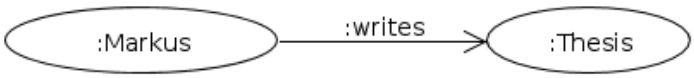

Figure 3: A small RDF graph about Markus who is writing a thesis

In figure 3 we can see an example for an RDF graph. Each node represent a thing or a person in the real world. A directed edge link nodes together if they have a relationship. In this example the fact "Markus is writing a thesis" is expressed with RDF. We call the nodes resources and the "write" edge a property of the "Markus" resource.

[14] taken from http://www.w3.org/2007/Talks/0130-sb-W3CTechSemWeb/layerCake-4.png
[15] [27] p. 26
[16] [18] Abstract
[17] [27] p. 17
[18] [41] p. 53
[19] [27] p. 37
[20] [31] section 3.1

With RDFS (RDF Schema) it is possible to define a terminological knowledge.[21] With the possibility to create a new vocabulary in a domain of interest RDFS becomes a knowledge representation language: An ontology language. You can compare this with object-orientated programming. First you define classes and attributes (terminology, what exists) and then you create instances resp. individuals of the classes and fill the attributes with data (knowledge, what you know). The set of defined classes and properties that can also be linked together (inheritance, class hierarchy, property hierarchy) is called T-Box (terminology box).

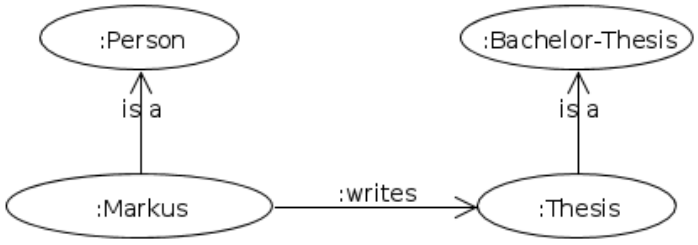

Figure 4: A small RDF graph about Markus how is a Person and the Thesis which is a Bachelor-thesis

With RDFS we can expand our previous example with classes: Person and Bachelor-Thesis (fig. 4). Now the system knows that Markus is a Person and the thesis is a bachelor-thesis. Moreover we can express constraints about properties.

But the expressiveness of RDFS is relatively low. That's why RDFS is called a lightweight ontology.

With RDF(S) we are now able to reason about our knowledge and can find implicit knowledge or invalid facts. Semantic Web Services can use reasoner to get more information and process the data in a more proper way.

To give a short example: If we say, that "every person is a human" and "Markus is a person" then the system can reason that "Markus is a human", though we have never told the system explicitly.

3.3.4 OWL

To add more expressiveness to RDF(S) OWL (Web Ontology Language) was invented.[22]

OWL adds more vocabulary like "relations between classes (...), cardinality (...), equality, richer typing of properties, characteristics of properties (...), and enumerated classes".[23]

We wouldn't go into detail because this richer expressiveness is not needed by this thesis. Moreover we never use OWL nor RDFS as a ontology language. In section 4.3 we will cover the approach WSMO [16] that use a own ontology language based on MOF [34].

[21] [27] p. 67
[22] [27] p. 125
[23] [33] section 1.2

4 Semantic Web Services

The Web Service "technology provides a standard and [is a] widely accepted way of defining (...) interfaces."[24] We have seen in section 2 how nowadays Web Services can be implemented. Now we add semantic to our Web Services. We will use ontologies as a knowledge representation model. This enables the use of reasoner to get implicit knowledge. Semantic Web Services can use this knowledge to better interact with each other.

Another step is to expand the expressiveness of Semantic Web Service descriptions. It will be possible to discover Semantic Web Services with logic inferences. This will improve the results of finding a specific Semantic Web Service. While WSDL don't express how to use operations of a Web Service or what sequence of operations are valid Semantic Web Service Technology will add the term *choreography* that expresses the interaction of Semantic Web Services.

4.1 Definition

A "service is the performance of some actions by one party to provide some value to another party."[25]

We call the performer of a service the *service provider* (like Amazon as a service provider for E-Commerce) and the one who receive the benefit *service requester* (like an agent of a customer that searching for a specific product).

Further we define the term *concrete service* as a service with concrete parameters (e.g. Amazon will sell the DVD "Titanic" for $14.96 and deliver it by standard mail service). But a customer often don't know all parameters. When a customer want to search a product (s)he will frequently only search this product by name. The customer use a abstraction and we call this *abstract service* (e.g. the customer searching for a film named "Titanic" and never told if it should be a DVD or how to deliver the product).

4.2 Life cycle

Let us now see how Semantic Web Services interact and what communication phases they have.[26]

4.2.1 Service Modeling Phase

In this phase both sides (provider and requester) model in a standard knowledge representation language what they do resp. what they want. The service requester will define a goal that is a abstract service description. A goal is a wish what value the requester want to have. Meanwhile a service provider will also define a abstract service, because he will provide a range of services and can't know what the requester exactly want to have.

4.2.2 Service Discovery Phase

Discovery means to determine if "the requirement description of a requester and the offer description of a provider are in some sense compatible(...)".[27] We have both descriptions: the abstract goal of the requester and the abstract service of the provider. There will be many service provider registered, so the requester want to get the best fitting provider (you can compare this with a shoppingtour when you searching for a shop that sells what you need). A centralized architecture implements discovery and acts like a service where provider can registrate a abstract service description and the service requester can discover a list of potential service provider.

[24] [41] p. 159
[25] [41] p. 162
[26] [41] section 6.3.7
[27] [41] p. 168

4.2.3 Service Definition Phase

In the discovery phase the requester never communicate with the provider. The requester only get "several provider which are potentially able to meet their needs".[28] Because we only match abstract services we want now find and define a concrete service that will be performed by the provider. This can be compared with our shopping example: We have found a shop and now we are talking to the vendor to negotiate what product we want to buy. In this example we can see that communication is involved. Both parties communicate about a concrete service in a one-to-one connection. This process is called *choreography*.

4.2.4 Service Delivery Phase

After a concrete service is negotiated a new communication will be started to deliver the value from provider to requester. But sometimes communication will not be necessary. We can think about the product that will shipped to the costumer. But this thesis will need delivery communication because we will work in a desktop environment. Again choreography will govern this communication.

4.3 WSMO

There are some approaches to make the vision described above real. To name some existing approaches: WSMO [16], OWL-S [5], SWSF [6], and WSDL-S [2]. This thesis will use WSMO, because WSMX [14] (Web Service Execution Environment) provides useful Java code that will be reused in this research.

WSMO[29] [16] (Web Service Modeling Ontology) is a specification based on MOF[30] [34] (MetaObject Facility) for implementing a Semantic Web Service framework.

While WSMO describes the structure of elements with MOF-based classes and attributes, WSML [15] (Web Service Modeling Language) provides a formal syntax and semantics for describing WSMO elements.

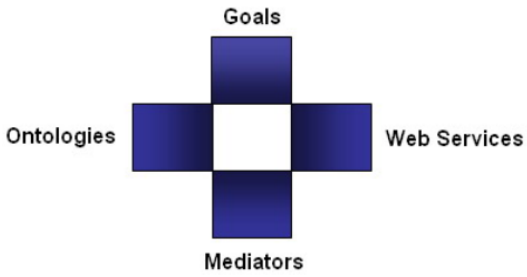

Figure 5: The four main elements of WSMO: Ontologies, Goals, Web Services and Mediators

This thesis will explain the four main elements of WSMO (fig. 5) and simultaneously show how to describe these elements in WSML. The examples are small so that the reader can easily understand the main concepts.

4.3.1 Ontologies

As already predicted in section 3.3.4 about OWL, WSMO uses a own model and language for describing ontologies. An Ontology is "a formal explicit specification of a shared conceptualisation of a domain of interest" [23]. But before we will go into detail what elements can be used in ontologies we should investigate how these elements are identified and how they can be annotated with non-functional properties.

[28] [41] p. 169

[29] `http://www.wsmo.org/`

[30] I want the reader to refer to MBSE (Model-based Software Engineering) `http://www.es.tu-darmstadt.de/lehre/mbse-v/`

IRI

WSMO uses IRIs to identify all elements. It is possible that WSMO elements have the same name in different contexts. To distinguish them namespaces are used. At the beginning of each WSMO document a namespace block should be defined.[31]

```
namespace {
    _"http://namespace.of.document/"
}
```

Listing 3: Namespace block with definition of own namespace

Every WSMO element defined in the document will have the namespace http://namespace.of.document/. Within the namespace block it is also possible to refer to other namespaces (e.g. foaf [12]).

```
namespace {
          _"http://namespace.of.document/",
    foaf  _"http://xmlns.com/foaf/0.1/"
}
```

Listing 4: Namespace block with foaf

Now we can use the short writing foaf#Person to refer to the concept http://xmlns.com/foaf/0.1/Person.

Non-functional property

Like every WSMO element can be identified with a IRI every WSMO element can have non-functional properties[32]. Non-functional properties are metadata for WSMO elements to describe for example contributor, coverage, creator, date, description, identifier, relation, source, subject, title, type, version. WSMO recommend the usage of Dublin Core [42], but it is also possible to define own non-functional properties.

Let me show you an example of an empty ontology with the non-functional properties title and description:

```
namespace {
          _"http://ontology.namespace/",
    dc    _"http://purl.org/dc/elements/1.1#"
}

ontology nameOfOntology
    nfp
        dc#title hasValue "A title of the ontology"
        dc#description hasValue "A description of the ontology"
    endnfp
```

Listing 5: Non-functional properties realized with Dublin Core

First we have add the Dublin Core namespace http://purl.org/dc/elements/1.1# to the namespace block.

In the nfp/endnfp block we use Dublin Cores titel and description definition to annotate the ontology nameOfOntology. This ontology is identified by the IRI http://ontology.namespace/nameOfOntology.

[31] [15] section 2.1.1
[32] [16] section 9 and [15] section 2.2.3

Concept

"Concepts constitute the basic elements of the agreed terminology for some problem domain."[33] A concept is a resource to compare this with RDF (we have seen an example in section 3.3.3). It is a terminology element for a specific domain (e.g. a person concept for a social network domain). Concepts can have attributes to describe characteristics. Moreover inheritance of concepts is possible. Attributes can be compared with properties in RDF. They have a name and a range to datatypes or concepts.

To describe an ontology we use the keyword ontology followed by a local name. Below we can define concepts with the concept keyword.

```
ontology World

    concept Human
```

Listing 6: The concept Human is part of the ontology World

```
ontology World

    concept Human

    concept Person subConceptOf Human
```

Listing 7: Inheritance is possible with the keyword subConceptOf

By using inheritance all instances of Person are also instances of the Human concept. Further more the Person concept inherits the signature and all constraints of Human.

Attributes can have data types or reference to other concepts. We use the keyword ofType to describe a data type relation. Otherwise we use impliesType to refer to a concept.

```
ontology World

    concept Human
        hasName ofType _string

    concept Person subConceptOf Human
        hasChild impliesType Human
        hasBirthdate ofType _date
```

Listing 8: Every Human has a name and Persons might have children and a birthdate

Instance

An instance is a representation of one or more concepts. The attributes of instances can be allocated with datavalues. A set of instances expresses the knowledge about a specific domain.

To instantiate a concept we create an instance with the "instance" keyword. To express the relation to a concept we use memberOf. With hasValue we allocate the attributes of the concept with values.

```
ontology World

    concept Human
        hasName ofType _string

    concept Person subConceptOf Human
        hasChild impliesType Human
```

[33] [16] section 4.4

```
        hasBirthdate ofType _date

    instance Markus memberOf Person
        hasName hasValue "Markus"
```

Listing 9: Markus is a real Person with the name “Markus”

Axiom

An axiom is a constrain inside the ontology and defined as a Logical Expression. Later we will see how Logical Expressions are described in WSML. A Semantic Web Service in WSMO will use axioms to express its capability.

Relation

“Relations are used in order to model interdependencies between several concepts (...).”[34] You can compare this with classical relations from mathematics.

WSML description of $Distance \subseteq City \times City \times \mathbb{R}$:

```
relation distance (ofType City, ofType City, impliesType _decimal)
   subRelationOf measurement
```

Listing 10: Distance relation between two cities taken from [15] section 2.3.2

Furthermore we can create instances of relations. “Instances of relations (...) can be seen as n-tuples of instances of the concepts (...)”[35]

```
relationInstance distance(Innsbruck, Munich, 234)
```

Listing 11: A concrete distance between Innsbruch and Munich taken from [15] section 2.3.3

4.3.2 Logical Expression

Goals and Web Services are service descriptions using logical expressions to express there capabilities. In order to understand Goals and Web Services we will first look how logical expressions are defined in WSMO ([16] section 8) and described in WSML ([15] section 3.6).

WSML Variants

To distinguish between the expressiveness of logic WSML uses variants. This thesis will cover only WSML-Core and WSML-Flight.

WSML-Core is the least expressive of the WSML variants. We will use it when we create ontologies because we don’t need a high logical expressiveness for this purpose.

WSML-Flight extends WSML-Core with meta-modeling, constraints and non-monotonic negation. The syntax is based on F-Logic [29], has the same semantic as Datalog and adds inequality and stratified negation.[36]

Variables

Varibles start with a question mark “?” and use the following symbols in the names: { a-z, A-Z, 0-9, _, - }.[37]

[34] [16] section 4.5
[35] [16] section 4.7
[36] [41] p. 196
[37] [16] section 8.1

```
?var ?var_with_underscore ?CamelCase
```

Listing 12: Variable examples

Anonymous IDs

"Anonymous IDs can be used to denote objects that exists, but do not need a specific identifier"[38]
They can be numbered or unnumbered.

```
_#1 _#2 //numbered
_#      //unnumbered
```

Listing 13: Anonymous ID examples

Membership Molecule

A molecule is a statement about a resource. A statement is closed by a dot ".". With the membership molecule annotation memberOf we can express what type an element has.

```
_"http://example.com/Markus" memberOf _"http://example.com/Person" .
?person memberOf _"http://example.com/Person" .
?p memberOf ex#Person . //if ex is a defined namespace with _"http://
   example.com/"
```

Listing 14: Person membership

Sometimes it is necessary to express multiple membership:

```
?person memberOf { _"http://example.com/Human", _"http://example.com/Agent"
    } .
```

Listing 15: A Person is at the same time a Human and an Agent

Attribute Value Molecule

We frequently need to query what values the attributes of resources have. Additionally the membership molecule can be combined.

```
//with namespace ex for _"http://example.com/" we can write it in a short
   way:
ex#Markus[ex#name hasValue "Markus"] memberOf ex#Person .
?person[ex#name hasValue ?name] .
```

Listing 16: Name attribute examples

To define a range of attributes we can write them comma-separated:

```
ex#Markus[ex#name hasValue "Markus", ex#lastName hasValue "Schroeder"]
   memberOf ex#Person .
```

Listing 17: Range of name attributes example

[38] [16] section 2.2

Conjunction and Disjunction

Molecules can be combined with conjunction (add) and disjunction (or). It is possible to share same variables in different molecules.

```
?einhausen[name hasValue "Einhausen"] memberOf City and Markus[livesIn
   hasValue ?einhausen] .
```

Listing 18: Markus lives in Einhausen

4.3.3 Goals and Web Services

Both Goals and Web Services are Web Service descriptions. While requesters model a goal to express what they want providers model a capability to say what they do.

Capability

"A capability defines the Web service by means of its functionality."[39] resp. "Goals can be descriptions of Web services that would potentially satisfy the user desires."[40] It is separated in four axioms expressed with Logical Expressions (see section 4.3.2):

Precondition is a condition about the information space before the service invocation.

Postcondition is a condition about the information space after the service invocation.

Assumption is used to describe the world state before the execution.

Effect is used to describe the world state after the execution.

Additionally we can use shared variables between precondition, postconditon, assumption and effect.

On the one side Web Services *have* a capability and on the other side Goals *request* capability. A discovery engine can use these two descriptions to find a match. That's why we can classify the capability description to the service discovery phase in section 4.2.2. We will go into detail in the discovery section 4.3.5.

Because every keyword is self-explanatory here is an example about a Web Service capability that register a child to german citizenship:

```
webService registerChild

    capability registerChildCapability

        sharedVariables ?child

        assumption //Child is not dead
            definedBy
                ?child[alive hasValue _boolean("true")] memberOf Child.

        effect //After the registration the child is a German citizen
            definedBy
                ?child[hasCitizenship hasValue oo#de] memberOf Child.
```

Listing 19: Simplified example from [15] section 2.4.1 about child citizenship registration

[39] [16] section 5.1
[40] [16] section 6

Interface

While the capability describes the functional aspects of a service description the interface points out how requesters can interact resp. communicate to achieve the provider's functionality. The interface description is used in the service definition phase mentioned in section 4.2.3. WSMO uses the term choreography to describe the communication behavior of the Web Service. This communication is state-based and that's why WSMO uses ASMs (Abstract State Machines) to implement choreography.[41]

Because ASMs and the description of ASMs is a complex topic we will cover it separately in the Choreography section 4.3.6.

4.3.4 Mediation

Because WSMO uses "strict decoupling, mediation addresses the handling of heterogeneities that naturally arise in open environments."[42] Whenever interoperability problems come up between ontologies, web services or goals mediation will resolve these obstacles.

The topic of this thesis is a implementation of Web Service technology on the scope of a desktop environment. We will assume that we will not need Mediation at all hence this thesis doesn't cover this topic.

4.3.5 Discovery

We have seen in section 2.3 about UDDI how simple Web Services are nowadays discovered. While Web Services using keyword matching and categorizations Semantic Web Services going further and using logical inferences.

With logical expressions within capabilities it is possible to describe an abstract service description. Service provider and service requester model there description in the Service Modeling Phase (section 4.2.1). Discovery will perform a comparing of the capability descriptions to "figure out which provided service is relevant for a specific request."[43] This is called *matching* and the result is a boolean value: if it is true then the service offer is relevant, else irrelevant.

WSMO uses a Description Logic based discovery framework that uses subsumption inferences. Subsumption checks if "the requested service capability is a specialization of the provided one or vice versa."[44] This inference is called "Entailment of Concept Subsumption" and can be applied in both directions: The requester's capability is a specialization of the provider's capability (this is called a plugin-match) or the provider's capability is a specialization of the requester's capability (this is called a subsumes-match). If the match is a plugin-match and at the same time a subsumes-match we talk about an exact match.

Plugin-Match

Each desire of the requester is covered by the provider. In any case the provider will be a good choice.

To give an example we will look at WSMO Use Case of Amazon E-commerce Service [32] where a simplified capability of the amazon service looks like this:

```
sharedVariables {?request}

precondition
    definedBy
        ?request memberOf am#helpRequest or
        //...
```

41 [37] section 1
42 [16] section 1 Centrality of Mediation
43 [41] p. 219
44 [41] p. 230

```
postcondition
    definedBy
        ( ?request[helpType hasValue ?type] memberOf am#helpRequest
          implies exists ?response
            (?response[responseType hasValue ?type] memberOf am#
               helpResponse)
        ) and
        //...
```

Listing 20: Part of the capability of Amazon E-commerce Service in WSML

The precondition (condition about the information space before execution) is that the amazon web service among other things expects a help response.

The variable "?request" is a shared variable. So the request found in the precondition is also the request in the postcondition.

The postcondition says that if the web service got a help request (memberOf am#helpRequest) then a response will be exist. This response will be a help response (memberOf am#helpResponse) and of the same type (look at the attributes helpType and responseType).

A requester who requests a helpResponse while having a concrete helpRequest will match with the provider above.

Subsumes-Match

Only a subset of the desires of the requester is covered by the provider. But this subset is not empty and that's why the provider matches with the requester.

To find out if the wish can be fulfilled by the provider the requester must contact the provider and negotiate a concrete service.

4.3.6 Choreography

While discovery finds matchings between capability descriptions to determine what provider provides the desired functionality, choreography coordinate a one-to-one communication between a service requester and a service provider. The models for the communication are ASMs (Abstract State Machines). We will first cover why WSMO uses ASMs and how ASMs are working.

ASM

ASMs are used "for abstraction, validation and verification of the system at a given stage in the development process."[45] Hence, WSMO uses ASMs for abstracting the communication between requester and provider in a stateful manner. This will be needed if both need a number of interaction steps to invoke and deliver a service.

The advantages of a ASM model are:[46]

- Minimality: ASMs use only a small set of modeling primitives.
- Maximality: But ASMs are very expressive and that's why they can be used to model any aspects around computation.
- Formality: ASMs are based on mathematical formalisms.

ASMs are split in two main categories: Single-Agent and Multi-Agent. WSMO uses Multi-Agent ASMs because they allow to describe collaborating ASMs. Multi-Agent ASMs again are divided into two categories: Synchronous and Asynchronous. For real Web Services spread over the internet an asynchronous

[45] [37] appendix A
[46] [37] section 1

communication will be needed because these agents are independent entities. In the small scope of WSMX or a desktop environment Synchronous Mulit-Agent ASMs will be sufficient.[47]

Basic Terminology of ASMs

While normal ASMs are using function names for describing a basic terminology WSMO uses Ontologies. Concepts and relations form a state signature for the ASM. A state is defined by instances of their concepts of ontologies. ASMs provide a way to give elements of a state signature roles (also called modes).[48] Roles describe the communicative activities of extensions of concepts and relations as follows:

- A "static" role means that it can't be changed by the ASM nor the environment around the ASM.
- A "in" role means that only the environment can change it.
- A "out" role means that only the ASM can change it.
- A "controlled" role means that only the ASM can change and read it.

```
in concept am#itemSearchRequest
out concept am#itemContainer
```

Listing 21: Role example about item search taken from [32]

For the amazon web service a item search request can't be changed but the response of a item container that will be send back.

Transition Rules of ASMs

To change a state transition rules can be applied to ASMs. A change in the state means changing instances of an ontology. That's why there exists the modifiers add, delete and update to add or delete instances or to update attributes of existing instances.

Further more there exists three different constructions that can be used:

"if Condition then Rules endIf" is a guarded transition rule where Condition is a Logical Expression and Rule another rule or a modifier.

"forall Variables with Condition do Rules(Variables) endForall" has the meaning that each variable satisfying the Condition is used in the Rules.

"choose Variables with Condition do Rules(Variables) endChoose" has the meaning that the system can choose one variable allocation that satisfying the Condition.

Execution Steps of an ASM

1. According to the current State evaluate the Conditions of the transition rules. After this only Rules with allocated variables existing.
2. Execute all Rules simultaneously.
3. If Step two was consistent then the ASM is in the next state.

These three steps will as long as executed until no rules can get from step one because all Conditions evaluate to false.

[47] [37] section A.2
[48] [24] section 3.3.1 role

Communication

For communication between provider and requester both represent an ASM. They describe a choreography that is symmetric to each other. As mentioned Concepts that marked as an “in” role can be received and concepts that marked as an “out” role can be sent.

First the requester send instances to the provider. Then the provider makes an execution step (see above) and modify instances with add, remove or update. Instances that marked as “out” are sent back to the requester. The requester also do an execution step. This will be done until communication ends because no instances are sent to each other.

WSMX is a mediating entity between the service requester and the service provider (look at [24]).

Example

For an example we look at the WSMO Use Case of Amazon E-commerce Service in [32]. Here is a simplified exemplar:

```
interface amazonWSInterface
    choreography
        stateSignature
            in concept am#itemSearchRequest
            out concept am#itemContainer
```

Listing 22: State signature of Amazon Web Service

In the state signature we have two concepts with different roles: A item search request can be received by the amazon web service that’s why it has the “in” role. After the search amazon sends back a container with items. So itemContainer has the “out” role.

```
        transitionRules
            if (?ItemSearchRequest memberOf am#itemSearchRequest) then
                add(_# memberOf am#itemSearchResponse)
            endIf
```

Listing 23: Transition rule of item search

This simple Transition Rule is used to catch an incoming search request for items. That means if the requester has send a itemSearchRequest-Instance then the if condition evaluate to true. Because ItemSearchResponse returns a list of items it inherit from itemContainer. Hence itemSearchResponse has the “out” role. It will be send back to the requester.

Someone will note that itemSearchResponse is always an empty response. That’s why the choreography is used to exchange events that represent communication. The real data will be send over a communication binding.

5 Linux

To transfer Semantic Web Services to the desktop we need an environment. In this section we will show the advantages of a Linux environment and why we can use it to realize the idea of "Semantic Desktop Services".

5.1 History

The following is a short summary of the History of Linux [26]:

Unix is an Operating System that was in earlier days very expensive and closed source. That's why Andrew S. Tanenbaum wrote "Minix" from scratch that is based on Unix ideas. But this operating system was only for academic use, so Minix failed to be a good operating system. Nevertheless the sourcecode of Minix was available.

At the same time Richard Stallman founded the GNU project [40]. His vision is that software should be free from restrictions so that everybody can modify and improve the code. The GNU project created a lot of useful unlicensed tools (e.g. GCC[49]), but a free operating system kernel was not available.

In 1991, Linus Benedict Torvalds wrote his own kernel "Linux" and distributed the source, so that others can help him. After few month the kernel became a stable software that compete with others. It was licensed under GNU General Public License [21].

Today nearly every distribution is based on GNU/Linux: the Linux kernel with GNU software tools.

5.2 Philosophy

Because in the time Linux was invented the operating system ideas influenced each other and that's why many philosophies were taken over. These concepts will help us to implement the Web Service technology much faster to the desktop environment. We will find many advantages in Linux listed in [36].

5.2.1 Modularity

"Write simple parts connected by clean interfaces."[50]

Complex software is separated in many modules or packages. This software parts need a clean interface to communicate. This approach of divide complexity has a long history and is well approved. Modularity allows to split software in such a way that changes on software parts don't effect the whole program, because interfaces abstract functionality.

We have seen this approach of modularity in the Web Service technology: With WSDL we can define clean interfaces that can be interpreted by machines. The value of service (the real execution) is another part. That's why Web Services fulfill modularity. With Semantic Web Services we have the possibility to add more information to the interface description. As a result Semantic Web Services can be connected by a machine in a automated way.

5.2.2 Composition

"Design programs to be connected with other programs."[51]

Today's programs on Linux machines are frequently implemented as filters[52]. This means that they can read and write simple, textual, stream-oriented formats. It is possible to connect these programs with pipes to transfer text from one program to another.

[49] http://gcc.gnu.org/
[50] http://www.faqs.org/docs/artu/ch01s06.html#id2877537
[51] http://www.faqs.org/docs/artu/ch01s06.html#id2877684
[52] http://www.faqs.org/docs/artu/ch07s02.html#plumbing

```
find | grep "pattern"
```

Listing 24: Classical piped programs: find and grep

An example will be “find” and “grep” that piped (with the “|” symbol) together. The program “find” returns a list of files and directories. This textual list is transferred to grep over a pipeline. On the other side “grep” accepts this list and searches for a pattern.

Moreover composition means that GUI (Graphical User Interface) and program logic are always separated. The program logic is again separated in application primitives organized into a library. Each primitive has a well-defined API (Application Programming Interface).

Web Services are also connected with other Web Services over choreography. They communicate like we have seen in the find-grep example. But new technologies allow to express a more complex communication behavior. The exchanged, textual format is replaced by ontology definitions. Additionally the design of composition makes it easy to reuse parts of programs in Web Services.

5.2.3 Representation

“Fold knowledge into data, so program logic can be stupid and robust.”[53]

This philosophy says that it is better to make the data structures complex and the logic simple because you can better handle data then program logic.

We can see this idea in the Semantic Web Services area too: Ontologies are complex data that can express many aspects with the help of logic to model nearly everything. So Web Services can be simple and do one task well. Using Ontologies means also that we have the possibility to reason more implicit knowledge.

5.3 Everything is a File

As [13] explains, the idea that “Everything is a File” is a key concept of Unix. Particularly it’s the idea that everything can have a file descriptor. This implies that files, pipes, processes, system information, serial devices, tape devices, disk devices, network sockets and so on can be accessed via file descriptors. This idea raises integrity among all devices connected with the system. That’s why you must never think about the kind of device. You read from it or write to it like it is a simple file.

Even internal data structures in the kernel (e.g. system information about processes) are emulate with file descriptors in the “/proc” filesystem [11]. Here we can find for each process a subdirectory named with the PID (Process ID). This filesystem doesn’t exist on harddrive. The kernel emulate a filesystem if you access it. So Linux abandon open, readable, dynamic knowledge about the whole system. Again this information is easily accessible by simple file reading.

On the other side configuration files for programs or system settings are always in a human readable format and mostly good documented (see next section about man pages).

To summarize, because everything is a file we have access to the knowledge of the whole system in a standardized way. However, everything is in a human readable format and that’s why it must be parsed to process the information with machines.

5.4 Man pages

Linux is self-documented with man pages [35] (manual pages). With a man page a user can find out how to use a command or how to read a configuration file.

Linux saves knowledge in human readable files. If we want to know for example information about user accounts in the system we can read the file “/etc/passwd”[54]:

[53] http://www.faqs.org/docs/artu/ch01s06.html#id2878263

[54] http://www.cyberciti.biz/faq/understanding-etcpasswd-file-format/

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/false
daemon:x:2:2:daemon:/sbin:/bin/false
```

Listing 25: Example of /etc/passwd

Though this file is human readable we don't now how the structure is defined and what's the meaning of it. We can search for the man page of passwd (with 'man 5 passwd') to get these informations:

```
NAME
    passwd - the password file
DESCRIPTION
    /etc/passwd contains one line for each user account, with seven fields
       delimited by colons (":"). These fields are:
    o   login name
    o   optional encrypted password
    o   numerical user ID
    o   numerical group ID
    o   user name or comment field
    o   user home directory
    o   optional user command interpreter
(...)
```

Listing 26: Passwd man page

With the informations taken from the man page we know now the meaning of each field.

The advantage is that we have non-binary files that we can parse with informations from man pages. Knowledge about the system is open and accessible. But every knowledge is human readable and hard to process via a machine. Further more there are no real connections between instances (passwd file) and classes (passwd man page).

5.5 GNU/Linux

GNU [40] (Gnu's Not Unix) is a project with the aim to creating a complete Unix-compatible software system.

Today Linux and GNU are joint to a full featured operating system. Richard Stallman mentioned in [39] that we should differ between kernel and application software. He requests that Linux should called "GNU/Linux", because the GNU project contributes many programs to the operating system.

Because GNU wants to be a Unix-like system we can find nowadays a lot of commands and philosophies taken from Unix. Since 1984 the GNU project developed a long list of useful tools [20]. These tools are available for free and are used in many distributions of Linux.

So the big advantage is that functionality capsuled in libraries and programs already exists. But they are mostly accessible via a terminal or a GUI. Moreover every input and output communication is human readable. That means only humans can reuse functionality in a GNU/Linux system.

5.6 Conclusion

We have seen that the philosophies of Linux are the same of Web Services:

- Make modular parts connected by clean interfaces.
- Design composition to connect programs resp. web services.
- Use a complex representation of knowledge in data structures.

We can find applied philosophical principles[55] in nowadays Linux distributions:

Many little functionalities capsuled in small programs makes it possible to find the right program for a specific problem, because “small is beautiful. Write programs that do as little as is consistent with getting the job done.” Moreover because “complex front ends (user interfaces) should be cleanly separated from complex back ends” machines can easily access the functionality without barriers caused by front ends.

The advantage of reading every knowledge about the system from files has a disadvantage because of the following principles:

- “Data streams should if at all possible be textual (so they can be viewed and filtered with standard tools)” and
- “Database layouts and application protocols should if at all possible be textual (human-readable and human-editable).”

That’s why a machine will never be able to reason over the system knowledge accessed from files.

To sum up, Linux provides human readable knowledge about the system in files and functionality in small tools to process this knowledge. If we can enable the knowledge for machines and reuse the tools in a automated way with Semantic Web Service technologies the vision of “Semantic Desktop Services” will become true.

[55] http://www.faqs.org/docs/artu/ch01s08.html

6 Semantic Desktop Services

Finally this thesis introduces the own approach called “Semantic Desktop Services”. We will first define what requirements we have to this new approach. Then we have a look at available technologies to program a prototype called “Engine”.

To use Web Services in a desktop environment we must first transform existing knowledge. On the other hand to process functionality in a semantic way we must annotate programs as well.

At the end we have a deeper look at the engine: The sections cover the components, explain inner workings with an execution semantic, present some results and close with graphical user interface ideas.

6.1 Vision

Today system admins have to know the tools they are using. They need a good understanding how the behaviors of this programs effect the system. Further more every program can be invoked in other ways (e.g. some programs are filters and others not). So a system admin must be well versed with the environment (s)he is using.

The main task of system admins is to write scripts (sequences of commands) to do a specific, desired job. To express this in an abstract level they compose functionality to fulfill a goal. A range of tools help them to create a valid composition: syntax highlight, syntax check, debugger, auto-completion, man pages, books about commands, etc.

But this thesis want to go further. As mentioned an admin has a vision what (s)he wants to do and writes complex scripts that do the job. What will be if the admin can simply express the vision as a goal and request the system to give an advice what can be done to fulfill this goal? We will get faster solutions to complex problems and much better support. The system can give a range of advices what the admin can do. Additionally reasoning helps to detect invalid compositions of commands or risky invocations (if we think about data deletion).

To test this new ideas we use the Linux access control system [30] as a use case. Because Linux is a multi-user operating system it differs between users and groups. It controls the access to files and programs. Every file, executable or directory is owned by a user and a group. Permissions can be set exactly to express read, write and execute permissions for the owner, group and others (all users on the system).

6.2 Technology

To implement this vision this thesis uses technologies provided by WSMO.

6.2.1 wsmo4j

Wsmo4j [17] is a reference implementation of WSMO in Java. The project is hosted and can be downloaded at `http://wsmo4j.sourceforge.net/` .

Every element described in section 4.3 about WSMO is represented by a Java Class. By the help of the “WSMOFactory” class we can create WSMO entities.[56] A parser allows us to parse WSML documents to get WSMO entities by description. Further more a serializer creates WSML documents out of a WSMO Java model. Another factory “DataFactory” can be used to create WSML data values (e.g. strings, decimals, integers, etc.). Logical Expressions, covered by section 4.3.2, can be parsed from text to model or created by means of the “LogicalExpressionFactory”.[57]

6.2.2 WSML2Reasoner

WSML2Reasoner [9] provides algorithms to transform WSML descriptions to reason about it. In doing so WSML2Reasoner uses existing reasoner engines. So it is possible to reason about Logical Expressions.[58]

[56] [17] p. 12
[57] [17] p. 22
[58] maybe you want to try out the online demo at `http://tools.sti-innsbruck.at/wsml/rule-reasoner/v0.2/`

The engine will need this to perform a plugin-match and to check if assumptions are valid in the facts.

6.2.3 WSMX

WSMX [1] "is an execution environment for dynamic discovery, mediation and invocation of web services" written in Java. Its architecture [44] is based on SOA (Service Oriented Architecture). That means WSMX has strong de-coupling components and standardization of external behavior of components. Because the source is open (available at `http://sourceforge.net/projects/wsmx/`) this thesis reuses code of the discovery component. WSMX is an executable environment with many other components that could be reused, but the complexity is not needed in this research.

6.3 Knowledge

The main problem is that knowledge in the Linux environment is human readable. To enable machines to reason and process this data we need to transform the textual files to a knowledge representation language. We have learned that Ontologies provide good knowledge representation possibilities. First, we need to extract the concepts of Linux to collect them in ontologies. Second, we parse files to get data primitives and create instances of these concepts.

6.3.1 From Man pages to Concepts

We have seen that man pages explains the structure and content of files in Linux. As an example I will use passwd to demonstrate how we could extract concepts from man pages:

```
NAME
    passwd - the password file
DESCRIPTION
    /etc/passwd contains one line for each user account, with seven fields
       delimited by colons (":"). These fields are:
    o   login name
    o   optional encrypted password
    o   numerical user ID
    o   numerical group ID
    o   user name or comment field
    o   user home directory
    o   optional user command interpreter
```

Listing 27: Passwd man page

Because the passwd file has a simple structure a translation is straight forward:

```
ontology PasswdOntology
  nfp
    dc#title hasValue "Passwd Ontology"
  endnfp

    concept User
        nfp
            dc#title hasValue "User Concept"
        endnfp

        login ofType _string
            nfp
                dc#title hasValue "login name"
```

```
        endnfp

    optEncPasswd ofType _string
        nfp
            dc#title hasValue "optional encrypted password"
        endnfp

    uid ofType _integer
        nfp
            dc#title hasValue "numerical user ID"
        endnfp

    gid ofType _integer
        nfp
            dc#title hasValue "numerical group ID"
        endnfp

    username ofType _string
        nfp
            dc#title hasValue "user name or comment field"
        endnfp

    home ofType _string
        nfp
            dc#title hasValue "user home directory"
        endnfp

    cmd ofType _string
        nfp
            dc#title hasValue "optional user command interpreter"
        endnfp
```

Listing 28: Ontology of passwd with User concept

For each field we create an attribute for the User concept. With non-functional properties we annotate the attributes with titles taken from the man page.

For simplicity the “home” attribute has the type string but actual we could use a Directory concept (e.g. “impliesType Directory”) from a Filesystem Ontology.

In future every man page about data structures could be transformed to concept descriptions. Extra axioms about the behavior of this concepts have to be added by user manually (e.g. every Person in the system is a User). So after a while the transformations could form a complete Linux Ontology. This ontology can be used for reasoning and knowledge exchange.

Automated Approach

In this research man pages are manually transformed into a WSMO ontology description. But we could think about an automated way of extracting human readable information. With NLP [28] (Natural Language Processing) it could be possible to get a range of auto-generated concepts. As an example DBpedia [10] extracts categorizations and articles from Wikipedia. With the DBpedia knowledge extraction framework the human readable, structured markup description is extracted to get a rich knowledge base. Man pages have also structure because of sections and standard textual expressions (for example the NAME section has nearly always a title and a description separated by a dash).

6.3.2 From Files to Instances

With a Linux Ontology we can now create instances of concepts. We find concrete data in files such as /etc/passwd holds information to user accounts:

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/false
daemon:x:2:2:daemon:/sbin:/bin/false
```

Listing 29: Example of /etc/passwd

Because /etc/passwd has a simple structure a parser will

- create a user account instance for each line and
- split the line with a colon separator ":" to get the data fields.

Another example we covered is the /proc filesystem where we can extract system information, for example memory usage and information about processes.

We have two approaches how we can load instances from files. Both attitudes use parsers to parse human readable files and extract data.

Update on change

One approach is to read all files and save the generated instances in a persistent triplestore. Triplestores are like databases with the focus on store and retrieve triples (like we have seen in section 3.3.3 about RDF). Files have to be monitored to hold the instances in the triplestore up to date. So if the Linux system changes a file the Semantic Desktop Service Engine will be invoked and parses this file anew.

The advantage of this approach is that the Linux system information is stored completely in a persistent triplestore. In the future Semantic Desktop systems could use a triplestore as a replacement for a filesystem. This causes new ways of data manipulation and reasoning. New program designs can be possible if we think that every tool can be a complete Web Service.

A disadvantage is that we must monitor the files to hold our triplestore consistent. That could rise other consistency problems. Every time a file changed it must be read completely because we don't know which textual information is translated to instance data. Monitoring all files can cause high performance problems.

Another disadvantage is the problem of a overflowed triplestore with instances. Reasoning over hundreds of instances drops the performance. A normal desktop PC hasn't the power to query instances of a complete system.

Parse when needed

Another approach would be to read information from the system when needed. That means for example if a Web Service needs information about user accounts then /etc/passwd will be parsed. After using the instances they will be deleted because the information maybe will become old.

That's why the advantage is that we will have always up to date data. Reasoner could be extended with parser support to get instances from textual files when needed. Moreover reasoning will become faster because the set of instances is relatively small.

But the disadvantage comes when we must parse information always anew. Files which never be changed will parsed over and over again. Maybe an intelligent system can recognize and prevent this.

6.3.3 Conclusion

With concepts from man pages and instances from files we can translate the knowledge of a Linux system to a semantic network. Data will then directly linked together. For example if a user is a member of a

group we have not to look at the group ID of this user and search for the group with the correspondent ID. Users and groups will be linked together. Same goes for other relationships (e.g. owner of a file, etc).

This tends to the vision of Semantic Desktop [38]. "A Semantic Desktop is a device in which an individual stores all her digital information like documents, multimedia and messages. These are interpreted as Semantic Web resources, each is identified by a Uniform Resource Identifier (URI) and all data is accessible and queryable as RDF graph."[59]

6.4 Functionality

While knowledge represents what we know Functionality shows what we can do with this knowledge. We add, modify or delete information in the system with tools. As mentioned we have a lot of possibilities with a range of GNU tools. If we think about user account information for example the tool "useradd" helps us to create new user accounts. But first we must annotate the commands machine processable to use them in an automated way. This research shows as an example how the "useradd" command can be annotated with WSMO Web Service descriptions.

6.4.1 The Command "useradd"

If we want to know how to use "useradd" we can investigate the man page with "man useradd".

```
NAME
       useradd - create a new user or update default new user information

SYNOPSIS
       useradd [options] LOGIN
       (...)
```

Listing 30: Man page of useradd

We discover in the SYNOPSIS section that the simplest call is adding a name (LOGIN) at the end of useradd. For example if we want to create a user named "markus" we call the command "useradd markus".

6.4.2 Non-Functional Properties

Now we can create a Web Service description about useradd with the help of the man page:

```
webService useradd
    nfp
        dc#title hasValue "useradd"
        dc#description hasValue "create a new user or update default new
           user information"
        sds#command hasValue "useradd ?name"
    endnfp
```

Listing 31: Web Service useradd with non-functional properties

The name of the Web Service is the same as the name of the tool. Additionally we add Dublin Cores title and description property. The first line in the NAME section could always be split with a dash "-" in title and description. The sds Namespace and the command property is a definition by myself. The command property will be used to invoke the command. "?name" is a variable that will be covered in the Capability section below.

[59] [38] p. 3

6.4.3 Capability

The capability describes what effect useradd has to the system or particularly to the knowledge. However we should also define the assumptions we have on the system. Because creating an existing user named “markus” causes an error message “useradd: user ’markus’ already exists”.

Shared Variables

Shared variables are shared between effect and assumption. Further more this thesis will also use shared variables in simple property descriptions (e.g. command or reason).

```
capability useraddCapability
    sharedVariables { ?name }
```

Listing 32: Capability description with shared variables of the useradd Web Service

As you can see the name of the user is a central information we will use and need.

Effect

The effect describes the system state after the execution of useradd.

```
effect
    definedBy ?iri[passwd#login hasValue ?name] memberOf passwd#User .
```

Listing 33: Effect Capability of the Web Service useradd

After execution useradd creates a user with a defined IRI (?iri). This instance is a member of the defined concept User from the passwd Ontology (memberOf passwd#User). Moreover the user instance has a login name (passwd#login). This name is saved in the ?name variable. It’s the same variable we use in the command property above.

Assumption

The assumption describes the system state before the execution of useradd. We have seen that useradd prevents us to create a user with the same name. That’s why we should create this assumption:

```
assumption
    nfp
        sds#reason hasValue "user ’?name’ already exists"
        sds#assert hasValue false
    endnfp

    definedBy ?iri[passwd#login hasValue ?name] memberOf passwd#User .
```

Listing 34: Assumption Capability of the Web Service useradd

Assumptions must be checked before the useradd command will be invoked. If a assumption fail we want to print out an error message for the user. That’s why we add to the assumption the non-functional property “reason”. Like the command property it has the ?name variable that will be replaced by the actual name.

The assert property means whether we assert the assumption to true or false. We must add this property because checking assumptions is based on making queries. In the assumption we defined the same statement as in the effect capability. Therefore the assumption will check if a user exists with the same name in the system. If we find a result (the assumption is true) the assumption should fail because useradd assumes that no user exists with the same name before execution. Thus we assert the assumption to false (sds#assert hasValue false).

6.4.4 Conclusion

With the help of this methods we can create for each tool on the system WSML Web Service descriptions. This must always be done manually because a machine can hardly compute capability descriptions.

If we take useradd as an example we can create more Web Services:

“groupadd” creates a group in the system. It’s not very different to “useradd”.

“gpasswd” adds a user to a group. The command assumes that user and group are existing in the system.

“chown” changes the owner or group of a file. It assumes that the user, group and file exists.

6.5 Engine

After translating Knowledge and Functionality from Linux to WSML descriptions we can now process them with WSMO technologies. For this research I programmed an engine based on sourcecode and libraries from WSMX.

6.5.1 Components

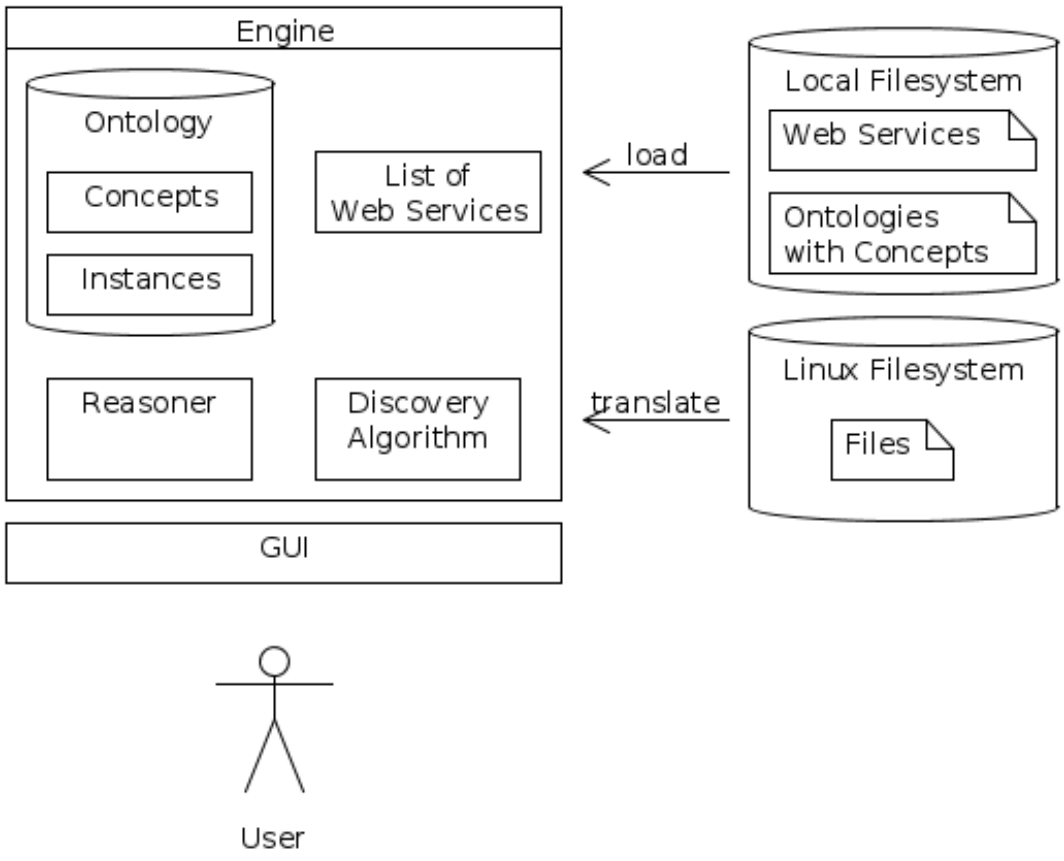

Figure 6: Diagram that shows the components of the engine

Figure 6 shows the main components of the engine and peripheries around. First of all, before we can process data, we must load it into the engine.

Web Services (like “useradd”) are described in WSML and saved in files (see fig. 6 on the top-right side). The engine reads these files from a local filesystem and parses them to get WSMO Java objects. They are stored in a list of available Web Services.

Manually defined ontologies (like the passwd ontology) can also be loaded and parsed into the engine. Every concept of every loaded ontology is stored in one big ontology. This makes it easier for the reasoner to reason about all knowledge.

On the other side knowledge from Linux must be translated and saved in the engine (see fig. 6 on the bottom-right side). Therefore translators existing that parse a specific Linux file and create WSMO Java objects with the help of wsmo4j. For example the engine provides the Java class "PasswdLinuxToWSMO-Translator" to parse the file /etc/passwd mentioned in section 6.3.2. The translators create instances of the loaded concepts and invoke a listener designed with the observer pattern. To recognize changings in files the translators use file monitors provided by Apaches Commons Virtual File System [4]. For example if a new user is added to the system the file /etc/passwd will be changed and the translators get an event to parse the file anew. The new generated instances will be replaced by the old ones. The engine uses the mentioned "Update on change" approach because it was implemented in short time.

After loading we have a list of WSMO Web Services and one big ontology containing concepts and instances (depicted in fig. 6 as a database).

On the front-end the engine provides a GUI for the user. This interface allows the user to write down goals or define them with graphs. Moreover loaded concepts and instances can be browsed in a tree. We will cover the aspects of the GUI later in detail in section 6.6.

The heart of the engine is the Discovery Algorithm. The main task of the Discovery Algorithm is creating a tree of command invocations based on a goal definition of the user. Therefor it needs the data we loaded at the startup of the engine. Because the execution semantic of the algorithm is the most interesting part of the engine we will go into detail in the next section.

6.5.2 Execution Semantic

The Execution Semantic of the Discovery Algorithm (fully depicted in fig. 7) is divided in tree levels.

In the first stage the algorithm iterates thru all Web Services loaded in the startup phase (mentioned above). This phase checks if a Web Service is useful for solving the given goal. There are little proves if the Web Service is properly loaded by checking existing capability. After this we figure out if the effect of a Web Service is relevant to the goal by performing a plugin-match (see section 4.3.5). This means the algorithm makes a query with the effect based on knowledge of the goal. The result is a set of allocated variables in the logical expression of the effect. Only if all variables are allocated (the effect solves the goal partially or completely) the Web Service is marked as usable. This stage could be compared with the Service Discovery Phase explained in the Life cycle section 4.2.2.

The next stage detects if the Web Service is executable. Possibly some assumptions are not valid in the facts. That's why this stage iterates thru all assumptions. It replaces all shared variables in the assumption with values from the effect variables. That's because we want to entail the assumption in the facts with values taken from the effect. For example we have a shared variable "name" that is allocated in the effect with the value "markus". If the assumption checks whether a named person exists the assumption needs to know the name. Only if all assumptions are valid this Web Service will be discovered because the Web Service is usable and executable.

But maybe one assumption is invalid. Then the algorithm runs to the third stage because it searches for other Web Services that could make the (currently) invalid assumption valid. This level starts recursively the Discovery Algorithm. The invalid assumption becomes the goal that should be solved.

At the end the execution semantic returns a list of advices. An advice is a tree of all discovered Web Services. Particularly it's a tree of commands that solves the goal.

6.5.3 Results

In this section this thesis presents results of the engine for some goals. The goal is always on top of the tree. The childrens are linux commands that solve the goal or reasons why a parent command can't be invoked.

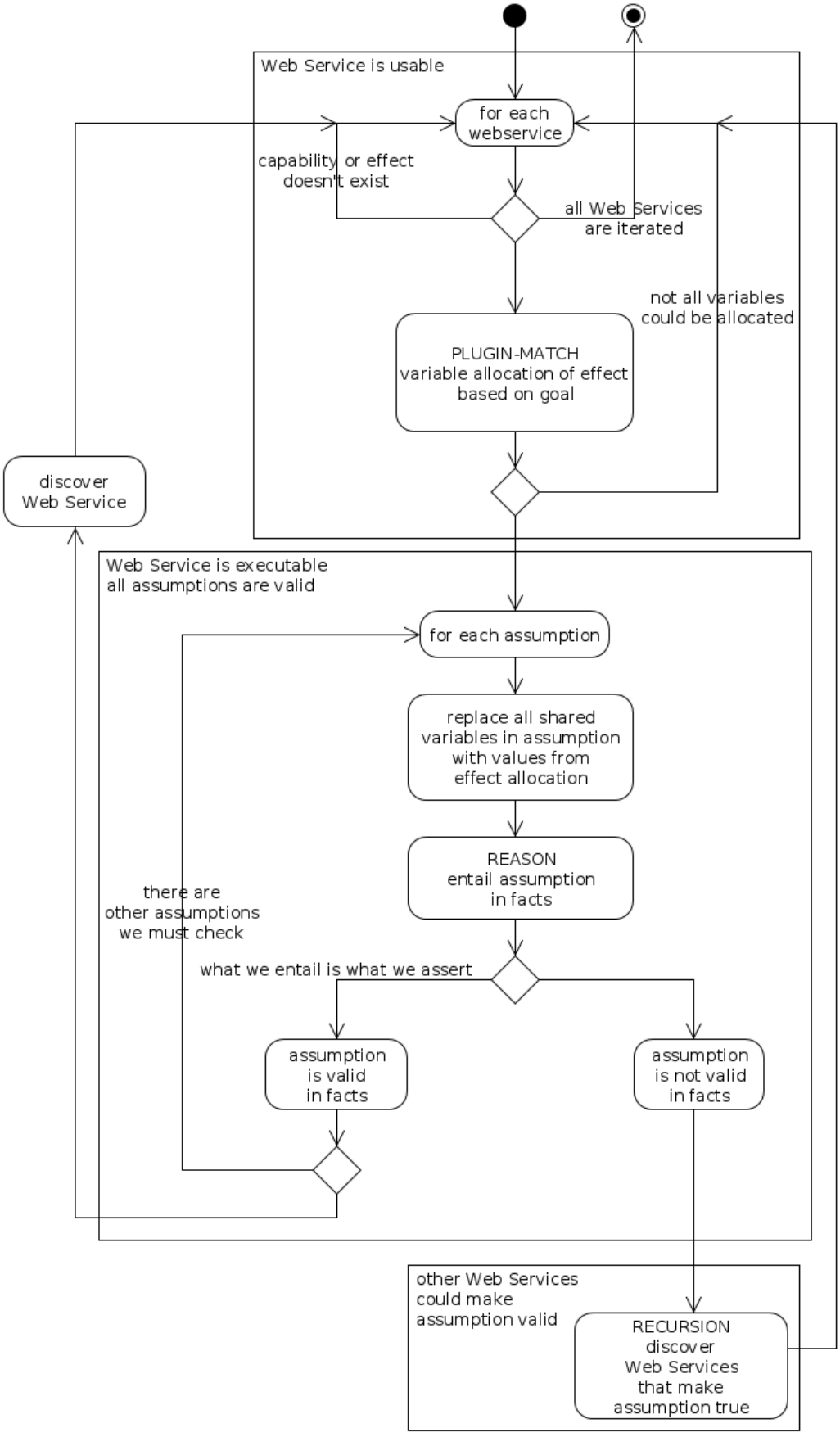

Figure 7: Diagram of the Execution Semantic of the Discovery Algorithm

```
_#1 memberOf passwd#User
```

Listing 35: The goal expresses to create an anonymous user

If the user defines such a goal (s)he wants to create a new user. But because no login name is defined the engine can't find any suitable Web Service.

```
_#1[passwd#login hasValue "new"] memberOf passwd#User.
\__ useradd new
```

Listing 36: The goal expresses to create an anonymous user with login name "new"

Now a login name "new" is defined. That's why the effect of useradd matches with the goal. Because no user named "new" exists in the knowledge base the engine suggests to create a new user with "useradd new".

```
_#1[passwd#login hasValue "root"] memberOf passwd#User.
\__ useradd root
    \__ user 'root' already exists
        \__ user 'root' already exists
```

Listing 37: The goal expresses to create an anonymous user with login name "root"

In this use case we want to create the superuser root. But after installation of a Linux system there is mostly a superuser named "root". The engine suggests "useradd root" but checked the assumptions of useradd and print out that "user 'root' already exists". In the next step the engine tries to find a Web Service to solve the problem. But the assumption is the same as the goal: a user root exists. That's why a second reason message is printed out and the recursion stops.

```
_#1[grp#groupName hasValue "newgroup"] memberOf grp#Group.
\__ groupadd newgroup
```

Listing 38: The goal expresses to create an anonymous group with group name "newgroup"

There is no big difference between useradd and groupadd. This should only demonstrate how to define a goal to create a group.

```
_#1[passwd#login hasValue "root"] memberOf passwd#User and
_#2[grp#member hasValue _#1, grp#groupName hasValue "bin"] memberOf grp#
   Group.
\__ gpasswd -a root bin
```

Listing 39: The goal expresses to add a user "root" to the group "bin"

Both the user "root" and the group "bin" existing in the system. That's why the engine suggests the command "gpasswd" to add (with the option "-a") the user "root" to the group "bin". Moreover the engine propose useradd and groupadd (not shown in the listing) but mentions that the user resp. the group already exists. That's because useradd or groupadd could solve the goal partially.

```
_#1[passwd#login hasValue "root"] memberOf passwd#User and
_#2[grp#member hasValue _#1, grp#groupName hasValue "unkown"] memberOf grp#
   Group.
\__ gpasswd -a root unkown
    \__ can't find group with name 'unkown'
        \__ groupadd unkown
```

Listing 40: The goal expresses to add a user "root" to a unknown group "unknown"

Now we want to add an existing user to a non-existing group. The Web Service gpasswd has the assumption that the group must exist. Therefore the reason "can't find group with name 'unkown' " is returned. The engine tries to find a Web Service to solve the problem of the non-existing group and finds groupadd. So if we invoke the commands from the bottom to the root of the tree the system fulfills the goal.

```
_#1[passwd#login hasValue "unknowuser"] memberOf passwd#User and
_#2[grp#member hasValue _#1, grp#groupName hasValue "unkowngroup"] memberOf
    grp#Group.
\__ gpasswd -a unknowuser unkowngroup
    \__ can't find user with name 'unknowuser'
    |   \__ useradd unknowuser
    \__ can't find group with name 'unkowngroup'
        \__ groupadd unkowngroup
```

Listing 41: The goal expresses to add an unknown user "unknownuser" to an unknown group "unknowngroup"

As expected if we also have a user who is unknown to the system the engine suggests to add the user too.

```
_#1[passwd#login hasValue "root"] memberOf passwd#User and
_#2[fso#owner hasValue _#1, fso#origName hasValue "/bin"] memberOf fso#File
   .
\__ chown root /bin
```

Listing 42: The goal expresses to make the user "root" owner of the file (resp. directory) "/bin"

The engine suggests the command chown (change owner) that will make root owner of the file /bin.

```
_#1[passwd#login hasValue "root"] memberOf passwd#User and
_#2[fso#owner hasValue _#1, fso#origName hasValue "/foo"] memberOf fso#File
   .
\__ chown root /foo
    \__ file '/foo' doesn't exist
```

Listing 43: The goal expresses to make the user "root" owner of the non-existent file "/foo"

If the file doesn't exist the Web Service chown returns a message. Maybe we could implement other Web Services like "touch" to create a file or "mkdir" to create a directory. Then this assumption could solve by one of these commands.

```
_#1[passwd#login hasValue "root"] memberOf passwd#User and
_#2[grp#groupName hasValue "bin"] memberOf grp#Group and
_#3[fso#owner hasValue _#1, fso#group hasValue _#2, fso#origName hasValue
   "/var"] memberOf fso#File.
\__ chown root:bin /var
```

Listing 44: The goal expresses to make the user "root" and the group "bin" owner of the file (resp. directory) "/var"

As expected the engine suggests chown. The command chown has two usages: make user owner of a file or make both user and group owner of a file. One advice is "chown root /var" that we have seen above. But because of another Web Service with another effect and command definition, the engine suggests also "chown root:bin /var". Hence, it is possible to define many Web Services for different invocations of commands.

6.6 Graphical User Interface

After explaining the inner workings of the engine we will show in this section how the user interacts with the GUI (Graphical User Interface).

Today's GUIs are mostly build of menus, text fields and buttons. Because GUIs are interfaces to the computer the user must know how to control these elements and how the computer understand this. GUI-Programs have a range of functions that can be accessed over menus. There are the typical menus like "File", "Edit", "View", etc. But if the range of functions grows menus will overflow and manually finding by iterating thru the menu items takes a long time.

This thesis shows a new way to interact with the GUI. The user has the possibility to search for concepts and instances. The ontology is a shared conceptualization between the computer and the user. If the user understands concepts of the ontology (s)he can better interact with the system.

Further more we will introduce how users can combine concepts and instances to define a goal using a graph. This enables an intuitive usage without learning programming languages. By composing nodes with relations the user expresses a goal. Searching for the right function in a menu that solves the goal is not necessary. In the future with the help of NLP we can imagine that users write down a desire in natural language.

With the idea of Semantic Desktop Services a new workflow accrues. The user defines goals and the computer suggests advices that solve the goal. It's a new way of communication with the system based on request and response. With learning algorithms the system can better guess proper advices.

6.6.1 Tree of Concepts and Instances

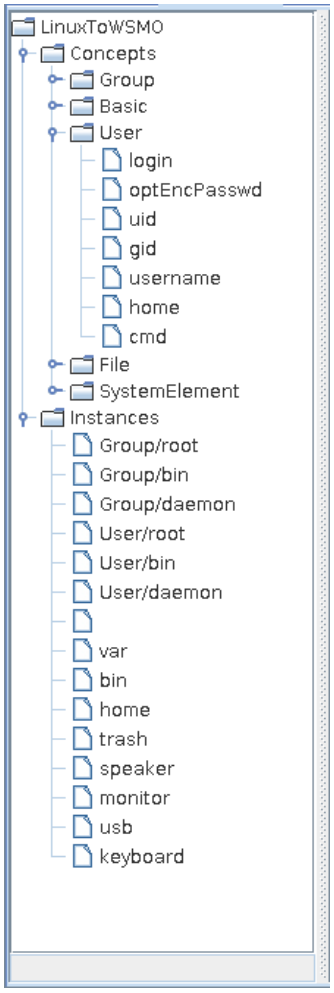

Figure 8: A list of loaded concepts and instances

The GUI of the engine provides a tree of concepts and instances on the left side (see fig. 8). The root of this tree is named “LinuxToWSMO” because the structure is a result of the linux data transformation. The tree is separated in two categories: Concepts and Instances.

In the Concepts part we find for each concept a node. The childrens of a concept node are attributes of this concept. This enables browsing in the concept definitions. More GUI elements such as tooltips could annotate the nodes with more informations. Additionally, we could think of a search engine where the user can find concepts, attributes and instances by name.

In the Instances part we see loaded instances from linux files. To hold the instance set minimal the engine loaded three Groups, Users and Files. For other goal definitions system elements such as trash, speaker, monitor etc. are added. The user can use this system elements to define a goal with resources from the PC. An example will be the usage of the trash instance: The desire of deleting a file can be expressed by the triple “<file> <moveTo> <trash>”.

The interesting part of the LinuxToWSMO-tree is that we can drag concepts and instances and drop them to build up a goal. The user has not to know or type a long URI to express what concept or instance (s)he means.

6.6.2 Goal definition with graph

We have seen that goal definitions are complex logical expressions. At this stage a user is forced to learn the formal syntax to define a valid goal. To prevent this we allow the creation of goals based on graphs.

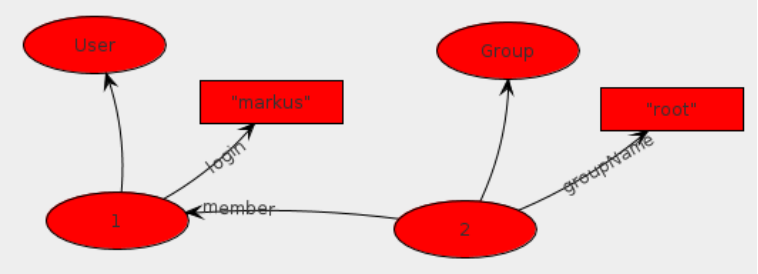

Figure 9: A goal definition represented by a graph

We have seen in section 3.3.3, demonstrated by the two figures 3 and 4, how RDF graphs look like. The GUI follows the same design: Ellipses represent concepts and instances while rectangles represent data values. A directed edge is used to show relationships of concepts and instances.

The user can add concepts or instances to the graph via drag&drop . New created instances are named with anonymous IDs. In figure 9 the user instance has the anonymous ID “1” while the group instance has the ID “2”.

With a right click on an instance a context menu opens and the user can select a valid attribute based on the concept of this instance. By doing so a relation pointing to a data value will be created. For example in figure 9 the user instance has the login relation that points to a data value resource with the value “markus”.

To create a relation between two instances the user can select those two instances and add a relation with the help of a context menu. In figure 9 this is demonstrated by the “member” relation between the user and the group instance.

It is also possible to transform the defined graph in a textual logical expression. If we transform the graph in figure 9 we get:

```
_#2[grp#member hasValue _#1, grp#groupName hasValue "root"] memberOf grp#
   Group and
_#1[passwd#login hasValue "markus"] memberOf passwd#User.
```

Listing 45: Result of graph to logical expression transformation

To sum up, without typing a logical expression in WSML syntax the user can define a goal. With the help of graphs we have a intuitive way to do this. The support of context menus helps to choose valid attributes.

6.6.3 Workflow

In this section we will see how the GUI elements working together. Figure 10 shows the complete GUI of the engine. Step by step we go thru the GUI elements and the workflow of the user:

First, the user has an intension what (s)he wants to do. We imagine as an example that the user wants to add the user “markus” to the group “root”. So (s)he must look at the concept part of the LinuxToWSMO tree. With drag&drop the User and Group concept can be dropped in the right panel.

The second step is the annotation of instances with data values. With a context menu the user adds a login attribute to the User instance and a groupName attribute to the Group instance. With the help of the text field on the left bottom side the user fills the attributes with data.

The third step is the connection of the User and Group instance. Also with a context menu the user can add the member relation.

After this interactions the goal is defined (see fig. 9). The user can now start the Discovery Algorithm.

As predicted the engine suggests advices. In our example there are three advices. The advices are separated in tabs. In each tab the user can see the command tree (many command trees are shown in section 6.5.3). On the right top side there are debug outputs of the Discovery Algorithm.

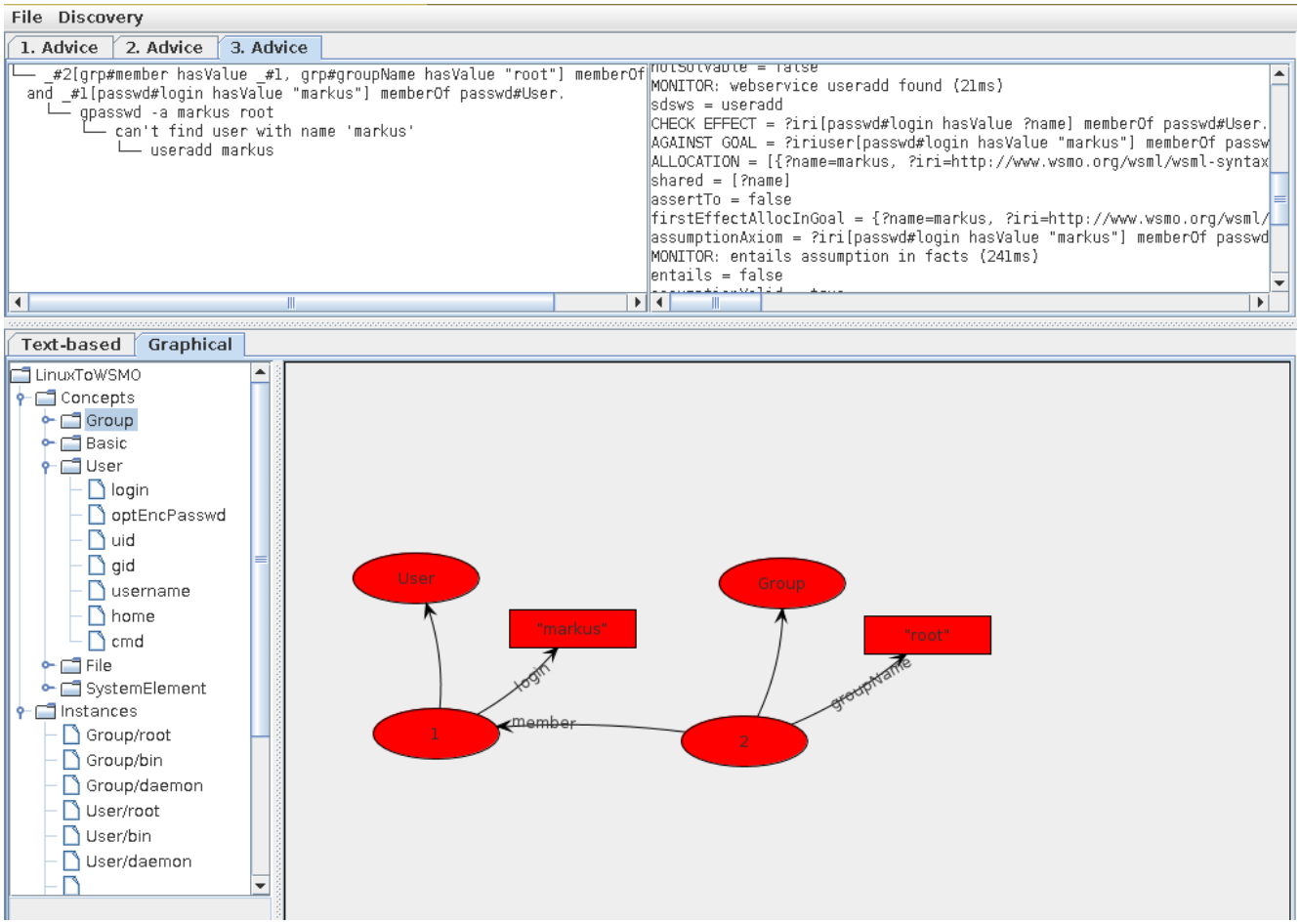

Figure 10: The Graphical User Interface of the Engine

7 Conclusion

We have seen how Web Services can be described and used. We defined the term Web Service and looked at an example taken from Amazon. With the Web Service Role Model we depicted the interaction between Web Services and a Discovery Agency.

To introduce Semantic Web Services we covered the Semantic Web. This thesis explained the main concepts with the help of the Layer Cake of the Semantic Web. The focus was to give the reader an idea of the term Ontology.

After we covered the Semantic Web we could introduce the definition of Semantic Web Services and the life cycle. The research reused the existing approach WSMO. To reuse this technology we explained the WSMO entities and we shown how to describe them with WSML. We have seen how WSMO implements Discovery and Choreography.

To come closer to the desktop environment we presented Linux as a suitable testbed. Many aspects of this operating system can be used to transfer them to the Web Service domain. But we discovered problems as well.

We tried to solve this problems by presenting the new Semantic Desktop Services approach. In the first step human readable knowledge from man pages and files was transformed to an Ontology. In the second step functionality was annotated via WSMO Web Service descriptions. To process this new semantic data this research introduced a prototype called “Engine”. We looked closer to the inner workings of the program and presented some results. At the end the Graphical User Interface of the engine was shown with screenshots.

7.1 Advantages

WSMO provides a lot of predefined MOF models, a own language WSML and the execution environment WSMX. It was easy to reuse parts of WSMX and the libraries. WSMX provides a good architecture to test new Web Service approaches.

Technologies such as Discovery and Choreography are well researched. It was no problem to adopt these ideas because WSMX shows in the source code how to implement them. After reading the code it was easy to get a good understanding how it works.

Linux has demonstrated that we have a lot of unused knowledge and functionality because of the lack of semantic annotations. With the help of Ontologies and Web Service descriptions we improved the processing of system knowledge. We enabled reasoning over Linux knowledge and automated composing of commands.

7.2 Disadvantages

The engine is only a prototype that misses a lot of features. The presented ontologies have a low expressiveness. They don’t have constrains or restriction. Too few translators are provided by the engine. So we can’t build up a big knowledge base. But if we would enlarge the set of instances we would be faced with performance problems.

This thesis presented only simple results. The command tree was not complex enough to show that this approach could be used in the future. Only a small range of tools was described as a Web Service. Further more only the access control domain was covered by some programs.

The Choreography approach was not reused in the engine. So communication between two Web Services was not minded. Instead, discovery was used to allocate variables for communication. But if complexity rises this approach will probably fail.

7.3 Outlook

Here we give an outlook what can be done to solve these problems in the future. This research collects ideas and improvements that will be covered in this section.

7.3.1 More complexity with more Web Services

The more Web Services are added to the engine the better are the results. More assumptions can be solved by executing other commands. That means the command trees becoming bigger and more complex. But also the amount of advices rises. So a user must again choose among possibilities what command tree has the best solution. An approach from the artificial intelligence domain can help to learn what command trees serve the best. A learner can also save the suggested trees for goals to look up if a goal appears twice. But more Web Services means that the performance of the algorithm drops if we iterate thru all Web Services.

7.3.2 Implement choreography

In the prototype there is no choreography implementation at all. The Web Services don't communicate or make contracts as mentioned in 4.2.3. But we have seen in section 4.3.6 that it would be possible.

But this causes more changes in the design and architecture. Services in the Web are interactive agents. They act like servers and give responses to requests. In contrast commands in a Linux environment can only be invoked. That's why a specific agent should run on the computer for each command. These command agents are acting like Services and communicate for example over localhost. Then choreography can govern this communication.

7.3.3 Completely or partially solved

The engine should detect whether the goal is completely or partially solved. The prototype finds all advices and moreover advices that will not solve the problem at all because of invalid assumptions. A learning algorithm could determine how good an advice solved the problem. Sometimes users are happy about solving the problem partially. Therefore a good filter strategy should applied.

Another possibility could be that a manual intervention is needed (if we think about password input for example). Then a partially solved goal could be completely solved by user interaction.

7.3.4 Other Domains

This thesis covered a small field of the Linux access control domain. But other domains could also be possible: filesystem modification, multimedia or communication applications.

For example if we look at an email client program of the communication domain. We need a lot of small functionalities which work together: One Web Service could implement IMAP (Internet Message Access Protocol) to transfer and receive emails. Another Web Service holds email account data. With the help of Ontologies we can model concepts like persons, email-accounts, emails and inboxes. The Web Services communicate over Ontologies and exchange instances of concepts. So it could be thinkable that an email program in the future could consist of many small Web Services which interact in an automated way.

References

[1] D. Aiken et al. Wsmx documentation, Feb. 2005. http://www.wsmo.org/TR/d22/v0.2/20050223/ [Accessed Oct. 6, 2012].

[2] R. Akkiraju et al. Web service semantics - wsdl-s, Nov. 2005. http://www.w3.org/Submission/WSDL-S/ [Accessed Sep. 24, 2012].

[3] Amazon. A translation approach to portable ontology specifications. https://affiliate-program.amazon.com/gp/advertising/api/detail/main.html [Accessed Sep. 21, 2012].

[4] Apache. Commons virtual file system. http://commons.apache.org/vfs/ [Accessed Oct. 12, 2012].

[5] A. Barstow et al. Owl web ontology language for services (owl-s), Nov. 2004. http://www.w3.org/Submission/2004/07/ [Accessed Sep. 24, 2012].

[6] S. Battle et al. Semantic web services framework (swsf), May 2005. http://www.w3.org/Submission/2005/07/ [Accessed Sep. 24, 2012].

[7] O. Berger. Adding adms.sw rdf descriptions of source packages to the pts, Aug. 2012. https://lists.debian.org/debian-qa/2012/08/msg00099.html [Accessed Oct. 16, 2012].

[8] O. Berger. Generating rdf description of debian package sources with adms.sw, Aug. 2012. http://www-public.it-sudparis.eu/~berger_o/weblog/2012/08/24/generating-rdf-description-of-debian-package-sources-with-adms-sw/ [Accessed Oct. 16, 2012].

[9] B. Bishop et al. Wsml2reasoner. http://tools.deri.org/wsml2reasoner/ [Accessed Oct. 6, 2012].

[10] C. Bizer et al. Dbpedia - a crystallisation point for the web of data. http://w.websemanticsjournal.org/index.php/ps/article/download/164/162 [Accessed Oct. 20, 2012].

[11] T. Bowden et al. The /proc filesystem, Oct. 1999. http://www.kernel.org/doc/Documentation/filesystems/proc.txt [Accessed Oct. 5, 2012].

[12] D. Brickley et al. Foaf vocabulary specification, Aug. 2010. http://xmlns.com/foaf/spec/20100809.html [Accessed Sep. 26, 2012].

[13] N. Brown. Ghosts of unix past: a historical search for design patterns, Oct. 2010. http://lwn.net/Articles/411845/ [Accessed Oct. 4, 2012].

[14] C. Bussler et al. Web service execution environment (wsmx), June 2005. http://www.w3.org/Submission/2005/SUBM-WSMX-20050603/ [Accessed Sep. 24, 2012].

[15] J. de Bruijn et al. Web service modeling language (wsml), June 2005. http://www.w3.org/Submission/2005/SUBM-WSML-20050603/ [Accessed Sep. 24, 2012].

[16] J. de Bruijn et al. Web service modeling ontology, June 2005. http://www.w3.org/Submission/2005/SUBM-WSMO-20050603/ [Accessed Jul. 7, 2012].

[17] M. Dimitrov et al. wsmo4j programmers guide, Nov. 2006. http://wsmo4j.sourceforge.net/doc/wsmo4j-prog-guide.pdf [Accessed Oct. 6, 2012].

[18] M. Duerst et al. Internationalized resource identifiers, Jan. 2005. http://www.ietf.org/rfc/rfc3987.txt [Accessed Sep. 23, 2012].

[19] D. C. Fallside et al. Xml schema part 0: Primer second edition, Oct. 2004. [Accessed Oct. 16, 2012].

[20] Free Software Foundation. Gnu manuals online. http://www.gnu.org/manual/manual.html [Accessed Oct. 4, 2012].

[21] Free Software Foundation. Gnu general public license, June 2007. http://www.gnu.org/copyleft/gpl.html [Accessed Oct. 4, 2012].

[22] S. Goedertier. Asset description metadata schema for software. http://joinup.ec.europa.eu/asset/adms_foss/description [Accessed Oct. 16, 2012].

[23] T. Gruber. A translation approach to portable ontology specifications, 1993.

[24] A. Haller et al. Wsmx choreography, June 2005. http://www.wsmo.org/TR/d13/d13.9/v0.1/ [Accessed Oct. 1, 2012].

[25] H.-J. Happel et al. Kontor: An ontology-enabled approach to software reuse.

[26] R. Hasan. History of linux. https://netfiles.uiuc.edu/rhasan/linux/ [Accessed Oct. 4, 2012].

[27] P. Hitzler et al. Semantic web, 2008.

[28] D. Jurafsky and J. H. Martin. Speech and language processing.

[29] M. Kifer et al. Logical foundations of object-oriented and frame-based languages, May 1995. http://www.cs.umbc.edu/771/papers/flogic.pdf [Accessed Sep. 26, 2012].

[30] S. Kleinman. Linux users and groups, Aug. 2009. http://library.linode.com/using-linux/users-and-groups [Accessed Oct. 20, 2012].

[31] G. Klyne et al. Resource description framework (rdf): Concepts and abstract syntax, Feb. 2004. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/ [Accessed Sep. 26, 2012].

[32] J. Kopecky et al. Wsmo use case: Amazon e-commerce service, Jan. 2006. http://www.wsmo.org/TR/d3/d3.4/v0.2/20060113/ [Accessed Oct. 3, 2012].

[33] D. L. McGuinness et al. Owl web ontology language, Feb. 2004. http://www.w3.org/TR/owl-features/ [Accessed Sep. 24, 2012].

[34] OMG. Metaobject facility. http://www.omg.org/mof/ [Accessed Jul. 7, 2012].

[35] J. Plötner et al. Linux - das umfassende handbuch. http://openbook.galileocomputing.de/linux/linux_kap06_006.html [Accessed Oct. 20, 2012].

[36] E. S. Raymond. The art of unix programming, Sept. 2003. http://www.faqs.org/docs/artu/index.html [Accessed Oct. 3, 2012].

[37] D. Roman et al. Ontology-based choreography, Feb. 2007. http://www.wsmo.org/TR/d14/v1.0/ [Accessed Sep. 28, 2012].

[38] L. Sauermann et al. Overview and outlook on the semantic desktop. http://courses.ischool.utexas.edu/Turnbull_Don/2008/fall/INF_385T-SW/readings/Sauermann-2005-Semantic_Desktop.pdf [Accessed Oct. 22, 2012].

[39] R. Stallman. Linux and the gnu system. http://www.gnu.org/gnu/linux-and-gnu.en.html [Accessed Oct. 4, 2012].

[40] R. Stallman. Initial announcement, Sept. 1983. http://www.gnu.org/gnu/initial-announcement.html [Accessed Oct. 4, 2012].

[41] R. Studer et al. Semantic web services, 2007.

[42] S. Weibel et al. Dublin core metadata for resource discovery, Sept. 1998. http://www.ietf.org/rfc/rfc2413.txt [Accessed Sep. 26, 2012].

[43] G. M. Weiss et al. Data mining, 2010. http://storm.cis.fordham.edu/~gweiss/papers/data-mining-chapter-2010.pdf [Accessed Oct. 16, 2012].

[44] M. Zaremba et al. Wsmx architecture, June 2005. http://www.wsmo.org/TR/d13/d13.4/v0.2/20050613/20050613_d13_4.pdf [Accessed Oct. 6, 2012].