Abstract

The Web Service Modeling Ontology (WSMO) provides the conceptual framework for semantically describing web services and their specific properties. Based on WSMO the Web Service Modeling Language (WSML) implements this conceptual framework in a formal language for annotating web services with semantic information.

In this document we present and discuss how semantic descriptions of web services based on WSMO and WSML can be exploited for various tasks in the design and dynamical composition of web-service-based software-systems.

In particular, we investigate and analyze what kind of statements need to be formally checked (resp. proven) in order to support the various tasks occurring during this dynamic system-construction process mechanically and allow these processes to access the semantic information captured in the description. These formal statements are called proof obligations.

Together, WSMO and WSML as well as the semantic tool suite described and developed within deliverable D5 constitute the necessary semantic infrastructure that is needed for making semantic web services come true. Only such an infrastructure allows us to exploit the full-potential of web service technology in the context of Enterprise Application Integration (EAI) and fully-dynamic eCommerce.
Contents

1 Introduction ................................................. 4
   1.1 Motivation ........................................... 4
   1.2 Goal of this document ................................. 6
   1.3 Overview of the paper ................................. 6

2 Semantic support in the context of WSMO and WSML ...... 7
   2.1 Motivation ........................................... 7
   2.2 Where can semantic annotation be exploited? ......... 7

3 What do we want to prove? .................................. 9
   3.1 What is a proof obligation? ............................. 9
   3.2 Service Discovery .................................... 9
      3.2.1 Goal-Capability-Matching ........................ 13
      3.2.2 Exploiting Mediators for Web Service Discovery . 18

4 Concrete Proof Obligations .................................. 20
   4.1 Proof obligations in the context of Service Discovery . 20
      4.1.1 Capability matches Goal .......................... 21
   4.2 Service discovery using logic programming .............. 25
      4.2.1 Goal as ground facts ............................. 26
      4.2.2 Transaction logic reasoning ........................ 28
      4.2.3 Open points ...................................... 30

5 Conclusions .................................................. 32
   5.1 What have we achieved so far? ........................ 32
   5.2 Future Work ........................................... 32

6 Acknowledgements ........................................... 34

A Further Notes on Service Discovery ......................... 35
1 Introduction

1.1 Motivation

The World Wide Web (WWW) has been invented in 1989 by Tim Berners-Lee [BLFD99] and since then changed the way people gather and access knowledge in everyday life drastically. At its invention, nobody was able to foresee the effects and influences of this new promising technology. In fact, the WWW can be considered as one of the most influential technologies that have been invented in the 20th century.

Nowadays the World Wide Web is certainly the biggest information repository on earth and contains a significant amount of the knowledge of mankind. But the major bottleneck when it comes to exploiting the represented information is how to find the required knowledge: the Web is still a mainly syntactically-driven information repository, where information is represented in form of web pages that actually lack to capture the semantics of their contents. As a consequence, the content of the pages is not easily accessible by computers but instead exclusively intended for consumption by human beings. Given the size and the rapid growth of the WWW, this approach doesn’t scale up.

This has lead to significant efforts on the invention of the so-called Semantic Web [BLHL01, DFvH03, FHLW03], that will enable computerized agents to easily gather information in the Web and exploit this knowledge on behalf of their human clients. By enriching the information repository WWW with machine-processible semantics, the annoying bottleneck shall become overcome.

From Web Services to Semantic Web Services. Web services add a new level of abstraction on the current Web. Whereas at the moment the Web mainly is constituted by simple documents (e.g. static and dynamic web pages), in the future the Web might become a repository of electronic (web) services. Each service represents a computational entity that can be accessed and invoked over the Internet. Thus, the Web might become a huge, distributed, heterogeneous and shared computational device.

Recently, web services gain more and more interest in the industry as well as among academics. This is partly due to what they promise: a uniform infrastructure for accessing software-systems over the Internet which can be used for integration of legacy systems – even across enterprise boundaries. Nonetheless, though web service simplify enterprise application integration drastically, the technologies around web services – namely SOAP, WSDL and UDDI – do not solve the integration problem by themselves. Again, the integration technologies are exclusively syntactical means that lack the consideration of semantics.

Moreover, given the growing industrial interest in web services, one can expect a huge amount of web services for various tasks and computations. Hence, the same bottleneck as for the conventional WWW arises: How can we find a web service that solves one of our problem at hand? The corresponding technologies SOAP, WSDL and UDDI can resolve the bottleneck only partially. Again, machine-processable semantics for web-services is needed in order to ensure access of web services for agents without human interaction. Adding semantics to web services is the main prerequisite for an approach that scales.

This semantically enriched web services are called Semantic Web Services. In a sense, we can use the following equation in order to describe the relationship between web services and semantic web services symbolically:

\[
\text{Semantic Web Service} = \text{Web Service} + \text{Semantic Annotation}
\]
And what do we do with all that Semantics? The major benefit of any form of specification is that by explicitly documenting what an artifact actually does, it becomes a lot easier for humans (who don’t know that artifact before) to comprehend what this thing actually can be used for.

Clearly, with a formal representation of any specification, the use of an artifact becomes more precise\footnote{Usually even unambiguous in a mathematical sense!} and eventually accessible for \textit{computerized} agents.

Knowledge about the capabilities of single objects can be exploited for combining these objects to systems which solve more complex problems than each of their elements.

In general, the system-construction process itself is a highly creative task – at least it requires an understanding of the semantics of the single elements that can be combined to a system. Regarding the design of software architectures for instance \cite{AC97a,AC97b} show the benefit of formal specification of architectural elements (in particular so-called \textit{connectors}) for automated and semi-automated validation of architecture designs and the potential impact on software quality in early phases of the software development process. In these works for example formal descriptions of interactions are exploited to check whether given elements of an architecture that are supposed to interact can actually interact without any problems.

Each software system is built for some purpose: to solve a specific well-defined problem of a client or organization. Software architectures themselves are the conceptual means for describing how the functionality of a system is provided and how the given requirements are actually satisfied. In this sense software architectures \cite{GS96,BCK03} bridge the gap between the user requirements and the level of computational units that provide some well-defined functionality and interact and communicate with each other.

In a world of rapidly changing requirements and incomplete knowledge (such as the Semantic Web with its open and heterogeneous user community), the problems to be solved most likely occur rather ad-hoc and can’t be foreseen in detail. In this case, the construction of a system that solves the problem at hand, in general can’t be done in advance, but has to be done on the fly instead. Thus, software-systems composed by services in the Web constitute a particularly interesting and challenging problem for the theory and practice of software architecture: how to dynamically construct systems out of existing ones in a goal-driven way.

In case of a formal specification, the knowledge about the possible elements of a system to solve a problem is represented in a \textit{machine-processable} manner.

On the other hand – when considering the semantic web –, all efforts in semantic annotation (resp. specification or description) of web services are far from being actually effective as long as there is no way for \textit{computerized} agents (instead of human beings) to \textit{actually process} this kind of information about the services. Only then, the dynamic composition of software-systems could be achieved.

In this context, „processing“ means to apply the formalized knowledge \textit{intelligently} to solve a given problem, that is, to actually exploit the semantics of the given description. In one way or the other, this drills down to performing simple inference steps over existing knowledge until enough knowledge has been generated to solve the task.

Thus, agents need support for \textit{reasoning about semantic web service descriptions} in order to effectively utilize semantic annotations.

The big picture. Together, WSMO and WSML as well as the semantic tool suite developed in the context of deliverable D5 constitute the necessary se-
Inferencing Support for SWS — Proof Obligations

Semantic infrastructure that is needed for making semantic web services come true. Such an infrastructure is a basic building block for semantically discovering services and dynamic composition of services. Thus, it is a major necessary prerequisite for enabling the exploitation of web service technology’s full potential in the context of Enterprise Application Integration and Electronic Commerce\(^2\) [Pen03].

1.2 Goal of this document

In this document we present and discuss how semantic descriptions of web services based on WSMO and WSML can be exploited for various tasks in the design and dynamic composition of web-service-based software-systems.

We identify the different phases (resp. domains) within the (dynamic) construction of a software system based on web services, where semantic information can be beneficially used to support a client in solving a problem at hand with a computerized agent. For the various domains, we describe what kind of reasoning task we have to cope with and what form of reasoning has to be carried out.

In particular, for all these domains we investigate and analyze what kind of statements need to be formally checked (resp. proven) in order to support the various tasks occurring during this dynamic system-construction process mechanically and allow these processes to access the semantic information captured in the description. These formal statements are called proof obligations.

1.3 Overview of the paper

Section 2 identifies the various areas in the process of dynamically building web-service-based applications where information about the semantics of a service as well as knowledge about its properties can be fruitfully applied.

In section 3 we define and describe the different reasoning tasks that have to be supported by a suite of inferencing tools in the context of WSMO and WSML. In each case we extract the corresponding statements that have to be proven formally and give a precise definition which is done in section 4.

Finally, in section 5 we summarize our current achievements and sketch future work and ideas.

\(^2\)One imagines for instance the potential benefit of dynamic configuration of supply chains for manufacturing companies under consideration of the current market state.
2 Semantic support in the context of WSMO and WSML

This section identifies the various areas in the process of dynamically building web-service-based applications where information about the semantics of a service as well as knowledge about his properties can be fruitfully applied.

2.1 Motivation

Having the overall vision of the Web Service Modeling Framework \[\text{[FB02]}\] in mind, let’s briefly look at a possible scenario of using semantic web services in order to solve a problem.

An example scenario. to be written!

Our scenario reveals the major application areas of semantic description of web services and shows what elements of these semantic descriptions can be exploited in what manner. Precisely here is where reasoning comes into play: Each time when we want to access and exploit the information contained in the semantic annotation of web services, we’ll most likely need to perform some form of reasoning.

Therefore, by abstracting from the specific details of the given scenario and analyzing the resulting generic model we are be able to identify the major proof obligations that we have deal with.

An abstract view on this scenario. to be written!

2.2 Where can semantic annotation be exploited?

According to this abstraction, we have to consider the following areas where elements of semantic descriptions of web services can be exploited. In each of these areas, we’ll then identify possible concrete reasoning tasks.

- Service Discovery
- Service Composition
- Service Invocation
- Service Mediation on the data, protocol and process level
- Service Execution Monitoring
- Service Compensation

Remark (Completeness of the list). Please note, that we don’t consider this list to be complete in any sense. During the course of this project, we are going to extend and possibly refine it as far as this is relevant for the use cases that we aim to address.

Furthermore, in future versions of this document we will only consider those areas wherein we are able to identify substantial or important concrete reasoning task.
**Current restrictions.** For the moment, we only consider WSMO-Standard \cite{KLR04}. This excludes some important aspects that are included in the WSMO extension WSMO-Full. These aspects are mainly concerned with the business or application layer of WSMO, e.g. concepts like contract, contract templates, negotiations and protocols for the pre-negotiation phase and the post-negotiation phase etc.

Currently, WSMO-Full is in a very early and rapidly evolving stage. In the final version of WSMO-Full it is our goal to cover all the related problems in detail. But at a first attempt, we focus here on specific reasoning tasks within WSMO-Standard, for instance goal-driven service discovery.

Moreover, in this version of the document, we focus on one specific aspect of the whole process: the service discovery task; and in particular on one specific aspect therein: the goal-capability-matching which lies in the very heart of the service discovery and is the most fundamental element of service discovery where automation is highly desired.
3 What do we want to prove?

Now we want to take a closer look into each of the application areas that have been revealed by our abstract model and for which semantic annotation has to be processed and exploited. For each of these domains, we want to clarify how we can actually apply the knowledge contained in the description of semantic web services within this context. Consequently, we derive the corresponding proof obligations in an informal or semi-formal way. A rigorous definition of these proof obligations can be found in section 4.

3.1 What is a proof obligation?

Given a set of semantic web services, our desire is to check and become sure about particular „real-world“ properties of elements of this set, of compositions of such elements as well as the relationship between such elements. As mentioned in section 1 we principally aim at automated or semi-automated construction of dynamic software-systems with well-understood properties, whose architectural paradigm is exclusively based on web-services.

The question whether a given composition of web service fulfills a requested task or whether two web services can interact with each other (and thus can actually be connected architecturally within a system) are examples for such real-world properties of interest.

As a first step, we have to come up with a precise mathematically-based definition of the properties of interest.

Afterwards, we principally have to translate this definition in statement of some formal language, that allows us to check the statement by using algorithms. The translation has to preserve the semantics of the original definition or can be strengthened. Strengthening of the definition can be necessary because of the available information about the single elements or computational complexity resp. efficiency of algorithmic checks, but should be kept to a minimum level. in a formal language is called proof obligation. We will learn about various examples for proof obligations in section 4.

3.2 Service Discovery

According to the above mentioned scenario, we have in principle two parties between which a matching has to be established: A requester who is looking for a concrete service that solves a specific problem and a set of services that offer specific functionality to their clients. Both parties are basically interested in interacting with each other. Moreover, both parties have certain assumptions resp. expectations on the cooperating party and its behaviour during the cooperation. These non-functional constraints have to taken into account within the matchmaking process as well.

Then, Service Discovery is the process of identifying possible candidates of services that can solve the requester’s problems. The identification of valid candidate services will at least take into account the functionality that is offered

More precisely, this means finding some criterion \( P \) with the property that if this criterion is valid then the original defined criterion \( C \) is valid. Please note, that it’s not required that the set of models is of \( P \) is subset of the models of \( C \). Both criterion might also use different formal languages
by a service. Furthermore, an additional selection function might be applied in a filtering sub-process. This selection function for instance might be based on non-functional characteristics of services (like response times, level of trust or reliability and user preferences).

Remark (Service Discovery vs. Requester-Provider-Matchmaking). Please note, that when we are considering a business environment the service discovery process is only one single element of the overall matchmaking process between service requester and service provider, since whether a service requester can – or is willing to – actually use one of the discovered services in order to solve his problem depends also additional information besides the functionality offered by a service which can’t be advertised in general beforehand but instead depends on the specific role of the requester within the interaction and the current state of the world when the service requester is looking for a service, e.g. financial aspects or a certain degree of trust between the two interacting parties. In particular, in a business setting this involves negotiation between the service requester and the service providers. Requester-Provider-Matchmaking is an important process that we have to address at a later stage when we consider WSMO-Full.

Thus, service discovery is basically based on two ingredients: A precise description of the needs of a client that requests some abstract service (whereby this service must not necessarily exist in a materialized form in the real-world) as well as precise descriptions of what the available and materialized services can provide.

Let $\mathcal{AS}$ denote the set of all semantic web services (resp. e-services)$^4$. 

The most fundamental aspect of service discovery then is the exclusive consideration of the basic functionality that is expected by requester and provided by a service (thus ignoring all non-functional characteristics), that is we only exploit the capabilities of services and the goal of the requester. We call this subtask Goal-Capability-Matching and explore it in detail in the following section 3.2.1.

Since Goal-Capability-Matching only uses information about service functionality, in general it only is able to identify a superset $\mathcal{CS}$ of all matching services that contains candidates for matching services. The actual set of services that match the client’s request $\mathcal{MS} \subseteq \mathcal{CS}$ in general depends on additional properties (like non-functional aspects of the service itself or the specific communication pattern of the service that is described its choreography) and hence usually is only a subset of these candidates.

If the requester doesn’t additional requirements besides the functional ones at all, then both sets coincide.

Otherwise, Goal-Capability-Matching has to be supplemented with a filter function$^5$ that takes into account the additional information on the client’s requirements. For different non-functional aspects $A_i$ of semantic web services (like costs, trust-level, availability, service-provider-related information etc.) we assume different filter functions $\text{sel}_{A_i}: \mathcal{P}(\mathcal{AS}) \rightarrow \mathcal{P}(\mathcal{AS})$.

---

$^4$From an system architecture point of view that means the set of all semantic web services that are registered at some sort of (central or distributed) registry. Only the registry allows the discovery component to take the semantic annotation of some service into account during the discovery process.

$^5$A filter function for web services is a function $f: \mathcal{P}(\mathcal{AS}) \rightarrow \mathcal{P}(\mathcal{AS})$ with the particular property $f(S) \subseteq S$ for all $S \subseteq \mathcal{AS}$. 

---
In fact, given some goal $G$, Goal-Capability-Matching itself induces a filter $\text{sel}_{\text{gen}}(G) : \mathcal{P}(\mathcal{AS}) \rightarrow \mathcal{P}(\mathcal{AS})$ on the set of all web services and thus can be seen as one of these filter functions.

We will consider such additional filters in future versions of this document if they are based on logical descriptions and reasoning. Other kinds of filters and their impact on a system architecture of a discovery component for semantic web services will be presented and discussed in more depth in deliverable D5.2 of WSML [Kel04] in the course of the WSMO project.

The single filter function (including Goal-Capability-Matching) can be combined to the overall filter function $\text{discovery}$ that represents the major part the discovery process in a formal way:

Let $\text{ServiceRequests}$ be the set of all possible service requests and $R \in \text{ServiceRequests}$ some service request. A service request itself at least includes the client’s goal $\text{goal}(R)$, as well as supplementary non-functional aspects $\text{aspect}_{A_i}(R)$ ($i = 1, \ldots, h$). Then the discovery of services that actually solve the problem specified by the user can be formally represented as a composition of functions as follows:

$$\text{discovery} : \text{ServiceRequests} \rightarrow \mathcal{P}(\mathcal{AS})$$

$$\text{discovery}(R) = ( \text{sel}_{\text{gen}}(\text{goal}(R)) \circ \text{sel}_{A_1}(\text{aspect}_{A_1}(R)) \circ \ldots \circ \text{sel}_{A_h}(\text{aspect}_{A_h}(R))) (\mathcal{AS})$$

**Applying Service Discovery.** For the full automation of service composition this identification of a set of services that solve the given problem is not enough: when dynamically composing the services (at runtime) we have to select one of these services concretely. Conceptually, this corresponds to applying some selection function on the set of discovered services. This selection function in general will be based on user preferences $P$ like minimum-costs, maximal-throughput, minimum-response-time, service-provider characteristics and so forth. We denote such a preference-based selection function by $\text{select}(P)$.

The preferences could be explicitly specified by the requester or implicitly given by tools that are used by the requester which for example could have learned user preferences during the clients behavior over a certain period of time. When no user preferences are available at all, we can use the random choice among the set of discovered services as the default selection function.

Then, the discovery process has to be extended by a preferences-based selection process such that we end up with a specifically selected service that solves the requester’s problem. Formally, we construct a function $\text{find}(R, P)$ that takes a user request $R$ and the preferences $P$ of a user and returns a particular web service that satisfies the request as well as the user preferences or ⊥ if no

---

6Please note, that a request $R$ is considered at this point as an abstract entity representing all the information that might be relevant for some client when specifying his requirements on service to be searched for. Since at present we do not know precisely what aspects we are taking into account here and since relevant information even might vary from the context in which discovery has to be performed, we decided to model $R$ instead of a tuple as an abstract entity to which a partial „projection” function can be applied in order to get the details about single aspects related to some request. This „projection” function is denoted by $\text{aspect}_{A_i}(R)$ ($i = 1, \ldots, h$). The concrete nature of the single aspects $A_i$ are not relevant for our discussion here and thus do not need to be defined here.

7A selection function on a set of web services is a function $f : \mathcal{P}(\mathcal{AS}) \rightarrow \mathcal{AS}$ with the particular property $f(S) \in S$ for all $S \subseteq \mathcal{AS}$.
such web service can be found:

\[
\text{find: } \text{ServiceRequests} \times \text{Preferences} \rightarrow ( \mathcal{AS} \cup \{\bot\}) \\
\text{find}(R, P) = \left\{ \begin{array}{ll}
\text{select}(P)(\text{discovery}(R)) & \text{if } \text{discovery}(R) \neq \emptyset \\
\bot & \text{otherwise}
\end{array} \right.
\]

In the simplest case, we assume that there is a single existing service that fits our needs. Then, we are looking for exact matches between the description of the requested service (the goal) and the specification of what is offered by existing and known web services (the capabilities). Thus, we have to support goal-capability-matching for a semantic-based service discovery.

However, in many cases, we will not be able to find a single service that exactly provides what we need. In order to resolve this deficiency, in principle there are two natural approaches (which themselves are not be considered directly as service discovery tasks):

- **Server-sided resolution: service composition** – Given the specification of the requesters needs, we try to combine several existing web services at design-time (represented by means of proxies) or on the fly at runtime in order to create a new web service that is adequate with respect to the request. On particular important case is when one of the combined services is a mediator service that overcomes differences in data, protocol or process representation.

- **Client-sided resolution: compute and describe mismatch** – Given the specification of the requesters needs and the description of a possible (e.g. almost matching) web service, we compute the existing mismatch and return it to the client. Afterwards it’s the client’s responsibility to use this information in a proper way to close the gap.

Clearly, the technique underlying the latter approach can also be used order to support the first approach: When we precisely know about the mismatch between what we need and what we have, it’s much easier to find the missing piece in the puzzle.

Therefore, we can consider the computation of a goal-capability-mismatch as a quite useful and relevant task that we would love to be supported – at least to some extent – by our framework. In principle, expect this problem to be an undecidable problem in most cases, but there might be interesting cases too, that be dealt with algorithmically. There might even be cases, when the description can be used in a constructive sense: for the automated generation of a mediator that bridges the gap.

In fact, this problem is a generalization of the the capability-goal-matching problem: If we can compute the mismatch between a goal and a capability then we can also solve the matching problem by testing whether there is no mismatch at all. Thus, it clearly is a computationally more complex task in general.

**Remark (On the the relation between Service Discovery & Service Composition).** We defined Service Discovery to be the process of identifying the set of all services which satisfy the client’s request. Using the corresponding discovery function, we are able to construct a function find which delivers a specific service to be used for satisfying the needs of the requester, if such a service exits.
What happens if such a service doesn’t exist in materialized form, that is as a single available service?

In approach sketched above, we only detect that the needs of the client can’t be satisfied by any existing (resp. known) service. On the other hand, in such a situation it might be possible to construct the needed (and currently non-existing) service from a set of existing services. That is we try to compose the needed service on-the-fly from existing ones. The process of combining available (and materialized) services into a beforehand non-existing (composed) service which satisfies the specified requirements is called service composition.

If we succeed we are able to return the composed (but temporary) service. We could also materialize the new service permanently by storing the description of its composition – in WSMO terms the so-called orchestration – as well as the resulting properties of the composed service and hence allow reuse of the composed service.

In case that we fail to find a composition, we signal to the requester that his needs can’t be satisfied by our infrastructure at the moment.

From this perspective, service composition can be exploited as a specific subtask during service discovery – a subtask that is needed to be performed if no existing service satisfies the client’s request. On the other hand, it is clear, that during the composition of services, that is when automatically or semiautomatically constructing the orchestration of a required but non-existing service, service discovery is one of the major tasks that has to be performed by humans or agents (resp. some composition engine).

To sum up, both processes, the discovery as well as the composition of services can fruitfully enrich each other in an application context.

We want to add another remarks here: the constraints specified by the request $R$ for service discovery may be more precise and ,,complete” when performed in the context of service composition as in the case of a pure request by a human used who searches and ,,browses” a service repository. The point we want to make here is that we consider service discovery in this document as a context-dependent process, that means the provided input as well as the expected output is expected to be of different quality in various contexts.

### 3.2.1 Goal-Capability-Matching

In this section we focus the discovery subtask Goal-Capability-Matching in detail on a semi-formal level. In particular, we describe goals, capabilities precisely, and define the notion of matching between these two concepts. The formal proofobligation is derived later on in section 4.1.1.

**Goals.** In WSMO-Standard [KLR04] the means for a service requester to express his desires are so-called goals. There, the definition of the term goal basically coincides with the one given in [FB02]:

*A goal specifies the objectives that a client may have when he consults a web service.*

By invoking a service, the requester wants to achieve something: either he wants to acquire additional knowledge (by analyzing the result resp. output of the invoked service) or the state of the real-world should be changed in some way. Indeed, both reasons can be considered as some kind of state change: The former can be consider as a state change in the information space whereas the latter can be considered as a state change in the real-world.
In contrast to [FB02], where a goal formally is defined by two logical expressions describing requirements on the pre-state and the post-state, in WSMO a goal only refers to the state that should be reached after invoking a service. In that state, the desires of the requester have to be satisfied.

To sum up, we consider a goal here as a description of possible states of the world (as well as the states of the information space) that a client wants to achieve (after invocation and execution of some service). Of course, the basic assumption here is that the requester wants to use a service in order to reach his goal.

Therefore in WSMO a goal $G$ is represented by a \textit{postcondition} (for the state of the information space) and an \textit{effect} (for the state of the real-world). Formally, postconditions and effects are represented by formulas in F-Logic [KILW95] (over some signature $\Sigma$): $\Phi^{\text{post}}$ and $\Phi^{\text{eff}}$.

In other words, by using some service the user wants to reach a state (in the real-world) where both the specified postcondition as well as the specified effects are satisfied. All such states are considered as valid with respect to the requester's goal, that is the goal is considered as being achieved, when the service execution results in such a particular state; the goal itself can be identified with the set of states that satisfy the requirements stated by postcondition and effects.

The basic vocabulary for the definition of these constraints on the state that has to be reached after the service execution originates from a set of domain ontologies $O_G = \{O_1, \ldots, O_n\}$, e.g. $\Sigma_{O_G} \subseteq \Sigma$. Since each ontology $O_i$ formally describes the interrelationship between the terms used for expressing the goal, they also have to be considered in this reasoning task. Without loss of generality, we consider an ontology $O_i$ as a set of F-Logic formulas (over signature $\Sigma_{O_G}$).

In general, the ontologies that are used for describing the goal can’t be integrated by simply joining them into one ontology: Usually, there will be conflicts, e.g. due to slight differences in the interpretation of terms, or some concepts that have actually a similar semantics in two or more ontologies are considered to be different because of different names. These problems with integration the imported ontologies are treated by one or more \textit{ontology mediators} $M_j$. We denote the set of used ontology mediators by $M_G$ and assume again that the mediation functionality provides by these mediators $M_j$ is represented by a set of formulas.

A goal then can be represented as a tuple of (sets of) F-Logic formulas (over some signature $\Sigma$):

$$G = (\Phi^{\text{post}}, \Phi^{\text{eff}}, O_G, M_G)$$

\textit{Capabilities.} The main concept in WSMO-Standard for describing what a service actually provides as its functionality is the so-called \textit{capability}. A capability basically consists of description what the service expects from its input (precondition\footnote{Could also be referred to as \textit{input-related preconditions}.}) and the state of the world (assumption\footnote{Could also be referred to as \textit{state-related preconditions}.}) at invocation time in order to provide its service properly, as well as the description of the output (postcondition\footnote{Could also be referred to as \textit{output-related postconditions}.}) it provides and the state of the real-world that the execution of the
service results in \( \text{effects}^{11} \)^{12}.}

Formally, each of these descriptions represents a constraint of a state (of the information space or the real-world) and is thus expressed by a F-Logic formula: \( \Psi_{\text{pre}}, \Psi_{\text{ass}}, \Psi_{\text{post}}, \Psi_{\text{eff}} \). As in the case of goals, for any capability \( C \) these formulas are expressed with respect to a set of ontologies \( O_C \) that have to be integrated by using some ontology mediators \( M_C \). Again, we consider both elements to be represented by a set (resp. sets) of F-Logic formulas.

Hence, a capability can be considered as a tuple of (sets of) F-Logic formulas too:

\[ C = (\Psi_{\text{pre}}, \Psi_{\text{ass}}, \Psi_{\text{post}}, \Psi_{\text{eff}}, O_C, M_C) \]

**Remark (Capabilities and actual Service Functionality).** Let us briefly discuss the relationship between the terms ,,capability” and ,,service”.

In WSMO each semantic web service specifies its capability. This means that the service provider announces that this web service provides a certain type of functionality. The actual functionality provided by the the web service indeed is defined by the concrete implementation of the service.

Semantically speaking, a service implementation defines a specific relation between our abstract state-space.\(^{13}\)

On the other hand, from a semantic point of view, a capability can also be seen as a characterization of a set of services with similar behaviour; it constrains the ,,allowed” state-changes when executing these services. In this sense capabilities and actual service functionalities are principally independent of each other.

More formally, we can give a formula in a Dynamic F-Logic\(^{14}\) which describes whether a service ,,respects” some capability: Given a concrete implemented service \( s \), that means a fixed relation \( I(s) \) over the state space. A service \( s \) provides a capability \( C \) if the following formula (in a Dynamic F-Logic) is valid (holds in all states wrt. this fixed interpretation of \( s \)):

\[
(\Psi_{\text{pre}}^{C} \land \Psi_{\text{ass}}^{C}) \rightarrow (\langle s \rangle (\Psi_{\text{post}}^{C} \land \Psi_{\text{eff}}^{C}) \land [\mathfrak{s}] (\Psi_{\text{post}}^{C} \land \Psi_{\text{eff}}^{C}))
\]

(1)

The formula can be read as follows: If the precondition and the assumption of the capability are met then every terminating execution\(^{15}\) of service \( s \) stops in a state that that satisfies the postcondition and the effects specified by the capability \( C \) \( \ldots [\mathfrak{s}] (\Psi_{\text{post}}^{C} \land \Psi_{\text{eff}}^{C}) \ldots \) and such an execution exists \( \langle s \rangle (\Psi_{\text{post}}^{C} \land \Psi_{\text{eff}}^{C}) \).

Please note, that in WSMF and WSMO capability descriptions are seperate from a specific services. A capability can semantically characterize more than one service in principle. Thus, by only considering the capability of a service, we do not know any details about a concrete service implementation providing this capability – more important, we even don’t care about that!

\(^{11}\)Could also be refered to as state-related postconditions.

\(^{12}\)Please note, that the distinction for pre-state-constraints and for post-state-constraints is not common in formal software specification. There preconditions and assumptions are usually combined in the notion of precondition and the term postcondition denotes both postconditions and effects at the same time.

\(^{13}\)The set of all states. Each state in this space represents knowledge about the real-world as well as the information space.

\(^{14}\)This logic can be seen as a logic that combines two orthogonal aspects in a single logical framework: the modelling primitives of F-Logic [KLW95] for states and a propositional Dynamic Logic [HKT00] for state-transitions. The details are not relevant here, since throughout this section, we use this formal language as a notation with precise semantics.

\(^{15}\)In general, we consider \( s \) to be nondeterministic.
In WSMO, per convention every service provider instead has to specify the capability of each (registered) service and there is exactly one such capability specification for each available service.

By attaching a capability description to a service, the service provider implicitly guarantees that the service implementation actually is correct in the sense that the above mentioned formal criterion (1) is fulfilled. This is our working assumption throughout this document and the WSMO project.

What does matching now precisely mean? Informally, we consider a service (described by its capability) to match a goal if the execution of the service (according to the capability) can solve a client’s problem (which is exclusively described by the goal), that is by execution of the service we can end up in a state where the goal is fulfilled.

Since we do not know anything about the concrete service (implementation), we do not know any particular details about (possible) executions of the service besides the pre-state-constraints and the post-state-constraints mentioned in the capability. Thus, we have to abstract from the actual service executions (in particular the set of halting states of a service) and consider the postcondition of the service capability as the main information about the execution. Then, a capability \( C \) matches a goal \( G \) if the capability post-state-constraints (postcondition and effects) imply the post-state-constraints of the goal.

In the model of WSMO-Standard, by using a goal the requester only specifies what he expects to achieve and not what he’s willing or able to provide, whereas whether „there is an execution of the service“ by which the needed poststate is reached depends on the state of the world as well as the client’s input when the service is being invoked. In that sense, our notion of matching between a goal and a capability is a relative one: The match can only be considered as valid (or meaningful) when the requester is able ensure the properties of the state of the world described by the preconditions and the assumptions in the capability when invoking the service. But as soon as this is guaranteed, the service can provide it’s functionality and thus there is such an execution. In that sense, the matching — though being relative to the prestate requirements — is actually established by the poststate requirements only.

Remark (Formal description of a goal-driven discovery request). Above, we have not given a formal semantics of the client’s discovery request. We want to do this here: Given a goal \( G \) by the user, the corresponding (goal-driven) discovery request \( R_G \) is:

\[
(\Psi_{G}^{pre} \land \Psi_{G}^{ass}) \to ((s) (\Phi_{G}^{post} \land \Phi_{G}^{eff}) \land [s] (\Phi_{G}^{post} \land \Phi_{G}^{eff}))
\]

All known semantic web services \( s_i \) (in a registry) are described by means of their capabilities \( C_i \) which semantically can be seen as the following valid formula:

\[
(\Psi_{C_i}^{pre} \land \Psi_{C_i}^{ass}) \to ((s_i) (\Phi_{C_i}^{post} \land \Phi_{C_i}^{eff}) \land [s_i] (\Phi_{C_i}^{post} \land \Phi_{C_i}^{eff}))
\]

To formally proof that (1) is fulfilled by some service (implementation) \( s \) it would be necessary to formally verify the service (implementation) wrt. the corresponding capability \( C \). This requires at least that source code of the service implementation is known and accessible. Service verification is at present not in the scope of the WSMO project.

And in general, an approximation of the possible service executions.
Please note, that this representation of the goal-capability-matching obviously suggest a natural criterion on the specification elements (goal and capabilities), which roughly (i.e. omitting pre-state-constraints for simplicity) is as follows:

\[(\Psi_{\text{post}}^{C_i} \land \Psi_{\text{eff}}^{C_i}) \rightarrow (\Phi_{\text{post}}^{G} \land \Phi_{\text{eff}}^{G})\]  

(2)

This is the criterion that we finally used in our informal definition of goal-capability-matching above. We will discuss later on whether this criterion is the only one, and if not in which relation this criterion stands to alternative ones (e.g. if it’s stronger or weaker etc.)

Alternative Definitions for the Notion of Matching. The definition of matching (2) between a capability and a service which has been discussed above is well known from Component-based Software Engineering research [ZW97] (and later adapted in the context of other papers on service discovery for web services [PKPS02, LH]) and commonly referred to as a Subsumes-Match:

Under the assumption that the service requester exactly describes his goal, the service certainly can be used to resolve the client’s goal but the service itself can only cover parts of the possible solutions: There are some acceptable states (in which the goal is achieved) but these states are not reachable by this service. As a consequence, a system should look for other (matching) services such that all the services together ideally cover the admissible goal states completely.

The above mentioned literature proposes additionally alternative definitions for matching with different properties:

1. **Exact-Match.** Like the Subsumes-Match but the considered service completely covers the goal. Formally that means:

\[ (\Psi_{\text{post}}^{C_i} \land \Psi_{\text{eff}}^{C_i}) \leftrightarrow (\Phi_{\text{post}}^{G} \land \Phi_{\text{eff}}^{G}) \]

(3)

2. **Plugin-Match.** Here one must show that the set of admissible states for resolving the goal is a subset of the states which can be reached by the service. Formally, that means:

\[ (\Psi_{\text{post}}^{C_i} \land \Psi_{\text{eff}}^{C_i}) \leftarrow (\Phi_{\text{post}}^{G} \land \Phi_{\text{eff}}^{G}) \]

(4)

In fact, unlike the Exact and Subsumes-Matching notions, this criterion does not guarantee that each execution of the service resolves the goal. It is only possible to execute the service such that the problem of the requester is solved. But there might be execution which end up in a state that is not admissible wrt. the goal. On the other hand, in principle we assume that it is possible to cover the goal completely by executions of this single service.

3. **Intersection-Match.** There is a common state between the set of admissible goal states and the set of post-states described by the capability. In other words,

\[ (\Psi_{\text{post}}^{C_i} \land \Psi_{\text{eff}}^{C_i}) \land (\Phi_{\text{post}}^{G} \land \Phi_{\text{eff}}^{G}) \]

(5)

---

18Whereas in this literature
19Please note, that this statement has been considerably simplified for the sake of comprehensibility. Later we will discuss this issue in depth and clarify that this notion of matching actually is a lot less. In fact, we argue that we can not conclude that this service can be used to achieve the goal in general, because the implicit assumption we used for simplification of the discussion above that every state satisfying the post-state-constraints of the capability is indeed reachable by execution of the service, is unrealistic!
is satisfiable.

In comparison to the Plugin-Match, the goal is only partially covered and there is are executions of the service which do not resolve the requester’s goal. Hence, this notion is weaker than the Plugin-Match.

4. Disjoint / Mismatch. There is no common state between the set of admissible goal states and the set of post-states described by the capability. In other words, the conjunction of the goal post-state-constraints and the capability post-state-constraints is unsatisfiable.

Clearly, we can not use the service to achieve the client’s goal at all.

The single matching-notions are usually totally-ordered (wrt. application domain considerations) such that this ordering can be used for ranking services that match a given goal (according to the above mentioned notions). Here we would propose the following total order on the above mentioned criterions:

$$Exact > Subsumes > PlugIn > Intersection$$

where $>$ can be interpreted as ”preferred over”.

We are going to discuss the single variants of matching in a future version of this document in more detail and clarify the precise interpretation and consequences of these notions.

3.2.2 Exploiting Mediators for Web Service Discovery

In the previous section, we have only taken into account the description of the capabilities of available services and the description of the requester’s goal. Only these description elements are exploited in order to locate services that fulfill the requirements expressed by the requester.

In this section, we discuss how to exploit the mediators defined in WSMO to provide an alternative approach to Web Service discovery. The mediators that are to be considered in this approach are ggMediators and wgMediators. We will first discuss the use of wgMediators and then we will analyze the role of ggMediators in the discovery process.

**Use of wgMediators** WSMO-Standard version 0.2 defines wgMediators as elements that link Web Services to goals, explicitly stating the difference (called reduction) between the two components and mapping different vocabularies. The most interesting point of this definition is that wgMediators provide an explicit link between Web Service capabilities and requester goals. The mediation between different vocabularies is not of interest for our discussion.

The approach presented in the previous section does not consider these explicit links between requester needs and Web Service offers. Clearly, these explicit links can be taken into account when locating suitable Web Services, as they describe a well-defined relationship between the request and the offer. In order to exploit these links, we will make two assumptions:

- **Use of pre-defined goals:** Explicit links can only be exploited if they are provided at design-time i.e. if they are available for the discovery process. In order to define such links at design-time, the goals and capabilities involved also have to be defined before-hand. In the case of capabilities it is clear that the service will describe its capability to be advertised. However, it can be argued that goals can be defined by the user right before submitting the query for suitable services, and therefore explicit
links would not be available. Therefore, our assumption is that some pre-defined goals are described at design-time and links are established between them and the capabilities of the published services.

- **Definition of reductions by the requester:** The pre-defined goals will be customized (if necessary) by the requester to describe his needs. This can be done by using ggMediators i.e. the user expresses its request by refining existing goals, and this refinement is expressed as the reduction in a ggMediator. Therefore, we assume that the requester will use pre-defined goals (with well-established links to the available services) and refine them using ggMediators.

We believe that these assumptions can be met in a real application. In fact, defining libraries of pre-defined goals will support the user in defining his needs for the following reasons:

- **Ease-of-use:** The requester can inspect available goals and, helped by non-functional properties of the goal such as the textual description, will locate the goals related to her request. This process is, specially for non-experts in logics, much easier than defining postconditions and effects from the scratch.

- **Reduction of effort:** The requester only has to specify postconditions and effects to refine existing goals. This effort is obviously lower than defining a complete new goal.

- **Reduction of errors:** The use of pre-defined and appropriately tested goals improves accuracy when defining the requester needs. Goals define from the scratch by the requester are more likely to be wrongly expressed.

For these reasons, we believe that the use of pre-defined goals is not only a reasonable assumption but also a feature that might be required by users.

Once pre-defined goals, wgMediators, and the reductions to the selected predefined goal are performed by the user, we can use this information for locating suitable services. In contrast to the scenario in section 3.2.1, now the search space is not the entire set of available services, but only those ones linked via a wgMediator to the pre-defined goal selected (and possibly refined) by the requester. In this way, we can considerably cut-off the search space in the discovery process.

**Use of ggMediators** So far, we have ignored ggMediators, that enable the definition of goals by reusing and refining existing ones. Nevertheless, this is an important point that deserves additional discussion in order to ground our alternative approach. As a goal can be described as successive refinements of other goals, it can happen that services linked to the goal(s) used to describe the selected goal can fulfill the user request. For our discussion, we will assume that the ggMediators define a non-directed acyclic graph, being the nodes of the graph the different goals defined, and the arcs the reductions captured in the ggMediators. We do not consider cycles as these would be an error when modelling goals. We consider the graph as non-directed, as the reduction expressed in the ggMediator is direction-independent.

WSMO-Standard version 0.2 [KLR04] defines the links established by wgMediators and ggMediators as capturing the difference between the linked components. However, this difference can reflect a reduction of functionality or an augmentation of functionality. The existence of both possibilities implies that the discovery process has to follow every path linked to the selected goal.
If we impose that the reductions expressed in ggMediators and wgMediators always capture a functionality reduction, then we have a tree instead of a graph. In this case, only the services linked to the selected goal and its parent goals (more general functionalities) have to be considered. Nevertheless, this does not completely fit to the definition of reduction given in WSMO-Standard version 0.2. Therefore, we will not make this assumption in order to be consistent with the definitions given in the specification of WSMO.

In this context, the search space is the one depicted in Figure 1. The box on the left represents the goals to be taken into account i.e. the ones linked to the selected goal via a ggMediator. The box on the right represents all the services linked to these goals via a wgMediator, that constitute the search space.

When comparing the use of mediators for Web Service discovery to the discover process without mediators, positive and negative aspects can be found for the former approach:

⊕ Reduced search space
⊕ Supports the user in defining her goals
⊖ The approach is less general

But, both approaches are compatible i.e. mediators can be exploited if a pre-defined goal is selected and refine, and the discovery in absence of mediators can be applied otherwise.

4 Concrete Proof Obligations

In section 3 we described the concrete reasoning tasks that we want to support on a precise but still semi-formal level. Now, this section is concerned with the precise formal definition of the corresponding proof obligations.

4.1 Proof obligations in the context of Service Discovery

In the following sections we give precise definitions of the proof obligations that belong to the single reasoning tasks identified for web service discovery in section 3.2.
4.1.1 Capability matches Goal

Let \( G \) be a goal and \( C \) be a capability. We start to develop a minimal definition for matching first and modify this notion afterwards as already outlined in section 3.2.

According to our definition above, \( C \) matches \( G \) if and only if for any service \( s \) that offers capability \( C \) there is an execution of the service\(^{20} \), which stops in a state that satisfies the requirements stated in the goal. According to semantics of a capability, the resulting state – the post-state of the execution of service \( s \) – satisfies at least the requirements stated as the postconditions and the effects of the capability.

Thus, given a specific service \( s \) that provides a capability \( C \), it’s obvious that the set of states which are reachable by executing \( s \) from any state satisfying the precondition of \( C \)\(^{21} \) is indeed a subset of the set of all states that satisfy the postcondition of \( C \). This situation is illustrated in figure 2.

If for the given service \( s \) the set of halting states has a common element with the set of states where the goal is satisfied, then the service can be used by the requester in order to achieve his goal. In other words, we are interested to check whether the intersection of the set of possible halting states \( S_{\text{halt},s} \) of \( s \) and the set of states satisfying the goal \( S_{\text{goal}} \) have a non-empty intersection:

\[
S_{\text{halt},s} \cap S_{\text{goal}} \neq \emptyset \tag{6}
\]

Here, the set \( S_{\text{halt},s} \) can be described by the following formula

\[
S_{\text{halt},s} = \{ s'' \in \text{States} \mid s' \text{ satisfies } \Psi_C^{\text{pre}} \land \Psi_C^{\text{ass}} \text{ and } \exists s'' : (s', s'') \in I(s) \}
\]

\(^{20}\)Please note, that the actual execution – and thus the state reached by this specific execution – depends on the concrete state of the world (in particular the input) where the service is being invoked!

\(^{21}\)The set of halting states of \( s \)
where States denote the set of all states and \( I(s) \subseteq States \times States \) the actual relation that represents the implementation of service \( s \).

According to the commonly used semantics of post-state-constraints (post-conditions and effects) by attach capability \( C \) to a service \( s \) it holds that

\[
S_{\text{halt}_s} \subseteq S_{\text{post}_C} := \{ s' \in States \mid s' \text{satisfies } \Psi^\text{post}_C \land \Psi^\text{eff}_C \}
\]  

(7)

where \( S_{\text{post}_C} \) denotes the set of states satisfying the post-state-conditions of a capability \( C \).

But at the level of capabilities, we don’t have any information about the specific service \( s \) that offers capability \( C \), hence, we don’t know anything about the relation \( I(s) \) and thus the possible sets of halting states of the services \( s \). The only information related to the set of halting states that can be read off the capability description itself is the postcondition. Thus, this is the only available information for developing the proof obligation for the goal-capability-matching task.

If we can show, that the set \( S_{\text{post}_C} \) of states satisfying the post-state-conditions of a capability \( C \) is a subset of the set \( S_{\text{goal}} \) of states that fulfill the goal, then this also holds for all services \( s \) (resp. service implementations \( I(s) \)) providing \( C \), and the matching is established.

Formally, if the precondition and assumptions of \( C \) are satisfiable and the goal \( G \) is satisfiable then:

\[
S_{\text{post}_C} \subseteq S_{\text{goal}} \Rightarrow (S_{\text{halt}_s} \cap S_{\text{goal}} \neq \emptyset \text{ for all services } s \text{ with capability } C)
\]

Clearly, this is a stronger criterion than the actual required one, e.g. there are some cases where a service (implementation) \( I(s) \) actually matches a goal, but this criterion is not satisfied. In particular this might happen, if the capability \( C \) represents a very vague statement (and thus the set \( S_{\text{post}_C} \) is quite big), but the set of halting states can be described in a much more precise way.

But it’s the weakest criterion that can be established with the given information on services by a capability, that means every weaker criterion would need specific information about the services \( s \) (resp. their implementation \( I(s) \)) that can provide the capability \( C \), more precisely, we would need more detailed information about the corresponding set of halting states \( S_{\text{halt}_s} \).

On the other hand, if the service has been described carefully and precisely then it is in principle possible that the the post-state-constraints of a service capability exactly describes the set of halting states of the service implementation. In this case, the two intersection criterion can be applied and the strengthening of the criterion is not necessary in principle.

Thus, we’ll use this stronger criterion when defining the proof obligation for Goal-Capability-Matching: Whatever state we consider, when the state satisfies the post-state-conditions of the capability, then it satisfies the goal. Since states are represented by F-Logic-Interpretations \( I \) (over signature \( \Sigma \)) we get as our proof obligation nothing else than a logical entailment between the requirements on the poststate by the capabilities and the goal:

\[
\Psi^\text{post}_C \land \Psi^\text{eff}_C \models \Phi^\text{post} \land \Phi^\text{eff}
\]

(8)

\[22\] Please note, that in WSMO a human being attaches a capability \( C \) to some web service \( s \). Thus, the attached capability \( C \) can in principle be arbitrary unspecific (wrt. to the actual semantic of the service that is defined by the service implementation).
In order to adequately reflect the conceptual difference between postconditions and effects, we will use the following criterion that is stronger than criterion \(8\):

\[
\Psi_{\text{post}} \models \Phi_{\text{post}} \quad \text{and} \quad \Psi_{\text{eff}} \models \Phi_{\text{eff}}
\]  

(9)

In summary, we have achieved the following: If \(C\) matches \(G\) (by proving the proof obligation) and if the requester is able to ensure the requirements on the pre-state when calling any service with capability \(C\), then (according to the semantics of the capability) this service will stop in a state that fulfills the postcondition and thus (because of our stronger criterion) the requirements specified in the goals are satisfied. In other words: The requester can use a service with capability \(C\) in order to achieve his goal!

Again, we have to take imported ontologies \(O\) and corresponding ontology mediators \(M\) also into account: In the proof obligation, they have to be considered as premisses, since the axioms of the ontologies as well as the description of the mediation constrain (or define) the basic semantics of the symbols in the ontologies and the definition of the capability and the goal in turn relies on this particular semantics.

Now, we want to define the actual proof obligation for checking Goal-Capability-Matching.

From Goal-Capability-Specifications to Proof Obligations. So far, we ignored possible variables in the formulae mentioned above and implicitly assumed them to be closed formulae. In fact, this is not always true:

For a capability \(C\) the formulae \(\Psi_{\text{pre}}, \Psi_{\text{ass}}, \Psi_{\text{post}}, \Psi_{\text{eff}}\) refer to the input parameters as well as the output value of a corresponding service. These formulae especially have to refer to the same values that are used for this parameters and the considered values should be independent of an interpretation \(I\) of the used signature \(\Sigma\). For this reason, input parameters are modelled as free variables in the capability description \(C\). In spite of that, the output value is the result of a computation that depends on the input of the service. Hence, it is represented as a function symbol\(^{23}\) with the input parameters as it’s arguments.

Let \(in_1, \ldots, in_n\) be the set of input parameters used in the capability \(C\) and \(out(in_1, \ldots, in_n)\) denote the output parameter (resp. return value) that is used when stating \(C\).

Since the notion of matching is about the possibility to execute any service with some capability in a certain way and each execution depends\(^{24}\) on the input provided by the client at invocation time, the input parameters that occur in the formulae of the capability description must be considered as existentially quantified in the proof obligation.

Please note, that the principle ideas for the proof obligation \((9)\) mentioned above still remain valid in principle. We can simply adapt the formal statements from above by considering inputs and outputs mentioned in the capability \(C\) in

\(^{23}\)For the moment, we don’t consider non-determinism here. Nevertheless, it can be easily modelled by using a relation symbol instead and adapting the proof obligation below accordingly.

\(^{24}\)Even stronger, we assume that the execution of the service is completely determined by the given input values.
the proof obligation. Because of the intended semantics of the free variables in the formulae mentioned in \( C \), we have to replace the logical entailment relation by the implication junctor.

Additionally, we create as our proof obligation a \textit{closed} formula by binding free variables (besides the input parameters) to existential quantifiers, that is we generate the existential closure \( \text{Cl}_\exists(\Phi) \) of the proof obligation \( \Phi \). This way, we ensure that the semantics of the proof obligation itself has no implicit components.

Eventually, we’re only interested in input values that satisfy the requirements on the pre-state as described in the capability \( C \). Only for such input values, the service is guaranteed to reach a halting state which satisfies the constraints on the post-state as described in \( C \).

As already discusses above, we can \textit{not} include all pre-state-constraints in the proof obligation: The assumptions in general can not be checked and such a check at discovery time has no benefit as long as you can not guarantee that the world stays the same until the actual invocation of the service; thus we will not include the assumptions mentioned in \( C \) in the proof obligation. Nevertheless, the pre-condition can be checked and thus we at least include it in the proof obligation. Please note, that this does not affect the correctness of our approach severely, since we already clarified that matching is a notion relative to the satisfaction of the constraints on the pre-state at service invocation time.

Because of this, we just get more candidates for possibly matching services.

**Proof Obligation.** Given a goal

\[
G = (\Phi_{\text{post}}, \Phi_{\text{eff}}, \Omega_G, \mathcal{M}_G)
\]

and a capability

\[
C = (\Psi_{\text{pre}}, \Psi_{\text{ass}}, \Psi_{\text{post}}, \Psi_{\text{eff}}, \Omega_C, \mathcal{M}_C)
\]

the proof obligation for \textit{Goal-Capability-Matching} is to prove the following relation

\[
\text{PO}^*_{\text{gcm}}(G, C) \equiv \{ \Omega_C, \Omega_G, \mathcal{M}_C, \mathcal{M}_G \} \vdash \text{Cl}_\exists(\exists \text{in}_1, \ldots, \text{in}_n: (\Psi_{\text{pre}} \land \Psi_{\text{ass}} (\Psi_{\text{post}} \rightarrow \Phi_{\text{post}}) \land (\Psi_{\text{eff}} \rightarrow \Phi_{\text{eff}})))
\]  

(10)

By using sound and complete deduction calculus \( \mathcal{C} \) that defines the provability relation \( \vdash \), this is equivalent to show within the calculus that the set of conclusions can be derived from the set of premises:

\[
\text{PO}^*_{\text{gcm}}(G, C) \equiv \{ \Omega_C, \Omega_G, \mathcal{M}_C, \mathcal{M}_G \} \vdash \text{Cl}_\exists(\exists \text{in}_1, \ldots, \text{in}_n: (\Psi_{\text{pre}} \land \Psi_{\text{ass}} (\Psi_{\text{post}} \rightarrow \Phi_{\text{post}}) \land (\Psi_{\text{eff}} \rightarrow \Phi_{\text{eff}})))
\]  

(11)

Such a calculus for full F-Logic is discussed in detail in [KLW95]. Hence, we can deal with the proof obligation \( \text{PO}^*_{\text{gcm}}(G, C) \) (and thus with this reasoning task) in an automated way, as it is needed for our purpose. Nevertheless, we have to keep in mind some practical issues.

**Practical concerns.** F-Logic is at least as expressive as classical First-Order Logic for which logical entailment is not decidable. It’s straightforward to see
that thus logical entailment for full F-Logic is not decidable, too. But because there is a sound and complete proof system for logical entailment, this relation is recursively-enumerable (resp. semi-decidable).

This has the following consequence for practical applications: Given a capability $C$ and a goal $G$ for which we want to know whether $C$ matches $G$. We can proceed as follows: Compute the proof-obligation $PO_{gcm}(G, C)$ and start an automated theorem prover (ATP) for full F-Logic. If $PO^*_{gcm}(G, C)$ holds, then the ATP will find a proof for $PO_{gcm}(G, C)$ in finite time. On the other hand, if $PO_{gcm}(G, C)$ doesn’t hold, then the ATP won’t stop in general. Unfortunately, even in the first case, the time for getting the answer from the ATP is undefined, that means there might be cases, where the prover needs a very long time to find a proof. In general, it’s not possible to distinguish both cases, if we don’t get an answer for a long time.

**How can we deal with this situation?** One pragmatic approach to this problem is to set and use a time-limit, for finding the answer. If the proof obligation has not been established within that period of time, we stop the prover and assume that the capability doesn’t match the goal (resp. at least we don’t assume that they match!). Obviously, this again restricts our notion of Goal-Capability-Matching further and is only applicable, when a suitable number of actual matchings can be detected in that way.

A second approach is to restrict the language for specifying goals and capabilities in such a way, that we end up with a language where our specific proof-obligation $PO_{gcm}(G, C)$ is decidable (for all $G$ and $C$ in the restricted language). This approach might be limited by the application domain requirements on the expressiveness of the language for specifying semantic services.

The following third approach can be seen as an intermediate approach between the two mentioned first: We restricts the syntax of the language for in such a way, that our proof-obligation is not guaranteed to be decidable but very efficient and specialized proof-search techniques can be applied. Thus we expect to find (existing) proofs several orders-of-magnitude faster in average.

In our opinion all theory is grey and the feasibility resp. the unfeasibility of one approach or the other has to be analysed by some case-studies.

### 4.2 Service discovery using logic programming

In this section we revisit the proof obligations presented in section 4 and we discuss their applicability when using F-Logic [KLW95] to express requester goals and service capabilities, and Flora-2 [YK] as the underlying inference engine.

Notice that we refer here to the proof obligations for goal-capability-matching. Our aim is to present to what extent these proof obligations can be tested using the reasoning services of Flora-2 and to introduce, where necessary, new assumptions that better define the semantics of goal-capability-matching in this context.

The proof obligation introduced in section 4 was as follows:

---

25 But it might stop for some particular cases.
26 Please note that this is different from having a language for which logical entailment is always decidable! Indeed, this would be even more restrictive.
27 For classical logic such an example would be SLD-Resolution which exploits syntactical characteristics of the Horn fragment of First-Order Logic in order to get rid of non-determinism when searching for a proof. This method is sound and complete for the Horn fragment, but not complete for arbitrary formulas in CNF.
PO\textsubscript{gcm}(G, C) \equiv \{O_C, O_G, M_C, M_G\} \vdash_C Cl_\exists(\exists in_1, \ldots, in_n: (\Psi_{\text{pre}} \land \Psi_{\text{ass}} (\Psi_{\text{post}} \rightarrow \Phi_{\text{post}}) \land (\Psi_{\text{eff}} \rightarrow \Phi_{\text{eff}})))

The proof obligation above expresses that we have a goal-capability-matching if we can prove that our knowledge base (ontologies and mediators) entails that exists a combination of inputs to the service such that the capability preconditions are satisfied, and the capability postconditions imply the goal postconditions, and the capability effects imply the goal effects.

In the following sections, we will discuss how this proof obligation can be tested using F-Logic and Flora-2.

4.2.1 Goal as ground facts

In the proof obligation above all the free variables, both in the capability and in the goal, are existentially quantified by the use of the existential closure of the proof obligation (Cl\exists(\Phi)). Notice that in the goal the presence of free variables is also possible, and that in the proof obligation these variables are existentially quantified.

The goal postconditions and effects can be seen as formulas constraining the set of all possible termination states, both in the information space and in the real world space. The proof obligation requires that there exists a certain combination of input values such that the preconditions of the capability hold, and the state defined by the capability postconditions (resp. effects) that result of the execution of a service presenting this capability for this combination of inputs, is in the set of valid termination states defined by the goal postconditions (resp. effects).

We first consider the simplest case in which the goal postconditions and effects are ground facts i.e. there are no free variables in the formulas expressing the goal postconditions and effects. That means that the goal postconditions (resp. effects) define a single valid termination state in the information space (resp. in the real world space) instead of a set of valid termination states.

For the definition of the capability postconditions (resp. effects), we still allow the use of arbitrary F-Logic formulas that constraint the set of possible termination states in the information space (resp. in the real world space) that result of the execution of a service presenting this capability.

In our first approach, we model goal postconditions and effects as ground facts that are asserted to the knowledge base, and capability postconditions and effects as queries over the knowledge base. We consider that we have a goal-capability-match when there is an answer to both the query expressed by the capability postconditions and the query expressed by the capability effects.

**What does this mean?** The capability postconditions and effects are formulas restricting the possible termination states in the information and real world spaces, respectively. When using them as a query against the knowledge base to which the goal postconditions and effects are asserted, we are checking if the goal is an answer to the capability’s query i.e. if the goal postconditions and effects fulfill the constrains defined by the capability postconditions and effects i.e. if the termination state expressed in the goal is among the possible termination states of the capability.

In terms of entailment checking, the use of the capability as a query and the goal as a ground fact can be seen as checking the following entailment:
\[ \text{PO}^{gcmgf}_{gen}(G, C) \equiv \{ \mathcal{O}_C, \mathcal{O}_G, \mathcal{M}_C, \mathcal{M}_G \} \land \Phi_{post} \land \Phi_{eff} \]

However, as preconditions are not part of the goal definition in WSMO [KLR04], they cannot be asserted to the knowledge base in the same way as goal postconditions and effects. Therefore, we cannot follow the same approach for checking the capability preconditions. For that reason, we leave them out at the moment, which leads to the following actual proof obligation:

\[ \text{PO}^{gcmgf}_{gen}(G, C) \equiv \{ \mathcal{O}_C, \mathcal{O}_G, \mathcal{M}_C, \mathcal{M}_G \} \land \Phi_{post} \land \Phi_{eff} \]

Please notice that:

- Preconditions are modelled as part of the capability definition, but they cannot be checked as long as there is no precondition defined in the goal and the actual input is not available.

- If the postconditions or effects refer to input values constrained by the precondition expression, these constraints will also be in the query and they have to be fulfilled by the goal facts.

As can be seen in equation 12, we have changed the direction of the entailment from equation 9. We are checking if the goal entails the capability instead of checking if the capability entails the goal. This might look counter-intuitive. One would expect that, as the capability reflects the functionality provided and the goal the functionality requested, the capability should be the one that fulfills the goal constraints and not the other way around. The reasons for the use of this new proof obligation 12 are:

- As query answering is the reasoning service provided by Flora-2, we need to model one side of the matching information (goal or capability) as a fact or set of facts, and the other side as a query over these facts. In order to keep the direction of the entailment of the proof obligation 9, the capability must provide the ground facts that will be asserted to the knowledge base. However, we believe that in the general case it is not realistic to expect the service provider to describe all the possible facts that will result from the execution of a service. As an example, imagine the capability of a service that provides flight tickets. In order to model the capability as a set of ground facts, the provider must list all the possible tickets (connections) that the service can provide, which in turn implies describing the whole database of the provider in the capability. We believe that in the general case the provider will describe in the capability the constraints that define the set of possible results of the service. In addition, we believe that it is more realistic to expect the requester to define the exact facts that he expects as the result of a service execution.

- Although the proof obligation 12 might look counter-intuitive, it is in fact checking if the goal expressed by the requester satisfies the constraints over the possible results of a given capability. We find this approach close to the user intuition.

Limitations
Incomplete facts. We assume that the goal will be described as a set of ground facts expressing the postconditions and effects. However, the requester might be interested in providing incomplete facts as a goal. As an example, a requester might be interested in a flight from Innsbruck to Frankfurt but he does not want to place any restriction on the carrier. In our approach, this can lead to two different situations:

- 1) If the capability does not place any constraint on the carrier, this constraint will not be in the capability query and the goal will be an answer to the query i.e. we will have a match.

- 2) If the capability restricts the carriers e.g. only offering flights with Iberia or Lufthansa, this constraint in the query will not be satisfied by the facts given in the goal and, therefore, the goal-capability-matching will fail. However, as the requester did not place any constraint on the carrier i.e. he does not care, this capability should have matched the goal.

Therefore, we cannot assure that the proof obligation [12] gives all the correct matching capabilities when the goal facts are partially specified, i.e. the approach is not complete.

Expressivity. As we rely on ground facts for describing the goal, the kind of goals a requester can describe is limited. Goals such as requesting a flight ticket for today or tomorrow, or giving a price limit instead of a fixed price, cannot be expressed.

4.2.2 Transaction logic reasoning

The approach presented in the previous section is extended here using transaction logic reasoning in order to augment the expressivity allowed for the description of the goal and in order to check the entailment in the direction given by equation [9].

Goals and capabilities are represented in the following way:

- Goal
  - Postcondition
    * $\neg goalPQuery$: a boolean combination of literals (including negated literals). A literal is an atomic formula or an F-logic molecule, possibly negated.
    * $goalPAxioms$: a set of rules that define some of the literals in $goalPQuery$. The rule heads are single positive literals or conjunctions of literals. Notice that rules that have conjunctions of literals in the rule heads can be replaced with sets of rules that have singleton literals, so this does not extend the expressive power.
  - Effect
    * $\neg goalEQuery$: a boolean combination of literals (including negated literals). A literal is an atomic formula or an F-logic molecule, possibly negated.
    * $goalEAxioms$: a set of rules that define some of the literals in $goalEQuery$. The rule heads are single positive literals or conjunctions of literals.

- Capability
D5.1 – Inferencing Support for SWS — Proof Obligations

- **Precondition**
  * \( \text{preQuery} \): a boolean combination of literals, like \( \text{goalQuery} \).
  * \( \text{preAxioms} \): rules that define some of the literals in \( \text{preQuery} \).
    The rule heads are single positive literals or conjunctions of literals.

- **Postcondition**
  * \( \text{postFacts} \): conjunction of positive literals, but can also contain disjunctions.
  * \( \text{postAxioms} \): rules that define some of the literals in \( \text{postFacts} \).
    Each rule can have disjunctions of conjunctions of positive literals.

- **Effect**
  * \( \text{effFacts} \): conjunction of positive literals, but can also contain disjunctions.
  * \( \text{effAxioms} \): rules that define some of the literals in \( \text{effFacts} \).
    Each rule can have disjunctions of conjunctions of positive literals.

For discovery, we test whether the goal queries for the postconditions and effects hold after the service execution. The service execution is simulated by using hypothetical updates i.e. we assume that the service presenting the capability executes for a given input and assert the postconditions and effects of the capability. The proof obligation is the following:

\[
PO_{\text{gmed}}(G, C) \equiv \\
\{ \mathcal{O}_C, \mathcal{O}_G, \mathcal{M}_C, \mathcal{M}_G \} \models \\
\Diamond (\text{insertrule}\{\text{preAxioms}[Input]\} \odot \text{preQuery}[Input]) \\
\odot \\
\Diamond (\text{insertrule}\{\text{postAxioms}[Input], \text{effAxioms}[Input], \text{goalP Axioms}, \text{goalEAxioms}\} \\
\odot \text{insert}\{\text{postF acts}[Input]\} \\
\odot \text{insert}\{\text{effF acts}[Input]\} \\
\odot \text{goalP Query}[Input]) \\
\odot \text{goalEQuery}[Input])
\] (13)

In the proof obligation, \( \Diamond \) is the possibility operator i.e. hypothetical assertions, \( \odot \) is the serial conjunction, and \( \text{formula}[Input] \) means that the variable \( Input \) in \( \text{formula} \) is replaced by the actual value of \( Input \).

Please notice that, as Flora-2 does not support hypothetical updates, we simulate them by inserting and deleting rules and facts from the knowledge base.

It can be seen that the proof obligation above represents a more intuitive way to check goal-capability-matching than the one presented in section 4.2.1, as we check if the result of the (hypothetical) execution of a service presenting this capability satisfies the constraints on the set of valid termination states expressed in the goal.

**Limitations**
The input has to be provided at discovery time. The presented approach requires that the input the requester is willing to provide is described at discovery time. Only in this case, the postconditions and effects of the capability can be asserted to the knowledge base, and the goal postconditions and effects can be used as queries. The same applies to the precondition, that can only be tested if the input to the service is provided.

We believe that assuming that the information provided for the goal (e.g. that the departure city has to be Innsbruck) will be provided as an input to the service is a realistic assumption, as this information is provided anyway in the description of the goal. However, a capability might request additional information that is not directly part of the goal e.g. login and password information. In this case, if we need that all possible inputs that might be required by the capabilities fulfilling the goal have to be present in the description of the goal, we clearly limit the set of capabilities that can be matched. We will not match a capability that can fulfill the goal if the required input information was not described in the goal. It might be unrealistic to expect the user to list all the possible inputs he can provide in order not to leave out possible matching capabilities. Furthermore, the capability must completely describe the function that maps the inputs of the service to the termination states in order to have completely grounded facts for the capability when the actual values for the input are provided.

In addition, as the input is not modelled as part of the goal in WSMO [KLR04], this approach requires this new element to be included for the modelling of goals in WSMO.

Incomplete facts. Symmetrically to what happened with the previous approach, if the goal postconditions and effects place some constraints that are not present in the (grounded) capability postconditions and effects, we will not have a match. For example, if the goal expresses a price constraint and the capability does not express any constraint on this property, the capability facts (grounded with the input information) will not be an answer to the goal query. Therefore, we have again that the matching is not complete for partially specified capabilities or goals. This problem can be overcome by requiring the goal and capability to explicitly represent "don’t care" properties by giving them a variable as value.

4.2.3 Open points

Use of ground facts. In the previous sections it can be seen that as long as we rely on query answering, we need ground facts either for the goal or for the capability. In Section 4.2.1 we rely on ground facts for the goal so that we can check if these facts fulfill the constraints expressed by the capability. In Section 4.2.2 we rely on grounding the postconditions and effects of the capability through the inputs given by the requester. Then we can check whether the resulting facts satisfy the goal constraints. In the first case, we limit the expressiveness of the goal and we do not have a complete procedure if incomplete facts are given in the goal. In the second case, we require an exhaustive description of all the information the requester is willing/able to provide.

Matching without input information. In the case where the actual input information is not required and, at the same time, we do not restrict the expressions in the goal to ground facts, we have a set of logical formulas restricting the set of states that are valid for the requester and the set of states that can result from the execution of a service providing the capability. In this case, two different possibilities appear:
• Compatible constraints. One possibility would be to check whether the constraints described in the goal do not contradict the constraints described in the capability. However, this can result in matching of capabilities and goals that have nothing in common, as they obviously do not express contradicting constraints although the capability does not fulfill the goal given. Therefore, this possibility cannot be considered.

• Common states. Another possibility is to check whether there can be common valid states for the capability and the goal i.e. if at least one valid state for the goal is a termination state for a service providing the capability. This can be done through query containment, checking if the postconditions and effects of the capability (viewed as queries) contain the postconditions and effects of the goal (viewed as queries). This would mean that the termination states of the capability are a (possibly not strict) superset of the valid states for the goal. We can also consider as a match the case where the goal queries contain the capability queries, although in this case combinations of inputs that do not lead to the fulfillment of the goal can exist.

Matching with input information. If the input information the requester is willing/able to provide is required, we can follow the approach of Section 4.2.2. As mentioned, this implies that the user have to describe in the goal all the possible input information he will provide. While assuming that the requester will list all this information for every goal can be unrealistic, assuming that the user will maintain a knowledge base with all this information is a more realistic assumption. In this case, additional input information can be requested from the user whenever necessary. Under the latter assumption, we can consider a two-phase provision of the requester input for the goal-capability-matching.

In the case where the capability postconditions and effects cannot be grounded with the input information given in the goal, the missing input information to ground these facts can be requested. If the requester is able to provide this additional information, we will check the matching with grounded facts. Otherwise, the matching will fail. A drawback of this approach is that it can become inefficient, as services that provide a completely different functionality to the one in the goal will in most cases request additional information to ground their postconditions and effects.

Relative matching. As mentioned in previous sections, the notion of goal-capability-matching is always relative, as it will depend on the actual input values and state of the world (for the satisfaction of the assumptions) at execution time. Furthermore, depending on the approach followed, it can be relative to the completeness of the facts given in the goal or to the provision of all the necessary input information.

These open points will be studied and possible solutions will be provided in future versions of this document. From the discussion above, we believe that the most effective solution would be checking the existence of common states. However, query containment is not implemented in Flora-2, and depending on the expressivity allowed for postconditions and effects it can become undecidable. Another possible direction is the two-phase matching where missing input information is in turn requested to the user.
5 Conclusions

5.1 What have we achieved so far?

Currently we mainly addressed the problem of service discovery in the context of WSMO, that is, given a description of what the requester wants to achieve (Goal) as well as a (possibly huge) set of descriptions of services that precisely state what the corresponding services provide as their functionality (Capability).

For service discovery, we distinguished two variants, which are basically orthogonal to each other and can be combined to a more efficient practical approach to identifying adequate services for some client request: One variant which makes no use of knowledge expressed by means of mediators between goals and capabilities as well as goals in WSMO on the one hand, and an approach that exploits the knowledge about mediators in WSMO in order to speed-up (resp. simplify) discovery by pruning the search space on the other hand.

The former has been discussed in detail in sections 3.2 and 4.1 where we mainly focussed on a specific and fundamental subtask (see section 4.1.1): Goal-Capability-Matching. The latter has been described on a conceptual level in section 3.2.2.

In section 4.2 we discussed an approach to service discovery based on logic programming techniques and Transaction Logic reasoning.

Furthermore, we identified an initial set of subtasks of service discovery, for which semantic support would be beneficial, and briefly discussed the relation between service discovery and service composition.

5.2 Future Work

Obviously, besides service discovery there are a number of very interesting domains in the context of semantic web services, where semantic support is useful and needed.

Naturally, service discovery is one of the most fundamental of these domains. Additionally, the corresponding description means in WSMO are fully defined and in a sense well-understood. For this reasons, we started with the investigation of this domain.

Clearly, we will look all other domains as well – but a necessary prerequisite for this is some mature intermediate state of the corresponding deliverables in WSMO. In particular, we aim at automated (or at least computer-supported) analysis of global and local properties of a software-system, which is composed by semantic web services.

\[28\] For instance, the description of orchestration and choreography of web services are currently not as well-developed!
But even for service discovery, there are some aspects besides the consideration of the pure functionality (Goal-Capability-Matching) that we haven’t treated in detail until now. In particular, we want to look at the „dynamic part” of web service descriptions in WSMO, that is the choreography, and see how we can include this information in a service discovery request which constrains the choreography of accepted services. This issue will mainly be relevant on the application level of WSMO, namely WSMO-Full.

Similarly, the corresponding techniques are also relevant for checking whether two given web services which are intended to interact with each other – for instance by dynamically composing them within a service-based system – can actually interact with each other (and thus can indeed be composed) which is a fundamental task for automated and semi-automated service composition.

For now, the detailed comparison to other approaches to service discovery in the web service areas as well as for other areas in computer science (like component systems) is missing and should be explained in future versions of this document. As an alternative, we are currently thinking about documenting a separate in-depth discussion of existing approaches to service discovery in form of a technical report of the DERI institute. In this case, we would refer to this technical report within this document.

To sum up, until now we covered only a small aspect of the whole picture. Certainly, a lot of interesting questions and difficult problems will arise along our way when we try to complete our investigation of the various areas where semantic annotation of web services can be exploited beneficially and we’re excited to see how (and to what extend) we can cope with them (in a practical setting).
6 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 programme.

The editors would like to thank all the members of the WSMO working group for their advice and input into this document.
A Further Notes on Service Discovery

This section describes and discusses several important questions related to service discovery in WSMO and WSML. It provides a more detailed explanation of several nuances which have been omitted in the main part of the text for the sake of simplicity. Nonetheless, we believe that the following sections are valuable in order to get a full understanding of the service discovery mechanism in WSMO as well as its limitations.

Precision and Recall

Information Retrieval Systems are assessed by means of two measures: Precision, which measures to what extent no inadmissible elements are discovered and Recall, which measures to what extent all admissible elements from some domain are discovered. Since service discovery can be seen as a particular retrieval task, we can apply the same assessment techniques here.

Preconditions & assumptions in the context of matching

As mentioned, the introduced notion of matching is a relative one - it’s relative to the ability of the requester to ensure the preconditions and the assumptions. The match is valid only if user ensures that the precondition and assumptions are met when actually invoking the service. What consequences has this fact? Indeed, the given explanation is a bit too sloppy: How is the user able to ensure these conditions at all?

With preconditions it's straightforward: Preconditions are defined as conditions that the input values of a user have to respect at invocation time in order to allow the service to work properly. Such a condition could be the requirement that a particular integer-valued input parameter that represents the day during a month of a specific date only takes values between 1 and 31. Since these conditions are explicitly mentioned and they only refer directly to the user input, the user can indeed easily ensure that the requirements are met by his input.

With assumptions the situation in general is more involved: Assumptions are constraints on the state of the world that the service imposes. Only when the assumptions are satisfied at invocation time, the service guarantees to work properly. Since assumptions refer to the „current“ state of the world, the user only has limited (or no) possibilities to influence the properties of the state of the world. In most cases, the user is not even interested in actively changing the state of the world such that the assumptions are met: Imagine for example the assumption of a book-selling service that enough copies of the desired book are available in the stock; here the user himself can not change the state of the stock. Even worse, he might not be able to get any information about the state of the book-sellers stock. If the user is really lucky, then he is indeed able (perhaps by using some additional services) to check whether the assumptions are fulfilled at invocation time. In general, this will not be the case.

But what consequence has this issue: According to our definition of matching between goal and capability, a service that provides the given capability would match the goal although (when it comes to the actual invocation) the service indeed can not be used to fulfill my goal, because the assumptions are not satisfied! And this discussion also shows, that if it’s not possible to automatically ensure the assumptions (by checking them) at invocation time, it’s not possible to automatically discover and invoke a service which offers this capability.
without having doubts that the service can’t be used for the intended purpose. That means fully-automated and dynamic system-construction at runtime is not possible in a reliable way under these circumstances. But it still is possible, if all assumption can be checked at invocation time.[29]

Naturally, we have to ask ourselves whether our definition for goal-capability-matching is indeed adequate for our purpose:

The discussion above clearly shows that the property of our criterion as being „only” relative to the precondition (and not explicitly taking into account the precondition for the decision on matching) is not a flaw of our definition but somehow inherent to all such formal criterions:

Discovery is a process that chronologically occurs before the corresponding service invocation. It’s never possible to have both at the same time. Thus, during the period of time between both activities the world might change and the results of the discovery process might become invalid (wrt. to the new state of the world – that is, the current one at invocation time).

Moreover, during service discovery usually not all information that (of course) is available when the user is actually invoking a service (actual state of the world at invocation time as well as all input values) are known at discovery time. For instance, one might think of an additional input parameter of a candidate service for which its concrete value at invocation time can not be determined by just using the goal specification given by the requester.

Thus, by taking preconditions and assumptions into account at discovery time, we basically have the same problem, since preconditions and assumptions have to be evaluated with respect to the concrete state of the world (and information space resp. input values) at the actual invocation time but this point in time is different from the point in time when discovery happens.

Nevertheless, checking the preconditions and assumptions at invocation time is a very important if we want to construct reliable and correct software-systems automatically on runtime by composing web-services.

Accuracy of the Service Capability

Up to now, the basic assumption has been that the capability accurately describes the functionality provided by a service.

Unfortunately, in a real environment we have to assume that there is a difference between the actually provided service functionality and the capability that has been assigned to a service by a service provider: A service actually does not provide precisely what the capability claims about the service.

Let us stress two observations: First, service providers (or specialists in formally describing services) are the responsibilities for defining and attaching a capability to a service. Second, service providers are usually interested that their services are discovered and (more important) used (for instance in order to make money). One (simple) way increase the probability of usage is to increase the number of discoveries, since discoveries are a need for service usage, if service providers and service requesters are strongly decoupled.

Hence, there are several reasons for such a difference with varying effects on discovery:

[29] Of course, all the problems with a changing world with agents that act concurrently that are well-known from information systems occur in this context as well!
• **Simplification of the service description.** The service provider is not willing to specify all the details related to his service, but decides to „graciously” simplify the description by abstracting from details.

One example would be that a service states to offer information all events in all cities in Europe although it is only capable of providing this kind of information for a (large) list of cities in Europe.

Clearly, the capability descriptions is a lot simpler and it increase the number of possible discoveries, which potentially is beneficial for the provider, but the discovery is less accurate, i.e. the precision of the discovery mechanism decreases.

• **Unawareness of incorrect descriptions.** The person who specifies the capability gives a wrong description, for instance because he is no expert in formal specification or he is not concentrated.

One common example is the semantically incorrect use of ”and” in everyday life: It often happens, that people use the connective ”and” when they actually (that is from a semantic perspective) mean ”or”. Another common problem is the use of (nested) quantifiers in a first-order language.

As before, the precision of the discovery component decreases.

• **Fake.** This is very similar to the first case, but related to even more inaccurate descriptions. The incorrectness of the description is not based on simplification, but one the intention to reach a maximum number of potential clients:

One can imagine an ”I provide everything in the area of tourism” service which actually is only capable of booking rooms in a few hotels in Tyrol.

Once again, the precision of the discovery component decreases.

Nonetheless, the basic assumption in this document and in the WSMO project at present is that the registry trusts a service provider when he attaches a capability to a service.
References


[LH] Lei Li and Ian Horrocks. A software framework for matchmaking based on semantic web technology.
