



Project Number: **215219**
 Project Acronym: **SOA4All**
 Project Title: **Service Oriented Architectures for All**
 Instrument: **Integrated Project**
 Thematic Priority: **Information and Communication Technologies**

D5.3.2 Second Service Discovery Prototype

Activity N:	A2 – Core Research and Development	
Work Package:	WP5 - Service Location	
Due Date:	31/08/2010	
Submission Date:	31/08/2010	
Start Date of Project:	01/03/2008	
Duration of Project:	36 Months	
Organisation Responsible of Deliverable:	KIT	
Revision:	1.0	
Author(s):	Sudhir Agarwal (KIT), Martin Junghans (KIT), Barry Norton (KIT)	
Reviewers:	Yosu Gorroñoigoitia (ATOS) Patrick Un (SAP)	

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
Dissemination Level		
PU	Public	x
PP	Restricted to other programme participants (including the Commission)	
RE	Restricted to a group specified by the consortium (including the Commission)	
CO	Confidential, only for members of the consortium (including the Commission)	

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
1.0	04.08.2010	Initial Version	Martin Junghans
1.1	09.08.2010	Executive Summary, Introduction	Sudhir Agarwal
1.2	18.08.2010	Conclusion and Outlook	Barry Norton
1.3	20.08.2010	Version for internal review	Sudhir Agarwal
1.4	27.08.2010	Addressed reviewers' comment, final version	Sudhir Agarwal

Table of Contents

EXECUTIVE SUMMARY	6
1. INTRODUCTION	7
1.1 PURPOSE AND SCOPE OF THIS DELIVERABLE	7
1.2 STRUCTURE OF THE DOCUMENT	7
2. PRACTICAL SEMANTIC WEB SERVICE DISCOVERY	8
2.1 REQUIREMENTS	8
2.2 SERVICE DESCRIPTIONS	11
<i>Formal Model of Web Services</i>	11
<i>Description Formalism</i>	13
<i>Modeling Example</i>	15
2.2 SERVICE REQUESTS	17
<i>Request Description Syntax</i>	17
<i>Semantics of Service Request</i>	19
2.3 MATCHMAKING	21
<i>Matching Properties</i>	21
<i>Matching Functionalities</i>	22
3. IMPLEMENTATION AND EVALUATION	25
3.1 INTEGRATION WITH SOA4ALL SERVICE REPOSITORY	25
3.2 INTEGRATION WITH WSML2REASONER	26
3.3 SERVICE TEMPLATES	26
3.4 PERFORMANCE RESULTS	28
3.5 USER INTERFACE	31
4. USE OF DISCOVERY WITHIN SOA4ALL	33
5. RELATED WORK	34
6. CONCLUSION AND OUTLOOK	36
REFERENCES	37

List of Figures

Figure 1: Discovery approaches that use the same formalism for an offer D and a request R are based on intersections (left). Using two different formalisms allows specifying the (exact) matches in the request (right).....	9
Figure 2: Formal Property-Based Model of Web Services	12
Figure 3: Actual Functioning of a Web Service	13
Figure 4: Abstract Formal Model of the Functionality of a Web Service	14
Figure 5: Mean query answering time against increasing number of Web service descriptions for three query sizes.....	30
Figure 6: Discovery user interface with request specification (left) and desired services and operations (right).	31
Figure 7: Specification of desired service classification.	32
Figure 8: Specification of desired functional and non-functional properties.	33

List of Tables

Table 1: Mapping between theory and implementation of service descriptions.	25
Table 2: RDF Schema for service template.	27
Table 3: An Example Service Template	28
Table 4: Query sizes tested in the experiment.	29

Glossary of Acronyms

Acronym	Definition
D	Deliverable
DL	Description Logic
EC	European Commission
ISBN	International Standard Book Number
NFP	Non-Functional Property
NFR	Non-Functional Requirement
POSM	Procedure Oriented Service Model
RDF	Resource Description Framework
RDFS	RDF Schema
REST	Representational State Transfer
SAWSDL	Semantic Annotations for WSDL
SDC	Semantic Discovery Caching
SOA	Service Oriented Architectures
SWRC	Semantic Web Research Community
UK	United Kingdom
WP	Work Package
WSDL	Web Service Description Language
WSL4J	WSMO-Lite for Java
WSML	Web Service Modeling Language
WSMO	Web Service Modeling Ontology

Executive summary

Discovery of service descriptions is a central task in Service Oriented Architecture (SOA). The service discovery component enables users to find services appropriate for their needs from a large pool of available services. In the previous deliverables, we have presented a full text based discovery and a preliminary semantic discovery. In this deliverable, we present a more sophisticated and expressive semantic discovery technique as well as its implementation details and integration within other SOA4All components and its use within SOA4All use cases. The semantic discovery component allows users to specify a structured query containing requirements on service classification, pre-conditions, effects and non-functional properties and finds the services that match the query by using the reasoning facilities provided by WP3.

1. Introduction

SOA4All will help to realize a world where billions of parties are exposing and consuming services via advanced Web technology. The outcome of the project will be a comprehensive framework and infrastructure that integrates four complimentary and revolutionary technical advances into a coherent and domain independent service delivery platform.

In such a setting users require mechanisms to find Web services that are suitable for their needs automatically. The mechanism should allow users to search for Web services by specifying desired functionality. In many practical scenarios, the non-functional properties of Web services are often crucial. For example, a user may not want to use a Web service for a particular purpose if it has low response time even though it offers the required functionality. Furthermore, Web services in general are not only data provision services, but can also cause changes. For example, a bank transaction service will cause a change in the account balance of the user.

The semantic discovery component presented in this deliverable extends the first discovery prototype in many ways. It allows users to specify a structured query containing requirements on service classification, inputs, outputs, pre-conditions, effects and non-functional properties and finds the services that match the query by using the reasoning facilities. Apart from the conceptual and theoretical details about the functioning of the discovery component, a prototypical implementation along with performance evaluation results of the discovery component is also presented. The discovery component is integrated with the iServe repository and the ontology reasoner, which means that it fetches the service descriptions from the iServe repository and uses the reasoner to reason about various properties of the descriptions obtained from the iServe repository.

1.1 Purpose and Scope of this Deliverable

SOA4All is proposing a new paradigm where billions of services will be available for the users to interact with them. Thus, in order to enable an interaction with the right services we will need an efficient methodology to discover relevant services from the end-user perspective on large number of available services. Discovery of Web services is one of most important steps performed by a user in the overall Web services usage life cycle. The discovered Web services are meant to be embedded in a larger process which is then executed. The current deliverable describes a sophisticated expressive semantic discovery of services. The results presented in this deliverable will be mainly used by service provisioning and service consumption platforms developed in WP2 as well as automatic compositions techniques developed in WP6. Furthermore, the discovery component plays a central role in the SOA4All use case scenarios. The discovery component itself relies on the iServe repository for obtaining the semantic descriptions of services, and in many cases the domain ontologies, and uses the reasoning facilities provided by WP3.

1.2 Structure of the Document

This document is structured as follows. Section 2 describes the conceptual and theoretical details of the semantic discovery approach. Section 3 describes the implementation and

evaluation details. In particular: (1) the mappings between the conceptual models from Section 2 and the concrete service description formalisms developed in WP3, (2) functioning of the discovery component with details about its integration with the WP2 iServe repository and the WP3 reasoning facilities, (3) performance evaluation and (4) Graphical user interface. In Section 4, we present details about the usage of the discovery component within SOA4All. We distinguish between the usage by other platform services, e.g. composition (WP6), and usage within the SOA4All use cases. In Section 5, we discuss some related work in detail and compare them with our work. In Section 6, we summarize our results and discuss briefly the next steps in the context of this task.

2. Practical Semantic Web Service Discovery

2.1 Requirements

In this section, we discuss in detail the limitations of existing semantic Web service discovery approaches and derive the requirements for our SOA4All approach.

Comprehensible Requests and Descriptions – Uniformity of Offer and Request. One common problem of almost all existing and well known approaches is that they apply the same formalism for describing service offers and service requests. Intuitively, a service description formalizes the actual values of the Web service properties and a service request specifies the acceptable value range of the properties. Therefore, using the same formalism with same interpretation for both service description as well as request does not correspond with the intuition of the requester. Such mismatch between the semantics of formalisms and the intuitive interpretation of the requester makes these approaches hard to use in practice. There exists a mismatch between the interpretation of D that describes a service and the tuple \mathcal{R} that describes a request, if the description of a service request uses the same formalism as the one used for service descriptions.

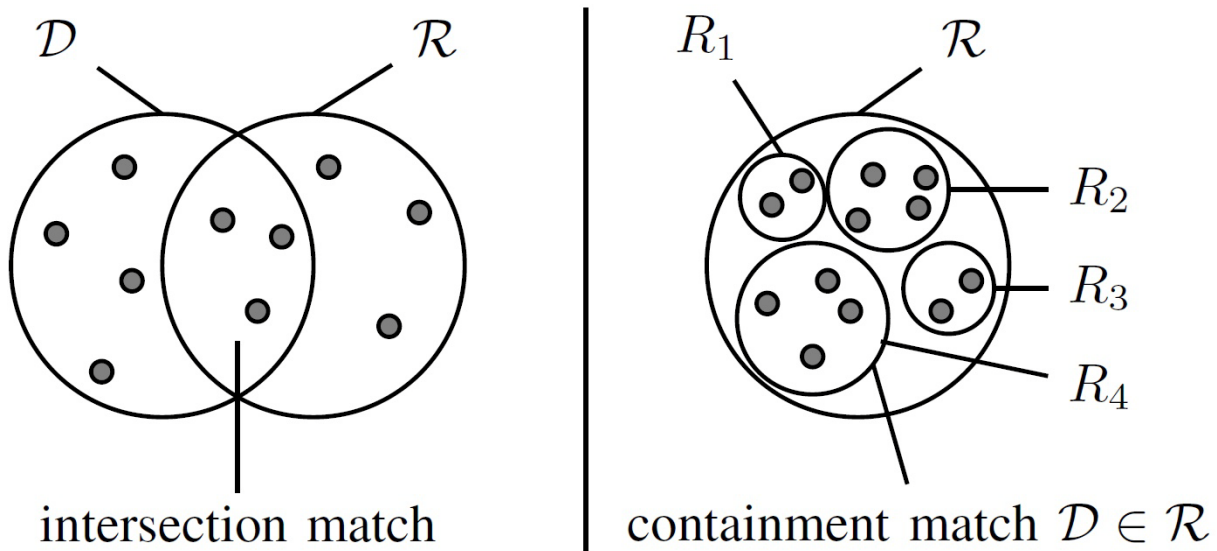


Figure 1: Discovery approaches that use the same formalism for an offer D and a request R are based on intersections (left). Using two different formalisms allows specifying the (exact) matches in the request (right).

If the same formalism is used for offers and requests, then a service request corresponds to a service offer description from which a set of desired services is derived. In our view, this is an impractical and unintuitive approach as we will further justify in the subsequent section. Consequently, we propose to use two distinct formalisms for service descriptions and request. It is more intuitively that a service description formalizes the actual functionality of a Web service and a service request describes the set of services that provide a requested functionality.

The left side of Figure 1 shows a service description D and a request R that are interpreted as a set of execution runs depicted by dots respectively. A match is given if there is an intersection between D and R . Degrees of matches, for instance plugin or subsume match, present different types of the intersection of both sets of runs. Although the notion of different types of matches was applied in many discovery approaches [2, 3, 4, 5, 6], intersection based approaches require further matchmaking to make sure that a service is applicable for all the desired tasks at hand. For example, if only a few runs of a request overlap with those of a match, then it is possible that the requested domain of inputs is not covered by the match, which in turn causes that the discovered service is not applicable. In addition, if the requester wants to prohibit specific service properties, then it is not possible to ensure that the matching services fulfill such requirements, which might not be covered by the intersection. Since we aim at service discovery that is expressive enough to cover practical use cases, the request formalism must be capable to let users specify desired properties and rule out undesired ones as well as the specification of alternative and preferred properties.

Consider for instance the functional description of a book selling Web services that requires the invoker to provide as input an ISBN book number of the book to sell. While the inputs of the service descriptions are consentaneously interpreted as required for invocation, the interpretation of the inputs specified in a request using the same formalism is not clear. A user might specify a set of inputs he or she is able to provide or the set of inputs that a service has to consume. From a user perspective, a request for book selling Web services may contain different interpretations of a set of inputs simultaneously. Either the user provides an ISBN number or alternatively author name and book title as inputs. This very simple example leads to the observation that a set of inputs is not a proper representation of inputs in a request. While inputs of a service description are usually interpreted as compulsory, the interpretation of inputs within a request however should not be solely compulsory since the user may also want to specify which inputs must not be provided for a service invocation, like the number of a bank account. The user should be able to specify as much information as he or she wants to provide in a service request to precisely characterize the desired set of discovery results.

Intersection-based approaches lack the ability to let requests exclude certain properties because not all requested properties need to be fulfilled by matching services. Furthermore, an intersection-based match cannot guarantee that a matching service can be successfully invoked as the execution run that the user wants to invoke may not comply with the requirements specified in the request. This can be since the desired run might not be member of the intersection between service offer and request. Thus, in order to guarantee applicability of a matching service, intersection-based approaches using the same formalism for offers and requests need to further check for applicability in a further step. Consequently, the freedom provided by the different matching degrees is not practicable for the purpose of service invocation. As an example, a service offers to ship goods from a city in the UK to another city in Germany. A user requests for a shipping provider that operates between European cities. Using intersection based matchmaking will identify the mentioned service as a match. However, if the requester wants to ship an item from Berlin to Hamburg, then the matched service offer fails.

The same conceptual mismatch between service descriptions and requests occurs in preconditions and effects. Employing the same formalism for the description of a service, a class of services, and a set of desired services is not appropriate, because their interpretation and their intended use are different. Requests conceptually differ from service functionality descriptions. We believe that this mismatch makes current functionality based formalisms difficult to use.

The right side of Figure 1 shows a more intuitive interpretation of a request (in analogy to database queries), in which a request is viewed as a set of all desired services a user is looking for. A desired service is described by a combination of desired properties depicted by the dots in the figure. Any service description from the pool of available service descriptions that describes a service contained in the set is considered as a match for the request.

Unified View on Functional and Non-Functional Properties. Non-functional properties (NFPs) are part of semantic service descriptions and supplement functionality descriptions of services. In contrast to functional descriptions that describe what a service actually does, NFPs describe manifold quality attributes of services. It can be observed that non-functional requirements (NFRs) are often referred to as soft criteria and exclusively considered for ranking [7]. It is not determined per se whether properties are interpreted as hard or soft requirements and it is likewise valid to perceive NFRs as hard requirements, too. For example, the NFP *availability* with a value of *0.99* can be considered for discovery as well. A request may specify that services have to offer an availability of at least *90%*, which is considered to be a hard requirement. When a user specifies that he prefers higher availability rates to lower ones then such requirements are referred to as a soft requirement and can be used to rank services.

Support for Services That May Cause Changes. Web services are not only information providing services, but many of them may cause changes, for example by creating a new order. Both, service descriptions and requests, have to capture the dynamic nature of services. Existing semantic discovery approaches do not support modeling the changes caused by a service execution nor do they support specifying desired and undesired changes in requests [3]. Although [5] addresses changes in the knowledge bases by the introduction of dynamic symbols, the discovery approach presented in [5] fails to reason about the dynamics of Web services.

2.2 Service Descriptions

In this section, we introduce a formalism to describe Web services semantically. We first present a formal model of a Web service that captures functional and non-functional properties in a unifying way. We then introduce a formalism for describing such models by presenting an abstract syntax and its semantics as a mapping to the formal model.

We present our formal model of Web services, the formalism to describe service properties including functionality descriptions. We consider atomic Web services that may require user inputs at service invocation time and provide outputs at the end solely. There are no user interactions in between, which allows us to describe service functionalities by the states before and after execution without stating anything about the intermediate states.

Formal Model of Web Services

We consider a finite set \wp of Web service property types and a finite set \mathcal{V} of value sets. Each property type $P \in \wp$ is associated with a value set $V_P \in \mathcal{V}$. We view a Web service as a finite set of property instances Q with each property instance $q \in Q$ being of a property type $t(q) \in \wp$ that is associated with a value $v_q \in V_{t(q)}$ (refer to Figure 2).

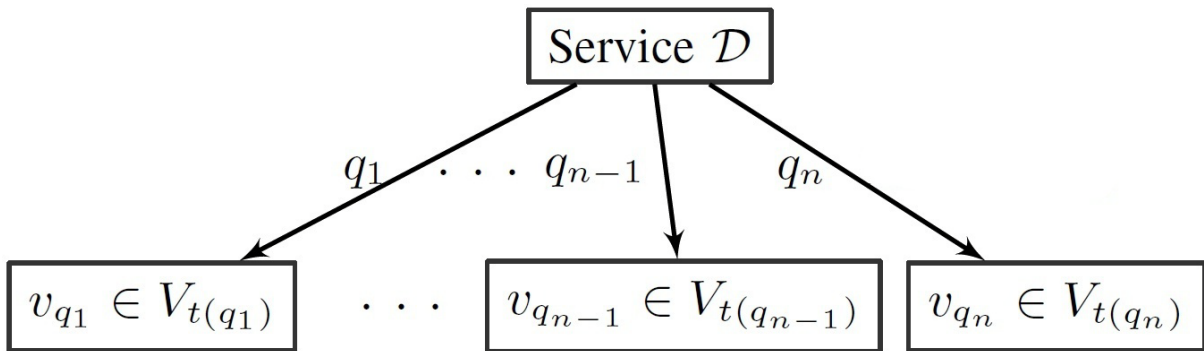


Figure 2: Formal Property-Based Model of Web Services

Of particular interest is the functionality or behavior of a Web service. We consider traditional Web services that have no user interactions during their execution. That is, we assume that inputs are provided with the invocation and outputs are returned at the end. In general, a Web service not only provides information but may also cause changes in the information state of the Web service provider.

Definition: Labeled Transition System (LTS). A labeled transition system $L = (S, T, \rightarrow)$ comprises a set S of states, a set T of transition labels and a labeled transition relation $\rightarrow \subseteq S \times T \times S$.

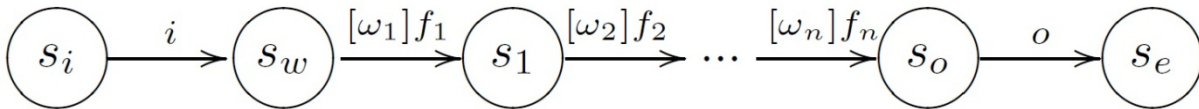


Figure 3: Actual Functioning of a Web Service

Figure 3 shows the formal model of the behavior of a Web service as a labeled transition system. The execution can be seen as a series of states:

- (1) s_i is the start state that contains the knowledge available to the Web service before the Web service is invoked by providing input parameters.
- (2) s_w describes the knowledge after the service invocation carrying the input values as well. This state contains the values of the input parameters in addition to all the knowledge of s_i .
- (3) A series of states s_1, \dots, s_n that occur during the execution of a Web service while computing the output values and performing any changes with actions $[\omega_1]f_1, \dots, [\omega_n]f_n$. An action f_i in a state takes place only if the condition ω_i is true in the state.
- (4) A state s_o after the Web service has performed all the changes and computed the output values. In this state the output operation o takes place, which emits the computed output values.
- (5) s_e denotes the end state that is equivalent to s_o since the output operation does not change the knowledge base.

The activities i, f_1, \dots, f_n and o can take place only if their respective beginning state is consistent. In particular, after inputs values are available in the state s_w , further execution of a Web service can take place only if s_w is consistent. Any conditions that need to hold on the input values are available as part of the knowledge in the state s_w .

While transitions i and o are communication activities for receiving values from and emitting values to the user resp., transitions f_1, \dots, f_n are local operations for navigating and performing computations within the knowledge base as well as adding or deleting facts in the knowledge base.

Description Formalism

The formal model of Web services as presented above cannot be described completely mainly because of the following reasons. (1) Service providers may not want to reveal the

exact internal sequence of local operations they perform. (2) Assuming a global set of property names is not feasible.

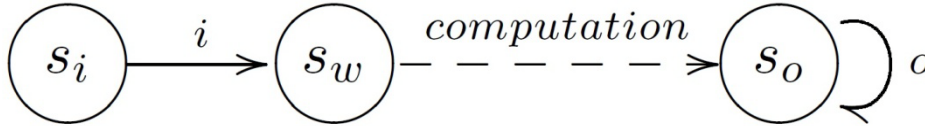


Figure 4: Abstract Formal Model of the Functionality of a Web Service

The first issue is relevant for functionality, which is in our model just one of many properties of a Web service. However, because of high importance of this property, we will deal with it in more detail in the subsequent sections. We address the second issue by modeling Web service properties as properties in WSML [11]. This allows us to use existing ontology reasoners to reason about Web service properties while not forcing a global set of property names. More precisely, we

- define for each value set $V \in \mathcal{V}$ an ontology concept V . Furthermore, we assume a set of common data types either available directly or modeled as ontology concepts.
- model for each property $P \in \mathcal{P}$ with range V_P a property P as an object property with range V_P if V_P is a set of individuals. Otherwise, if V_P is a data type, we model a property P as a data type property with range V_P .

Furthermore, since ontology languages allow alignment of concepts and properties in subclass-of and subrelation-of relationships respectively, we achieve interoperability among properties without demanding a global set of property names. We now turn our attention to modeling functionality of a service semantically.

Inputs. Inputs of a Web service are simply described as a set of variable names.

State after Input. After a Web service has received inputs, it reaches the state s_w , where the actual execution of the functionality of the Web service starts. This state contains

- ontology individuals describing the knowledge of the Web service before the input activity.
- input variables as ontology individuals. Note that even though inputs are concrete values that are unknown at the time of service description, we can differentiate inputs from other individuals, since we know from the set of inputs which of the individuals are inputs.
- formulas that represent the overall condition derived from the conditions w_1, \dots, w_n and actions f_1, \dots, f_n .

State Before Output. After a Web service has performed all the operations (computing the output and performing any changes), it is ready for emitting output values to the user. This state s_o describes the state obtained after performing the operations f_1, \dots, f_n to the state s_w and the subsequent states s_1, \dots, s_{n-1} .

Outputs. Outputs of a Web service are simply described as a set of individuals that are returned to the user.

Modeling Updates. The four properties described above model the functionality of a Web service. Changes caused by a Web service are modeled implicitly as difference between the pre-state and the post-state. Note, that a change can only be either addition or deletion of a concept or property instances. We model such changes with axioms about the existence and non-existence of the concept or property instances in the pre- and post-state.

The sets of inputs I and outputs O denote the set of inputs required by a service and the set of outputs returned by the service after its successful execution, respectively. They assign the service's inputs and outputs to variable names that can be referenced in preconditions and effects and also allow us to distinguish inputs and outputs from instances that already exist in the provider's knowledge bases. The types of inputs and outputs are specified in preconditions and effects.

The pragmatics of the logical formula that represents the precondition is restricted to the description of (i) requirements on inputs, like their types, relationships among them, conditions on the values of the inputs, and (ii) conditions that must hold in the state after inputs from the perspective of service providers. A precondition describes the state from the perspective of the service provider(s) before a service can be successfully invoked. In contrast to [20], by preconditions and effect we do not intend to describe global states that model the knowledge bases of the entire world as perceived by an external observer which certainly causes several problems. An effect formula is restricted to describe (i) constraints on returned outputs, (ii) the relation between inputs and outputs, and (iii) changes made by the service in the knowledge bases of service providers.

Modeling Example

We now give an example of service description with our formalism. Imagine a service s that has one operation that requires three inputs: user ID, password, and the ISBN number of a book. The service creates a shipping order for the given book if it is available to the user's address. The service then returns an invoice to the user about the order and price details. The Web service has inputs

$$I = \{id, pwd, isbn\}.$$

The precondition ϕ below states that the described service requires that the user $user$ with ID id is registered and authenticated by its password pwd . Furthermore, for a successful execution the service requires that the book $book$ with ISBN $isbn$ is in stock. If the user $user$ has already any reward points, they can be referenced by the variable $rewards$.

$$\begin{aligned} \phi \equiv & \text{isRegistered}(id) \wedge \text{isAuthenticated}(id, pwd) \wedge \text{Book}(book) \wedge \\ & \text{ISBN}(isbn) \\ & \wedge \text{hasISBN}(book, isbn) \wedge \text{isAvailable}(book) \\ & \wedge \text{User}(user) \wedge \text{hasID}(user, id) \wedge \text{hasRewards}(user, rewards) \end{aligned}$$

The effect ψ states that in the state s_e there exists an order `order` about product `book` (which is made sure by the precondition to be the ordered book) which is supposed to be shipped to the user's address denoted by `address`. Furthermore, it states that there exists an invoice about the order and price, which equals the price of the book. The user already collected `rewards` reward points before the service is executed. And as ψ guarantees, the user receives for each Euro spent for the book another reward point to its balance after service execution.

$$\begin{aligned} \psi \equiv & \text{Order}(\text{order}) \quad \wedge \quad \text{containsProduct}(\text{order}, \quad \text{book}) \quad \wedge \\ & \text{isShippedTo}(\text{order}, \quad \text{address}) \\ & \wedge \text{containsOrder}(\text{invoice}, \text{order}) \wedge \text{containsPrice}(\text{invoice}, \text{price}) \\ & \wedge \text{hasPrice}(\text{book}, \text{price}) \quad \wedge \quad \text{hasAddress}(\text{user}, \quad \text{address}) \\ & \wedge \text{Invoice}(\text{invoice}) \wedge \text{hasRewards}(\text{user}, \text{rewards}+\text{price}) \end{aligned}$$

Another example that also needs to model updates is a service that deletes a user subscription. Hence, the precondition contains $\exists u:\text{User}(\text{user}) \wedge \text{hasID}(\text{user}, \text{id})$ and the effect contains $\nexists u:\text{User}(\text{user}) \wedge \text{hasID}(\text{user}, \text{id}) \dots$. To explicitly create a subscription that did not exist before, the conditions apply to precondition and effect vice versa. Coming back to the book selling service, it returns the book and the corresponding invoice to the user, which is modeled by

$$O = \{\text{book}, \text{invoice}\}.$$

Apart from the functionality described above, the NFPs of the service s state that it does not accept credit cards, delivers within 2 days and has availability 0.90. With our formalism these characteristics can be modeled as follows.

$$N = \{ \text{acceptsCreditCard}(s, \quad \text{false}), \quad \text{deliversBefore}(s, \quad 2), \\ (\text{availability}(s, 0.90)) \}$$

2.2 Service Requests

A service request aims at specifying constraints in order to restrict the set of available Web services to the set of desired Web services. We first describe the description syntax of requests and afterwards investigate in detail how constraints on functional and non-functional properties can be expressed. Then the semantics of requests is introduced.

In the following, we denote a service request by $\mathcal{R} = (I_{\mathcal{R}}, O_{\mathcal{R}}, \phi_{\mathcal{R}}, \psi_{\mathcal{R}}, N_{\mathcal{R}})$. We apply a formalism to requests that differs from that of service descriptions as the request models sets of services with sets of desired property values (see the right side of Figure 1). Such intention is analogous to database queries and we claim that this approach is more intuitive to use than applying the same formalism to model both offer and request, however with dissimilar interpretations.

Request Description Syntax

We aim at requests that constrain properties of services in a unifying way. That is, can comprise constraints on the functionality and NFPs as well logical combinations of different types of constraints. Besides requesting for inclusions and exclusions of desired property values, requests also allow for the specification of certain combinations of desired property values. For example, a user might accept a longer delivery time if the service offers credit card payment.

A request always refers to a desired service s of type `Service` (defined in a service model ontology like WSMO-Lite [21]) in order to refer to its properties and specify constraints on their values by expressions of the form `hasProperty(s, propertyValueSet)`. By this, the request description structure reflects the formal service model from the previous section. The following example sketches a request $\mathcal{R} = (I_{\mathcal{R}}, O_{\mathcal{R}}, \phi_{\mathcal{R}}, \psi_{\mathcal{R}}, N_{\mathcal{R}})$ for a book selling service.

$$\begin{aligned}
 \mathcal{R} &\equiv \text{Service}(s) \wedge \\
 &\wedge \text{hasInputs}(s, \text{isbn} \wedge \text{id} \wedge \neg \text{bday}) \\
 &\wedge \text{hasOutputs}(s, \text{book} \wedge \text{inv}) \\
 &\wedge \text{hasPrecondition}(s, \dots \text{ISBN}(\text{isbn}) \wedge \text{Birthday}(\text{bday}) \wedge \text{hasRewards}(\text{user}, r_{pre})) \\
 &\wedge \text{hasEffect}(s, \text{Book}(\text{book}) \wedge \text{hasISBN}(\text{book}, \text{isbn}) \wedge \text{Invoice}(\text{inv}) \wedge \\
 &\text{hasRewards}(\text{user}, r_{post}) \wedge \text{lessThan}(r_{pre}, r_{post})) \\
 &\wedge \text{hasPrice}(s, p) \wedge \text{hasDeliveryTime}(s, dt) \wedge \text{lessThan}(dt, 7) \wedge \\
 &(\text{acceptCreditCard}(s, \text{true}) \vee \text{acceptCreditCard}(s, \text{false}) \wedge \text{lessThan}(dt, 3))
 \end{aligned}$$

Constraints on functional properties are expressed by logic expressions. This example states that the desired sets of inputs must accept an ISBN and must not require any date of birth information. Furthermore, the desired service must return a book that is identified by the

given ISBN as well as an invoice. The service must provide a reward program and the user's number of rewards has to be increased after the purchase.

Constraints on NFPs. Non-functional requirements, denoted by $N_{\mathcal{R}}$, constrain values of NFPs such that the request again describes a set of desired values. Analogously to service descriptions, each desired service can be described by a finite set of property instances Q in a request. A property instance $q \in Q$ of a property type $t(q) \in \wp$ can be restricted to a set of desired values $V_{R,q} \subseteq V_{t(q)}$.

As an example, the maximum delivery time of 7 days of the desired service s can be requested by stating

$$\text{hasDeliveryTime}(s, dt) \wedge \text{lessThan}(dt, 7).$$

Constraints on Functionality. Service descriptions D may contain a description of the sets of inputs I , set of outputs O , precondition ϕ and effect ψ . A request \mathcal{R} describes a set $I_{\mathcal{R}}$ of desired sets of inputs and a set $O_{\mathcal{R}}$ of desired sets of outputs by logic formulas. The description also may include required or exclude unacceptable inputs or outputs of a desired service as the above example request shows.

In turn, it allows precise description of all possible matches as we consider the equality match solely. Prior approaches like [19] classified different interpretations into types of matches. Since these matching types are implemented by a discovery algorithm, the interpretation of the request is not as clear to the user as first order logic expressions are. Furthermore and as already mentioned above, these approaches cannot guarantee that discovered services fulfill all requested properties if all kind of intersections are considered as matches.

The request analogously describes a set $\phi_{\mathcal{R}}$ of precondition descriptions and a set $\psi_{\mathcal{R}}$ of effect descriptions. Using formulas expressed in a logic like first order logic to specify the set of requested preconditions and effects not only allows for precisely expressing which conditions must hold in the pre- and post-state, but also for excluding services with undesired conditions. For example, the negation of the existence of a `bday` prevents matching Web services that require a user registration for the order of the specified book. Consequently, the query in the above example request prevents Web service requiring a date of birth information being a match for the request.

The precondition ϕ and effect ψ of a service offer D describe the knowledge before and after service execution, respectively. The collection of facts of on ontology based knowledge base is often called A-Box. That is, ϕ and ψ are interpreted as A-Box statements of the knowledge base. Whereas the requests $\phi_{\mathcal{R}}$ and $\psi_{\mathcal{R}}$ for preconditions and effects are interpreted as queries to restrict the set of available services to those that have required knowledge in their A-Boxes described by ϕ and ψ , respectively.

Semantics of Service Request

The semantics \mathfrak{S} of a request maps a set of property-values sets into the formal model that is described by sets of desired property-value sets. Translated into the formal model, a request is a set of sets of property instances $q \in Q$ of type $t(q)$, which is assigned to a value $v \in V_{R,q}$ that is member of the desired value set $V_{R,q}$.

Let $q = (p, V_R)$ denote a requested property instance of user's concern. $V_{R,q} \subseteq V_{t(q)}$ is the set of acceptable property values of the property $p = t(q) \in \mathcal{P}$. Then, the interpretation

$$\mathfrak{S}: \left\{ \mathbb{Q}: \mathbb{Q} \subseteq \bigcup_{q \in Q} \{q\} \times 2^{V_{R,q}} \right\} \rightarrow \left\{ \mathbb{Q}: \mathbb{Q} \subseteq \bigcup_{q \in Q} \{q\} \times V_{R,q} \right\}$$

of a property request is $(p, V_{R,q})^{\mathfrak{S}} = \{(p, v) | v \in V_{R,q}\}$, which is the set of property-value pairs that is constructed by considering each value $v \in V_{R,q}$ that is member of the set of acceptable values individually.

This interpretation provides us means to formalize properties of services. The desired value set of the functionality property is discussed now.

The request \mathcal{R} embraces constraints on the functional properties by describing desired sets $I_{\mathcal{R}}$, $O_{\mathcal{R}}$, $\phi_{\mathcal{R}}$ and $\psi_{\mathcal{R}}$ of input sets, output sets, preconditions and effects, respectively. Note, that a simpler approach for matching functionality on the basis of explicit functional classification has been provided in the SOA4All deliverable D5.3.1. The desired functionality $(I_{\mathcal{R}}, O_{\mathcal{R}}, \phi_{\mathcal{R}}, \psi_{\mathcal{R}})$ is translated into a set \mathcal{L} of LTS's, which formally models a set of desired service executions that fulfill the requirements in \mathcal{R} .

An LTS $L_R \in \mathcal{L}$ models a service execution in terms of the formal model depicted in Figure 3. $L_R = (S_R, W_R, \rightarrow_R)$ models the functionality of one particular service configuration $R \in \mathcal{R}$ and is defined as follows.

$$\begin{aligned} S_R &= \{s_i, s_w, s_o, s_e\} \\ W_R &= \{R_{in}, R, R_{out}\} \\ \rightarrow_R &= \{(s_i, R_{in}, s_w), (s_w, R, s_o), (s_o, R_{out}, s_e)\} \end{aligned}$$

A request describes requirements on the precondition and effect by $\phi_{\mathcal{R}}$ and $\psi_{\mathcal{R}}$. The states in L_R that are described by the knowledge that holds at that time consequently have to fulfill the requirements and constraints of $\phi_{\mathcal{R}}$ and $\psi_{\mathcal{R}}$. Each desired service execution that fulfills the request \mathcal{R} is modeled by one LTS in the set \mathcal{L} . The semantics maps the requested functionality $(I_{\mathcal{R}}, O_{\mathcal{R}}, \phi_{\mathcal{R}}, \psi_{\mathcal{R}})$ into (i) a set of transitions modeling the input operations described by $I_{\mathcal{R}}$, (ii) a set of transitions modeling the output operations described by $O_{\mathcal{R}}$, (iii) a set of pre-states s_w described by $\phi_{\mathcal{R}}$, and (iv) a set of post-states s_o described by $\psi_{\mathcal{R}}$.

Then, an LTS is constructed for each combination of input transition, output transition, pre-, and post-state. In each LTS, the state s_i is implicitly derived from the description of s_w and

the input operation that is that $(s_i, R_{in}, s_w) \in \rightarrow_R$ is a consistent transition in L_R and is described by $I_{\mathcal{R}}$. The knowledge modeled by s_i can be derived from the knowledge in s_w except for the knowledge derived from the input operation. Similarly, the end state s_e can be derived from s_o as the output operation does not change the knowledge. The remaining transition $(s_w, R, s_o) \in \rightarrow_R$ abstracts from the individual local operations and is implicitly described by the states before and after service execution.

Consequently, the set of possible values of the property instance that models the functionality is the set of all labeled transition systems.

2.3 Matchmaking

The task of service discovery is to find matches between service offers and requests. After we introduced both description formalisms and their translations into a common model, we define in this section a match and how it can be determined.

A match is given if a service \square meets all requirements of a request \mathcal{R} , i.e., the property values of the service are in the set of desired values of the request of all requested property instances. Within the two formal models of service descriptions and requests, a match is computed by checking whether the service description is contained in the set of desired service descriptions of the request. Since we consistently modeled each property of a request as a set of desired values, the matchmaker checks for a containment relation between service description and request. This applies to all property-value pairs as well as to the LTS-based formal interpretation of offered and requested functionalities.

Matching Properties

We first investigate how to match properties in general. The subsequent section discusses functionalities in detail.

A service description was interpreted as a set

$$Q_W^{\mathfrak{S}} \subseteq \bigcup_{P \in \wp} P \times V_P$$

of property instances comprising functional and non-functional properties in a unifying way. Within the formal model of service descriptions, the property instances $q \in Q$ model the assignment of a property $t(q)$ to a value $v_{t(q)} \in V_{t(q)}$. Constraints on properties Q_R in a request are formalized as a set of values assigned to a property.

$$Q_R \subseteq \bigcup_{P \in \wp} P \times 2^{V_P}$$

$$Q_R^{\mathfrak{S}} \subseteq \{Q: Q \subseteq\} \bigcup_{P \in \wp} P \times V_P$$

As can be easily derived from these three equations, a set of property instances $Q_W^{\mathfrak{S}}$ of a Web service description matches the requested properties $Q_R^{\mathfrak{S}}$ if and only if $Q_W^{\mathfrak{S}} \in Q_R^{\mathfrak{S}}$. I.e., there exists a set $Q'_R \in Q_R^{\mathfrak{S}}$ of property instances that equals the set $Q_W^{\mathfrak{S}}$. Then there exists $q_R \in Q'_R$ for each property instance $q_w \in Q_W^{\mathfrak{S}}$ with $q_R = q_w$ and this in turn means that resp. types $t(q_R) = t(q_w)$ and values $v_{q_R} = v_{q_w}$ of both property instances are equal per definition.

Matching Functionalities

Let L_D denote the interpretation of the functionality description (I, O, ϕ, ψ) of service w , i.e., L_D is the value of the functionality property of w . Also, let the set \mathcal{L} of LTS' denote the interpretation of the requested functionality $(I_R, O_R, \phi_R, \psi_R)$. Analogously, \mathcal{L} corresponds to the set of desired values of the service functionality property. Then, a service functionality matches a requested functionality if and only if $L_D \in \mathcal{L}$. That is, $\exists L_R \in \mathcal{L}: L_R \equiv L_D$. In order to define a match, we now define the equivalence between labeled transition systems.

Definition: Equivalence of States. Two states s_i and s_j are equivalent if the knowledge $s_i^{\mathfrak{S}}$ and $s_j^{\mathfrak{S}}$ that holds in the respective states is equivalent.

$$s_i \equiv s_j \Leftrightarrow s_i^{\mathfrak{S}} \models s_j^{\mathfrak{S}} \wedge s_j^{\mathfrak{S}} \models s_i^{\mathfrak{S}}$$

Definition: Equivalence of Transition Labels. Two transition labels w_i and w_j are equivalent if they both either simulate an input or an output operation and the set of variables that are modeled by the transitions are equivalent.

$$w_i \equiv w_j \Leftrightarrow w_i^{\mathfrak{S}} \models w_j^{\mathfrak{S}} \wedge w_j^{\mathfrak{S}} \models w_i^{\mathfrak{S}}$$

If both labels model local operations, then they are also considered to be equivalent. Otherwise, two labels are not equivalent.

Definition: Equivalence of Labeled Transitions. Two labeled transitions are equivalent if both start and both end states as well as the transition labels are equivalent.

$$(s_i, w_i, s_j) \equiv (s_k, w_k, s_l) \Leftrightarrow s_i \equiv s_k \wedge s_j \equiv s_l \wedge w_i \equiv w_k$$

Now we are able to define the equivalence between the L_D and $L_R \in \mathcal{L}$, which corresponds to a match between the offered and the requested functionality. Therefore, let the superscripts D and R link the symbols to the corresponding LTS' L^D and L^R . Two LTS' L^D and L^R are equivalent if and only if the following conditions hold: (i) Equivalence of the corresponding four states $s_i^D \equiv s_i^R$, $s_w^D \equiv s_w^R$, $s_o^D \equiv s_o^R$, $s_e^D \equiv s_e^R$, and (ii) Equivalence of transitions such that

$$\begin{aligned} (s_i^D, w_{in}, s_w^D) &\equiv (s_i^R, R_{in}, s_w^R) \\ (s_w^D, w, s_o^D) &\equiv (s_w^R, R, s_o^R) \\ (s_o^D, w_{out}, s_e^D) &\equiv (s_o^R, R_{out}, s_e^R) \end{aligned}$$

Now we further show how a match within the formal model corresponds to a match in terms of the service and request descriptions. Basically, the containment relation $L^D \in \mathcal{L}$ corresponds to the query answering task of a reasoner that computes the match. Therefore, the reasoner identifies a model for the request query by binding variables of the request to individuals modeled in the A-Box of the knowledge base that represents the service

description. To show the equivalence of the match in the formal model and the match identified by a reasoner, both directions of the implication between them are discussed. For simplicity, we now only consider knowledge bases that model the pre-states before execution. The post-states are treated analogously.

Assuming that there is a match in the formal model, i.e., $L^D \in \mathcal{L}$, then it holds that $s_w^D \equiv s_w^R$ as defined above. Consequently, the information content $KB(s_w^D)$ and $KB(s_w^R)$ that model the pre-states s_w^D and s_w^R is equivalent, i.e., it holds that $KB(s_w^D) \models KB(s_w^R) \wedge KB(s_w^R) \models KB(s_w^D)$. Obviously, then there exists a variable binding to answer the query against the knowledge base $KB(s_w^D)$ that models the service description.

In reverse, if there exists a variable binding to answer the query, then the knowledge base $KB(s_w^D)$ is a model of the request and contains at least the information that has to be satisfied to fulfill the request. Due to the definition of the request semantics, there also exists an LTS in \mathcal{L} which contains a pre-state s_w^R that is modeled by a knowledge base $KB(s_w^R)$ that is equivalent to $KB(s_w^D)$. The reason why there exists such LTS is that the model \mathcal{L} of a request contains all the LTS' with all possible states and respective knowledge bases that are model of the request. Hence, if there exists a variable binding to answer the query, then $L^D \in \mathcal{L}$.

The same argumentation applies to the remaining states of the LTS'. It remains to show that matching inputs and outputs is also guaranteed. As current reasoners only model static knowledge, we model the additional information that input and output operations provide in the knowledge bases $KB(s_w^D)$ and $KB(s_e^D)$ that model the respective states subsequent to the operation. In the pre-state knowledge base $KB(s_w^D)$ we therefore explicitly add the information about inputs and their values. In the end state knowledge based $KB(s_e^D)$ after the output operation, we only mark the instances that are emitted. Because the values of the outputs were already computed by the sequence of local operations before the post-state s_o , no further changes are performed by the output operation. By modeling inputs and outputs within the knowledge bases that model the states of the transition systems, the presented matchmaking already covers matching inputs and outputs. As a consequence, the match between functionalities can be computed by using query answering (i.e., instance checking) as the reasoning task.

Matchmaking, as introduced above, detects the match between the example service description and the request described in previous sections. For instance, the NFPs N match the NFRs $N_{\mathcal{R}}$, because the offered service is that fast that it delivers in less than three days although it does not accept a credit card. Furthermore, the precondition also matches the request as it only requires that the service identifies books by ISBN and the effect as it delivers the proper book with an invoice and the user receives reward points. However, a book selling service that identifies book by author name and book title for example would not match the example request \mathcal{R} .

3. Implementation and Evaluation

In this section, we introduce the implementation of the presented discovery approach by explaining the creation of the reasoner's knowledge base, which is derived from the services descriptions retrieved from a repository of semantic service descriptions. Within the project, we use the WSMML reasoner framework (See <http://tools.sti-innsbruck.at/wsmml2reasoner>, [28], [27] for reasoning and installation details) and the WSMML-DL language dialect [29] for Web service descriptions within project.

We present the implementation of the matchmaking algorithm within the SOA4All discovery component and provide performance results at the end. The semantic discovery is developed as Web service so that it can be used by other SOA4All components with standard Web protocols. The semantic discovery component offers a range of methods for various purposes related to the discovery of Web services.

Semantic Web service descriptions use the WSMO-Lite service ontology [36, 21] in conjunction with the Procedure Oriented Service Model (POSM)¹. POSM is a service model that represents the structure of a service description containing concepts like Service, Operation, etc. POSM refers to the concepts provided by WSMO-Lite. By this, POSM allows us to describe services semantically using WSMO-Lite Annotations without annotating a WSDL document. In order to bridge the gap between the formal models on a theoretical level that were introduced above and the concrete implementation, we provide the mapping between the two levels as depicted in Table 1 (posm and wl abbreviate the namespaces of POSM and WSMO-Lite respectively).

Table 1: Mapping between theory and implementation of service descriptions.

Formal model	Elements of Semantic Service Descriptions
I	posm:hasInputMessage and posm:Message
O	posm:hasOutputMessage and posm:Message
ϕ	posm:hasCondition and wl:Condition
ψ	posm:hasEffect and wl:Effect
N	wl:NonFunctionalParameter

3.1 Integration with SOA4All Service Repository

Semantic service descriptions are stored in the SOA4All service repository (available at <http://iserve.kmi.open.ac.uk/>). Semantic service descriptions can be created by means of the SOA4All Studio. Therefore, given OWL-S Profile, SAWSDL, and WSDL service descriptions can be imported. Semantic service descriptions within the SOA4All service repository are modeled using the minimal service model. (For more information about the current version of

¹ <http://www.wsmo.org/ns/posm>

the minimal service model we have to refer to the documentation provided at the following Web page (http://iserve.kmi.open.ac.uk/wiki/index.php/Simple_vocabulary). As the discovery component relies on the POSM developed by Work Package 3, the semantic service descriptions derived from the SOA4All service repository are translated into the proper model. The translation is provided by the WSL4J API, which was developed in collaboration with Work Package 3 (see appendix to D5.4.2).

Service descriptions are retrieved from the repository in form of ontologies serialized in RDF. We use the RESTful service interface of the service description repository in order to obtain a list of currently registered Web service descriptions. The resource 'services' located at <http://iserve.kmi.open.ac.uk/data/services/> provides a list of available service description within the repository. By integration of the latest semantic space developments, a notification mechanism (for new service descriptions) will shortly be provided by the iServe repository. It will enable us to continuously run discovery upon the most current set of available service descriptions. Each service description retrieved from the repository is parsed by WSL4J API, which then simply allows us to serialize the parsed service description into a RDF ontology using the POSM service model.

3.2 Integration with WSML2Reasoner

The service discovery component uses the reasoner to provide intelligent matching of requests and services. Different reasoning functionalities, such as subsumption reasoning, satisfiability checking, instance retrieval, etc. are required by the discovery component depending on the model used to specify services and user requests. To achieve one of the functional requirement mentioned before, namely accuracy, the discovery solution must integrate and use the reasoning support. A clear defined interaction that results in a well defined interface between the two components is required. The translated service description ontologies retrieved from the service repository are imported into a WSML reasoner instance using the reasoner's RDF parser functionality.

Each service description retrieved from the SOA4All repository was translated into a POSM service description in form of an ontology. We refer to the documentation of POSM for details on the modeling of service descriptions within an ontology serialized in RDF. These POSM ontologies are then imported into the reasoner and build the knowledge base on which reasoning tasks can be performed later. Within the reasoner's knowledge base, properties of a service description *D* are modeled as properties of the service instance within the WSML representation of the service description ontology.

3.3 Service Templates

A simple user interface that is part of the SOA4All Studio allows to formulate and submit service requests that may contains the combination of constraints on the desired NFPs, inputs, outputs, precondition, and effect. The request information is encapsulated into a so-called service template² object. A service template is defined by an RDFS ontology (see [30],

² <http://www.wsmo.org/ns/service-template>

though the definition has evolved) and contains the elements inputs, outputs, requirements, and preferences. These elements of a service template are defined by the respective properties as shown in Table 2 taken from [30]. Obviously, inputs $I_{\mathcal{R}}$ and outputs $O_{\mathcal{R}}$ of a service request are captured by the respective elements within a service template. Desired precondition $\phi_{\mathcal{R}}$ and effects $\psi_{\mathcal{R}}$ as well as non-functional requirements $N_{\mathcal{R}}$ are encapsulated to the requirements field of service templates by using multiple instances of the `hasRequirement` property. Preconditions and effects are logical expressions and are encoded as string representation of WSMML axioms. The preferences are used for service ranking. The functional category is used for classification based discovery (refer to SOA4All deliverable 5.3.1).

Table 2: RDF Schema for service template.

<code>ServiceTemplate</code>	<code>rdf:type</code>	<code>rdfs:Class</code>	<code>.</code>
<code>hasFunctionalCategory</code>	<code>rdf:type</code>	<code>rdf:Property</code>	<code>.</code>
<code>hasInput</code>	<code>rdf:type</code>	<code>rdf:Property</code>	<code>.</code>
<code>hasOutput</code>	<code>rdf:type</code>	<code>rdf:Property</code>	<code>.</code>
<code>hasPreference</code>	<code>rdf:type</code>	<code>rdf:Property</code>	<code>.</code>
<code>hasRequirement</code>	<code>rdf:type</code>	<code>rdf:Property</code>	<code>.</code>

The discovery engine translates the user request into proper WSMML syntax and sends this WSMML query to the reasoner. The reasoner executes the query upon its knowledge base, which models all service descriptions. In order to answer the query, the reasoner determines for each service w modeled in the knowledge base, whether the descriptions of inputs I , outputs O , precondition ϕ , effect ψ , and N is a model for the requested combination of inputs $I_{\mathcal{R}}$, outputs $O_{\mathcal{R}}$, precondition $\phi_{\mathcal{R}}$, effect $\psi_{\mathcal{R}}$, and NFRs $N_{\mathcal{R}}$, respectively. I.e., the reasoner checks, for example, whether the requested preconditions $\phi_{\mathcal{R}}$ is fulfilled by the facts in the A-Box that were introduced by the precondition ϕ of the service w . Therefore, the reasoner checks all potential variable mappings between query and service description including input and output variables. If there is a variable mapping that fulfills the conditions of precondition and effect and also the requested sets of inputs and outputs cohere to this mapping, a match is identified. Finally, a list of matching services from a repository of generated Web service description is retrieved from the discovery engine and displayed to the user.

```

@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix wsml: <http://www.wsmo.org/wsml/wsml-syntax#>.
@prefix wsl: <http://www.wsmo.org/ns/wsmo-lite#>.
@prefix st: <http://www.wsmo.org/ns/service-template/0.1#>.
@prefix sf: <http://www.service-
finder.eu/ontologies/ServiceCategories#>.
@prefix sr: <http://seekda.com/ontologies/RankingOntology#>.
@prefix pref: <http://www.wsmo.org/ontologies/nfp/preferenceOntology#>.
@prefix bs: <http://www.example.com/bookselling#>.
@prefix ex: <http://localhost:8081/DisCloud/serviceTemplates/BSS#>.

```

```
ex:stBSS a st:ServiceTemplate;
    st:hasFunctionalCategory sf:BookSellingService;
    st:hasInput bs:isbn, bs:id;
    ex:hasOutput bs:book, bs:inv;
    st:hasRequirement [rdf:type wsl:Condition; rdf:value "?isbn
memberOf _\"http://www.example.com/bookselling#ISBN\" ?bday memberOf
_\"http://www.example.com/bookselling#Birthday\" and
?user[_\"http://www.example.com/bookselling#hasRewards\" hasValue
?rpre]"^^wsm:Literal].

    st:hasRequirement [rdf:type wsl:Effect; rdf:value
"?book[_\"http://www.example.com/bookselling#hasISBN\" hasValue ?isbn]
memberOf _\"http://www.example.com/bookselling#Book\" and ?inv memberOf
memberOf _\"http://www.example.com/bookselling#Invoice\"
?user[_\"http://www.example.com/bookselling#hasRewards\" hasValue
?rpost] and ?rpre[_\"http://www.example.com/bookselling#lessThan\"
?rpost]"^^wsm:Literal].

st:hasRequirement rdf:value
"?s[_\"http://www.example.com/bookselling#hasDeliveryTime\" hasValue
?dt] and ?dt[_\"http://www.example.com/bookselling#lessThan\"
_int(\\"7\")]"^^wsm:Literal].

st:hasRequirement rdf:value
"?s[_\"http://www.example.com/bookselling#acceptCreditCard\" hasValue
_boolean(\\"true\")]"^^wsm:Literal].

...
```

Table 3: An Example Service Template

In order to reason about updates caused by the execution of a Web service, the discovery engine manages the differentiation between individuals that change during execution. As per convention, changing instances have different symbols indicated by the suffixes `pre` and `post` in above sections. This allows to reason about changes although reasoners are not capable to model dynamics in knowledge bases. We therefore introduced the association between the changed instances, which is managed by the discovery engine.

3.4 Performance Results

In the following we performed some tests on the implementation of the semantic service discovery. As the presented formal approach already guarantees the applicability of discovered services for the given problem described by the request, we tested the performance of the discovery implementation. That is, we measured the time between submitting a request and the retrieval of the discovery results. Typically, semantic matchmaking highly depends on the performance of the reasoner, which depends on the size of the knowledge base among other and was already subject to performance evaluation in [28].

Given that service descriptions crawled by seekda mainly represent the information derived from WSDL service descriptions, we decided to synthesize rich semantic service descriptions in a fairly large scale to perform our measurements. Also the SOA4All service description repository did not provide us a large number of semantic service descriptions at that stage. Therefore, we created a set of randomly generated service descriptions with varying size ranging from 5,000 to 30,000 descriptions, which is approximately the number of currently available Web service according to seekda (Trends available at http://webservices.seekda.com/about/web_services). We used the Semantic Web for Research Community (SWRC) ontology [12] as domain knowledge to model service descriptions. It provides classes and properties to express individual types and conditions.

The synthetic semantic service descriptions provide one operation each. An operation expects 1 to 8 inputs and returns 1 to 8 outputs. The precise numbers were randomly chosen. Note that we refer to message parts of the input or output message of the service operation. Each input and output is assigned to a random concept of the SWRC ontology within the description of precondition and effect respectively. Then, we further randomly generated up to 8 further variables for each precondition and each effect. Each variable or input/output parameter can have an association with another one within the description of preconditions and effects.

Further, we generated up to 6 non-functional properties per service. Non-functional properties are modeled by an instance of an ontology concept which is associated to a random precise value within the range of property values.

We measured the mean query answering time of the reasoner on a quad core Xeon CPU (2.33GHz) powered machine with 2GB RAM. Queries of three different sizes (small, medium, large) were sent to the reasoner's knowledge base.

Table 4: Query sizes tested in the experiment.

		Small	Medium	Large
Query size				
	Variables	6	9	12
	Relations	9	12	15
	NFRs	2	4	6

Small, medium, and large conjunctive queries with various numbers of variables (including inputs, and outputs) and relationships among them within the desired precondition and effect description were tested in this experiment. Table 1 lists the precise number of terms of the individual queries. As depicted in Figure 4, the time to answer these queries range from 2.8s, 4.2s, 5.0s with 5,000 service descriptions to 17s, 23s, 33s with 30,000 descriptions for small, medium, large sized queries, resp.

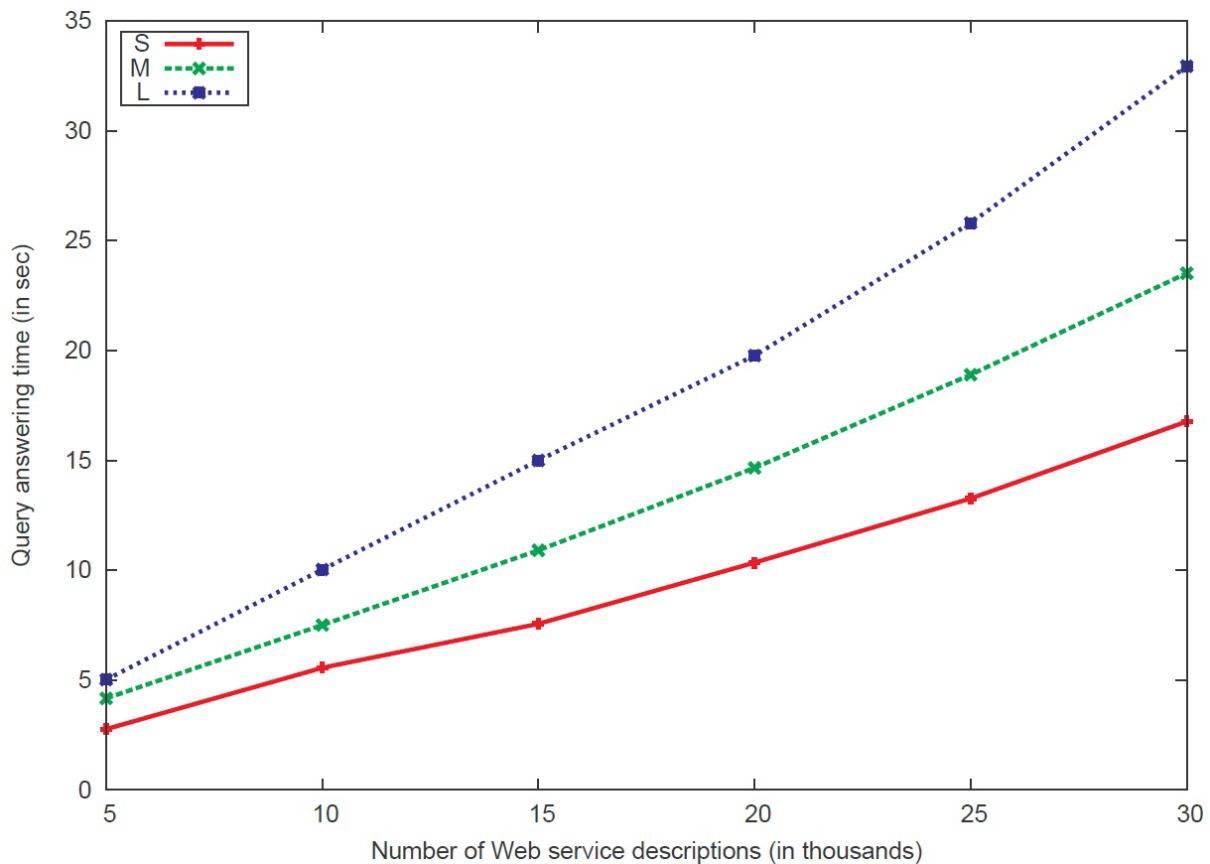


Figure 5: Mean query answering time against increasing number of Web service descriptions for three query sizes.

Note, the purpose of Figure 4 is to show the feasibility of the presented discovery approach. It is clear, that query answering time measure highly depends on size and structure of the used domain ontologies, size and complexity of the query and service descriptions. Nevertheless, these results can be significantly improved by the introduction of indexing structures, increasing the computational power, and distributing the reasoning process [13] among various options.

Finally, we want to provide some references to related semantic discovery approaches which provide some performance results. Such results comprise time measures for the computation of the desired service functionalities. We want to emphasize that results, esp. the answering time, are not comparable at all. Among many different reasons, the usage of different hardware and the varying complexity of semantic service descriptions, requests, and domain ontologies lead to incomparability. However we want to provide a hint on the very rough ranges of query answering times of other prominent Web service discovery approaches.

[35] provides experimental results from a test bed related to the European INFRAWEBs project. The authors conclude that matching a WSML goal against semantic service descriptions scales up to 1,000 service descriptions with a result within 5 seconds.

Stollberg et al. reports mean query answering times of 72 seconds against 2K Web service descriptions when using a naïve approach that they reduced to 0.3 seconds by using semantic discovery caching (SDC) [33]. As we will further discuss in the related work section of this document, the SDC technique relies on a pre-computed caching structure within a global hierarchy of goals. The query is answered by a simple lookup which leads to the fairly fast response time.

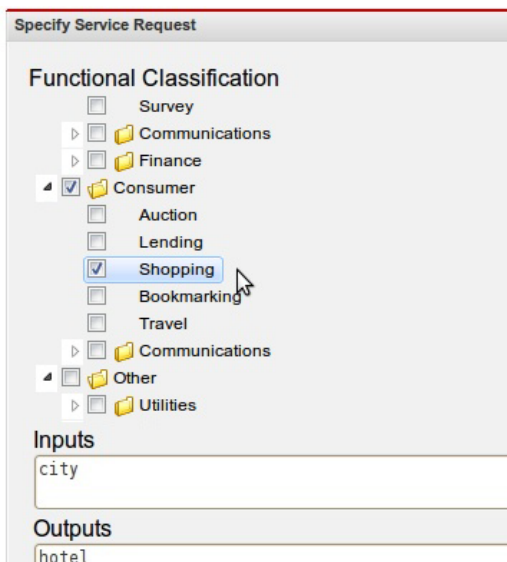
Another example for an approach that relies on a pre-computation phase is provided in [34]. An experiment shown in 34] with 2K Web service descriptions measured a classification phase with more than 160 seconds and query answering time of about 20ms.

Figure 6: Discovery user interface with request specification (left) and desired services and operations (right).

3.5 User Interface

The semantic Web service discovery component of SOA4All comes with a Web based graphical user interface (see Figure 5). It is implemented with the Google Web toolkit and an integrated part of the SOA4All Studio. The interface allows users to enter a request for service functionalities by specifying:

- The functional classification of the service. Classification based discovery was already discussed in [31]. It allows selecting a set of classes displayed to the user as depicted in Figure 5. Services descriptions contain an assignment to a subset of available classes.
- The desired functional and non-functional properties of a request \mathcal{R} . It includes the fields for inputs, outputs, precondition, effect, and non-functional requirements. In order to support users in expressing the conditions, elements of registered (domain) ontologies are automatically suggested for completion as depicted in Figure 8.
- Preferences that are used to rank the set of results according to what the user describes in this field. Preferences are discussed in the ranking deliverable [32].



Specify Service Request

Functional Classification

- Survey
- Communications
- Finance
- Consumer
 - Auction
 - Lending
 - Shopping
 - Bookmarking
 - Travel
- Communications
- Other
 - Utilities

Inputs

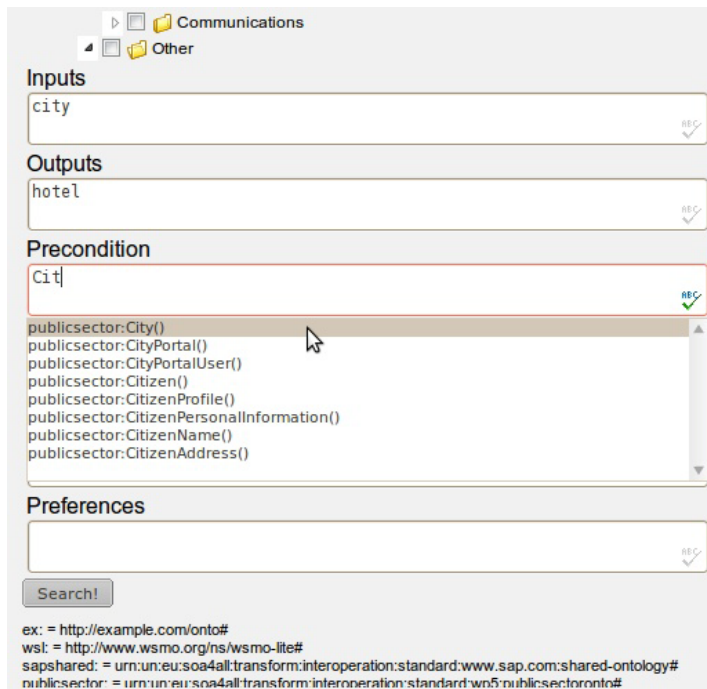
city

Outputs

hotel

Figure 7: Specification of desired service classification.

After submitting the request to the discovery service, the user interfaces retrieves a set of services and operations that fulfill the request. If preferences were specified, then the displayed result set is an ordered list with descending adherence to the preferences. In addition, some further information about a selected service and operation are display on the right side of the Web interface, which is omitted in Figure 6 due to the space constraints.



The screenshot shows a web-based configuration interface for a service. It is organized into several sections:

- Inputs:** A text field containing the value "city".
- Outputs:** A text field containing the value "hotel".
- Precondition:** A text field containing "Cit". Below this is a scrollable list of ontology classes: `publicsector:City()`, `publicsector:CityPortal()`, `publicsector:CityPortalUser()`, `publicsector:Citizen()`, `publicsector:CitizenProfile()`, `publicsector:CitizenPersonalInformation()`, `publicsector:CitizenName()`, and `publicsector:CitizenAddress()`. A mouse cursor is positioned over the first item.
- Preferences:** An empty text field.

At the bottom, there is a "Search!" button and a list of ontology URIs:

- ex: = `http://example.com/onto#`
- wst: = `http://www.wsmo.org/ns/wsmo-lite#`
- sapshared: = `urn:un:eu:soa4all:transform:interoperation:standard:www.sap.com:shared-ontology#`
- publicsector: = `urn:un:eu:soa4all:transform:interoperation:standard:wn5:publicsector:onto#`

Figure 8: Specification of desired functional and non-functional properties.

If the requesting user is logged in into the SOA4All Studio with an OpenID account, the discovered operations can be added to the personalized list of favorite service operations (Figure 5). The favorites list, which is a module within the SOA4All Studio, is for example also used in the SOA4All Process Editor and allows to bind process activities to user's favorite service operations.

4. Use of Discovery within SOA4All

Service ranking and selection component (WP5) will order the relevant services that were identified by the service discovery solution and finally select the best fit. From an architectural perspective, service discovery is expected to return a set of service descriptions that belong to the services matching the request.

The service discovery solution should support the service construction (WP6) in dynamic and adaptive composition and reconfiguration of constructed services in reaction to environmental changes. Parametric templates that represent service compositions as well as the composition optimizer require as bidding time concrete services that will be identified by the service discovery solution.

All use cases potentially use discovery, especially: (WP7) in locating relevant services within an enterprise and within e-Government scenarios, the longest-standing use of discovery in use case demonstrators; (WP8) in locating, for example, third party services (SMS, etc.) in geographical regions where Ribbit does not provide these. WP9 has so far concentrated more on potential uses of ranking since the scenarios are based on hand-built compositions of tightly-controlled services.

5. Related Work

Service-oriented computing is an interdisciplinary paradigm that revolutionizes distributed software development. The success encountered by the Web has shown that tightly coupled software systems are only good for niche markets, whereas loosely coupled software systems can be more flexible, more adaptive and often more appropriate in practice. Applications that adopt service-oriented architectures (SOA) can evolve during their lifespan and adapt to changing or unpredictable environments more easily. When properly implemented, services can be discovered and invoked dynamically using non-proprietary mechanisms, while each service can still be implemented in a black-box manner. Despite these promises, service integrators, developers, and providers need to create methods, tools, and techniques to support cost-effective development, as well as the use of dependable services and service-oriented applications.

Web service discovery deals with finding appropriate Web services for a task at hand and is one of the central components needed for developing service-oriented applications. It is the task of identifying service descriptions from a pool of descriptions that fulfill the request and is realized by matching service descriptions against a service request.

Universal Description, Discovery and Integration (UDDI) [1] was the first attempt to provide users with a system for finding Web services. However, UDDI discovery requires a lot of manual effort for finding the right services, mainly due to its lack of support for use of heterogeneous terminologies and the lack of formal description of the functionality of Web services in its underlying model. For example, UDDI is not able to deal with synonyms or relations between terms that describe services.

Since the advent of the Semantic Web, many semantic Web service discovery approaches have been proposed to deal with heterogeneity in the terminology used in different services. Some of them consider the functionality description of services, which allows for automated tasks like service composition. Automation requires functionality-based discovery, since simple matching of input and output types still requires a lot of human effort to figure out whether the matching Web services offer the desired functionality.

Functionality based semantic service discovery allows for automated tasks like service composition. The common model to describe the functionality of a software artifact is represented by inputs, outputs, preconditions, and effects, or shortly denoted by (I, O, P, E) (analogously (I, O, ϕ, ψ)). Inputs denote the set of user-provided message parts at Web service invocation. Outputs describe the set of values returned to the user after service execution. Preconditions and effects describe the information states of the world before and after service execution, resp., by logical formulas. Semantic Web service discovery approaches compute the match between a service offer that describes the functionality of the service and a service request.

OWL-S Matchmaker uses OWL-S profile for describing Web service offers as well as requests [2, 16]. Even though OWL-S Profile has elements for preconditions and effects, the

OWL-S matchmaker uses types of input and output parameters only. The approach presented in [3] models Web services as well as requests as description logic (DL) classes and bases the matchmaking on the intersection of service offer and request, which is computed by a DL reasoner. Such approaches fail to reason about the dynamics of Web services, since DL reasoners cannot reason about changing knowledge bases. The approach in [26] deals with variables, but is limited to Web services that do not change the world and, thus, can be described by a query. Efficient semantic discovery approaches that can deal with functionality of Web services are presented in [21]. Efficiency is achieved by pre-computing a classification of services in a hierarchy of goal templates. However, the requirement of such a classification hierarchy hinders the usability of creating service descriptions and requests since it is not feasible to maintain a global hierarchy in a decentralized and open environment of the Web. Furthermore, [21] do not support matching of inputs and outputs nor do they deal with the possible inconsistency between functional description of Web services and their classification.

The description of the functionality of a software by preconditions and effects was introduced by [14]. In contrast to description and discovery approaches in the field of software specification, the assumption of a closed world does not hold for Web services. The ability to model side effects to the world and the consideration of background knowledge thus were not considered. Zaremski and Wing consider different match types based on the implication relations between preconditions and postconditions of software library components and a query [15].

Martin et al. presents a discovery approach in [16] that is based on OWL-S and describes services functionalities semantically by inputs, outputs, preconditions, and effects. This approach interprets preconditions as constraints that need to be satisfied for the service requester only and effects as side effects of the service execution on the world. In our approach we model conditions that hold at the service provider side since those conditions can be evaluated during service invocation and execution time.

The semantic Web community with focus on languages provides description logic (DL) based description approaches [17,2,18]. The approach by Li et al. in [3] represents objects like inputs and outputs as concepts in description logics. This approach further combines the use of DL with DAML+OIL and DAML-S and defines different matching degrees. Service description and request are similarly structured comprising inputs, outputs, preconditions, and effects. Discovery, i.e., matchmaking is based on the intersection of service offer and request and is reduced to checking subsumption of input and output types. However, DL-based approaches fail to reason about the dynamics of Web services, since DL reasoners cannot reason about changing knowledge bases. Consequently, more recent research activities concentrate on more detailed formalisms, for instance the state-based perspective on Web services that is discussed below. These models allow modeling the dynamics of Web services.

The state-based service discovery approach [5,19] developed by Stollberg et al. uses a state-based formal model of service descriptions [20]. The functionality of a Web service is formally described by the set of possible Web service executions while each normal execution of a Web service is determined by its start and end state. The discovery algorithm relies on the assumption that the precondition ϕ logically implies the effect ψ of a Web service execution. Modeling a transition as a logical implication $\phi \Rightarrow \psi$ can be problematic, e.g., in case of a Web service that deletes a certain fact, the existence of the fact would imply non existence of the fact, e.g., a user subscription would imply that the user is not subscribed anymore.

In contrast to the state-based approaches, Hull et al. propose a matching technique for stateless Web services in [26] with the restriction to conjunctive queries, since the query containment problem is decidable for such queries. In our approach, we can deal with stateful services since our discovery approach is not based on query subsumption but on query answering.

Goal-driven approaches like [5,24,25] do not explicitly specify inputs as parts of the goal. However, a goal needs to be mapped to a request for finding appropriate Web services. In such a request, constraints on inputs can be useful, in particular if a user wishes to exclude a particular input parameter. In goal based approaches, goals are mapped to predefined goal templates that are used to find appropriate Web services. However, the usability of one global hierarchy of goal templates is hardly feasible in an open environment like the Web. One major difference between our approach and the goal based approaches is that we interpret inputs, outputs, preconditions, and effects of descriptions and requests differently, namely the former as a pair of states the latter as a pair of queries.

Non-functional properties were not used for Web service discovery so far. In OWL-S non-functional properties are considered as human-readable metadata, e.g., service name. WSML [11] does not include NFPs into the logical model. Consequently, no reasoning on them is possible. The WSMO specification defined NFPs, however there is so far no prominent implementation available that considers them, such as the Internet Reasoning Service [22]. O'Sullivan et al. [23] described a set of NFPs relevant for Web services and their modeling, which were formalized in a WSMO deliverable.

6. Conclusion and Outlook

In this deliverable we have introduced the latest work on discovery and its evaluation. We discussed some drawbacks of the existing approaches that hinder them to be accepted in practical settings. Then we presented in detail the theory and implementation of our approach. The theoretical part explains the meaning of service descriptions, service requests and matching between the two with example illustrations. The implementation part explains the programming interface provided the discovery component, the graphical user interface as well the details of the integration of the service discovery component with WSML ontology reasoners and iServe repository. We also provide results of the performance evaluation.

Plans for the rest of the project involve addressing scalability at a different level, based on the Discovery Cloud (DisCloud) service template repository documented in the SOA4All Deliverable D5.4.2 [32]. DisCloud brokers service templates long-term, rather than ad hoc, against matching services. Within the lifetime of a service template there are two kinds of application of the discovery component that represent the significant computational load and potential difficulty in applying semantic discovery, with the characteristics detailed in this deliverable, at the scale foreseen in the SOA4All vision.

In particular semantic discovery is applied at the following stages:

- When a new *service template* is uploaded to DisCloud, semantic discovery is used against every *service description* of a matching functional classification.
- When a new *service description* is uploaded to iServe, DisCloud will be notified, and every *service template* of a matching functional classification will be checked against this new service.

The insight to be pursued is that both of these problems can be reduced to a *map* over the other type of resource, followed by a simple *reduce*. The important consideration, as with any MapReduce problem, is locality of data, i.e. that the computation is reasonably well isolated from the communication of large amounts of data (i.e., instead a well-scaling problem will place a virtualised image of the computation to be apply on the node with the data on which it must operate).

SOA4All has been given, via an application to the OpenCirrus consortium, access to a large computing cluster with support for Hadoop (Yahoo!'s open source MapReduce implementation) and Eucalyptus, a 'private cloud' interface-compatible clone for the Amazon Web Services cloud.

One evaluation that will be carried out is whether the SOA4All semantic bus offers the best means to distribute service template descriptions, or whether this is best achieved with a file-oriented distribution as in the Hadoop file system, or a 'resource as bucket' model of the Amazon S3 storage cloud (as supported by Eucalyptus). Others will be how efficiently both of the above points of computation will scale using a Hadoop-based approach to distributing the problem.

References

1. UDDI, "UDDI Executive White Paper," UDDI.org, Tech. Rep., Nov. 2001. [Online]. Available: http://uddi.org/pubs/UDDI_Executive_White_Paper.pdf
2. K. Sycara, M. Paolucci, A. Ankolekar, and N. Srinivasan, "Automated Discovery, Interaction and Composition of Semantic Web Services," in Journal of Web Semantics, vol. 1, no. 1, Dec. 2003, pp. 27–46.
3. L. Li and I. Horrocks, "A Software Framework for Matchmaking Based on Semantic Web Technology," Int. J. Electron. Commerce, vol. 8, no. 4, pp. 39–60, 2004.
4. I. Constantinescu, W. Binder, and B. Faltings, "Flexible and Efficient Matchmaking and Ranking in Service Directories," in ICWS '05: Proceedings of the IEEE International Conference on Web Services. Washington, DC, USA: IEEE Computer Society, 2005, pp. 5–12.
5. M. Stollberg, M. Hepp, and J. Hoffmann, "A Caching Mechanism for Semantic Web Service Discovery," in The Semantic Web. 6th Int. Semantic Web Conf., ser. LNCS 4825, K. Aberer and et al., Eds. Busan, Korea: Springer, 2007, pp. 480–493.

6. S. Grimm, S. Lamparter, A. Abecker, S. Agarwal, and A. Eberhart, "Ontology Based Specification of Web Service Policies," in *INFORMATIK 2004 - Proceedings of Semantic Web Services and Dynamic Networks*, ser. LNI, vol. 51. GI, September 2004, pp. 579–583.
7. S. Agarwal, S. Lamparter, and R. Studer, "Making Web services tradable - A policy-based approach for specifying preferences on Web service properties," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 7, no. 1, pp. 11–20, Januar 2009.
8. M. Junghans and S. Agarwal, "Towards Practical Semantic Web Service Discovery," in *7th Extended Semantic Web Conference*, ser. LNCS. Springer, 2010.
9. M. Junghans, S. Agarwal, "Web Service Discovery Based on Unified View on Functional and Non-Functional Properties," *Proceedings of Fourth IEEE International Conference on Semantic Computing (IEEE ICSC2010)*, Carnegie Mellon University, Pittsburgh, PA, USA, September, 2010.
10. W3C OWL Working Group, *OWL 2 Web Ontology Language: Document Overview*. W3C Recommendation, 27 October 2009, available at <http://www.w3.org/TR/owl2-overview/>.
11. J. de Bruijn, D. Fensel, M. Kerrigan, U. Keller, H. Lausen, and J. Scicluna, *Modeling Semantic Web Services: The Web Service Modeling Language*. Berlin: Springer, 2008.
12. Y. Sure, S. Bloehdorn, P. Haase, J. Hartmann, and D. Oberle, "The SWRC Ontology - Semantic Web for Research Communities," in *Proc. of the 12th Portuguese Conference on Artificial Intelligence – Progress in Artificial Intelligence (EPIA 2005)*, ser. LNCS, vol. 3803. Springer, December 2005, pp. 218–231.
13. J. Bock, "Parallel Computation Techniques for Ontology Reasoning," in *ISWC '08: Proceedings of the 7th International Conference on the Semantic Web*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 901–906.
14. C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.
15. A. M. Zaremski and J. M. Wing, "Specification matching of software components," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 4, pp. 333–369, 1997.
16. D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, and K. Sycara, "Bringing Semantics to Web Services: The OWL-S Approach," in *SWSWPC*, ser. LNCS, J. Cardoso and A. Sheth, Eds., vol. 3387. Springer, 2004, pp. 26–42.
17. B. Benatallah, M.-S. Hacid, A. Leger, C. Rey, and F. Toumani, "On automating Web services discovery," *The VLDB Journal*, vol. 14, no. 1, pp. 84–96, 2005.
18. J. Gonzalez-castillo, D. Trastour, and C. Bartolini, "Description Logics for Matchmaking of Services," in *KI-2001 Workshop on Applications of Description Logics*, 2001.
19. M. Stollberg, U. Keller, H. Lausen, and S. Heymans, "Two-phase Web Service Discovery based on Rich Functional Descriptions," in *Proc. of the 4th European Semantic Web Conf.* Springer, 6 2007.
20. U. Keller, H. Lausen, and M. Stollberg, "On the Semantics of Functional Descriptions of Web Services," in *Proc. of the 3rd European Semantic Web Conf.*, 2006.
21. T. Vitvar, J. Kopecký, J. Viskova, and D. Fensel, "WSMO-Lite Annotations for Web Services," in *5th European Semantic Web Conf.*, ser. LNCS 5021. Springer, 2008.
22. J. Domingue, L. Cabral, S. Galizia, V. Tanasescu, A. Gugliotta, B. Norton, and C. Pedrinaci, "IRS-III: A broker-based approach to semantic Web services," *Web Semant.*, vol. 6, no. 2, pp. 109–132, 2008.
23. J. O'Sullivan, "Towards a precise understanding of service properties," Ph.D. dissertation, Queensland University of Technology, 2006.
24. R. Lara, M. Corella, and P. Castells, "A Flexible Model for Locating Services on the Web," *Int. J. Electron. Commerce*, vol. 12, no. 2, 2008.
25. U. Keller, R. Lara, H. Lausen, A. Polleres, and D. Fensel, "Automatic Location of Services," in *Proceedings of the 2nd European Semantic Web Symposium (ESWS2005)*, Heraklion, Crete, 5 2005.
26. D. Hull, E. Zolin, A. Bovykin, I. Horrocks, U. Sattler, and R. Stevens, "Deciding Semantic Matching of Stateless Services," in *Proc. of 21st Nat. Conf. on Artificial intelligence (AAAI'06)*. AAAI Press, 2006.
27. Winkler, Daniel and Pressnig, Matthias. *Reasoner Framework Report - Installation and Configuration*. 2010. Technical Report.
28. Winkler, D., Pressnig, M., *D3.2.7 Second Prototype for Description Logic Reasoner for WSML DL v2.0*. 2010. SOA4All Project Deliverable.
29. Bishop, B., Fischer, F., Hitzler, P., Kroetzsch, M., Rudolph, S., Trimponias, Y., Unel, G., *Defining the Features of the WSML-DL v2.0 Language*. 2009. SOA4All Project Deliverable.
30. Reto Krummenacher, John Domingue, Carlos Pedrinaci, Elena Simperl. *SOA4All: Towards a Global Service Delivery Platform. Towards the Future Internet – Emerging Trends from European Research*. Edited by Georgios Tselentis, Alex Galis, Anastasius Gavras, Srdjan Krco, Volkmar Lotz, Elena Simperl, Burkhard Stiller, Theodore Zahariadis. pp. 161 – 172. 2010.
31. S. Agarwal, M. Junghans, O. Fabre, I. Toma, *D5.3.1 First Service Discovery Prototype*. 2009. SOA4All Project Deliverable.
32. S. Agarwal, M. Junghans, B. Norton, *D5.4.2 Second Ranking Prototype*. 2010. SOA4All Project Deliverable.
33. Stollberg, M.: Martin Hepp. *Semantic Discovery Caching: Prototype & Use Case Evaluation*. Technical Report DERI-2007-03-27, DERI (2007)
34. Lara R. *Two-phased Web Service Discovery*. *Proceedings of AI-Driven Technologies for Services-Oriented Computing Workshop at AAAI-06*, Boston, USA. 2006.
35. Kovacs, L.; Micsik, A.; Pallinger, P., *Two-phase Semantic Web Service Discovery Method for Finding Intersection Matches using Logic Programming*. *Workshop on Semantics for Web Services (SemWS'06)*, 2006.
36. Kopecký, J., Vitvar, T., Fensel, D. *D3.4.2 WSMO-Lite: Lightweight Semantic Descriptions for Services on the Web*, SOA4All Project Deliverable, 2009.