WSMX Deliverable

D10 v0.2 Semantic Web Service Discovery

WSMX Working Draft – October 3, 2005

Authors:
Uwe Keller
Rubén Lara
Holger Lausen
Axel Polleres
Livia Predoiu
Ioan Toma[a]

[a]In alphabetic order

Editors:
Rubén Lara
Holger Lausen
Ioan Toma

Reviewers:

This version:
http://www.wsmo.org/TR/d10/v0.2/20051003/

Latest version:
http://www.wsmo.org/TR/d10/v0.2/

Previous version:
http://www.wsmo.org/TR/d10/v0.2/20050307/

Copyright © 2004 DERI ®, All Rights Reserved.
DERI liability, trademark, document use, and software licensing rules apply.

Abstract

Semantic web services promise to add automation and dynamics to current web service technologies, considerably reducing the effort required to integrate applications, businesses and customers. One of the key tasks in the integration process is to locate services that can fulfill the application, business or customer needs. With current web service technologies, this is done mainly manually, which reduces the accuracy of the search and requires considerable effort. Semantic web services aim at providing formal descriptions of requests and web services that can be exploited to automate several tasks in the web services usage process, including dynamic discovery of services.
WSMO discovery provides a conceptual model for service discovery that exploits WSMO formal descriptions of goals and web services. In this document, and based on the theoretical foundations provided by WSMO and WSMO discovery, we investigate the implementation of a discovery engine that can provide dynamic web service discovery. We focus on the language requirements to formalize WSMO goals and web service capabilities, on the integration of different approaches to web service discovery defined by WSMO discovery, and on the definition of the interface and architecture of the discovery engine.

Copyright © 2004 DERI ®, All Rights Reserved.
DERI liability, trademark, document use, and software licensing rules apply.

Contents

1 Introduction **4**

2 Expressivity and Reasoning Support Required **5**
2.1 Keyword-based Discovery . . . 5
2.2 Discovery Based on Simple Descriptions of Services . . . 6
2.2.1 Non-restricted Expressivity . . . 6
2.2.2 DL Restrictions . . . 7
2.3 Discovery Based on Rich Descriptions of Services . . . 8
2.3.1 Set-based modelling . . . 8
2.3.2 Transaction Logic . . . 8

3 Integration **10**
3.1 Description of goals and web services . . . 10
3.1.1 Modelling Styles . . . 10
3.1.2 WSML Variants . . . 10
3.2 Ranking . . . 11

4 Interface **12**

5 Architecture **13**
5.1 Components . . . 13
5.2 Requirements on other components . . . 14

6 Testing and Benchmarking **15**
6.1 Set Based Modelling . . . 15
6.1.1 Use Cases Tested . . . 15
6.1.2 Evaluated Reasoners . . . 18
6.1.2.1 Racer . . . 18
6.1.2.2 Pellet . . . 20
6.2 Benchmarking . . . 22

7 Related efforts **23**
7.1 WSML Reasoning Implementation . . . 23
7.2 WSMX . . . 23
7.3 WSMO4J . . . 23

8 Conclusions and Future Work **24**
8.1 Future Work . . . 24

Bibliography **26**
.1 Abstract syntax for the example . . . 27

1 Introduction

WSMO discovery [Keller et al., 2004a] defines a conceptual framework for locating services that (totally or partially) fulfill a requester goal. This conceptual framework distinguishes three major steps in discovery: goal discovery, web service discovery, and service discovery. Goal discovery is about abstracting a concrete user goal to a pre-defined, reusable, and formalized goal. Web service discovery deals with the matching of formalized goals and formalize web services, selecting web services that can potentially be used to get the desired service. The last step, service discovery, uses the web services matched in the previous step to access the real services behind such web service interfaces, finally checking what services fulfill the requester goal.

In addition, [Keller et al., 2004a] discusses different approaches to the second step of the conceptual model, i.e. web service discovery, which offer different complexity, accuracy levels, and efficiency. Different notions of match and their respective formalizations are discussed for these approaches, providing a theoretical basis for web service discovery.

The aim of this document is to design and implement a discovery engine with well-defined semantics that can be applied in a wide range of scenarios based on the theoretical foundation provided by [Keller et al., 2004a].

In this version of the document, we will restrict ourselves to (different forms of) web service discovery. Further theoretical developments in goal and service discovery, to be provided by future versions of WSMO discovery, will eventually lead to an engine implementation covering the complete discovery process as defined in [Keller et al., 2004a].

The document is structured as follows: Chapter 2 further defines the scope of Web Service discovery and discusses the reasoning support and the language expressivity required for implementing web service discovery. Chapter 3 discusses how the different approaches to web service discovery can work together. The interface of the discovery engine is presented in Chapter 4. Chapter 5 defines the architecture of the engine and poses requirements on components outside it. The use cases and benchmarking of the implemented engine will be presented in Chapter 6. The relation of our discovery engine to the WSMO reference implementation WSMX[1], the WSMO reasoning implementation, and other related efforts is discussed in Chapter 7. Finally, Chapter 8 concludes the document and presents our future work.

[1] `http://www.wsmx.org/`

2 Expressivity and Reasoning Support Required

In this chapter, we discuss the language expressivity required for the different approaches to web service discovery described in [Keller et al., 2004a], together with the reasoning support necessary to find a match in each of these approaches.

The WSML family of languages [de Bruijn et al., 2004] offers a strictly layered set of languages with different expressivity. The different approaches to web service discovery will require different features from the language and, therefore, will use different WSML variants.

In addition, the proof obligations associated to each of the approaches to web service discovery can require different reasoning support to find a proof for a match. Therefore, the reasoning support needed and the available tools offering such support must be identified.

Eventually, the WSML reasoning implementation [de Bruijn, 2004] will offer the reasoning support required for each of the WSML variants, providing the reasoning engine that will be used by the WSMO discovery engine. However until specific WSML reasoning engines are available, our implementation will rely on existing tools. We will orient ourselves along the lines of the language layering approach and syntax proposed in [de Bruijn et al., 2004]. For a first test of feasibility, translations to the actual used implementations will be initially done manually or by simple scripts in order to achieve a working prototype. These efforts and experiments shall later on lead to more complete tool support and more efficient interfacing to the required reasoning engines.

In the following, we will discuss issues related with each of the web service discovery approaches. Here, we will not cover issues such as publication of web services or necessary registries; these issues will be discussed later in this document. Instead, we will assume that a goal and the set of web services to be considered for the matching process are already defined and available to the discovery engine, and we will focus on what language expressivity these require and on what reasoning support the engine needs to find a match.

2.1 Keyword-based Discovery

As described in [Keller et al., 2004a], **keyword-based discovery** is one possible approach to do web service discovery. Although it does not use well-defined semantics and is limited because of the ambiguity of natural language, keyword-based discovery can be used to filter and rank quickly the huge amount of available goal and service descriptions.

In order to design a good keyword-based discovery mechanism some questions must be answered. For example, over what properties of goal/service formalized description the keyword-based discovery must be performed? It is known that each component description in WSMO [Roman et al., 2004] has a `nonFunctionalProperties` section. A basic and intuitive approach is to match the keywords provided by the requester against the keywords from `dc:subject` or against extracted keywords from natural language descriptions provided in `dc:description` elements of `nonFunctionalProperties`. The same approach can be performed by considering other parts of goal/service description, for example `dc:description` elements of axioms that describe goal postconditions or

service capabilities and postconditions. One aspect that must be taken into account is that `nonFunctionalProperties` descriptions are not mandatory. So in case the author of goal/service dose not provide any `nonFunctionalProperties` description, this goal/service can not be discovered using this approach.

Alternatively keyword-based discovery can be used to find relations between keywords provided by the user and concepts from the ontologies that are used to formalize goals/services. This basically implies to match keywords from the user request against the concept names from the ontologies.

Finally, another approach that can be considered is a keyword-based discovery over the logical formulae that are used in goal and service descriptions. A keyword-based on the predicates used is as well possible because usually their names are chosen having in mind the semantics of what these predicates actually do. In essence, a keyword-based search can be performed matching the keywords that the user specified on the one hand and the names of the predicates on the other hand. A pure keyword-based matching as described above can be easily improved using different techniques like: stemming, part-of-speech tagging, phrase recognition, synonym detection etc.

Another question that must be answered is what tools can be used in keyword-based discovery. As a starting point we are considering Zettair [Zettair, 2003], a search engine formerly known as Lucy.

Zettair is a compact and fast search engine that allows indexing and searching of HTML (or TREC[1]) collections. Its primary feature is the ability to handle big collections of documents in a very fast and efficient manner. First an index structure is created and then this structure is queried and an answer is returned. Zettair accepts simple (non-nested) boolean, and phrase queries.

Keyword-based discovery uses basically textual descriptions that are not expressed using logics. In consequence this approach does not pose any requirement on the WSML variant to be used.

2.2 Discovery Based on Simple Descriptions of Services

In this section we consider the approaches for semantics-based discovery for simple semantic annotations of web services that are discussed in [Keller et al., 2004a].

2.2.1 Non-restricted Expressivity

In [Keller et al., 2004a] we discussed a set based modelling approach for web services and outlined its formalization in First-order languages. There we did not impose any restrictions on the language to be used for defining web services and goal. Thus, it is in general required to use a fully-fledged theorem prover for First-order languages in order to check the proof obligations that have been given in that document. Hence, the candidate WSML variants that we consider for this approach basically are WSML Core, WSML DL as well as the monotonic part of WSML Full.

There are a bunch of candidate reasoners available, that could be used for a prototype implementation. We will have a closer look at two specific systems, namely Vampire and SNARK, and the features they support.

Future versions of this document will elaborate on the selection of a concrete theorem prover for a prototype and specific aspects of how to implement the

[1] `http://trec.nist.gov/`

checks for the proof obligations given in [Keller et al., 2004a].

2.2.2 DL Restrictions

In the previous section, no restriction is posed on the allowed expressivity to describe goals and web services and, therefore, a first-order theorem prover is in principle required.

However, if we restrict the expressivity to WSML DL i.e. to a syntactical variant of OWL DL, we can efficiently exploit subsumption reasoning. Available DL reasoners such as RACER[2], FACT++[3] or Pellet[4] provide support for different DL language. RACER provides efficient subsumption reasoning for $\mathcal{SHIQ}$ with incomplete reasoning for nominals. FACT++ supports $\mathcal{SHIF}(\mathcal{D})$. Pellet provides sound and complete reasoning for OWL DL without nominals ($\mathcal{SHIN}(\mathcal{D})$) and OWL DL without inverse properties ($\mathcal{SHON}(\mathcal{D})$), and sound but incomplete for full OWL DL ($\mathcal{SHOIN}(\mathcal{D})$).

In [Li and Horrocks, 2003], RACER is used to classify web services in the T-Box, which is time-consuming but can be done off-line when publishing them. Once the T-Box is classified, experimental results show that checking the subsumption relation between the user request and the web services in the T-Box can be done within 20 milliseconds.

In additon, and as it was shown in the Semantic Web Fred (SWF)[5] project (see [Keller et al., 2004b]), using more expressive logics (First Order Logic) for set based modelling has consequences on the computational complexity on the one hand, but on the other hand the use cases tested did not show the need for additional expressivity wrt. the expressivity supported by current DL reasoners. In addition, the testings done in the context of the SWF project show that using a theorem prover for set-based discovery in principle implies checking every available service individually. For a reduced test set of four available services, the theorem prover required between one and two seconds to determine the existence of a matching service. Under the assumption that eventually a big number of services will be available to the discovery engine, a faster filtering of relevant services before evaluating them one by one is essential to make the discovery scalable.

These results clearly suggest that using DL reasoners to index web services and/or pre-defined goals can be a useful and efficient mechanism to restrict more detailed (and probably more expensive) checking to a (ideally small) subset of all the available web services.

The use of pre-defined, formalized goals is expected in WSMO discovery. If we restrict such goals to be described in WSML DL, we can classify them offline in a DL reasoner T-Box. When publishing a web service, its subsumption relationship with the classified pre-defined goals can be checked, and a wgMediator[Roman et al., 2004] can be generated to link the web service to a pre-defined goal it (totally or partially, depending on the subsumption relation computed) fulfills. In this way, and as goal discovery should result on a (refined) pre-defined goal, and web service discovery is expected to require matching these refined pre-defined goals against published web services, web service discovery is reduced to exploring the web services linked via wgMediators to the used pre-defined goal.

Another option is to classify the web services when they are published. It is

[2] http://www.sts.tu-harburg.de/~r.f.moeller/racer/
[3] http://owl.man.ac.uk/factplusplus/
[4] http://www.mindswap.org/2003/pellet/index.shtml
[5] http://www.deri.at/research/projects/swf/

expected that the number of web services available will be considerably higher than the number of pre-defined goals and, therefore, we would have to deal with bigger T-Boxes and worse classification times. However, these classification times do not necessarily affect the time to answer an incoming goal. Another consideration is that in this case the subsumption relation between a refined pre-defined goal and the classified web services would have to be computed for each discovery request, while in the previous solution subsumption checking is only required once for each web service, and only at publication time. However, notice that in this case we directly obtain the subsumption relation between the concrete refined pre-defined goal and the web services, while in the previous case the direct relation between those is not known but only their relation via the pre-defined goal.

For this approach, the candidate WSML variant is WSML DL, and the candidate reasoners are RACER, FACT++ and Pellet. From these two, RACER and Pellet provide the closest features to the requirements for a WSML DL reasoner, namely reasoning for WSML DL with limited support for nominals, while FACT++ does not support neither nominals nor number restrictions. For these reasons, only RACER and Pellet have been (partially) tested so far (see Section 6).

2.3 Discovery Based on Rich Descriptions of Services

2.3.1 Set-based modelling

In [Keller et al., 2004a] we discussed the extension of the set based modelling approach for web services to rich sematic descriptions which capture the actual relationship between inputs and outputs/effects of web service executions as well and gave the formalization in first-order languages.

In principle, the language expressivity needed for expressing the discussed proof obligations is the same as in the case of simple semantic annotations (without restrictions on the service and goal descriptions), that means we have to do reasoning over First-order formulae. Thus, it is in general required to use a fully-fledged theorem prover for First-order languages in order to check the proof obligations that have been given in that document. Basically, the very same techniques and systems that we discussed in Section 2.2.1 can be used for an implementation here.

The candidate WSML variant that we consider for this approach basically is the monotonic part of WSML Full.

2.3.2 Transaction Logic

The use of transaction logic [Bonner and Kifer, 1998] for web service discovery has been described in [Keller et al., 2004a]. An implementation of a slightly different proof obligation for web service discovery using transaction logic has been presented in [Kifer et al., 2004].

At the moment, is not completely clear what WSML variant will be required. Based on the work done in [Kifer et al., 2004], we envision that built-in equality and meta-modelling will be required. However it is not clear whether non-stratified negation will be needed. In addition, rules reification will be most likely required for the description of pre-defined goals and web services. This is not explicitly considered in any of the WSML variants, but it could be added

to WSML-Rule.

Therefore, we estimate that WSML-Flight or, more likely, WSML-Rule will be used. Additionally, transaction logic will be obviously used to check the proof obligation defined in [Keller et al., 2004a]. However, this does not pose any extra requirement on the language needed to describe goals and web services, but only on the language supported by the prover. Currently, $\mathcal{FLORA}$-2[6] is the only prover which supports transaction logic.

Future versions of this document will further explore the need for rule reification in WSML-Rule and will include the modelling and testing of some simple examples with $\mathcal{FLORA}$-2.

[6] http://flora.sourceforge.net

3 Integration

As discussed in the previous chapter, different approaches to web service discovery will require different WSML variants and different reasoning support. In this chapter, we outline how the different approaches can work together.

3.1 Description of goals and web services

Goals and web services might be described in different ways depending on the kind of web service discovery to be used. For example, postconditions and effects will not include the relation to the input when using simple semantic annotations for discovery, while this relation needs to be described if the functionality of the web service and not only the kind of results it provides is to be considered. In addition, the approaches discussed in the previous chapter will use different WSML variants. This section discusses these issues.

3.1.1 Modelling Styles

Here, we will discuss whether it is possible to use a single goal and web service description for all the kinds of web service discovery and, if so, what kind of translation or preprocessing is needed to adapt the single description to the needs of the concrete approach used.

Ideally, we will need only one service description and, if it is given in a formalism too specific for a certain approach (e.g. if we want to do DL based discovery, but have a rich description of the service using full first-order logic, we would like to extract a abstracted DL description directly from the full description. This abstraction needs to be formalized and its usefulness needs experimental proof.

3.1.2 WSML Variants

In future versions of this deliverable, we will discuss how the different WSML variants used for the different kinds of web service discovery can be made compatible.

Any kind of web service discovery will in principle use the same domain ontologies. Therefore, it is very relevant to be able to describe such ontologies only once, without requiring the duplication of the ontologies to fit the needs of the different approaches. It is foreseen that, provided that such ontologies are described in WSML-Core, a single definition of the ontology will be sufficient. Furthermore, and according to the study presented in [Volz, 2004], most of the currently available ontologies will be expressible in WSML-Core.

However, the descriptions themselves will require different WSML variants. We will study in the future what the implications of this are.

3.2 Ranking

If the difference approaches to web service discovery are to be combined, a ranking of the web services matched should be clearly defined. This requires not only to rank web services matched using e.g. keyword-based discovery, but also to rank these with respect to the matching results given by e.g. discovery based on transaction logic.

If future versions of WSMO also allow for the representation of user preferences and soft constraints (cf. [Arroyo et al., 2004]) in goals descriptions, these should also influence the ranking.

Future versions of this document will investigate this issue.

4 Interface

In this chapter, we discuss what interface the discovery engine will use. We expect the interface to adhere to the following rough description:

Given a formalized goal, an approach to web service discovery (keyword-based, simple, rich), and the notion of match expected to be used at that level (exact, plug-in, etc.), the engine should return a list of web services that are a (total or partial) match, together with a ranking of such web services.

The interface should also be suitable for goal discovery. For this, we expect the interface to accept a keyword description of a goal or a simple goal and to return the (ranked) matching pre-defined goals.

The detailed interface will be described in future versions of this document.

We will have a look at already developed interfaces in different projects, i.e. Semantic Web Fred (SWF), WSMX, and Semantic Web enabled Web Services (SWWS)[1] and take the approaches into account when making a more detailed design.

[1] `http://sws.semanticweb.org/`

5 Architecture

In this chapter, we outline the architecture of the discovery engine, which at the moment only includes the components necessary for web service discovery. Future versions will extend it to goal discovery and service discovery.

5.1 Components

The high-level architecture of the discovery engine is depicted in figure 5.1.

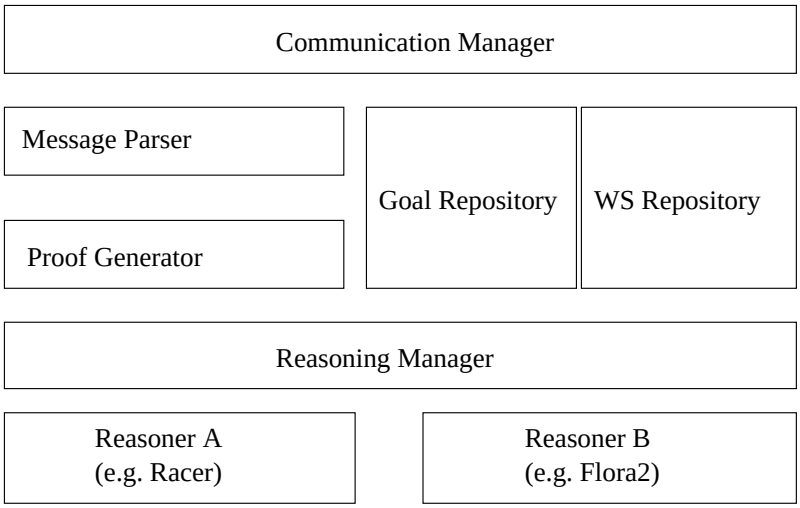

Figure 5.1: Discovery Engine Architecture

The Communication Manager handles the incoming requests, that can be for example goals, the exact messages will be defined in 4. The Message Parser analyzes the incoming message and extracts the message components such as postconditions or effects and hands them to the Proof Generator.

Note: Here we might add a translation step, which transforms the logical expression contained in the incoming WSML messages to the target format of the reasoner. It could be argued that this step is better done within the reasoning manager, however it will be more efficient to construct the proof obligation in the native format of the target reasoner.

Depending on the requested type of match the Proof Generator constructs logical formulas e.g. a subsumption check between two class description for a plug-in match. This step already needs the web service descriptions. Those will also be translated into the format required by the target reasoner.

The Reasoner Manager abstracts from the different interfaces of the reasoners and offers a common API to the Proof Generator. Furthermore such manager is responsible for load balancing the reasoners and network protocol related issues.

Notice that in the architecture the web services repository is part of the discovery engine i.e. we see the discovery engine integrated with the registry. The reason is that if the registry does something more than giving all the web

services (at the same time or one by one) to the discovery engine, it already performs some kind of discovery e.g. keyword-based discovery such as UDDI.

Another option is to assume a registry that contains the complete descriptions of the web services together with a unique identifier for every of them, with an API that provides the complete description of a web service given its identifier. Then, the discovery engine would only need to store a copy of the relevant information for discovery of every web service in the registry, together with its identifier. Complete descriptions would only be retrieved from the registry for matching web services. However, this approach needs to solve synchronization problems between the registry and the (partial) copy stored in the discovery engine.

As outlined in Figure 5.1, we do not only assume a web service repository part of this registry but also a goal repository. This goal repository shall serve to store use of pre-defined, formalized goals mentioned in Section 2.2.2. Abstraction from concrete user goals or natural language descriptions to such pre-defined goals is part of goal discovery. Having a repository of goals allows pre-classification at publish time and linking services from the repository to these pre-defined goals. This linking can also take place at publishing, allowing for more efficient discovery and pre-filtering over the more abstract pre-defined goals in the repository upon a concrete request. Furthermore, concrete requests can, during goal discovery re-use and abstract or refine existing goals from the repository and be published/classified within this goal repository in a closed-loop fashion. We have in the current version not foreseen or discussed the removal of goals or services and the concrete realization of the goal repository and linking between service and goal repository. These issues will be treated in more detail in the context of goal discovery in future versions of this document. For the moment we assume only matching between formalized goals and services, but since we estimate the goal repository being a crucial component for scalable discovery, we already foresee it here. In a first attempt it can simply store given user goals for later re-use or adaption.

The final use we will make of the pre-defined goal repository will also depend on our findings regarding the efficiency of the different types of web service discovery, as discussed in Section 2.2.2.

5.2 Requirements on other components

Here we describe the requirements posed by the discovery engine, both at the interface and functionality level, on other components e.g. registry or ontology repository.

6 Testing and Benchmarking

In this section we follow a bottom-up approach by generating examples that are applied to existing reasoners in order to evaluate which expressiveness of logical descriptions we can use for an implementation based on available technologies.

6.1 Set Based Modelling

First we concentrate ourselves on the set-based modelling approach. We believe it offers a good trade-off between computational complexity and expressivity. Furthermore we restrict ourselves in this version of the document to Description Logics, as DL reasoners allow to off-line classifying the available services and, after these are classified, the response times to determine the subsumption relation between the services and an incoming request are reasonably low [Li and Horrocks, 2003].

6.1.1 Use Cases Tested

In this section we will provide a small use case for discovery. We will specify different goals and services and check the different notions of match. The scenario is loosely based on [Stollberg et al., 2004]. However, in terms of domain ontologies it is greatly simplified in order to concentrate on the proofs necessary for discovery. The complete example, described in abstract syntax, can be found in Appendix **??**.

Domain Knowledge The domain ontology describes a Trip as having a start and an end location; furthermore each trip needs to have exactly one start and one end Location. The subclass NightTrip describes a Trip during the night.

We also describe locations and different sets of Cities; example instances for German and Austrian cities are added in order to check the reasoner support for nominals.

The Explicit disjointness axioms are necessary. Otherwise, all service and goal descriptions would overlap, since the Unique Name Assumption (UNA) does not hold for the DL reasoners considered[1].

[1]Notice that they do employ the UNA at the instance level.

$$
\begin{aligned}
Trip &\sqsubseteq (= 1 start.Location)\\
Trip &\sqsubseteq (= 1 end.Location)\\
NightTrip &\sqsubseteq Trip\\
City &\sqsubseteq Location\\
EuropeanCity &\sqsubseteq City\\
GermanCity &\sqsubseteq EuropeanCity\\
AustrianCity &\sqsubseteq EuropeanCity\\
GermanCity \sqcap AustrianCity &\sqsubseteq \bot\\
Trip \sqcap Location &\sqsubseteq \bot\\
\{Salzburg\} \sqcap \{Innsbruck\} &\sqsubseteq \bot\\
\{Hamburg\} \sqcap \{Frankfurt\} &\sqsubseteq \bot\\
\text{GermanCity(Hamburg), GermanCity(Frankfurt)}&\\
\text{AustrianCity(Innsbruck), AustrianCity(Salzburg)}&
\end{aligned}
$$

Generic Goals As described in [Roman et al., 2004]goals are reusable descriptions of a state that a user wants to achieve. They are expressed by means of effects and postconditions. The separation between postconditions and effects is a conceptual decision, however since they can be dealt with in the very same way [Keller et al., 2004a] for simplicity we only consider postconditions for now.

The three goals represent an international trip from somewhere in Austria to somewhere in Germany; a national trip within Austria and a trip between any two European cities.

$$
\begin{aligned}
Goal\_TripAT2DE &\equiv \exists hasPostCondition.(Trip \sqcap \\
&\qquad \exists start.AustrianCity \sqcap \exists end.GermanCity)\\
Goal\_TripAustria &\equiv \exists hasPostCondition.(Trip \sqcap \\
&\qquad \exists start.AustrianCity \sqcap \exists end.AustrianCity)\\
Goal\_TripEurope &\equiv \exists hasPostCondition.(Trip \sqcap \\
&\qquad \exists start.EuropeanCity \sqcap \exists end.EuropeanCity)
\end{aligned}
$$

Specific Goals In opposite to the generic goals the specific goals are not intended for reuse, but represent refined generic goals that are tailored to the users specific needs. For our preliminary tests we used nominals for the refinement of generic restrictions, e.g. restricting the start location from any Austrian city to a concrete instance (Innsbruck). We believe this is a natural way of refinement and additionally also foresee the need of individuals for rich service descriptions. Since different service descriptions will eventually share the same domain ontologies we use individuals also for the set based discovery.

$$
\begin{aligned}
Goal\_TripIBK2FRA &\equiv \exists hasPostCondition.(Trip \sqcap \\
&\qquad \exists start.\{Innsbruck\} \sqcap \exists end.\{Frankfurt\})\\
Goal\_TripIBK2SZG &\equiv \exists hasPostCondition.(Trip \sqcap \\
&\qquad \exists start.\{Innsbruck\} \sqcap \exists end.\{Salzburg\})
\end{aligned}
$$

Web Services In order to illustrate all different matches we include four web services in our example. A VTA service that provides tickets for Germany and Austria, the national ticket providers for Germany and Austria (DB resp. OEBB) and a special service that provides tickets for Trips during the night but only from Salzburg or Innsbruck to some German City.

$$\begin{aligned}
Service_VTA &\equiv \exists hasPostCondition.(Trip \sqcap \\
&\quad \exists start.(GermanCity \sqcup AustrianCity) \sqcap \\
&\quad \exists end.(GermanCity \sqcup AustrianCity)) \\
Service_DB &\equiv \exists hasPostCondition.(Trip \sqcap \\
&\quad \exists start.GermanCity \sqcap \exists end.GermanCity) \\
Service_OEBB &\equiv \exists hasPostCondition.(Trip \sqcap \\
&\quad \exists start.AustrianCity \sqcap \exists end.AustrianCity \\
Service_DBNight &\equiv \exists hasPostCondition.(NightTrip \sqcap \\
&\quad \exists start.\{Innsbruck \sqcup Salzburg\} \sqcap \exists end.GermanCity)
\end{aligned}$$

Matchmaking Given this set of services we can now analyze the different kinds of matches according to the definitions given in [Keller et al., 2004a]. Keller et al. provide a formalization based on first order logic, the examples given above can be rewritten, e.g. for a web service to the form $\mathcal{W} : \forall x.(\psi(x) \leftrightarrow ws_{\mathcal{W}}(x))$.

We start with the strongest notion match (equivalence) and iteratively proceed to non-matching services.

Equivalence Match An equivalence (or exact) match corresponds to $Service \equiv Goal$. For the given set of services the following match can be deducted:

$$(1) Goal\_TripAustria \equiv Service\_OEBB$$

Plug-in Match Plug-in matches are defined as $Goal \sqsubseteq Service$, i.e. the relevant set of objects of the web service is a superset of the relevant set of objects of the goal. The VTA Service can provide all goals, additionally the national ticket provider for Austria also provides Tickets for Innsbruck to Salzburg.

$$\begin{aligned}
(2) Goal_TripAT2DE &\sqsubseteq Service_VTA \\
(3) Goal_TripIBK2FRA &\sqsubseteq Service_VTA \\
(4) Goal_TripAustria &\sqsubseteq Service_VTA \\
(5) Goal_TripIBK2SZG &\sqsubseteq Service_VTA \\
(6) Goal_TripIBK2SZG &\sqsubseteq Service_OEBB
\end{aligned}$$

Subsumes Match Subsumes matches are defined as $Service \sqsubseteq Goal$, i.e. the relevant set of objects of the web service is a subset of the relevant set of objects of the goal. For example the goal to travel within Europe can be partially fulfilled by all available services.

$$\begin{aligned}
(7) Service_VTA &\sqsubseteq Goal_TripEurope \\
(8) Service_DB &\sqsubseteq Goal_TripEurope \\
(9) Service_OEBB &\sqsubseteq Goal_TripEurope \\
(10) Service_DBNight) &\sqsubseteq Goal_TripEurope
\end{aligned}$$

Intersection Match Intersection matches are defined as $\neg(Service \sqcap Goal \sqsubseteq \bot)$. We only consider know those matches that are not already mentioned above:

$$(11) \neg(Goal\_TripIBK2FRA \sqcap Service\_DBNight \sqsubseteq \bot)$$

All other pairs of Services and Goals are disjoint, i.e. have no common instance.

6.1.2 Evaluated Reasoners

In the following we do a two-fold evaluation. First we analyze if the Reasoners are capable of classifying the given goals and services correctly and, second, we evaluate if the implementation offers support for the queries needed for the different matching notions.

For the first step we created an OWL description of the DL expressions and gave the complete set of axioms (domain knowledge, services and goals) into the reasoner for classification. From the resulting subsumption tree equivalent, Plug-in and subsumes matches can be easily seen. For testing intersection between services and goals theoretically each goal-service pair needs to be checked for the satisfiability of its intersection. Since this is expensive and the concrete goals are not known a priori, we insert for each Goal G its negation $\neg G$ [Li and Horrocks, 2003][2]. All Services that are subsumed by $\neg G$ are disjoint with respect to G and, accordingly, all Services that have not been found by the previous matching notions and are not disjoint with G are classified as intersection matches.

For the first test, we choose Racer and Pellet. Racer is well known for its optimizations, Pellet was chosen since the OWL Tests [3] of the W3c indicated a high coverage of OWL DL.

6.1.2.1 Racer

Classification Result Racer performed as indicated by the hierarchy of Figure 6.1.

With respect to the expected classification (matches) it can be seen that matches (1), (2), (4), (7), (8) and (9) were correctly detected, while the others failed. Failed matches correspond to the goals using nominals.

The RACER manual [Haarslev et al., 2004] acknowledges the problems with the use of nominals i.e. using instances in class definitions and, in fact, it states that "the use of DAML files with individuals in concept terms with the RACER system is not recommended"[4].

Nominals are treated as disjoint (atomic) concepts, and they are also represented in the ABox as instances of a concept with the same name [Haarslev et al., 2004]. When trying a subset of the previous example including the services and domain knowledge but not the goals, with the introduction of a new service using as nominals Hamburg and Frankfurt, we get the hierarchy of Figure 6.2.

We can see that concepts are created for the nominals used. However, these concepts are not subsumed by the concepts GermanCity and Austrian City. On the contrary, when getting their definition using RICE[5] and the RACER syntax:

[2] Notice that [Li and Horrocks, 2003] states that services subsuming but not equal to $\neg G$ are the intersection matches. However, this is proved to be wrong by our example of intersection match.

[3] `http://www.w3.org/2003/08/owl-systems/test-results-out`

[4] Similar restrictions apply to the use of OWL.

[5] A RACER client (`http://www.b1g-systems.com/ronald/rice/`).

Figure 6.1: Taxonomy inferred by Racer

(get-concept-definition Frankfurt)

We get:

$$Frankfurt \equiv \neg OTHER0$$

where $OTHER0$ is a concept introduced by racer with empty definition. However, and as expected, if we explicitly introduce new concepts for the instances of Innsbruck, Salzburg, Frankfurt and Hamburg and describe the services and goals in terms of these concepts, all the matches expected are correctly found by RACER.

In conclusion, RACER did not deal properly with the nominals introduced in our examples, but when replaced by concepts the inferences drawn by RACER were the expected ones.

Interface RACER provides a HTTP-based DIG [Bechhofer, 2000] interface. In addition, it provides file and TCP interfaces, accepting KRSS-based [Patel-Schneider and Swartout, 1993] (RACER), RDFS, DAML+OIL, OWL and DIG syntaxes.

We have experimented the use of RACER to classify the test services to later ask for the subsumption relation between the concepts describing the goals and the services in the hierarchy. RACER provides a sufficient interface for this task. As an example, given the goal $Goal\_TripAustria$ we have posed the following queries (using RACER syntax and TCP interface) to RACER:

Equivalence matches We request the concepts that are synonyms of (equivalent to) the goal:

(concept-synonyms (SOME hasPostCondition
(AND Trip (SOME start AustrianCity) (SOME end AustrianCity))))

getting as a result the service $Service\_OEBB$, as expected.

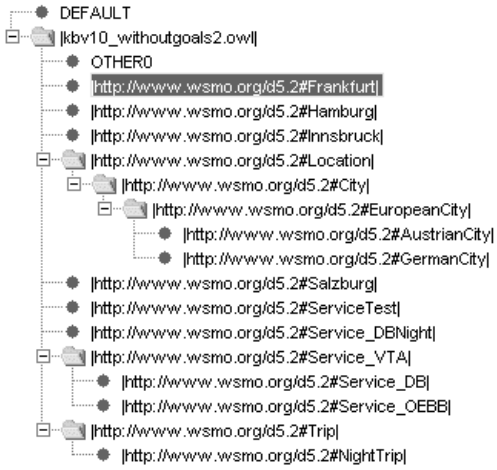

Figure 6.2: Concepts introduced by RACER for nominals

Plug-in matches We request the concepts that are ancestors of (subsume) the goal:

(concept-ancestors (SOME hasPostCondition
(AND Trip (SOME start AustrianCity) (SOME end AustrianCity))))

getting as a result the service $Service\_VTA$, as expected.

Subsumes matches We request the concepts that are descendants of (subsumed by) the goal:

(concept-descendants (SOME hasPostCondition
(AND Trip (SOME start AustrianCity) (SOME end AustrianCity))))

getting none as a result, as expected.

Intersection matches We request the concepts that are descendants of (subsumed by) the negated goal:

(concept-descendants (NOT (SOME hasPostCondition
(AND Trip (SOME start AustrianCity) (SOME end AustrianCity)))))

getting $SERVICE\_DB$ as a result, as expected. Notice that we get the services that are disjoint, as explained before.

6.1.2.2 Pellet

Classification Result Pellet performed as indicated by the hierarchy tree depicted in figure 6.3. As described in the introduction of this section, we classified Goals ($Goal\_xxx$), service ($Service\_xxx$) and the negation of Goal ($NOT\text{-}Goal\_xxx$). We can see that the matches (1) to (10) have been correctly recognized, however Pellet failed to find correct intersection matches using the negated Goals and identified too many intersection matches, e.g. $NOT\text{-}Goal\_IBK2FRA$ does not subsume $Service\_OEBB$, although it should. Please notice that this only occurs when using nominals; for class descriptions without nominals Pellet correctly provides the expected classifications.

However, when testing Pellet with explicit satisfiability tests for the intersection of all goals with all services it correctly detected satisfiable and non satisfiable intersections. Unfortunately using these explicit statements requires not only one query, but n queries (n being the number of Services in the registry), for a single instantiated goals, thus reducing the scalability of our approach.

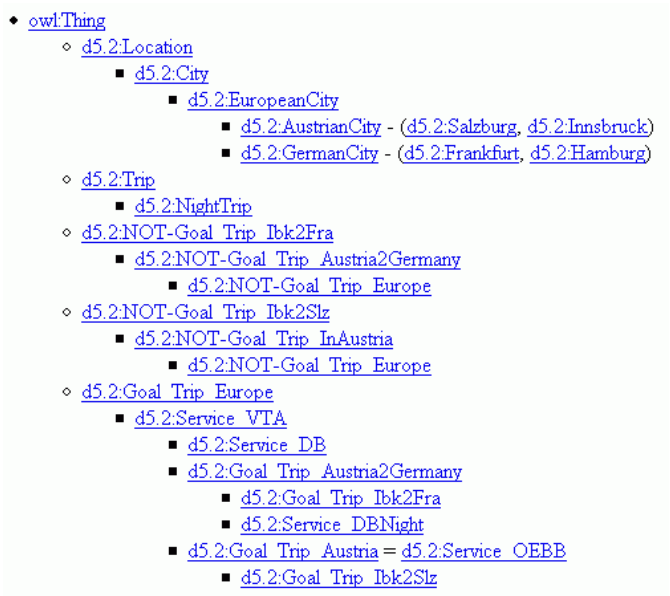

Figure 6.3: Taxonomy inferred by Pellet

Interface Pellet does not support the DIG interface, but can be used directly from a programming environment using its Java API. This API implements all interfaces as they are defined in the OWL API[6]. In the following we briefly discuss how the available methods can be used for checking the different types of matches.

$$\begin{aligned} Set &: equivalentClassesOf(OWLDescriptiond) \\ Set &: ancestorClassesOf(OWLDescriptiond) \\ Set &: descendantClassesOf(OWLDescriptiond) \end{aligned}$$

The API assumes that the component calling it has already an in memory model of the descriptions that need to be classified. Each such description is of type $OWLDescription$. The usage of these methods for equivalence, plug-in, and subsumes matches are straightforward. To check intersection matches we can either requests the concepts that are descendants of the negation of the OWL Description, or using $bolean : isConsistent(OWLDescription\, d1)$ to check each intersection separately. Negation and intersection of $OWLDescription$ can be achieved by using the API ($OWLNot$ and $OWLDisjointClassesAxiom$).

[6]They implement all interfaces specified in the package "org.semanticweb.owl.inference", cf. `http://owl.man.ac.uk/api.shtml`

6.2 Benchmarking

The benchmarking of the engine considering the different types of web service discovery and the use cases above will be included in this section.

7 Related efforts

In this chapter, we outline what the efforts related to the discovery engine are, and what is the nature of such relation.

7.1 WSML Reasoning Implementation

In [de Bruijn, 2004] an ongoing effort to provide a reasoning implementation for the different WSML variants is presented. The reasoner that will result from this work is of high relevance for the implementation of the discovery engine. As discussed in previous sections, our first outline of the discovery engine considers different existing reasoners for providing the reasoning support required for web service discovery. However, as [de Bruijn, 2004] will provide an integrated reasoner for all the WSML variants, it is our intention to use this reasoner for the final implementation of the discovery engine.

7.2 WSMX

The Web Services Execution Environment (WSMX) is an execution environment for dynamic mediation, selection and invocation of web services. WSMX is the reference implementation of Web Service Modeling Ontology (WSMO) [Roman et al., 2004].

The current version of WSMX includes a very basic discovery implementation. Part of the current discovery functionality in WSMX is provided by WSMX Matchmaker. This component implements a very simple matchmaking algorithm based on string matching.

The integration of the discovery engine with WSMX will be studied in the future version of this document.

7.3 WSMO4J

WSMO4J[1] will provide a data model in Java for WSML and (de-)serializers for the different WSML syntaxes. It will additionally provide wrappers for different Reasoners, the first one according to the current plans will be KAON2[2]. Others are not yet scheduled by the core development team, however also third parties can extend this open source API.

The discovery engine will rely on WSMO4J for the parsing of incoming messages and use its Java model for internal processing. This will include the transformation of logical formulas. The fine grained in memory representation of the expressions can be used to facilitate the mapping.

[1] http://wsmo4j.sourceforge.net/

[2] `http://sourceforge.net/projects/kaon2/`

8 Conclusions and Future Work

In this deliverable we have provided a first overview of how the WSMO discovery engine will perform web service discovery, and of the WSML variants and reasoning support required. We have briefly outlined the problems that are to be solved to integrate the different approaches to web service discovery. In addition, we have presented our first ideas on the interface and architecture of the engine. In this version, we have focused on set-based discovery with a simple test example and we have evaluated two existing DL reasoners. The conclusion of our experiments is that current reasoners present various limitations when dealing with nominals. Finally, we have identified related efforts and how they are expected to contribute to the development of our discovery engine.

8.1 Future Work

The immediate plans for the next version are to evaluate how the limitations of current DL reasoners to work with nominals can be overcome. Here, we foresee two main lines we can follow:

- Forbidding the use of nominals for set-based discovery and considering instance values only in a more detailed evaluation of the services filtered using DL reasoners, or
- providing an algorithm for replacing instances in services descriptions and requested goals by corresponding concepts, as was already tried for our test example.

While the first approach limits the accuracy of the (efficient) filtering of services using set-based discovery, the second one can require the introduction of a high number of new concepts for the instances existing in the domain ontologies used.

In addition to the previous point, we will concentrate in the next version on outlining how the rich logical expressions for describing services and goals will look like, and how these can be abstracted to DL expressions, and on making explicit our assumptions on the discovery engine and how these condition the architecture proposed.

Future work also includes a more detailed evaluation of the languages required, further ideas on how the different approaches to web service discovery can be integrated, and more detailed interface and architecture definitions. Continuing the modelling of test examples for the different approaches and the testing of the reasoners required will be also included in future versions of this document. The evolution of WSMO discovery [Keller et al., 2004a] will be closely followed and this document updated accordingly. Finally, our work will lead to an implemented engine for WSMO discovery.

Acknowledgements

The work is funded by the European Commission under the projects DIP, Knowledge Web, SEKT, SWWS, and Esperonto; by Science Foundation Ireland

under the DERI-Lion project; and by the Vienna city government under the CoOperate program.

The editors would like to thank to all the members of the WSML working group for their advice and input into this document.

Changelog

The following major updates have been done since the October 29th version of the deliverable:

- The testing of a concrete tool (Zettair) for keyword-based discovery has been proposed in Section 2.1.
- The discussion on the reasons to try DL reasoners have been updated in Section 2.2.2.
- A goal repository has been introduced in the architecture and possibilities for exploiting it have been discussed in Section 5.1.
- We have introduced a simple test example for set-based discovery using DL reasoners and experimented with RACER and Pellet, identifying limitations in the use of nominals and possibilities to overcome them (Section 6.1.1.
- Appendix **??** with the test example in abstract syntax has been added.
- The plans for the next version are outlined in Section 8.1.

Bibliography

[Arroyo et al., 2004] Arroyo, S., Bussler, C., Kopecký, J., Lara, R., Polleres, A., and Zaremba, M. (2004). Web service capabilities and constraints in WSMO. In *W3C Workshop on Constraints and Capabilities for Web Services*, Oracle Conference Center, Redwood Shores, CA, USA.

[Bechhofer, 2000] Bechhofer, S. (2000). The dig description logic interface: Dig/1.0. Technical report, University of Manchester.

[Bonner and Kifer, 1998] Bonner, A. and Kifer, M. (1998). A logic for programming database transactions. In Chomicki, J. and Saake, G., editors, *Logics for Databases and Information Systems*, chapter 5, pages 117–166. Kluwer Academic Publishers.

[de Bruijn, 2004] de Bruijn, J., editor (2004). *WSML Reasoning Implementation*. WSML Working Draft D16.2v0.1. Available from http://www.wsmo.org/2004/d16/d16.2/v0.1/.

[de Bruijn et al., 2004] de Bruijn, J., Foxvog, D., Lausen, H., Oren, E., Roman, D., and Fensel, D. (2004). The WSML Family of Representation Languages. Deliverable D16v0.2, WSML, `http://www.wsmo.org/wsml/`. Available from `http://www.wsmo.org/2004/d16/v0.2/`.

[Haarslev et al., 2004] Haarslev, V., Moeller, R., and Wessel, M. (2004). *RACER Users Guide and Reference Manual Version 1.7.19*.

[Keller et al., 2004a] Keller, U., Lara, R., and Polleres, A., editors (2004a). *WSMO Discovery*. WSMO Working Draft D5.1v0.1. Available from `http://www.wsmo.org/2004/d5/d5.1/v0.1/`.

[Keller et al., 2004b] Keller, U., Stollberg, M., and Fensel, D. (2004b). Woogle meets semantic web fred. In *Proceedings of the Workshop on WSMO Implementations (WIW 2004)*, CEUR-WS.org/Vol-113/.

[Kifer et al., 2004] Kifer, M., Lara, R., Polleres, A., Zhao, C., Keller, U., Lausen, H., and Fensel, D. (2004). A logical framework for web service discovery. In *Workshop on Semantic Web Services at ISWC 2004*.

[Li and Horrocks, 2003] Li, L. and Horrocks, I. (2003). A software framework for matchmaking based on semantic web technology. In *Proceedings of the 12th International Conference on the World Wide Web*, Budapest, Hungary.

[Patel-Schneider and Swartout, 1993] Patel-Schneider, P. and Swartout, B. (1993). *Description Logic Knowledge Representation System Specification from the KRSS Group of the ARPA Knowledge Sharing Effort*.

[Roman et al., 2004] Roman, D., Lausen, H., and Keller, U. (2004). Web service modeling ontology standard (WSMO-standard). Working Draft D2v1.0, WSMO. Available from http://www.wsmo.org/2004/d2/v1.0/.

[Stollberg et al., 2004] Stollberg, M., Lausen, H., Polleres, A., and Lara, R. (2004). Wsmo use case modeling and testing. Working Draft D3.2v0.1, WSMO. Available from http://www.wsmo.org/2004/d3/d3.2/v0.1/.

[Volz, 2004] Volz, R. (2004). *Web Ontology Reasoning with Logic Databases*. PhD thesis, AIFB, Karlsruhe.

[Zettair, 2003] Zettair (2003). The Zettair Search Engine. http://www.seg.rmit.edu.au/zettair/.

.1 Abstract syntax for the example

```
Namespace(rdf   = <http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
Namespace(xsd   = <http://www.w3.org/2001/XMLSchema#>)
Namespace(rdfs  = <http://www.w3.org/2000/01/rdf-schema#>)
Namespace(owl   = <http://www.w3.org/2002/07/owl#>)
Namespace(a     = <http://www.wsmo.org/d5.2#>)

Ontology( <http://www.wsmo.org/d5.2>

 ObjectProperty(a:end
  domain(unionOf(a:Trip))
  range(a:Location))
 ObjectProperty(a:hasPostCondition Functional)
 ObjectProperty(a:start
  domain(unionOf(a:Trip))
  range(a:Location))

 Class(a:AustrianCity partial
  a:EuropeanCity)
 Class(a:City partial
  a:Location)
 Class(a:EuropeanCity partial
  a:City)
 Class(a:GermanCity partial
  a:EuropeanCity)
 Class(a:Goal_TripHam2Fra complete
  restriction(a:hasPostCondition someValuesFrom(intersectionOf(
    a:Trip restriction(a:end value (a:Frankfurt))
    restriction(a:start value (a:Hamburg))))))
 Class(a:Goal_TripIbk2Fra complete
  restriction(a:hasPostCondition someValuesFrom(intersectionOf(
    restriction(a:start value (a:Innsbruck)) a:Trip
    restriction(a:end value (a:Frankfurt))))))
 Class(a:Goal_TripIbk2Germany complete
  restriction(a:hasPostCondition someValuesFrom(intersectionOf(
    restriction(a:start value (a:Innsbruck)) a:Trip
    restriction(a:end someValuesFrom(a:GermanCity))))))
 Class(a:Goal_TripIbk2Slz complete
  restriction(a:hasPostCondition someValuesFrom(intersectionOf(
    restriction(a:start value (a:Innsbruck)) a:Trip
    restriction(a:end value (a:Salzburg))))))
 Class(a:Goal_TripInAustria complete
  restriction(a:hasPostCondition someValuesFrom(intersectionOf(
    restriction(a:start someValuesFrom(a:AustrianCity))
    a:Trip restriction(a:end someValuesFrom(a:AustrianCity))))))
 Class(a:Goal_TripInEurope complete
  restriction(a:hasPostCondition someValuesFrom(intersectionOf(
    a:Trip intersectionOf(restriction(a:start someValuesFrom(a:EuropeanCity))
    restriction(a:end someValuesFrom(a:EuropeanCity)))))))
 Class(a:Goal_TripInGermany complete
  restriction(a:hasPostCondition someValuesFrom(intersectionOf(
    a:Trip restriction(a:start someValuesFrom(a:GermanCity))
    restriction(a:end someValuesFrom(a:GermanCity))))))
 Class(a:Location partial)
 Class(a:Service_DB complete
  restriction(a:hasPostCondition someValuesFrom(intersectionOf(
    restriction(a:end someValuesFrom(a:GermanCity))
    a:Trip restriction(a:start someValuesFrom(a:GermanCity))))))
 Class(a:Service_DBNight complete
  restriction(a:hasPostCondition someValuesFrom(intersectionOf(
    restriction(a:end someValuesFrom(a:GermanCity))
```

```
   restriction(a:start someValuesFrom(oneOf(a:Salzburg a:Innsbruck)))))))
Class(a:Service_OEBB complete
 restriction(a:hasPostCondition someValuesFrom(intersectionOf(
   restriction(a:start someValuesFrom(a:AustrianCity))
   a:Trip restriction(a:end someValuesFrom(a:AustrianCity))))))
Class(a:Service_VTA complete
 restriction(a:hasPostCondition someValuesFrom(intersectionOf(
   restriction(a:end someValuesFrom(unionOf(a:AustrianCity a:GermanCity)))
   a:Trip restriction(a:start someValuesFrom(unionOf(a:AustrianCity a:GermanCity)))))))
Class(a:TrainTrip partial
 a:Trip)
Class(a:Trip partial
 restriction(a:end cardinality(1))
 restriction(a:start cardinality(1)))

Individual(a:Frankfurt
 type(a:GermanCity))
Individual(a:Hamburg
 type(a:GermanCity))
Individual(a:Innsbruck
 type(a:AustrianCity))
Individual(a:Salzburg
 type(a:AustrianCity))

DisjointClasses(a:AustrianCity a:GermanCity)
DisjointClasses(a:AustrianCity a:GermanCity)
```