International Journal of Semantic Computing

A CACHING TECHNIQUE FOR OPTIMIZING AUTOMATED SERVICE DISCOVERY

MICHAEL STOLLBERG

SAP Resesearch Center Dresden,
Chemnitzer Str. 48, 01187 Dresden, Germany
michael.stollberg@sap.com

JÖRG HOFFMANN

Equipe MAIA, INRIA Nancy,
615 rue du Jardin Botanique, 54600 Villers-les-Nancy, France
joerg.hoffmann@loria.fr

DIETER FENSEL

Semantic Technology Institute, University of Innsbruck,
Technikerstrae 21a, 6020 Innsbruck, Austria
dieter.fensel@sti2.at

Received (07 January 2011)
Revised (25 March 2011)
Accepted (Day Month Year)

The development of sophisticated technologies for service-oriented architectures (SOA) is a grand challenge. A promising approach is the employment of semantic technologies to better support the service usage cycle. Most existing solutions show significant deficits in the computational performance, which hampers the applicability in large-scale SOA systems. We present an optimization technique for automated service discovery – one of the central operations in semantically enabled SOA environments – that can ensure a sophisticated performance while maintaining a high retrieval accuracy. The approach is based on goals that formally describe client objectives, and it employs a caching mechanism for enhancing the computational performance of a two-phased discovery framework. At design time, the suitable services for generic and reusable goal descriptions are determined by semantic matchmaking. The result is captured in a continuously updated graph structure that organizes goals and services with respect to the requested and provided functionalities. This is exploited at runtime in order to detect the suitable services for concrete client requests with minimal effort. We formalize the approach within a first-order logic framework, and define the graph structure along with the associated storage and retrieval algorithms. An empirical evaluation shows that significant performance improvements can be achieved.

Keywords: Service Discovery; Semantic Matchmaking; Goals; Optimization; Scalability.

1. Introduction

Service-oriented architectures (SOA) are the latest design paradigm for IT systems, proclaiming the use of services as the basic building blocks. While the basic technology stack around SOAP, WSDL, and UDDI enables the technical provision and use of Web services, the development of sophisticated SOA technology with accurate support for both service providers and clients remains a grand challenge [2, 11].

For this, the concept of Semantic Web Services (SWS) employs semantic technologies for supporting the service usage cycle, with the aim of faciliating flexible detection and consumption of services by providing automated mechanisms for the discovery, composition, and execution of Web services that leverage on formal, ontology-based descriptions [30, 4].

A central operation in SWS environments as envisioned by promiment approaches (esp. OWL-S [21], WSMO [14], and SAWSDL [12]) is *discovery*, which is concerned with the automated detection of suitable Web services for a given client request. Despite remarkable research results on semantic matchmaking as the base technique, most existing solutions lack computational performance. This hampers the usability in large-scale SOA applications, because the discovery task is usually performed as the first processing step that needs to consider all available Web services and thus forms a scalability bottleneck.

To overcome this problem, we present a two-phased approach for automated Web service discovery that is extended with a caching mechanism for enhancing computational performance. At design time, the suitable Web services for goals – i.e. generic and reusable descriptions of client objectives – are determined by semantic matchmaking. The result is captured in a directed acyclic graph that organizes the goals in subsumption hierarchies and captures the relevant knowledge on the usability of the available Web services. At runtime, the captured knowledge is leveraged in order to minimize the computational effort for detecting the suitable Web services for actual client requests, which are defined by instantiating goals with concrete inputs. The graph is updated whenever goals or Web services are added, removed, or changed, hence forming a cache for future searches. In comparison to existing optimization techniques, this approach allows us to perform the discovery task in an efficient and scalable manner while maintaining the high retrieval accuracy that can be obtained by employing semantic matchmaking techniques.

This article presents the main results of a PhD thesis [27], providing a revision of a preliminary approach presented in [28] along with a comprehensive evaluation. It is structured as follows: Section 2 introduces the problem setting and provides an overview of the approach. Section 3 defines the formal foundations and the semantic matchmaking techniques that we use for Web service discovery. Section 4 specifies the graph structure for caching discovery results, and Section 5 presents the optimized runtime discovery that utilizes the captured knowledge. Section 6 provides an empirical evaluation, and Section 7 positions our approach within related work. Finally, Section 8 concludes the article.

2. Overview

The following introduces the research context. We first explain the SWS approach and identify the need for optimizing automated Web service discovery, and then outline our approach for this.

2.1. *Semantic Web Services*

The overall aim of Semantic Web Services (SWS) – considering particularly the initial research efforts around OWL-S [21] and WSMO [14] – is to automate the Web service usage cycle for clients. This shall overcome the limitations of the classic Web service technology stack where the detection of suitable Web services is left to manual inspection. For this, inference-based techniques are developed for automating the central tasks of the Web service usage cycle by leveraging formal, ontology-based descriptions of Web services, client requests, and other relevant resources.

Figure 1 illustrates the workflow that is commonly realized in existing SWS environments (e.g. [13, 9]). For solving a given client request, at first the *discovery* component detects the potential candidates out of the available Web services. Then, the *selection and ranking* component chooses to most suitable candidate with respect to non-functional aspects, and the *behavioral compatibility* component checks whether the communication between the provider and the requester carried out successfully. Alternatively, the *composition* component automatically combines several Web services in order to solve the client request. The above components may utilize *mediation* facilities for handling potentially occurring mismatches that hamper successful interaction. Finally, the detected Web services are executed.

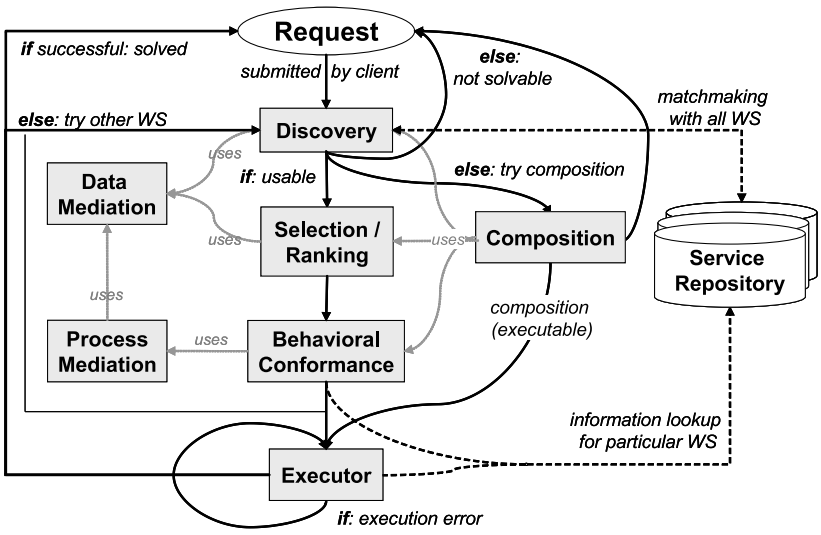


Fig. 1. Automated Web Service Usage with SWS

The discovery component is a bottleneck for scalability of such systems: it must consider all available Web services, and perform potentially expensive matchmaking for determining the suitable candidates out of them. This becomes in particular relevant for larger SOA applications as well as for advanced SWS techniques where automated discovery is a central and often performed operation (e.g. [3], [15]).

In consequence, two central requirements arise for automated discovery engines: (1) a high retrieval accuracy for determining candidate services, and (2) a high computational performance to ensure that the discovery task is performed in an efficient and scalable manner. While the former can most suitably be achieved by employing semantic matchmaking techniques that work on exhaustive formal and ontology-based descriptions, the latter requires optimization for reducing the necessary matchmaking effort. Most existing works address either of the two requirements separately. However, a sophisticated solution needs to meet both in order to assure the functional and the operational reliability of automated discovery engines.

The present work provides an integrated framework for this. The following gives an overview of our solution that is explained in detail in the subsequent sections.

2.2. *Approach for Scalable Automated Discovery*

The overall aim is to facilitate automated service discovery with high accuracy in a scalable manner, therewith satisfying the requirements identified above. For this, we follow the approach of working with highly expressive formal descriptions of services and client requests envisioned by prominent SWS frameworks: these allow achieving a retrieval accuracy for service discovery, but require optimization in order to be applicable in larger SOA systems.

Following the conception that underlies OWL-S and especially by WSMO, we take a goal-driven approach where a goal formally describes the objective that a client wants to achieve by using Web services, and the system automatically determines and executes the necessary Web services for solving the goal. This allows clients to request and consume Web services in a problem-oriented manner, abstracting from technical details for the actual invocation. To better support this, we distinguish *Goal Templates* as generic and reusable descriptions of client objectives that are stored in the system, and *Goal Instances* that describe concrete client requests and are defined by instantiating a goal template with concrete inputs.

On this basis, we separate the discovery task into two phases: at design time, the suitable Web services for Goal Templates are discovered by semantic matchmaking of their formal functional descriptions, while the actual Web services for solving a specific Goal Instance are determined at runtime. The latter is the time critical operation with respect to the operational reliability for solving concrete client requests. We optimize this by capturing relevant results from design time discovery, and then exploiting this in order to minimize the necessary matchmaking effort for runtime discovery. This adapts the concept of caching to Web service discovery, hence we refer to it as *Semantic Discovery Caching* (SDC).

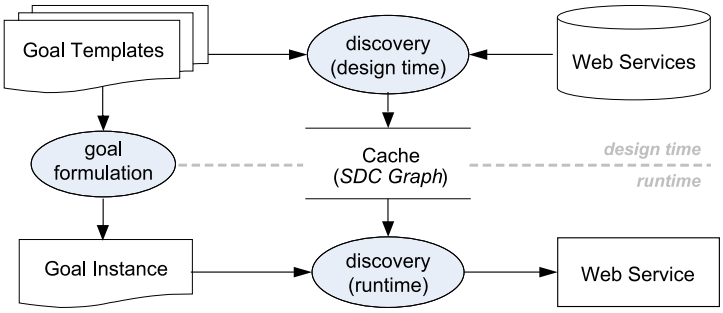


Fig. 2. Overview of the Approach

Figure 2 provides an overview of the framework as a dataflow diagram. The Goal Templates and the Web services along with their formal descriptions are stored in the system. The *SDC graph* forms the cache for the design time discovery results. This is defined as a directed acyclic graph (DAG) that properly describes the relevant relationships: it organizes the Goal Templates in subsumption hierarchies with respect to the requested functionalities, and captures the relevant knowledge on the functional usability of the available Web services in form of directed arcs annotated with the respective matching degree. At runtime, clients formulate concrete requests by creating a Goal Instance, and the runtime discovery determines the actually suitable Web services by utilizing the knowledge kept in the SDC graph.

This approach is suitable for satisfying both requirements identified above in an integrated manner. The remainder of the article first explains the technical solution in detail (Sections 3 - 5), then evaluates the achievable performance increase (Section 6), and finally positions the approach within related work (Section 7).

3. Semantic Web Service Discovery

This section defines semantic matchmaking techniques for automated Web service discovery, summarizing the approach presented in [29]. We follow a standard approach where the requested and provided functionalities are formally described in terms of preconditions and effects. We use classical first-order logic (FOL) as the specification language. This is undecidable in the general case, but provides high expressivity and serves as a logical umbrella for most of the ontology languages developed for the Semantic Web [6] that are used by prominent SWS frameworks.

3.1. *Foundations*

According to the common understanding, we consider Web service discovery to be primarily concerned with functional aspects. This relates to *what* a service can do, respectively *what* a client wants to achieve. If a service does not provide the functionality that is necessary to solve a goal, then it is not usable, and further investigations on other aspects become obsolete [24].

As the conceptual model that underlies our approach, we consider an actual execution of a Web service W to denote a sequence $\tau = (s_0, \ldots, s_m)$ that is observable in the world. In the context of discovery, we are merely interested in the start- and end states, because these constitute the provided functionality that can be described in terms of preconditions and effects. We thus define $\mathcal{T} = (s_0, s_m)$ as an abstraction that contains all observable executions with the same start- and end state. Then, we can consider the functionality provided by W as the set of all possible solutions, denoted by $\{\mathcal{T}\}_W$.

Analogously, we consider the functionality requested by a Goal Template G as the set $\{\mathcal{T}\}_G$ of all its possible solutions, i.e. changes of the world from a given start-state to a desired final state wherein the client objective is achieved. Then, a Web service W is usable for solving a Goal Template G if there is at least one possible execution of W that is a solution for G. Formally, we define the basic matching condition under functional aspects as $match(G, W) : \exists \mathcal{T}.\ \mathcal{T} \in (\{\mathcal{T}\}_G \cap \{\mathcal{T}\}_W)$.

A Goal Instance $GI(G, \beta)$ describes a concrete client request by instantiating a suitable Goal Template G with an input binding β, i.e. an assignment of concrete values to the inputs defined in G. We require goal instances to be valid instantiations, denoted by $GI(G, \beta) \models G$, which is given if β satisfies the conditions defined in G. Then, the solutions for $GI(G, \beta)$ are a subset of those for its corresponding goal template, i.e. $\{\mathcal{T}\}_{GI(G,\beta)} \subset \{\mathcal{T}\}_G$. For solving a Goal Instance, a suitable Web service W is invoked with the input binding defined in $GI(G, \beta)$. Thus, we define the basic matching condition as $match(GI(G, \beta), W) : \exists \mathcal{T}.\ \mathcal{T} \in (\{\mathcal{T}\}_{GI(G,\beta)} \cap \{\mathcal{T}\}_{W(\beta)})$ where $\{\mathcal{T}\}_{W(\beta)} \subset \{\mathcal{T}\}_W$ is the set of possible executions of W when it is invoked with a particular β.

This constitutes the basis of our two-phased discovery framework: the suitable Web services for goal templates can be determined at design time, and only these need to be inspected to detect the actually suitable Web services for goal instances at runtime. The following defines the necessary formal descriptions and semantic matchmaking techniques for evaluating these conditions.

3.2. *Functional Descriptions*

Following the common approach for semantically enabled Web service discovery (e.g. [22, 20]), we describe the functionality provided by a Web service as well as the one requested by a goal in terms of inputs, outputs, preconditions, and effects on the basis of ontologies.

Formally, we define a functional description $\mathcal{D}$ over a signature Σ with FOL as the underlying logic by four elements: a set of input variables $IN = \{i_1, \ldots, i_n\}$, a set of output variables $OUT = \{o_1, \ldots, o_m\}$, a precondition ϕ^{pre} that constrains the possible start states, and an effect ϕ^{eff} that constrains the possible end states. Each element is defined on the basis of consistent background knowledge Ω that is defined in terms of ontologies. In order to precisely describe the dependency of the start- and end states, IN occur as free variables in both ϕ^{pre} and ϕ^{eff}. For a

formula ϕ, we denote the set of free variables by $free(\phi)$ and the set of quantified variables by $quant(\phi)$.

Definition 3.1. *Let Σ be a signature and Ω be a consistent ontology. A functional description over Σ and Ω is a 4-tuple $\mathcal{D} = (IN, OUT, \phi^{pre}, \phi^{eff})$ where*

(1) IN is the set of input variables $i_1, \ldots, i_n$
(2) OUT is the set of output variables $o_1, \ldots, o_m$
(3) ϕ^{pre} is the precondition, a FOL formula with $free(\phi^{pre}) = IN$ and $quant(\phi^{pre}) \cap OUT = \emptyset$
(4) ϕ^{eff} is the effect, a FOL formula with $free(eff) \subseteq IN$ and $quant(eff) \supseteq OUT$.

An input binding that is defined by a Goal Instance and then used to invoke a Web service is a total function $\beta : \{i_1, \ldots, i_n\} \to \mathcal{U}$ that instantiates each IN-variable with an object of the universe $\mathcal{U}$.

Fig. 3. Examples for Functional Descriptions

Goal Template G "ship a package of any weight in Europe"		**Web Service** W "shipment in Germany, max 50 kg"	
Ω:	location & shipment ontology	Ω:	location & shipment ontology
IN:	{?s,?r,?p,?w}	IN:	{?s,?r,?p,?w}
OUT:	{?o}	OUT:	{?o}
ϕ^{pre}:	$sender(?s) \wedge in(?s, europe)$ $\wedge\, receiver(?r) \wedge in(?r, europe)$ $\wedge\, package(?p) \;\wedge weight(?p, ?w)$.	ϕ^{pre}:	$sender(?s) \wedge in(?s, germany)$ $\wedge\, receiver(?r) \wedge in(?r, germany)$ $\wedge\, package(?p) \;\wedge weight(?p, ?w)$ $\wedge\, maxWeight(?w < 50kg)$.
ϕ^{eff}:	$\exists ?o, ?price.\ shipmentOrder(?o, ?p)$ $\wedge\, from(?p, ?s) \;\wedge to(?p, ?r)$ $\wedge\, costs(?o, ?price)$).	ϕ^{eff}:	$\exists ?o, ?price.\ shipmentOrder(?o, ?p)$ $\wedge\, from(?p, ?s) \;\wedge to(?p, ?r)$ $\wedge\, costs(?o, ?price)$).

Figure 3 shows examples of functional descriptions for the shipment scenario defined in the SWS challenge [23]. The goal template describes the objective of shipping a package of any weight in Europe. This is defined by four input variables that are constrained in the precondition, and the effect states that the desired output is a shipment order for the package. The Web service offers a more restricted functionality for shipping packages of maximal 50kg within Germany. This is described analogously. The *location & shipment ontology* defines the relevant background knowledge; in FOL, we define ontology concepts as unary predicates, and attributes as well as relations as n-ary predicates. Note that the input variables occur as the only free variables in both ϕ^{pre} and ϕ^{eff}; these will be instantiated by input bindings defined in goal instances for shipping a particular package. We here model the outputs with existential quantification; Definition 3.1 allows arbitrary

FOL formula for this.

We define the formal meaning of a functional description as an implication between the precondition and the effect. This means that a Web service W provides the functionality described by $\mathcal{D}_W$, denoted by $W \models \mathcal{D}_W$, if and only if for all $\mathcal{T} \in \{\mathcal{T}\}_W$ holds that if $s_0 \models \phi^{pre}$ then $s_m \models \phi^{eff}$. In order to deal with functional descriptions in terms of model-theoretic semantics, we present this as a FOL formula $\phi^{\mathcal{D}_W}$ of the form $\phi^{pre} \Rightarrow \phi^{eff}$. Then, $W \models \mathcal{D}_W$ is given if and only if every $\mathcal{T} \in \{\mathcal{T}\}_W$ is represented by a Σ-interpretation that is a model of $\phi^{\mathcal{D}_W}$.

Analogously, the possible solutions for a Goal Template G are represented by the models of the FOL formula $\phi^{\mathcal{D}_G}$ that defines the implication semantics for the functional description $\mathcal{D}_G$. On this basis, we can formally define the goal instantiation $GI(G, \beta) \models G$ to be given if $\phi^{\mathcal{D}_G}$ is satisfiable under the input binding β. As a prerequisite for meaningful reasoning, we consider all functional descriptions to be *consistent*, meaning that $\phi^{\mathcal{D}}$ is satisfiable under an input binding β. Otherwise, a Web service $W \models \mathcal{D}$ would not realizable, and there would not be any solution for a goal [17].

3.3. *Semantic Matchmaking*

We now define the semantic matchmaking techniques for our 2-phased discovery framework. For this, we specify proof obligations on the basis of the formal functional descriptions defined above that evaluate the matching conditions discussed at the beginning of this section. The following first defines the matchmaking on the goal template level for discovery at design time, and then for runtime discovery on the level of goal instances.

3.3.1. *Goal Template Level*

As discussed above, we consider a Web service W to be usable for a Goal Template G if at least one of its possible executions is a solution for G. We express this in terms of matching degrees that properly describe all possible situations: four degrees distinguish situations where a match is given (*exact, plugin, subsume, intersect*), while *disjoint* denotes that W is not usable for solving G.

Table 1 shows the definition of the matching degrees, and depicts their meaning in terms of the set-theoretic relation between the possible executions of W and the possible solutions of G. We define the proof obligations on the basis of the formula $\phi^{\mathcal{D}} := \phi^{pre} \Rightarrow \phi^{eff}$ that reflects the formal meaning of the functional descriptions as explained above, along with a quantification of the input variables. This ensures that the signatures of $\mathcal{D}_G$ and $\mathcal{D}_W$ must be compatible in order to determine a match, and, furthermore, that all free variables that can occur in the preconditions and effects are bound. Thus, we can apply automated theorem proving for the matchmaking. The conditions for the degrees *exact, plugin* and *subsume* require logical entailment, while *intersect* as the weakest degree where a match is given requires a satisfiability test. We always use the highest possible degree to properly

Table 1. Definition of Matching Degrees

Denotation	Definition	Meaning
exact$(\mathcal{D}_G, \mathcal{D}_W)$	$\Omega \models \forall\beta.\ \phi^{\mathcal{D}_G} \Leftrightarrow \phi^{\mathcal{D}_W}$	$\{\mathcal{T}\}_G = \{\mathcal{T}\}_W$
plugin$(\mathcal{D}_G, \mathcal{D}_W)$	$\Omega \models \forall\beta.\ \phi^{\mathcal{D}_G} \Rightarrow \phi^{\mathcal{D}_W}$	$\{\mathcal{T}\}_G \subseteq \{\mathcal{T}\}_W$
subsume$(\mathcal{D}_G, \mathcal{D}_W)$	$\Omega \models \forall\beta.\ \phi^{\mathcal{D}_W} \Rightarrow \phi^{\mathcal{D}_G}$	$\{\mathcal{T}\}_G \supseteq \{\mathcal{T}\}_W$
intersect$(\mathcal{D}_G, \mathcal{D}_W)$	$\bigwedge \Omega \wedge \phi^{\mathcal{D}_G} \wedge \phi^{\mathcal{D}_W}$ is satisfiable	$\{\mathcal{T}\}_G \cap \{\mathcal{T}\}_W \neq \emptyset$
disjoint$(\mathcal{D}_G, \mathcal{D}_W)$	$\bigwedge \Omega \wedge \phi^{\mathcal{D}_G} \wedge \phi^{\mathcal{D}_W}$ is not satisfiable	$\{\mathcal{T}\}_G \cap \{\mathcal{T}\}_W = \emptyset$

denote the usability of a Web service by considering the following formal relations: (1) *plugin* $\wedge$ *subsume* $\Leftrightarrow$ *exact*, (2) *plugin* $\Rightarrow$ *intersect*, (3) *subsume* $\Rightarrow$ *intersect*, and (4) $\neg$ *intersect* $\Leftrightarrow$ *disjoint*.

We remark that similar matching degrees have been defined in other works before (e.g. [22, 20]). However, our matchmaking conditions are defined over more expressive functional descriptions, and thus can warrant a higher retrieval accuracy for the discovery task. We shall discuss this in more detail below in Section 7.

3.3.2. *Goal Instance Level*

For runtime discovery, we consider a Web service to be *usable for* a Goal Instance $GI(G, \beta)$ if it can provide a solution for $GI(G, \beta)$ when invoked with the respective input binding. Formally, this is given iff the formula $\Omega \wedge [\phi^{\mathcal{D}_G}]_\beta \wedge [\phi^{\mathcal{D}_W}]_\beta$ is satisfiable. This means that, under consideration of the ontology, there must be a common model for $\phi^{\mathcal{D}_G}$ and $\phi^{\mathcal{D}_W}$ when the functional descriptions are instantiated with the input binding defined in the Goal Instance (denoted by $[\phi^{\mathcal{D}}]_\beta$).

However, we can simplify the necessary matchmaking effort by using the knowledge on the usability of W for the corresponding Goal Template G as follows.

Theorem 3.1. *Let G be a Goal Template, and let $GI(G, \beta)$ be a Goal Instance such that $GI(G, \beta) \models G$. Let W be a Web service, and let $\mathcal{D}_W$ be a functional description such that $W \models \mathcal{D}_W$.*
W is usable for solving $GI(G, \beta)$ if and only if:

(1) ***exact***$(\mathcal{D}_G, \mathcal{D}_W)$ *or*
(2) ***plugin***$(\mathcal{D}_G, \mathcal{D}_W)$ *or*
(3) ***subsume***$(\mathcal{D}_G, \mathcal{D}_W)$ *and $\Omega \wedge [\phi^{\mathcal{D}_W}]_\beta$ is satisfiable, or*
(4) ***intersect***$(\mathcal{D}_G, \mathcal{D}_W)$ *and $\Omega \wedge [\phi^{\mathcal{D}_G}]_\beta \wedge [\phi^{\mathcal{D}_W}]_\beta$ is satisfiable.*

Referring to [27] for the proof, this states that only those Web services that are usable for a Goal Template G are potentially usable for a Goal Instance $GI(G, \beta)$, and that the actual usability can be determined with reduced matchmaking ef-

fort. The *disjoint* degree describes the situation where $\{\mathcal{T}\}_G \cap \{\mathcal{T}\}_W = \emptyset$. Here, W can not provide any solution for the Goal Instance because $GI(G, \beta) \models G$ implies that $\{\mathcal{T}\}_{GI(G,\beta)} \subseteq \{\mathcal{T}\}_G$. Under the *exact* and the *plugin* degree, it holds that $\{\mathcal{T}\}_{GI(G,\beta)} \subseteq \{\mathcal{T}\}_G \subseteq \{\mathcal{T}\}_W$. Thus, in these situations W is usable for any valid instantiation of G, and we do not need to perform any additional matchmaking at runtime. Under the *subsume* degree, all executions of W are solutions of G but not vice versa. The modelling sample above in Figure 3 is an example for this situation. Consider a Goal Instance that defines $\beta = \{?s|paris, ?r|vienna, ?p|aPackage, ?w|3.1kg\}$. Although this properly instantiates the Goal Template, β does not provide valid inputs for invoking the Web service (which is restricted to Germany); thus, it is not usable here. Under the *intersect* degree, the complete matching condition explained above needs to checked, because there can be solutions for $GI(G, \beta)$ that can not be provided by the Web service and vice versa.

This provides the first step towards optimizing Web service discovery at runtime. The caching mechanism presented in the following builds upon this in order to minimize the necessary matchmaking effort.

4. The Semantic Discovery Cache

This section introduces the Semantic Discovery Caching technique (SDC) for enhancing the computational performance of automated Web service discovery. The following defines the SDC graph that captures the relevant knowledge from design time discovery, specifies the algorithms for its creation and maintenance, and analyzes its formal properties.

4.1. *Overview*

The approach for optimizing the discovery task is to reduce the search space and minimize the necessary matchmaking operations by exploiting the relationships between goal templates, goal instances, and Web services. The central element for this is the SDC graph that organizes goal templates in a subsumption hierarchy with respect to their semantic similarity, and captures the relevant knowledge on the functional usability of the available Web services for the goal templates that is obtained from design time discovery.

As the constituting notion, we consider two Goal Templates G_i and G_j to be similar if they have at least one common solution. We express this in terms of *similarity degrees* $d(G_i, G_j)$ where $d = \{exact, plugin, subsume, intersect\}$ that denote the matching degree between the functional descriptions $\mathcal{D}_{G_i}$ and $\mathcal{D}_{G_j}$ and are formally defined analogously to Table 1 above. In order to enable efficient search, we define the SDC graph such that the only occurring similarity degree is $subsume(G_i, G_j)$. If this is given, then (1) the solutions for the child G_j are a subset of those for the parent G_i, and thus (2) the Web services that are usable for G_j are a subset of those usable for G_i.

In consequence, the SDC graph consists of two layers. The upper one is the *goal graph* that properly describes the semantic similarity among all existing goal templates and organizes them in subsumption hierarchies by directed arcs of the form (G_i, G_j). The lower layer is the *discovery cache* that explicates the usability of each available Web service W for every goal template G by directed arcs that are annotated with the usability degree $d(G, W)$ with $d = \{exact, plugin, subsume, intersect\}$ as defined in Table 1. It omits those arcs for which the usability degree can be directly inferred by rules of the form $d(G_i, G_j) \wedge d(G_i, W) \Rightarrow d(G_j, W)$. Therewith, the SDC graph forms a directed acyclic graph (DAG) that defines the minimal set of arcs necessary to perform efficient runtime discovery on the basis of Theorem 3.1.

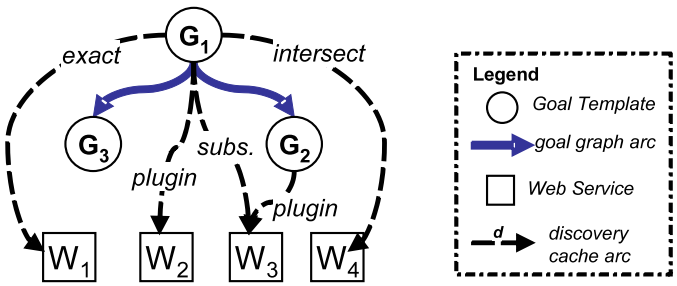


Fig. 4. Example of a SDC Graph

Figure 4 illustrates the structure of the SDC graph for our running example. There are three goal templates: G_1 for package shipment in Europe, G_2 for Switzerland, and G_3 for Germany. Their similarity degrees are $subsume(G_1, G_2)$ and $subsume(G_1, G_3)$. Thus, G_1 becomes the root node of the SDC graph, and G_2 and G_3 become its child nodes in the goal graph. Let W_1 be a Web service for package shipment in Europe. Its usability degree for G_1 is *exact*, which is explicated in the discovery cache. From this, we can infer that the usability degree of W_1 for G_2 and G_3 is *plugin*; these arcs are omitted in the SDC graph. The same holds for W_2 where $plugin(G_1, W_2)$, while under the usability degrees *subsume* and *intersect* the precise usability degree for the child nodes of G_1 can not be inferred directly.

The following defines the structure, constructs, and background of the SDC graph formally.

4.2. *The SDC Graph*

As outlined above, the SDC graph is a directed acyclic graph (DAG) with the goal graph as the inner layer and the discovery cache as the outer layer. The following formally defines its structure, and we explain the constructs and properties in more detail below.

Definition 4.1. *Let $d(G_i, G_j)$ be the similarity degree of goal templates G_i and G_j, and let $d(G, W)$ be the usability degree of a Web service W for a Goal Template G. Given a set $\mathcal{G}$ of goal templates and a set $\mathcal{W}$ of Web services, the SDC graph is the directed acyclic graph $(V_\mathcal{G} \cup V_\mathcal{W}, E_{sim} \cup E_{use})$ where:*

1) $V_\mathcal{G} := \mathcal{G} \cup \mathcal{G}^I$ is the set of inner vertices, where

- *$\mathcal{G} = \{G_1, \ldots, G_n\}$ are the goal templates; and*
- *$\mathcal{G}^I := \{G^I \mid G_i, G_j \in \mathcal{G}, d(G_i, G_j) = \mathsf{intersect}, G^I \equiv G_i \wedge G_j\}$ are the intersection goal templates for $\mathcal{G}$*

2) $\mathcal{W} := \{W_1, \ldots, W_m\}$ is the set of leaf vertices that represent Web services

3) $E_{sim} := \{(G_i, G_j) \mid G_i, G_j \in V_\mathcal{G}\}$ is the set of directed arcs constituting the goal graph, where

- *$d(G_i, G_j) = \mathsf{subsume}$; and*
- *not exists $G \in V_\mathcal{G}$ such that $d(G_i, G) = \mathsf{subsume}$, $d(G, G_j) = \mathsf{subsume}$*

4) $E_{use} := \{(G, W) \mid G \in V_\mathcal{G}, W \in V_\mathcal{W}\}$ is the set of directed arcs constituting the discovery cache, where

- *$d(G, W) \in \{\mathsf{exact}, \mathsf{plugin}, \mathsf{subsume}, \mathsf{intersect}\}$; and*
- *not exists $G_i \in V_\mathcal{G}$ such that $d(G_i, G) = \mathsf{subsume}$, $d(G_i, W) \in \{\mathsf{exact}, \mathsf{plugin}\}$.*

4.2.1. *The Goal Graph*

The goal graph organizes the existing goal templates in subsumption hierarchies that properly reflect their similarity with respect to the requested functionalities. This constitutes the skeletal structure of the SDC graph; the discovery cache amends this with the relevant knowledge on the usability of the available Web services.

We define the goal graph by directed arcs (G_i, G_j) that represent $subsume(G_i, G_j)$ as the only occurring similarity degree. This facilitates efficient search, because under this degree the Web services that are usable for the child G_j are always a subset of those usable for the parent G_i as it holds that if $\{\mathcal{T}\}_{G_i} \supseteq \{\mathcal{T}\}_{G_j}$ and $\{\mathcal{T}\}_{G_i} \cap \{\mathcal{T}\}_W = \emptyset$ then also $\{\mathcal{T}\}_{G_j} \cap \{\mathcal{T}\}_W = \emptyset$. To avoid redundancy, we describe functional subsumption hierarchies among several goal templates by the minimal set of goal graph arcs (*cf.* clause 3 of Definition 4.1).

In order to properly describe the functional similarities for any set of goal templates, we handle the other possible similarity degrees as follows: if $exact(G_i, G_j)$, only one of the goal template is kept while the other one is redundant for our purposes; if $plugin(G_i, G_j)$, we store the opponent arc (G_j, G_i). If $disjoint(G_i, G_j)$, then both goal templates are kept as separated nodes. Thus, the SDC graph can have disconnected subgraphs where each one contains a set of semantically similar goal templates along with their usable Web services.

The only critical similarity degree is $intersect(G_i, G_j)$, which denotes that G_i and G_j have a common solution but there are also exclusive solutions for each. This can cause cycles in the SDC graph which hamper its search properties. To avoid this, we create an *intersection goal template* $G^I(G_i, G_j)$ (*cf.* clause 1). Formally, this is defined as the conjunction of the functional descriptions of the original goal templates, i.e. $\phi^{\mathcal{D}_{G^I(G_i,G_j)}} = \phi^{\mathcal{D}_{G_i}} \wedge \phi^{\mathcal{D}_{G_j}}$. Therefore, $G^I(G_i, G_j)$ describes exactly the common solutions of G_i and G_j, i.e. $\{\mathcal{T}\}_{G^I(G_i,G_j)} = \{\mathcal{T}\}_{G_i} \cap \{\mathcal{T}\}_{G_j}$. In consequence, it holds that $subsume(G_i, G^I(G_i, G_j))$ and also $subsume(G_j, G^I(G_i, G_j))$. Thus, $G^I(G_i, G_j)$ becomes a child node of both G_i and G_j in the goal graph, as illustrated in Figure 5. Intersection goal templates are merely logical constructs in the SDC graph; their their functional descriptions do not have to be materialized.

This is applied for every occurring *intersect* similarity degree, so that eventually all similar goal templates are organized in a subsumption hierarchy without cycles. With this handling of the possible similarity degrees, the goal graph provides a general purpose structure for organizing any set of goal templates in redundancy-free subsumption hierarchies wherein every goal template is allocated at a precisely defined position. It however can contain disconnected subgraphs and child nodes can have multiple parents, so that in general the SDC graph is a directed acyclic graph but not a tree.

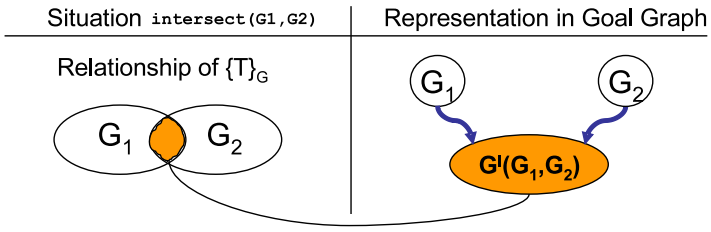


Fig. 5. Intersection Goal Templates

4.2.2. *The Discovery Cache*

The discovery cache captures the relevant knowledge on the functional usability of the available Web services for the existing goal templates, which is obtained from design time discovery. This is defined by direct arcs $d(G, W)$ annotated with the respective matching degree, so that the Web services denote the leaf nodes of the SDC graph (*cf.* clause 2 of Definition 4.1).

The main purpose is to enable efficient runtime discovery in accordance to Theorem 3.1, *cf.* Section 3.3.2. For this, we only define discovery cache arcs when the basic matching condition is satisfied, i.e. where $d \in \{\mathsf{exact}, \mathsf{plugin}, \mathsf{subsume}, \mathsf{intersect}\}$. Moreover, we only define the minimal set of arcs that is necessary to deduce the precise matching degree of every Web service for every goal template (*cf.* clause 4). This

Table 2. Inference Rules for Usability Degrees under $subsume(G_i, G_j)$

(1)	$exact(G_i, W)$	$\Rightarrow$	$plugin(G_j, W)$.
(2)	$plugin(G_i, W)$	$\Rightarrow$	$plugin(G_j, W)$.
(3)	$subsume(G_i, W)$	$\Rightarrow$	$exact(G_j, W) \vee$ $plugin(G_j, W) \vee$ $subsume(G_j, W) \vee$ $intersect(G_j, W) \vee$ $disjoint(G_j, W)$.
(4)	$intersect(G_i, W)$	$\Rightarrow$	$plugin(G_j, W) \vee$ $intersect(G_j, W) \vee$ $disjoint(G_j, W)$.
(5)	$disjoint(G_i, W)$	$\Rightarrow$	$disjoint(G_j, W)$.

is facilitated by general rules for inferring the usability degree among semantically similar goal templates.

Table 2 above lists the most important rules: if a Web service W is usable for a Goal Template G_i under the degree *exact* or *plugin* and G_i is a parent of another Goal Template G_j in the goal graph (i.e. $subsume(G_i, G_j)$), the only possible usability degree of W for the child G_j is *plugin* because it then holds that $\{\mathcal{T}\}_W \supseteq \{\mathcal{T}\}_{G_i} \supset \{\mathcal{T}\}_{G_j}$ (*cf.* rules 1 and 2). Thus, the arc (G_j, W) is not explicitly defined in the discovery cache, and this is applied for all occurrences of directly inferable discovery cache arcs. We refer to [27] for the formal definition of all inference rules that are used by the SDC technique.

4.3. *SDC Graph Management*

In order to serve as a suitable search index for goals and Web services, it is necessary that the SDC graph describes the relevant relationships correctly and at all times. For this, we provide algorithms for creating and maintaining the SDC graph that ensure its structure as defined above whenever a goal template or a Web services is added, removed, or modified.

Figure 6 shows the algorithm for creating an SDC graph. This is done by the subsequent insertion of new goal templates, following the rules explained above. At first, the new Goal Template G_{new} is allocated at the correct position in the goal graph. For this, we check its semantic similarity with the already existing root nodes, and then position G_{new} in the goal graph. The sub-procedures implement the rules for handling the possible similarity degrees. In particular, `child node insertion` positions G_{new} as a new child node by subsequently inspecting its semantic similarity with the goal templates in an already existing subsumption hierarchy, and `insersectGT insertion` handles all occurrences of the *intersect* similarity by defining the necessary intersection goal templates. After that, the discovery cache is created, respectively updated. We distinguish two procedures for this: if G_{new} has

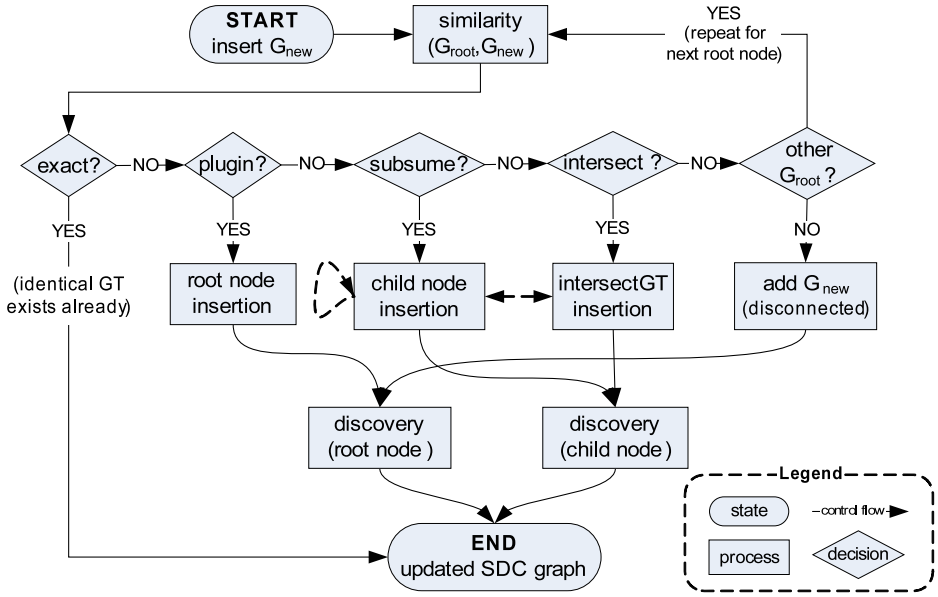


Fig. 6. SDC Graph Creation Algorithm

been inserted as a child node in the goal graph, the `child node discovery` merely needs to inspect the Web service that are usable for its direct parents (*cf.* rule 5 in Table 2). Otherwise, the `root node discovery` needs to inspect all available Web services, because there might be Web services that are only usable for G_{new}.

The algorithms for removing or modifying goal templates and Web services work analogously. They mainly remove the obsolete nodes and arcs, and revise the structure of the SDC graph if necessary. We refer to [27] for the exhaustive specification of all SDC graph management algorithms. These can be integrated with the system repositories, so that the updates are triggered whenever a change on the resources occurs.

4.4. *Formal Analysis*

We conclude the specification of the SDC graph with discussing its properties as a search index for goals and Web services. In particular, we show that the SDC graph holds all relevant knowledge that is relevant for optimizing discovery operations in a concise and redundancy-free manner, and we analyze its asymptotic complexity.

4.4.1. *Inferential Completeness and Minimality*

The SDC graph as defined above exposes two properties that are relevant for its application purpose. The first one is that we can infer (a) the precise similarity between every pair of goal templates, and (b) the precise usability degree of each

Web service for every existing goal template. This knowledge is necessary for optimizing Web service discovery, in particular at runtime; we thus refer to this as the *inferential completeness* of the SDC graph. The second property is that the SDC graph describes this knowledge in a concise and redundancy-free manner: every arc in the SDC graph is necessary to establish the inferential completeness, and every additional arc that describes a correct relationship between the goal templates and the Web services is redundant because it can be inferred from the existing arcs. We call this the *inferential minimality* of the SDC graph, which greatly eases its maintainability. The following states this formally.

Theorem 4.1. *Let* $SDC = (V_{\mathcal{G}} \cup V_{\mathcal{W}}, E_{sim} \cup E_{use})$ *be the SDC graph over a set of goal templates* $\mathcal{G}$ *and a set of Web services* $\mathcal{W}$*. Let* $A_{SDC} = \{d(x,y) | d \in \{exact, plugin, subsume, intersect\}, x \in \mathcal{G}, y \in \{\mathcal{G}, \mathcal{W}\}, (x,y) \in (E_{sim}, E_{use})\}$ *be all atoms defined in SDC. Let* $\mathcal{IR}$ *be the set of all inference rules that hold between* $d(x,y)$*, and let* $cl^{\star}(A_{SDC}) = \{d(x,y) | A_{SDC} \wedge \mathcal{IR} \models d(x,y)\}$ *be the deductive closure of* A_{SDC} *over* $\mathcal{IR}$*.*

SDC is inferentially complete and minimal:

(1) $d(x,y) \in cl^{\star}(A_{SDC})$ *if and only if* $d(x,y)$ *is* `true` *for* $\mathcal{G} \times \mathcal{W}$*; and*
(2) for all $d(x,y) \in A_{SDC} : cl^{\star}(A_{SDC} \setminus d(x,y)) \subset cl^{\star}(A_{SDC})$*.*

Referring to [27] for the proof, the rationale for this is as follows. Every atom $d(x,y) \in A_{SDC}$ describes a relationship that is obtained from semantic matchmaking, and thus is `true` for $\mathcal{G} \times \mathcal{W}$ by definition. The deductive closure $cl^{\star}(A_{SDC})$ is the set of all atoms $A_{SDC} \cup \{d^{\star}(x,y)\}$ where $d^{\star}(x,y)$ can be deduced from the inference rules for usability degrees (*cf.* Table 2). By definition, only those deducible atoms are `true` for $\mathcal{G} \times \mathcal{W}$ that correspond to arcs omitted in the SDC graph. Hence, $cl^{\star}(A_{SDC})$ contains all atoms that are necessary to correctly describe all relationships among $\mathcal{G}$ and $\mathcal{W}$, therewith constituting the inferential completeness. The removal of a single arc from the SDC graph disables this property: the respective atom $d(x,y)$ does not exist any more in A_{SDC}, and thus the related deducible atoms $d^{\star}(x,y)$ that are necessary to assure the inferential completeness can not be determined by the inference rules any longer. This shows the minimality.

4.4.2. *Asymptotic Complexity*

We now analyze the structural complexity of the SDC graph. For this, we consider the insertion of goal templates as the most expensive operation: it requires a possibly complete traversal of the goal graph in order to allocate the new goal template at the proper position, and then the discovery cache must be updated (*cf.* Figure 6). All other management operations on SDC graphs are significantly less expensive. For the analysis, we consider the number of necessary matchmaking operations as the central indicator, independent of the specification language used for goal and Web service descriptions.

We can distinguish two situations for inserting a new Goal Template G_{new} with respect to the number of necessary matchmaking operations. The first one is when G_{new} becomes a new root node in the SDC graph. For this, we need to first ensure that there is no root node G_{root} in the SDC graph with $subsume(G_{root}, G_{new})$, because then G_{new} would become a child node of this. Then, we need to inspect all existing Web services for G_{new} using the root-node-discovery procedure explained above. The other situation is when G_{new} becomes a child node in the goal graph. This requires expectably less matchmaking effort, because we only need to inspect the Web services that are usable for the existing goal templates. The necessary steps for this are: (1) find the existing root node G_{root} where $subsume(G_{root}, G_{new})$, (2) allocate G_{new} in the subgraph of G_{root} by subsequently checking the similarity with the already existing child nodes, and (3) check the usability of those Web services $\mathcal{W}_{G_p} \subseteq \mathcal{W}$ that are usable under the *subsume* or the *intersect* degree for the goal templates G_p that now are parents of G_{new}.

The following defines this in terms of the O-notation, using standard notions from graph theory: the diameter of the goal graph $diam(SDC_{GG})$ is the maximal distance between a root and a leaf node, and the branching factor $b(SDC_{GG})$ is the maximal number of children of a goal template in the goal graph [8].

Proposition 4.1. *The computational costs for inserting a new goal template G_{new} into a SDC graph are*

- $O(|\mathcal{G}_{root}| + |\mathcal{W}|)$ *when G_{new} becomes a new root node of SDC*
- $O(|\mathcal{G}_{root}| + (diam(SDC_{GG}) * b(SDC_{GG})) + |\mathcal{W}_{G_p}|)$ *where* $(G_p, G_{new}) \in E_{sim}, \mathcal{W}_{G_p} = \{W \mid d(G_p, W) \in \{subsume, intersect\}\}$ *otherwise.*

This indicates that the complexity for creating and managing SDC graphs is relatively high. However, this is performed at design time, and thus does not hamper the runtime efficiency for solving goal instances. Note that the matchmaking is undecidable in general when using classical FOL as defined in Section 3. In [27], we define modelling restrictions for functional descriptions which ensure that the proof obligations are decidable as `NExpTime-complete` problems, which is complexity class of prominent ontology languages for the Semantic Web [7].

5. Optimized Runtime Discovery

This section presents the optimization of the runtime discovery task by exploiting the knowledge kept in the SDC graph. We specify the algorithms for this, analyze their complexity, and illustrate them in our example.

5.1. *Discovery Algorithm*

As outlined above, the approach for optimizing Web service discovery at runtime is to minimize the computational effort by exploiting the SDC graph. In particular, we

use the captured knowledge to (1) *reduce the search space* by considering the most specialized goal template because for this only the minimal number of Web services needs to be inspected, and (2) to *minimize the number of necessary matchmaking operations* by first inspecting Web services for which no matchmaking is required at runtime. Listing 1 shows how this is implemented in the algorithm for discovering one usable Web service for a given goal instance.

```
1  discoverSingleWS(GI(G,β)) {
2    if ( validInstantiation (GI(G,β))) = false ) then
3      return ' invalid goal instance ';
4    if ( lookup(G) = W ) then return W;
5    while ( (G,Gc) in goalgraph ) {
6      if ( validInstantiation (GI(Gc,β) ) then G = Gc;
7      forall ( (G,W) in discoverycache ) {
8        if ( degree(G,W) = (exact,plugin) ) then
9          return result = W; } }
10   if ( checkOtherWS(G,β)) = W ) then return W;
11   else { return "no_Web_service_found"; }
12 lookup(G) {
13   result = empty;
14   forall ( (G,W) in discoverycache ) {
15     if ( degree(G,W) = (exact,plugin) ) then return result = W;
16     else { forall ( Gp and (Gp,G) in goalgraph ) { lookup(Gp); } } }
17   if ( result = empty ) return result ; }
18 checkOtherWS(G,β)) {
19   result = empty;
20   forall ( (G,W) in discoverycache and degree(G,W) = subsume ) {
21     if ( satisfiable (W,β)) ) then return result = W; }
22   forall ( (G,W) in discoverycache and degree(G,W) = intersect ) {
23     if ( satisfiable (G,W,β)) ) then return result = W; }
24   if ( result = empty ) then return result ; }
```

Listing 1. Algorithm for SDC-enabled Runtime Discovery

The input of the main procedure is a Goal Instance $GI(G, \beta)$ for which a Web service shall be found. At first, we check if this is a valid instantiation of the corresponding goal template G (lines 2-3); this must be given to perform discovery by semantic matchmaking. Then, the algorithm consists of three methods with increasing matchmaking efforts that hence are invoked successively if the preceding one has not been successful.

The first one is the *lookup*-method (line 4). This searches for a Web service W that is usable for the given goal template G under the *exact* or the *plugin* degree: we know that this W is usable for $GI(G, \beta)$ without the need of matchmaking at runtime (*cf.* clauses 1 and 2 in Theorem 3.1). Lines 12-17 define the algorithm for finding such a W in the SDC graph. We first inspect the discovery cache arcs for the given goal template G. As soon as an arc annotated with *exact* or *plugin* is detected, the respective Web service is returned as the discovery result for $GI(G, \beta)$. If this is not successful, we continue the search for the direct parents G_p of G until reaching a root node of the SDC graph.

If *lookup* as the most efficient method is not successful, we continue with the *refinement*-method. This successively replaces the given goal template G with a child node G_c of which the goal instance is a valid instantiation, i.e. where $subsume(G, G_c)$ and $GI(G, \beta) \models G_c$ (lines 5-9). This reduces the relevant search

space, because fewer Web services are usable for G_c (*cf.* rule 5 in Table 2). By definition of the SDC graph, we need to consider only *one* G_c for which this is given: each pair (G_{c1}, G_{c2}) of direct children of G is *disjoint* unless there is an intersection goal template G^I for them (*cf.* clause 3 in Definition 4.1). In the latter case, G^I is found by traversing either the path $G \rightarrow G_{c1} \rightarrow G^I$ or the path $G \rightarrow G_{c2} \rightarrow G^I$ because $GI(G, \beta) \models G^I(G_{c1}, G_{c2})$ only if $GI(G, \beta) \models G_{c1}$ and $GI(G, \beta) \models G_{c2}$. We thus can traverse the goal graph until finding the most adequate goal template for the given goal instance. In each refinement step, we search for Web services that are usable for G_c under the *exact* or *plugin* degree with a reduced version of the *lookup*-method (lines 7-9).

If this should not be successful, we finally inspect those Web services that are usable for the (possibly refined) goal template under the *subsume* or the *intersect* degree. This requires additional matchmaking, and lines 18-24 show the algorithm that performs the necessary satisfiability tests (*cf.* clauses 3 and 4 of Theorem 3.1). A Web service for which this holds is immediately returned as the discovery result. If this is not successful, then then a Web service for solving $GI(G, \beta)$ does not exist.

The algorithm for finding all Web services works analogously: after validating the goal instantiation, we first refine the goal template towards the most specialized one, and then collect the results of the *lookup*- and the *checkOtherWS*-method as the discovery result.

5.2. *Complexity Analysis*

We now analyze the computational costs of the algorithms, which corresponds to the search complexity of the SDC graph. As above, we consider the number of required matchmaking operations as the main indicator.

The *lookup*-method does not require matchmaking, thus we disregard it here. The *refinement*-method traverses the goal graph in a depth-first manner until finding the most specialized goal template G', performing the goal instantiation check. The *checkOtherWS*-method needs to perform matchmaking for all Web services that usable for G' under the *subsume* or *intersect* degree.

Proposition 5.1. *The costs for finding a usable Web service W for a Goal Instance $GI(G, \beta)$ in a SDC graph are*

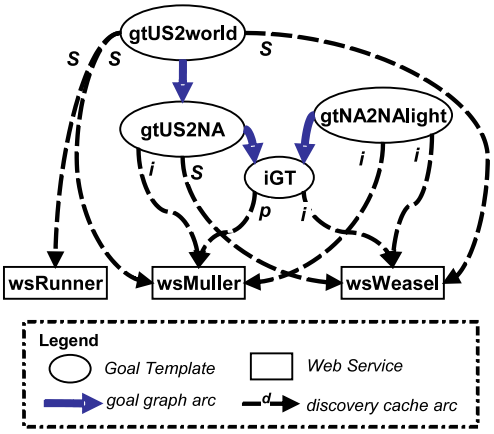
$$O(\ (diam(SDC_{GG}) * b(SDC_{GG})) + |\mathcal{W}_{G'}|\ ) \text{ where}$$
$$\mathcal{W}_{G'} = \{W \mid d(G', W) \in \{subsume, intersect\}\}$$

This indicates a high search complexity of the SDC graph in general. However, our hypothesis is that in most application scenarios the relationship of the actual goal templates and Web services allows the construction of SDC graphs where $diam(SDC_{GG}), b(SDC_{GG}) \ll |\mathcal{G}|$ and also $|\mathcal{W}_G| \ll |\mathcal{W}|$, so that a significant performance increase can be achieved with the SDC technique. We shall evaluate this below in Section 6.

5.3. *Illustrative Example*

In order to illustrate the above definitions, the following explains the creation of the SDC graph and the optimized runtime discovery for the shipment scenario introduced above. We here consider a condensed scenario setting that appears to be sufficient for demonstrating the operating principles of the SDC technique.

Figure 7 provides the relevant information for our discussion. Following the original scenario from [23], we consider three goal templates and three Web services for package shipment. The goal templates are (1) `gtUS2world` for shipping packages of any weight from the USA to anywhere in the world, (2) `gtUS2NA` for package shipment from the USA to North America, and (3) `gtNA2NAlight` for shipping light packages within North America. The Web services are (1) `wsMuller` that offers shipment from the USA to almost anywhere in the world for packages with a maximal weight of 50 kg, (2) `wsRunner` for shipping packages of any weight from the USA to all continents apart from North America and Africa, and (3) `wsWeasel` that offers shipment within the USA for packages of any weight. The functional descriptions of all resources are analog to the ones illustrated Figure 3, so that we can apply the semantic matchmaking techniques as defined in Section 3.3.



Goal Template	Sender	Receiver	Weight
gtUS2world	USA	world	heavy
gtUS2NA	USA	NA	heavy
gtNA2NAlight	NA	NA	light

Web Service	Sender	Receiver	Weight
Muller	USA	AF,EU,NA,AS	w50lq
Runner	USA	EU,AS,SA,OC	heavy
Weasel	USA	USA	heavy

Usability	wsMuller	wsRunner	wsWeasel
gtUS2world	subsume	subsume	subsume
gtUS2NA	intersect	X	intersect
gtNA2NAlight	intersect	X	subsume
iGT	plugin	X	intersect

Fig. 7. Overview of Illustrative Example

The SDC graph for this scenario is structured as follows: `gtUS2world` is a root node with `gtUS2NA` as a child, because this defines a proper specialization of the requested functionality. The similarity degree between `gtUS2world` and `gtNA2NAlight` is *intersect*: their common solutions are shipment orders for light packages from the USA to North America. However, the *insersectGT insertion* procedure (*cf.* Figure 6) allocates the resulting intersection goal template `iGT` as a common child of `gtUS2NA` and `gtNA2NAlight`, because their similarity degree is also *intersect* with exactly the same common solutions. The discovery cache captures the

design time discovery results. We observe that for the more general goal templates all Web services are merely usable under the *subsume* and *intersect* degree, while for the more specialized ones we also find *plugin* matches.

Let us now consider a concrete client request for shipping a package of 5.16 kg from San Francisco to New York City. This is described as a Goal Instance $GI(\texttt{gtUS2world}, \beta)$ with the input binding $\beta = (?s|sanFrancisco, ?r|newYorkCity, ?w|5.16\ kg)$. Here, `gtUS2world` is not the most appropriate one among the existing goal templates. We assume this to be a typical situation in real-world settings, e.g. when the request is created out of a client application that needs to support shipment to any location in the world.

The optimized runtime discovery algorithm from Listing 1 will find a usable Web service for this goal instance as follows. It commences with checking the goal instantiation condition: $GI(\texttt{gtUS2world}, \beta) \models \texttt{gtUS2world}$ is given because San Francisco is located in the USA, New York City is located in the world, and the package weight of 5.16 kg is included in the weight class `heavy`. Then, it tries to find a suitable Web service via the *lookup*-method. This is not successful because all Web services are only usable under the *subsume* degree. Hence, we continue with the *refinement*-method. In the first iteration, this will define `gtUS2NA` as the new goal template: this is a direct child of `gtUS2world` in the SDC graph, and $GI(\texttt{gtUS2world}, \beta) \models \texttt{gtUS2NA}$ is satisfied. Although this results in a reduction of the search space, the *lookup*-method for `gtUS2NA` is also not successful because the only existing usability degrees are *subsume* and *intersect*. Thus, we continue with the second iteration of the *refinement*-method, which defines the intersection goal template `iGT` as the new corresponding goal template. For this, `wsMuller` is usable under the *plugin* degree. This is detected by the *lookup*-method, and the algorithm returns this Web service as the discovery result for the Goal Instance $GI(\texttt{gtUS2world}, \beta)$. The algorithm for finding all usable Web services will also detect `wsWeasel` to be usable because it offers shipment within the USA for packages of any weight; for this, the *checkOtherWS*-method needs to be applied because the usability degree `wsWeasel` for `iGT` is *intersect*.

6. Evaluation

This section evaluates the SDC technique, in particular its suitability for optimizing automated Web service discovery. In order to quantify the achievable performance increase for runtime discovery as the time critical task, we compare our optimized discoverer with not optimized engines in larger application scenarios.

The following explains the experiment setup and presents the comparison test results. Afterwards, we discuss the practical relevance of our approach.

6.1. *Experiment Setup*

The overall aim of the SDC technique is to optimize automated Web service discovery at runtime, which is the time critical operation in our 2-phased framework

(*cf.* Section 2.2). Hence, the primary focus of this evaluation is to quantify the increase in the computational performance that can be achieved with the for the SDC-enabled runtime discovery presented above. For this, a sophisticated benchmarking technique does not exist, as present comparison measurements for comparing semantically enabled discovery merely focus on the retrieval accurcary but do not cover performance or scalability aspects (e.g. [19]). Also, a comparsion with other optimzation techniques for automated Web service discovery appears to not be feasible, because existing approaches commonly lack of retaining a high retrieval accuracy (see Section 7 for a detailed related work discussion).

In consequence, the following compares the performance of the SDC-enabled runtime discovery with not optimized runtime discovery engines along with a detailed examination on the impact of exploiting the SDC graph. The first comparison engine is a naive runtime discoverer that does not apply any optimization techniques. It retrieves the available Web services in a random order, and performs the basic matchmaking to determine their usability for a given goal instance (*cf.* Section 3.3.2). It uses the same matchmaker and infrastructure as the SDC engine. For the comparison test, we define a set of goal instances for the shipment scenario, and inspect the behavior of the engines for increasingly larger sets of available Web services. This test allows us to quantify the absolute performance increase, and we discuss this in terms of the following standard criteria: *efficiency* as the time required for completing a discovery task, *scalability* as the ability to deal with a large search space of available Web services, and *stability* as a low variance of the execu-

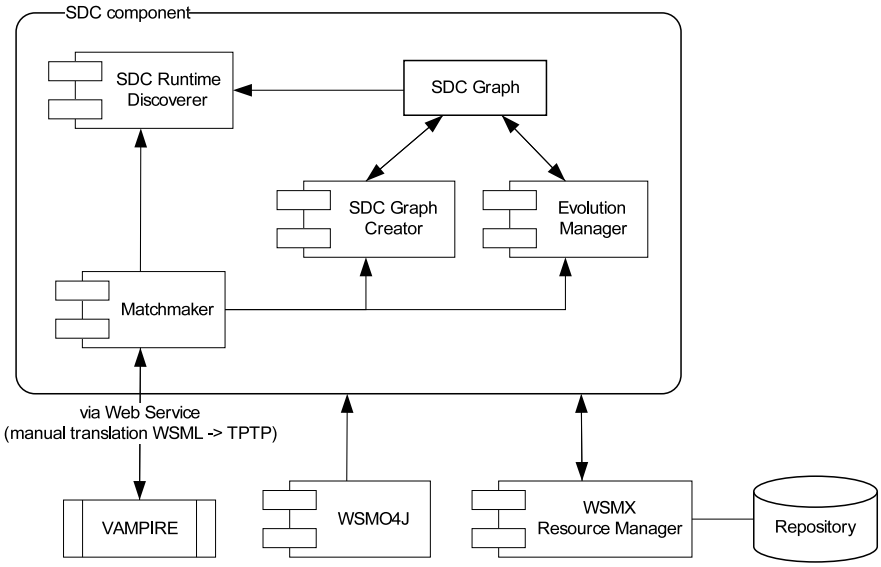


Fig. 8. SDC Prototype – Technical Architecture

tion time of several invocations [10]. In a second comparison test, we compare our engine with a runtime discoverer that implements our 2-phase framework but does not exploit the SDC graph for optimization. This allows us to evaluate the relative performance increase that can be achieved with the SDC technique. All implementations along with the original evaluation data and a detailed documentation are available online at `www.michael-stollberg.de/phd/`.

For conducting the evaluation test, we implemented all engines as discovery components in WSMX, the reference implementation of the WSMO framework (`www.wsmx.org`). Figure 8 shows the technical architecture of the SDC prototype that contains all components explained above: the `Matchmaker` implements all necessary semantic matchmaking techniques on the basis of VAMPIRE [25], a resolution-based automated theorem prover for classical first-order logic with equality that allows us to realize the matchmaking techniques exactly as specified in Section 3. The `SDC Graph Creator` and the `Evolution Manager` implement the management algorithms for SDC graphs (*cf.* Section 4.3), and the `SDC Runtime Discoverer` implements the algorithms for optimized runtime discovery (*cf.* Section 5).

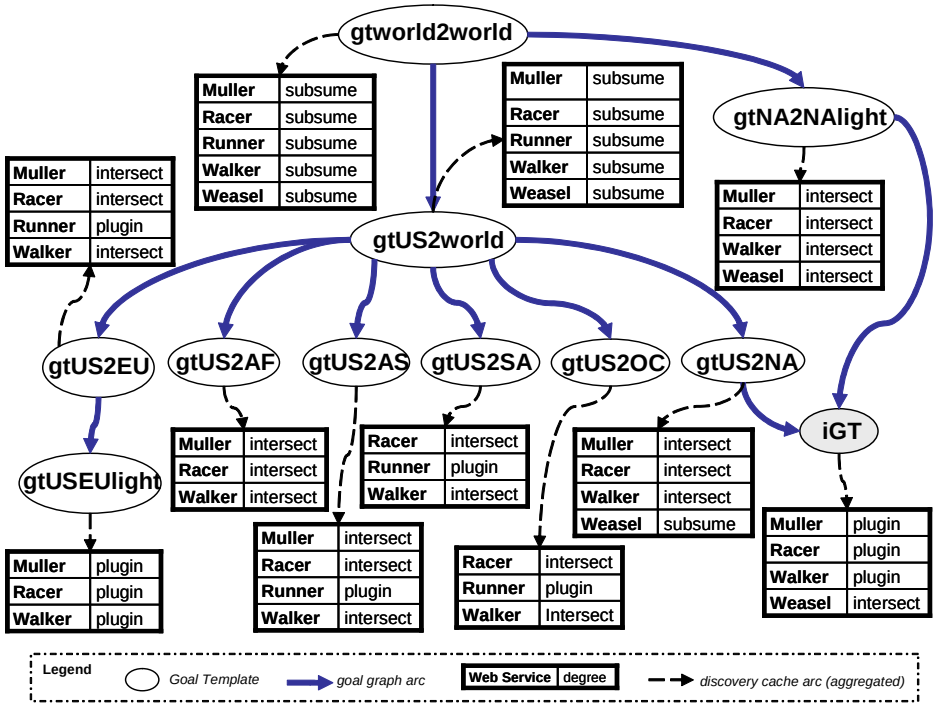


Fig. 9. SDC Graph for Shipment Scenario

6.2. *Use Case*

As the use case, we consider the shipment scenario that we have already discussed as the running example above. We here use the original data set as defined in [23]. This consists of 5 Web services for shipping packages: in addition those discussed in Section 5.3, `Racer` and `Walker` offer worldwide shipment of packages with a maximal weight of 70 kg, respectively 50 kg.

As the use case, we consider the shipment scenario that we have already discussed as the running example above. We here use the original data set as defined in [23]. This consists of 5 Web services for shipping packages: in addition those discussed in Section 5.3, `Racer` and `Walker` offer worldwide shipment of packages with a maximal weight of 70 kg, respectively 50 kg.

We define a set of goal templates, and create the SDC graph shown in Figure 9. Here, `gtworld2world` is the most general goal template and thus is the root node; `gtUS2world` is the same as defined above, and its direct children request package shipment from the USA to specific continents. The SDC graph shows properly defines all relevant relationships. Its creation with our prototype takes 48.11 seconds; the average time for inserting a single goal template is ca. 5 seconds, and 110 milliseconds for a single matchmaking operation.

For the comparison tests, we define a set of 10 goal instances as shown in Figure 10. Each goal instance is a valid instantiation of its corresponding goal template. The figure also enlists the usable Web services for each goal instance, representing the results of runtime discovery runs: these are identical for all compared engines.

Goal Instance	corresp. Goal Template	Input Binding			usable Web Services	
		sender	receiver	weight	name	total
gi1	gtUS2AF	San Francisco	Tunis	1 kg	Muller Racer Walker	3
gi2	gtworld2world	Los Angeles	Luxembourg City	1.5 kg	Muller Racer Runner Walker	4
gi3	gtUS2world	Berkley	Tunis	50.5 kg	Racer	1
gi4	gtUS2EU	Paolo Alto	Bristol	4.3 kg	Racer Runner	2
gi5	gtUS2NA	Los Angeles	New York City	5.5 kg	Muller Racer Walker Weasel	4
gi6	gtUS2world	Monterey	Berlin	60 kg	Runner Runner	1
gi7	gtUS2world	Santa Barbara	Sydney	17.3 kg	Racer Runner Walker	3
gi8	gtUS2SA	San Francisco	Quito	7.58 kg	Racer Runner Walker	3
gi9	gtUS2AS	Stanford	Beijing	57.8 kg	Racer Runner	2
gi10	gtUS2EUlight	San Francisco	Amsterdam	9.99 kg	Muller Racer Runner Walker	4

Fig. 10. Goal Instances for Comparison Tests

6.3. *Comparison Test Results*

The following presents and discusses the results of the two comparison tests. We here summarize the central findings, referring to [27] for further details.

6.3.1. *SDC vs. Naive Engine*

We commence with the comparison of the SDC runtime discoverer with the not optimized engine. Here, the main interest is the behavior of the engines within larger search spaces of available Web services. For this, we define incrementally larger search spaces (from 10 up to 2000 Web services): each set contains the 5 Web services from the scenario description, while all others do not offer package shipment and thus are not usable. In order to obtain statistically valuable results, we performed 50 repetitions of every discovery task. Table 3 shows the aggregated results of all test runs for the discovery of a single Web service.

We can make two important observations from this comparison. The first one is that an optimization of automated Web service discovery techniques appears to be necessary in order to warrant the operational reliability even in relatively small application scenarios. The naive engine requires between 0.1 and 26 seconds for a search space of 200 Web services (these are the actually measured minimal and maximal values) with a very high variance among the individual invocations. This performance appears to be not acceptable, in particular when considering industrial SOA systems that usually encompass more than 1000 Web services.

Table 3. Single WS Discovery – Aggregated Comparison Results(all values in seconds)

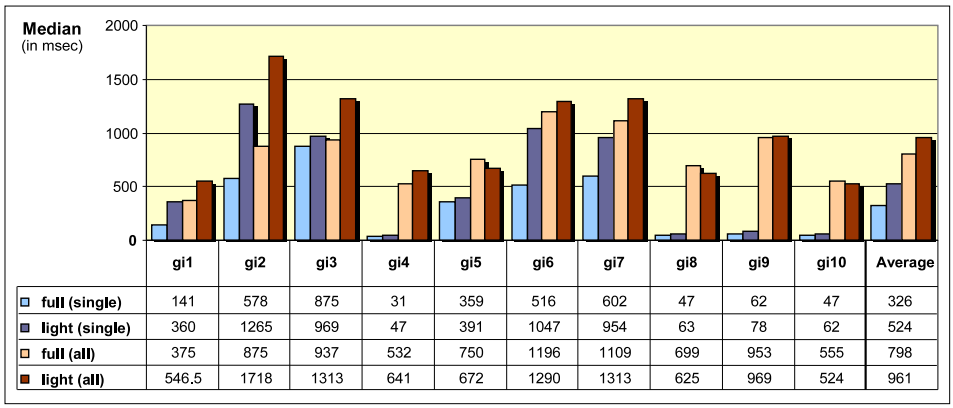
no. of WS	**Engine**	**Mean** μ	**Median** $\bar{x}$	**Standard Deviation** σ
10	SDC	0.28	0.27	0.03 (11.74 %)
	naive	0.41	0.39	0.21 (51.71 %)
50	SDC	0.28	0.28	0.09 (11.79 %)
	naive	2.00	1.79	1.29 (64.59%)
100	SDC	0.29	0.28	0.03 (11.53 %)
	naive	3.96	3.68	2.55 (64.48 %)
200	SDC	0.29	0.28	0.03 (11.51 %)
	naive	7.61	6.67	5.05 (66.35 %)
500	SDC	0.29	0.29	0.04 (13.42 %)
	naive	18.33	15.61	13.26 (72.34 %)
1000	SDC	0.29	0.29	0.04 (14.79 %)
	naive	37.69	33.22	26.28 (69.70 %)
2000	SDC	0.31	0.29	0.05 (18.03 %)
	naive	72.96	65.55	52.13 (71.45 %)

The second observation is that the SDC technique appears to be a sophisticated optimization technique. Considering the standard criteria, the results show that the SDC-enabled engine is sufficiently *efficient* because it performs the discovery task in 300 milliseconds in average, it warrants the *scalability* because the required times are independent of the search space size, and it exposes a high *stability* among several invocations with marginal variations in the actual processing times.

The comparison for the discovery of all Web services provides even more significant results. The SDC engine requires about 500 milliseconds, independent of the search space size. The naive engine always needs to inspect all available Web services, so that its processing times grow proportionally with the search space size.

6.3.2. *SDC vs. SDC_{light}*

In the above test, a considerable part of the achieved performance improvements result from our two-phased discovery model where only the usable Web services of the corresponding goal template need to be inspected as potential candidates for the given goal instance. Thus, the second comparison test focuses explicitly on the performance increase that can be achieved by exploiting the SDC graph. The comparison engine uses the design time discovery results but does not reduce the search space with the *refinement*-method from the optimized runtime discovery algorithm (*cf.* Listing 1).



	gi1	gi2	gi3	gi4	gi5	gi6	gi7	gi8	gi9	gi10	Average
full (single)	141	578	875	31	359	516	602	47	62	47	326
light (single)	360	1265	969	47	391	1047	954	63	78	62	524
full (all)	375	875	937	532	750	1196	1109	699	953	555	798
light (all)	546.5	1718	1313	641	672	1290	1313	625	969	524	961

Fig. 11. Performance Comparison SDC vs. SDC light

Figure 11 shows the comparison test results for both the discovery of a single and of all Web services. We observe that also here the full SDC-enabled engine is faster in average. However, there are cases where the search space reduction is more expensive than inspecting the candidates for the originally defined goal template (e.g. the discovery of all Web services for `gi5` in the test scenario). This effect disappears when more Web services are usable for the goal templates. Moreover, the perfor-

mance increase by omitting unnecessary matchmaking operations becomes more significant when dealing with more complex ontologies and functional descriptions for which the matchmaking is more expensive.

6.4. *Discussion*

The comparison tests reveal that the SDC technique can achieve substantial performance improvements. However, it adopts the concept of caching, and thus the enhancements for a specific application are dependent on the actual resources: the maximal optimization is achievable when there are several similar goal templates and many Web services with similar functionalities.

In order to determine the practical relevance, we conducted an explorative applicability study for the Verizon SOA system. Referring to [27] for details, this contains around 1500 Web services for homogenizing the data management among more than 650 distributed sales stores. The client applications use them for creating, modifying, and querying information on products, customers, and sales orders. The structure of these client requests is very similar, and the Web services mainly differ in the details of the input and output types. We can semantically describe these on the basis of an already existing business taxonomy. Then, we can construct an SDC graph that exposes a fine-grained subsumption hierarchy, and thus significant performance improvements can be achieved with the SDC technique. Similar improvements can be expected in other SOA applications, in particular in industrial solutions where larger numbers of similar requests and services need to be handled. Besides, the goal-based approach can enhance the system flexibility by overcoming the deficits of hard-wired service invocations.

This indicates that optimization of semantically enabled techniques for automated Web service discovery appears to be necessary for the employment in larger SOA systems that can be found in industrial settings, because the Verizon SOA system exceeds the dimension where non-optimzed discovery engines expose an acceptable performance (cf. Table 3 above). However, the general decision for employing SWS also determined by other factors, particularly regarding the business need for high-accuracy service discovery and the effort required of creating the richer semantic descriptions of the existing services and client requests.

7. Related Work

This section discusses related work. Although there is a wealth of insightful research on semantically enabled Web service discovery, we are not aware of any approach that addresses the performance challenge in a similar way. The following outlines the background of our approach and positions it within other works.

Our approach has been inspired by the WSMO framework that promotes goal-driven approach for Semantic Web Services [14]. We integrated the results of several associated works on automated Web service discovery, provided a general purpose formalization in first-order logic, and extended this with the caching mechanism

for optimization. Our prototype is implemented as a component of the reference implementation WSMX [13], and thus can be integrated with other related SWS environments, e.g. the IRS system that provides a goal-based broker for Semantic Web Services [9].

The first area of related work is automated Web service discovery by semantic matchmaking. This has been subject to several research efforts that provide valuable insights. Rooted in formal software specification (e.g. [33]), the matching degrees were already defined in early works (e.g. [20, 22]). However, these merely perform matchmaking on the separate elements of functional descriptions; the dependencies among the inputs, outputs, preconditions and effects are not considered. This has been addressed in later works, e.g. [18, 16]. Inspired by [17], our contribution to this is the semantic matchmaking techniques that work on sufficiently expressive functional descriptions and thus ensure a high retrieval accuracy for both design time and runtime discovery.

The second area is concerned with the optimization of automated Web service discovery. Although commonly considered to be essential for making SWS techniques applicable in real-world scenarios (e.g. [24, 4]), only a few works address this challenge. The majority follows the categorization approach already supported in UDDI: the available Web services are organized in hierarchical categorization schemes on the basis of ontologies; this is used to perform the pre-filtering of candidates which are then inspected by semantic matchmaking (e.g. [26, 32, 31, 1]). The central deficit is the imprecision of keyword-based annotations, which can lead to incorrect filtering results that conflict with the results of discovery by semantic matchmaking of rich functional descriptions. Thus, this approach is more problematic than ours in terms of ensuring a high retrieval accuracy.

An approach that leverages on formal functional descriptions to overcome the imprecision problem is presented in [5]. This defines a search tree where so-called *interval constraints* organize the provided functionalities in a subsumption hierarchy and the actual Web services are the leaf nodes. However, the interval constraints describe the inputs, outputs, preconditions, and effects in a disconnected manner. This is significantly less expressive than our functional descriptions, so that the index structure as well as the obtainable discovery results are less precise than with the SDC technique. Furthermore, the search tree needs to be traversed down to the leaf nodes in order to find a Web service. This could be enhanced by adapting our caching mechanism, which facilitates discovery by lookup without matchmaking.

8. Conclusions

This article has presented *Semantic Discovery Caching* (SDC) as a novel optimization technique for automated, semantically enabled Web service discovery. The aim is to enable the detection of suitable services for specific client requests with a high retrieval accuracy in a scalable manner, therewith facilitating the employment of Semantic Web Service techniques within larger SOA systems.

For this, we have proposed a two-phased discovery framework where client requests are described as goals that abstract from technical details. At design time, the suitable Web services for goal templates as generic and reusable descriptions of client objectives are discovered. The results are captured in a graph structure that organizes similar goal templates in subsumption hierarchies and keeps the relevant knowledge on the usable Web services in a redundancy-free manner. This is called the SDC graph, which serves as cache for the results of service discovery activities that can be performed at design time. At runtime, concrete client requests are described by goal instances that instantiate a goal template, and the discovery task is optimized by exploiting the knowledge kept in the SDC graph.

We have defined the relevant constructs and matchmaking techniques in a first-order logic framework, therewith allowing the applicability to various SWS frameworks that utilize higher-level Semantic Web languages. We further have formally defined the SDC graph along with the basic management techniques, and specified the algorithms for optimized runtime discovery. An empirical evaluation has shown that an optimization of semantically enabled service discovery techniques is necessary to allow the applicability in larger SOA systems, and that the SDC technique can achieve significant improvements in performance and scalability for this.

Acknowledgements

This work has been partially funded by the European Commission under the research projects SUPER, SHAPE, and INDENICA. The authors thank Uwe Keller and John Domingue for fruitful discussion and feedback.

References

[1] W. Abramowicz, K. Haniewicz, M. Kaczmarek, and D. Zyskowski. Architecture for Web services Filtering and Clustering. In *Proc. of the 2nd International Conference on Internet and Web Applications and Services (ICIW 2007), Mauritius*, 2007.

[2] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Data-Centric Systems and Applications. Springer, Berlin, Heidelberg, 2004.

[3] P. Bertoli, J. Hoffmann, F. Lecue, and M. Pistore. Integrating Discovery and Automated Composition: from Semantic Requirements to Executable Code. In *Proc. of the IEEE 2007 International Conference on Web Services (ICWS'07), Salt Lake City, USA*, 2007.

[4] J. Cardoso and A. Sheth. *Semantic Web Services, Processes and Applications*. Semantic Web and Beyond. Springer, 2006.

[5] I. Constantinescu, W. Binder, and B. Faltings. Flexible and Efficient Matchmaking and Ranking in Service Directories. In *Proc. of the 3rd International Conference on Web Services (ICWS 2005), Florida, USA*, 2005.

[6] J. de Bruijn. Logics for the Semantic Web. In J. Cardoses, editor, *Semantic Web: Theory, Tools and Applications*. Idea Publishing Group, 2006.

[7] J. de Bruijn and S. Heymans. Logical foundations of (e)RDF(S): Complexity and Reasoning. In *Proc. of the 6th International Semantic Web Conference (ISWC 2007, Seoul, Korea*, 2007.

[8] R. Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer, Heidelberg, 3. edition, 2005.
[9] 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. *Journal of Web Semantics*, 2008.
[10] C. Ebert, R. Dumke, M. Bundschuh, and A. Schmietendorf. *Best Practices in Software Measurement*. Springer, 2004.
[11] T. Erl. *Service-Oriented Architecture (SOA). Concepts, Technology, and Design*. Prentice Hall PTR, 2005.
[12] J. Farrell and H. Lausen. Semantic Annotations for WSDL and XML Schema. W3C Recommendation 28 August 2007, 2007. online: `http://www.w3.org/TR/sawsdl/`.
[13] D. Fensel, M. Kerrigan, and M. Zaremba. *Implementing Semantic Web Services - The SESA Framework*. Springer, 2008.
[14] D. Fensel, H. Lausen, A. Polleres, J. de Bruijn, M. Stollberg, D. Roman, and J. Domigue. *Enabling Semantic Web Services. The Web Service Modeling Ontology*. Springer, Berlin, Heidelberg, 2006.
[15] M. Hepp, F. Leymann, J. Domingue, A. Wahler, and D. Fensel. Semantic Business Process Management: A Vision Towards Using Semantic Web Services for Business Process Management. In *Proc. of the IEEE International Conference on e-Business Engineering (ICEBE 2005), October 18-20, 2005, Beijing, China*, 2005.
[16] D. Hull, E. Zolin, A. Bovykin, I. Horrocks, U. Sattler, and R. Stevens. Deciding Semantic Matching of Stateless Services. In *Proc. of the 21st National Conference on Artificial Intelligence (AAAI'2006)*, 2006.
[17] U. Keller, H. Lausen, and M. Stollberg. On the Semantics of Funtional Descriptions of Web Services. In *Proc. of the 3rd European Semantic Web Conference (ESWC 2006), Montenegro*, 2006.
[18] M. Kifer, R. Lara, A. Polleres, C. Zhao, U. Keller, H. Lausen, and D. Fensel. A Logical Framework for Web Service Discovery. In *Proc. of the ISWC 2004 workshop on Semantic Web Services: Preparing to Meet the World of Business Applications, Hiroshima, Japan*, 2004.
[19] U. Küster and B. König-Ries. Measures for Benchmarking Semantic Web Service Matchmaking Correctness. In *Proc. of the 7th ExtendedSemantic Web Conference (ESWC 2010), Heraklion, Crete, Greece*, June 2010.
[20] L. Li and I. Horrocks. A Software Framework for Matchmaking based on Semantic Web Technology. In *Proc. of the 12th International Conference on the World Wide Web, Budapest, Hungary*, 2003.
[21] D. Martin. OWL-S: Semantic Markup for Web Services. W3C Member Submission 22 November 2004, 2004. online: `http://www.w3.org/Submission/OWL-S/`.
[22] M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic Matching of Web Services Capabilities. In *Proc. of the 1st International Semantic Web Conference, Sardinia, Italy*, 2002.
[23] C. Petrie, T. Margaria, H. Lausen, and M. Zaremba (eds.). *Semantic Web Services Challenge*. Springer, 2009.
[24] C. Preist. A Conceptual Architecture for Semantic Web Services. In *Proc. of the 2nd International Semantic Web Conference (ISWC 2004)*, 2004.
[25] A. Riazanov and A. Voronkov. The Design and Implementation of VAMPIRE. *AI Communications*, 15(2):91–110, 2002. Special Issue on CASC.
[26] N. Srinivasan, M. Paolucci, and K. Sycara. Adding OWL-S to UDDI – Implementation and Throughput. In *Proc. of the First International Workshop on Semantic Web Services and Web Process Composition at the ICWS 2004, San Diego, Califor-*

nia, USA, 2004.

[27] M. Stollberg. *Scalable Semantic Web Service Discovery for Goal-driven Service-Oriented Architectures*. PhD thesis, Semantic Technology Institute, University of Innsbruck, Austria, 2008.

[28] M. Stollberg, M. Hepp, and J. Hoffmann. A Caching Mechanism for Semantic Web Service Discovery. In *Proc. of the 6th International Semantic Web Conference (ISWC 2007), Busan, Korea*, 2007.

[29] M. Stollberg, U. Keller, H. Lausen, and S. Heymans. Two-phase Web Service Discovery based on Rich Functional Descriptions. In *Proc. 4th European Semantic Web Conference (ESWC 2007), Innsbruck, Austria*, 2007.

[30] K. Sycara, M. Paolucci, A. Ankolekar, and N. Srinivasan. Automated Discovery, Interaction and Composition of Semantic Web services. *Journal of Web Semantics*, 1(1):27–46, September 2003.

[31] B. Tausch, C. d'Amato, S. Staab, and N. Fanizzi. Efficient Service Matchmaking using Tree-Structured Clustering. In *Poster at the 5th International Semantic Web Conference (ISWC 2006), Athens, Georgia (USA)*, 2006.

[32] K. Verma, K. Sivashanmugam, A. Sheth, A. Patil, S. Oundhakar, and J. Miller. METEOR-S WSDI: A Scalable P2P Infrastructure of Registries for Semantic Publication and Discovery of Web Services. *Journal of Information Technology and Management*, 6(1):17–39, 2005.

[33] A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.