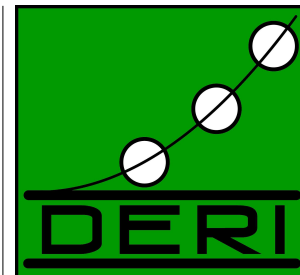


DERI – DIGITAL ENTERPRISE RESEARCH INSTITUTE



SEMANTIC DISCOVERY CACHING (SPECIFICATION)

Michael Stollberg

DERI TECHNICAL REPORT 2007-02-03

FEBRUARY 2007

DERI – DIGITAL ENTERPRISE RESEARCH INSTITUTE

DERI Galway
University Road
Galway, Ireland
www.deri.ie

DERI Innsbruck
Technikerstrasse 21a
Innsbruck, Austria
www.deri.at

DERI Korea
Yeonggun-Dong, Chongno-Gu
Seoul, Korea
korea.deri.org

DERI Stanford
Serra Mall
Stanford, USA
www.deri.us

SEMANTIC DISCOVERY CACHING:
CAPTURING AND REUSE OF WEB SERVICE DISCOVERY RESULTS
FOR IMPROVING THE COMPUTATIONAL EFFICIENCY OF
SERVICE-ORIENTED ARCHITECTURES

Michael Stollberg¹

Abstract. This technical report provides the specification of *Semantic Discovery Caching* (short: SDC). As a part of the author's PhD work, this is a technique for enhancing the efficiency and scalability of Web service discovery processes in service-oriented architectures. The approach is to capture discovery results for goals (formalized client requests) and, on this basis, perform Web service discovery for new, semantically similar requests. This allows to (1) attain the highest possible efficiency via *discovery-by-lookup*, meaning to perform Web service discovery without invoking a matchmaker, and (2) to increase scalability by *reducing the search space for Web service discovery* on the basis of clustering Web services with respect goals that can be solved with them. As a continuation of previous works, this document specifies the central constructs and operations of the SDC technique. Future research will present a prototype implementation within WSMX along with a demonstration throughout an illustrative example, and evaluate the SDC technique by discussing its applicability in real-world SOA applications.

Keywords: Service-oriented Architectures, Semantic Web Services, Web Service Discovery, Semantic Matchmaking, Goals, Goal-oriented Architectures, Efficiency, Scalability, Semantic Caching, Goal Graph

¹Digital Enterprise Research Institute (DERI) Innsbruck, University of Innsbruck, Technikerstraße 21a, A-6020 Innsbruck, Austria. eMail: michael.stollberg@deri.org.

Acknowledgements: This material is based upon works supported by the EU under the DIP project (FP6 - 507483). The author dedicates special thanks to Martin Hepp and Jörg Hoffmann for valuable discussion on the presented work. Besides, the work has greatly benefited from interesting discussions with Michael Genesereth (Stanford University).

Copyright © 2007 by the authors

Contents

1	Introduction	1
1.1	Context – A Goal-based Approach for Semantic SOA	2
1.2	Two-phase Web Service Discovery	4
1.2.1	Concepts and Approach	4
1.2.2	Formal Functional Descriptions	5
1.2.3	Semantic Matchmaking	7
2	Requirements Analysis	10
2.1	Web Service Discovery	10
2.2	Efficiency and Scalability	12
2.2.1	Computational Efficiency for Web Service Discovery	13
2.2.2	Scalability for Web Service Discovery	14
2.3	Goal-Based Discovery Caching	14
2.3.1	Similarity of Goals	15
2.3.2	Goal-based Index of Web Services	15
2.4	Operations and Technical Integration	16
3	The SDC Graph	18
3.1	Definition	18
3.1.1	Goal Template Similarity	18
3.1.2	Elements and Structure	20
3.2	Inference Rules for Web Service Usability Degree	22
3.3	Formal Properties and Refinement	24
3.3.1	Initial SDC Graph	24
3.3.2	Resolution of Intersect Arcs in Goal Graph	27
3.3.3	Illustrative Example	32
3.3.4	Formal Properties of Refined SDC Graph	33
4	Web Service Discovery with SDC	36
4.1	Goal Instance Formulation and Goal Template Discovery	37
4.1.1	Matchmaking for Goal Template Search	38
4.1.2	Algorithm for Goal Template Search	39
4.2	Web Service Discovery – Goal Template Level	41
4.2.1	Matchmaking for Web Service Usability Degree Determination	41
4.2.2	Algorithm for Discovering All Usable Web Services for a New Goal Template	43
4.3	Web Service Discovery – Goal Instance Level	46
4.3.1	Matchmaking and Procedure	46
4.3.2	Algorithm for Runtime Web Service Discovery	47
4.3.3	Computational Efficiency Analysis	49
5	SDC Graph Maintenance	51
5.1	Iterative Creation of the SDC Graph	51
5.1.1	Algorithm for Goal Template Insertion	52
5.1.2	Example	58
5.2	Evolution in Dynamic Environment	59
5.2.1	Goal Template Removal and Update	59
5.2.2	Changes on Available Web Services	62
5.3	Advanced Management	65
5.3.1	Goal Template Learning	65
5.3.2	Integration with Non-Functional Discovery	66

5.3.3 SGC Graph Clearing	66
6 Summary and Future Work	67
REFERENCES	71
APPENDIX	72
A Matching Degrees Overview	72
B Proof of Theorem 3.1: Inference Rules for Usability Degrees	74
C Complete SDC Algorithm	77

List of Figures

1	WSMO Discovery Framework	2
2	Realization of WSMO Discovery Framework	3
3	Illustration of $W \models_{\mathcal{A}} \mathcal{D}$	6
4	Abstract Architecture for Semantic Web Services	10
5	Goal Instance Resolution Procedure	12
6	SDC Allocation in SWS Environments	16
7	Example of an SDC graph	21
8	Initial Structure of the Goal Graph	25
9	Examples for Cycles and I-arc Concatenations in Initial Goal Graph	26
10	Disconnected Sub-Graphs in Goal Graph	26
11	Resolution of Intersect Arcs in the Goal Graph	27
12	Resolution of Undesirable Situations in the Goal Graph	29
13	Resolution of Cycles in the Goal Graph	31
14	Example for Resolving a Cycle in the Initial Goal Graph	32
15	Illustration of Goal Template Search	34
16	Omittance of WG Mediators in Discovery Cache	35
17	Operations for Web Service Discovery with SDC	37
18	Illustration of Goal Formulation Procedure	38
19	Runtime Operations for SDC-enabled Web Service Discovery	47
20	Illustrative Example for Runtime Web Service Discovery	49
21	Possible Situations for New Root Node Insertion	53
22	Possible Situations for Insertion of a New Child Node	55
23	Example for Iterative SDC Graph Creation	58

List of Tables

1	Definition of Matching Degrees for $\mathcal{D}_G, \mathcal{D}_W$	7
2	Overview of Requirements for Semantic Discovery Caching	17
3	Definition and Meaning of Goal Similarity Degrees	19
4	Goal Templates, Similarity Degree, and Intersection Goal Templates in Example	32
5	Definition and Relationship of Usability Degrees	42
6	Definition of Matching Degrees for $\mathcal{D}_1, \mathcal{D}_2$	72
7	Meaning of Matching Degrees for the Usability of a Web service for solving a Goal	73
8	Meaning of Matching Degrees for Semantic Similarity of Goals	73
9	Syntax for Pseudo Code used in Algorithm Definitions	77

1 Introduction

This technical report specifies *Semantic Discovery Caching*, short: SDC, a technique for capturing and reuse of Web service discovery results. This shall allow to decrease the computational costs of Web service discovery procedures and thereby enhance the efficiency and scalability of service-oriented architectures (SOA). Embedded in a goal-based approach for Semantic Web services, the SDC technique captures knowledge on discovered Web services for generic, reusable goal descriptions and enables efficient runtime discovery. This technique is a central part of the author's PhD work.

The aim of the SDC technique is to decrease the size of the search space for Web service discovery and to enhance its runtime efficiency. We therefore create an index of available Web services with respect to the goals that can be solved by them. The starting point for this is the semantic similarity of goal descriptions. Two goals are considered to be similar if they have at least one common solution. Thus, in the majority of cases, the same Web services can be used to solve them. We distinguish two notions of goals: a *goal template* is the generic description of a client objective that is defined at design time and kept in the system; a *goal instance* denotes a concrete client request that is created at runtime by instantiating a goal template with concrete input values. Because of their formal relationship, it always holds that only those Web service usable for a goal template are possibly usable for any of its goal instances. The indexing structure for clustering Web services with respect to the solvable goals is the so-called *SDC graph*. It consists of a set of trees wherein the inner nodes are goal templates and the leaf nodes are the Web services usable for the goal template at the parent node. The arcs are directed connections between the nodes that define the similarity between goal templates, respectively the usability of a Web service, in terms of the matching degree between their formal functional descriptions. This is the minimal knowledge relevant for decreasing the search space and increasing the runtime efficiency of Web service discovery.

The SDC technique is allocated in a SOA system as an intermediate for Web service discovery mechanisms to access and search Web service repositories. Usable Web services for goal instances and for semantically similar goal templates are detected on the basis of knowledge in the SDC graph. This is a novel approach that introduces the concept of semantic caching into the area of semantic SOA technology. By nature, the achievable increase for efficiency and scalability of Web service discovery processes is dependent on the number and relationship of Web services and goals in concrete SOA applications. Our hypothesis is that those situations wherein the SDC technique can achieve an adequate efficiency increase correlate with the most common situation in typical real-world SOA applications.

This report provides the complete specification and evaluation of the SDC technique. A detailed evaluation in form of a real-world applicability study as well as detailed discussion of related work will be addressed in a later stage of research. The document is structured as follows. The remainder of this section briefly recalls the research context. Then, Section 2 discusses the design and determines the arising requirements for the SDC technique. Section 3 defines the elements of the SDC graph and discusses its formal properties. Then, Section 4 specifies the Web service discovery operations that work on the SDC graph, and Section 5 defines the algorithms for maintenance of the SDC graph in its dynamically changing environment. Finally, Section 6 summarizes the report.

1.1 Context – A Goal-based Approach for Semantic SOA

The SDC technique is allocated in the Web service discovery part of a goal-based approach for semantically enabled SOA technologies that is promoted by the Web Service Modeling Ontology WSMO (*cf.* www.wsmo.org). The overall aim is to enable automated discovery, composition, and execution of Web services with semantic technologies for realizing the vision of service-oriented architectures [10].

In contrast to most other approaches for Semantic Web services (e.g. OWL-S [25], SWSF [3], or WSDL-S [1]), WSMO does not only provide an ontology-based description model for Web services but integrates *goals* and *mediators* as additional top level elements. Their intended usage is:

goal-driven Web service usage: a client shall formulate the objective to be achieved in terms of a goal, and a WSMO-enabled system solves this by automatically discovering, composing, and executing appropriate Web services on basis of formal, declarative descriptions. The aim is to enable problem-oriented usage of Web services: the client can concentrate on the problems to be solved while all details on the automated usage of Web services are handled by the system.

mediation-enabled Web service usage: to establish interoperability between Web services and goals if this is not given a priori, mediators connect potentially heterogeneous elements and apply semantically enabled techniques for handling and resolving mismatches on the data and the process level [36].

One of the central reasoning tasks in Semantic Web services is discovery, commonly understood as the detection of those Web services out of the available ones that can be used for solving a given goal. Adopted from the heuristic classification problem solving method, WSMO proposes a discovery procedure as shown in Figure 1 (taken from [17]). The framework distinguishes the following elements: a *goal* is an abstract, reusable description of a client objective; a *Web service* is a software artifact that has an abstract description and provides access to real-world *services* that can solve a goal. The first process is *goal discovery*, which is concerned with formulating a client objective in terms a goal. This is achieved by associating the concrete client desire with an generic, reusable goal description. The second process is *Web service discovery*, which is understood as the detection of usable Web services for solving a goal by semantic matchmaking of their abstract descriptions with a primary focus on the provided and requested functionality. Finally, *refinement* is concerned with determining those real-world service that are associated with a discovered Web service and can be used to solve the client's desire. This encompasses compatibility tests for all non-functional aspects, such as behavioral conformance, quality-of-service, and financial aspects.

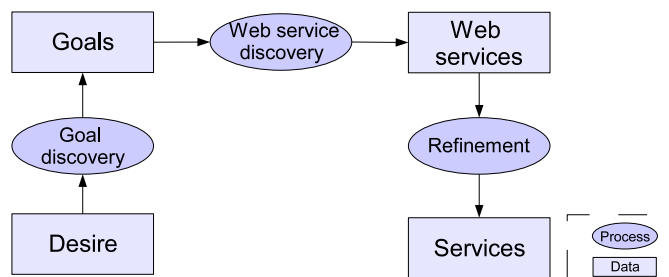


Figure 1: WSMO Discovery Framework

The PhD work of the author presents one possible realization of this abstract framework with a special focus on the quality and efficiency of Web service discovery. As conceptually compatible extensions, the central contributions of the work are:

1. **A refined goal model:** WSMO initially describes goals by the same description model as used for Web services. A refinement towards a more accurate model on the basis of experiences from respective development efforts is presented in [42]. The central extension is the differentiation of *goal templates* and *goal instances*: the former are generic objective descriptions that are defined at design time and kept in the system; the latter denote concrete client requests that are created at runtime by instantiating a goal template. A goal template described the requested functionality in terms of preconditions and effects and optionally a desired workflow that shall be sustained during the goal resolution. A client interface provides the counterpart of the Web service interface for invocation and consumption of its functionality (i.e. the choreography interface in WSMO terms). A goal instances instantiates a goal template by defining an input binding, i.e. an assignment of concrete values for the required inputs.
2. **Two-phase Web Service with sophisticated semantic matchmaking:** a semantically enabled Web service discovery has been developed. This defines semantic matchmaking for discovery on the goal template and the goal instance level, and integrates both into a two-phased Web service discovery for the extended goal model. Following the WSMO approach, this focusses on formally described functionalities requested in goals and provided by Web services. The complete discovery approach is presented in [41], along with a detailed report in [39]. Section 1.2 recalls the central definitions that are relevant in the context of this report.
3. **The Semantic Discovery Caching (SDC) technique:** This technique captures Web service discovery results on the goal template level, and utilizes this knowledge to enhance the efficiency of the discovery process. While [35] discusses the problem statement, the overall research approach, and related work, the SDC technique is specified and evaluated in this report.

Figure 2 shows the refinement of the WSMO discovery framework with the three extensions. The goal discovery process is replaced by the creation of goal instances: the client browses existing goal templates, and formulates the objective to be achieved by defining concrete input values for the goal description. This can be supported by graphical user interfaces as provided in IRS [6] or SWF [43], which eases the goal formulation for clients. Web service discovery is performed in a two-phased manner: at design, usable Web services for goal templates are determined; usable Web services for a concrete goal instance are determined at runtime, whereby the discovery result for the corresponding goal template serves as a pre-filter. The SDC technique therefore captures the relevant knowledge for enabling efficient runtime discovery.

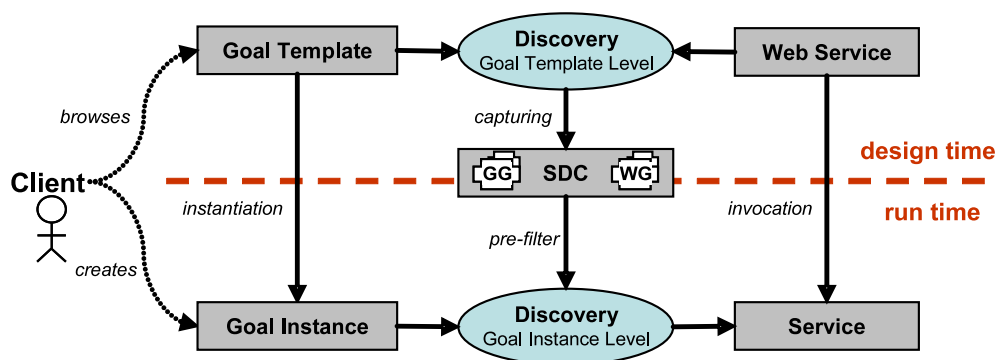


Figure 2: Realization of WSMO Discovery Framework

1.2 Two-phase Web Service Discovery

In order to provide a self-contained documentation, the following briefly recalls the central aspects and definitions of the two-phased Web service discovery with goal templates and goal instances. We refer to [41] for a comprehensive presentation, and to [39] for a detailed report on the Web service discovery realization.

1.2.1 Concepts and Approach

The discovery approach is defined in *Abstract State Spaces* (short: ASS) [18]. This defines a state based model of the world that Web services act in with precisely defined formal semantics. Therein, a particular execution of a Web service or a composition of Web services denotes a sequence of state transitions $\tau = (s_0, \dots, s_m)$, i.e. a change of the world from a start state s_0 to a termination state s_m that is triggered by the invocation with concrete inputs. The overall functionality provided by a Web service is the set of all its possible executions; we denote this as $\{\tau\}_W$.

A goal is the formal description of the desire of the client to get from the current state of the world into a state wherein the objective is satisfied. A goal template specifies conditions on the start state and the desired final state wherein the objective is considered to be solved. Hence, a goal template \mathcal{G} formally describes a set of possible solutions that we denote as $\{\tau\}_{\mathcal{G}}$. At runtime, a client creates a goal instance $GI(\mathcal{G})$ by assigning concrete values to the input variables defined in the corresponding goal template \mathcal{G} . We refer to this as an input binding β , so that a goal instance formally is a pair $GI(\mathcal{G}) = (\mathcal{G}, \beta)$ and its possible solutions are a subset of those for \mathcal{G} , i.e. $\{\tau\}_{GI(\mathcal{G})} \subset \{\tau\}_{\mathcal{G}}$. The input binding β defined in $GI(\mathcal{G})$ subsequently constitutes the concrete input values for invoking a Web service in order to solve the goal instance by a real-world service.

The aim of Web service discovery is find a Web service that can provide a τ that is a solution for the goal. We thus specify the meaning of a match for Web service discovery as follows.

Definition 1.1 (Meaning of a Match). *Let W be a Web service, \mathcal{G} a goal template, and $GI(\mathcal{G})$ a goal instance that instantiates \mathcal{G} with an input binding β . Let $\tau = (s_0, \dots, s_m)$ be a sequence of states in an Abstract State Space \mathcal{A} .*

We define the following sets:

$$\begin{array}{llll} \{\tau\}_{\mathcal{G}} & := & \text{possible solutions for } \mathcal{G} & \{\tau\}_{GI(\mathcal{G})} \subset \{\tau\}_{\mathcal{G}} & := & \text{possible solutions for } GI(\mathcal{G}) \\ \{\tau\}_W & := & \text{possible executions of } W & \{\tau\}_{W(\beta)} \subset \{\tau\}_W & := & \text{possible executions of } W \\ & & & & & \text{when invoked with } \beta \end{array}$$

We define the usability of a Web service for solving a goal as:

- (i) $match(\mathcal{G}, W) : \exists \tau. \tau \in (\{\tau\}_{\mathcal{G}} \cap \{\tau\}_W)$
- (ii) $match(GI(\mathcal{G}), W) : \exists \tau. \tau \in (\{\tau\}_{GI(\mathcal{G})} \cap \{\tau\}_{W(\beta)})$

This defines the basic matching conditions for Web Service discovery. Clause (i) states that a Web service W is usable for solving a goal template \mathcal{G} if there exists at least one execution of W that is a possible solution for \mathcal{G} . Clause (ii) defines that W is usable for solving a goal instance $GI(\mathcal{G})$ if there is at least one execution of W that is also a solution for $GI(\mathcal{G})$ when W is invoked with the inputs defined in $GI(\mathcal{G})$. Moreover, the following holds because of $\{\tau\}_{GI(\mathcal{G})} \subset \{\tau\}_{\mathcal{G}}$:

1. $match(GI(\mathcal{G}), W) \Rightarrow match(\mathcal{G}, W)$, i.e. a Web service that is usable for solving a goal instance is also usable for the corresponding goal template, and, as the logical complement

2. $\neg match(\mathcal{G}, W) \Rightarrow \neg match(GI(\mathcal{G}), W)$, i.e. that a Web service that is not usable for a goal template is also not usable for any of its goal instances. This constitutes the foundation of our two-phase discovery approach, because we can use the Web service discovery result on the goal template level as a pre-filter for the goal instance level discovery.

We define semantic matchmaking techniques to evaluate the matching conditions from Definition 1.1 on the basis of formal descriptions of goals and Web services. Without such techniques, we would need to perform test runs of a Web service for determining its usability for solving a goal. We focus on the requested and provided functionalities. This is widely considered as the primary aspect of interest for Web service discovery; other aspects such as behavioral conformance test, quality-of-service, financial, and the non-functional context are dealt with in subsequent usability tests [29]. Due to the high precision and recall rates that are achievable with matchmaking on formal functional descriptions, this replaces keyword-based discovery techniques. Subsequently, the SDC technique replaces Web service repository categorization by an index structure of the available Web services with respect to the goals that can be solved by them.

1.2.2 Formal Functional Descriptions

A functional description formally describes the overall functionality provided by a Web service, respectively possible solutions of a goal. This serves as the basis for semantic matchmaking of requested and provided functionalities. We apply functional descriptions as defined in the Abstract State Space model (ASS) mentioned above, which specifies them on the level of state changes and defines precise formal semantics for such functional descriptions [18].

An Abstract State Space \mathcal{A} is defined over a signature Σ and some domain knowledge Ω . A functional description is described as a 5-tuple $(\Sigma, \Omega, IF, \phi^{pre}, \phi^{eff})$. The signature Σ differentiates *static symbols* Σ_S that are not changed, *dynamic symbols* Σ_D that are changed by execution of a Web service, and Σ_D^{pre} that denote the interpretation of a dynamic symbol in the start state. Preconditions ϕ^{pre} and effects ϕ^{eff} are defined as statements in a logic $\mathcal{L}(\Sigma)$. $IF = (i_1, \dots, i_n)$ is a set of variables that denote all required inputs. To explicitly specify the deterministic dependency between the start- and end-states with respect to input values, they can occur as the only free variables in ϕ^{pre} and ϕ^{eff} . An input binding $\beta : \{i_1, \dots, i_n\} \rightarrow \mathcal{U}_{\mathcal{A}}$ is a total function that assigns objects of the universe of \mathcal{A} to each IF -variable. Finally, the symbol *out* denotes the computational outputs that are constrained by ϕ^{eff} .

The meaning of a functional description is defined with respect to the start- and the end-state of a sequence of state transitions. Formally, a $\tau = (s_0, \dots, s_m)$ in \mathcal{A} is considered to satisfy the described functionality if and only if it holds that if $s_0 \models_{\mathcal{L}(\Sigma)} \phi^{pre}$ then $s_m \models_{\mathcal{L}(\Sigma)} \phi^{eff}$. Here, $s \models_{\mathcal{L}(\Sigma)} \phi$ expresses that the formula ϕ is satisfied by the universe $\mathcal{U}_{\mathcal{A}}$ in a state s under the logic $\mathcal{L}(\Sigma)$. We refer to this as *implication semantics*: if the precondition is satisfied in s_0 , then s_m will satisfy the effect; otherwise, we can not make any statement about the behavior of the described functionality. Because the IF -variables occur as free variables in both the precondition ϕ^{pre} and the effect ϕ^{eff} , the end-state s_m is completely dependent on the start-state s_0 . This reflects the deterministic nature of functionalities provided by Web services.

While functional descriptions in the ASS model are defined independent of the specification language for preconditions and effects, we use classical first-order logic (FOL, [33]) for illustration throughout this work. In order to ease the handling of functional descriptions, we describe them as a first-order logic structure that maintains the formal semantics as defined in the ASS model. Definition 1.2 specifies the structure of a functional description, and Definition 1.3 defines its formal meaning for describing the overall functionality of a Web service.

Definition 1.2 (Functional Description). A functional description is a 4-tuple $\mathcal{D} = (\Sigma, \Omega, IF, \phi^{\mathcal{D}})$ such that:

- (i) Σ is a signature consisting of Σ_S (static symbols), Σ_D (dynamic symbols), and Σ_D^{pre} (pre-variants of dynamic symbols)
- (ii) $\Omega \subseteq \mathcal{L}(\Sigma)$ defines consistent domain knowledge
- (iii) IF is a set of variables i_1, \dots, i_n that denote all required input values; an input binding $\beta : \{i_1, \dots, i_n\} \rightarrow \mathcal{U}_A$ is a total function that assigns objects of the universe of A to each IF -variable
- (iv) $\phi^{\mathcal{D}}$ is a FOL formula of the form $[\phi^{pre}]_{\Sigma_D^{pre} \rightarrow \Sigma_D} \Rightarrow \phi^{eff}$ such that
 - ϕ^{pre} is the precondition with IF as the only free variables
 - ϕ^{eff} is the effect with IF as the only free variables and the outputs denoted by the predicate out
 - $[\phi]_{\Sigma_D^{pre} \rightarrow \Sigma_D}$ is the formula ϕ' derived from ϕ by replacing every dynamic symbol $\alpha \in \Sigma_D$ by its corresponding pre-variant $\alpha_{pre} \in \Sigma_D^{pre}$.

Definition 1.3 (Formal Semantics of a Functional Description). Let W be a Web service with $\{\tau\}_W$ as the set of its possible executions in an Abstract State Space \mathcal{A} . Let $\mathcal{D} = (\Sigma, \Omega, IF, \phi^{\mathcal{D}})$ be a functional description. Let $\Omega_A = \Omega \cup [\Omega]_{\Sigma_D^{pre} \rightarrow \Sigma_D}$ be the domain knowledge extended with $\alpha_{pre} \in \Sigma_D^{pre}$.

W provides the functionality described by \mathcal{D} , denoted by $W \models_{\mathcal{A}} \mathcal{D}$, if and only if:

- (i) every Σ -interpretation I with $I \models \Omega_A$ and $I, \beta \models \phi^{\mathcal{D}}$ under every input binding $\beta : IF \rightarrow \mathcal{U}_A$ represents a $\tau \in \{\tau\}_W$, and
- (ii) every $\tau \in \{\tau\}_W$ is represented by a Σ -interpretation I with $I, \beta \models \phi^{\mathcal{D}}$ and $I \models \Omega_A$ under every input binding $\beta : IF \rightarrow \mathcal{U}_A$.

This defines that a Web service W provides the functionality described by \mathcal{D} if and only if every Σ -interpretation I, β that is a model of $\phi^{\mathcal{D}}$ describes a $\tau = (s_0, \dots, s_m) \in \{\tau\}_W$. Such a Σ -interpretation describes the objects that exists in the end-state s_m if W is executed for a particular input binding β in a specific start state s_0 . For the implication semantics from clause (iv) in Definition 1.2, it holds that $I, \beta \models \phi^{\mathcal{D}}$ if $I, \beta \models \phi^{pre}$ and $I, \beta \models \phi^{eff}$; if $I \not\models \phi^{pre}$, we can not make any statement about the end-state of a τ . Hence, if a $\tau \in \{\tau\}_W$ can be described by a Σ -interpretation I with $I, \beta \models \phi^{\mathcal{D}}$, then it satisfies the described functionality; if there is a $\tau \in \{\tau\}_W$ that cannot be described by such a Σ -interpretation, then W does not provide the described functionality. Figure 3 illustrates this, while we refer to [39] for the formal explanation of this definition and its relationship to the ASS model.

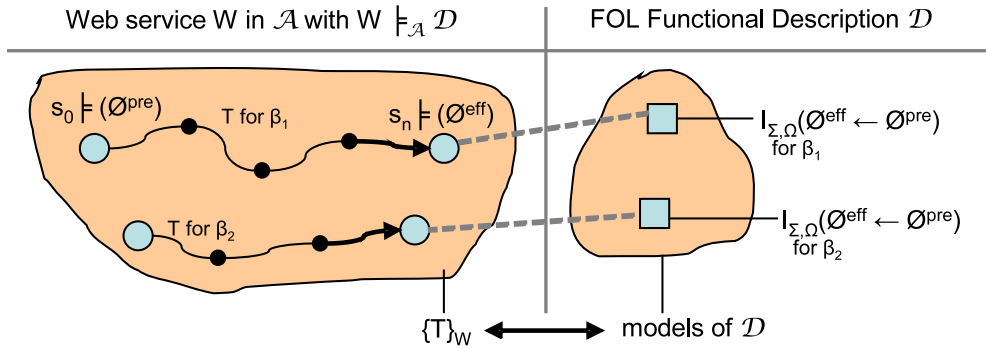


Figure 3: Illustration of $W \models_{\mathcal{A}} \mathcal{D}$

The meaning of a functional description \mathcal{D}_G of a goal template \mathcal{G} is analogous. Here, $\{\tau\}_G$ is the set of sequences of state transitions that are solutions for \mathcal{G} such that every $\tau \in \{\tau\}_G$ corresponds to a Σ -interpretation that is a model of \mathcal{D}_G . To precisely evaluate the usability of a Web service, in some cases we need to consider the concrete value assignments for the *IF*-variables. These are provided by the creation of a goal instance $GI(\mathcal{G})$ that defines an input binding β for the *IF*-variables in \mathcal{D}_G of the corresponding goal template \mathcal{G} . Subsequently, this β constitutes the inputs for invoking a Web service in order to solve $GI(\mathcal{G})$. We discuss this in more detail below in the context of discovery on the goal instance level.

1.2.3 Semantic Matchmaking

The following recalls the specification of the matchmaking techniques for Web service on the goal template, on the goal instance level, and their integration for the two-phased discovery approach outlined above. We defines the techniques on functional descriptions and input bindings as specified above, which provide sufficiently rich descriptions of possible Web service executions and possible solution for goals. We refer to [41] and [39] for more exhaustive explanations and illustrative examples.

Goal Template Level Discovery.

We express the usability of a Web service W for solving a goal template \mathcal{G} in terms of matching degrees. The distinct degrees denote specific relationships between the possible executions $\{\tau\}_W$ of W and possible solutions $\{\tau\}_G$ for \mathcal{G} . Four degrees – *exact*, *plugin*, *subsume*, *intersect* – denote different situations wherein the matching condition in clause (i) of Definition 1.1 is satisfied; the *disjoint* degree denotes that this is not given. In our two-phase discovery, these matching degrees serve as a pre-filter for determining the usability of a Web service W for solving a goal instance $GI(\mathcal{G})$ that instantiates the goal template \mathcal{G} .

We define the criteria for each degree over \mathcal{D}_G and \mathcal{D}_W from Definition 1.2, along with an explicit quantification of input bindings β . As the condition for the *exact* degree, $\Omega_A \models \forall \beta. \phi^{\mathcal{D}_G} \Leftrightarrow \phi^{\mathcal{D}_W}$ defines that every possible execution of W is a solution for \mathcal{G} and vice versa. We assume that all functional descriptions \mathcal{D} are consistent, i.e. that there exists a Σ -interpretations I under a β that is a model of $\phi^{\mathcal{D}}$. Representing a refinement of the matching degree definitions from [17], we therewith obtain a precise means for differentiating the usability of a Web service on the goal template level. Table 1 provides a concise compilation of the matchmaking degree definitions.

Table 1: Definition of Matching Degrees for $\mathcal{D}_G, \mathcal{D}_W$

Denotation $\mathcal{D}_G = (\Sigma, \Omega, IF, \phi^{\mathcal{D}_G})$ $\mathcal{D}_W = (\Sigma, \Omega, IF, \phi^{\mathcal{D}_W})$	Definition $\beta : IF \rightarrow \mathcal{U}_A$ $\phi^{\mathcal{D}} = [\phi^{pre}]_{\Sigma_D^{pre} \rightarrow \Sigma_D} \Rightarrow \phi^{eff}$ $\Omega_A = \Omega \cup [\Omega]_{\Sigma_D^{pre} \rightarrow \Sigma_D}$	Meaning for $\{\tau\}_G, \{\tau\}_W$ with $W \models_A \mathcal{D}_W$
exact ($\mathcal{D}_G, \mathcal{D}_W$)	$\Omega_A \models \forall \beta. \phi^{\mathcal{D}_G} \Leftrightarrow \phi^{\mathcal{D}_W}$	if and only if $\tau \in \{\tau\}_G$ then $\tau \in \{\tau\}_W$
plugin ($\mathcal{D}_G, \mathcal{D}_W$)	$\Omega_A \models \forall \beta. \phi^{\mathcal{D}_G} \Rightarrow \phi^{\mathcal{D}_W}$	if $\tau \in \{\tau\}_G$ then $\tau \in \{\tau\}_W$
subsume ($\mathcal{D}_G, \mathcal{D}_W$)	$\Omega_A \models \forall \beta. \phi^{\mathcal{D}_G} \Leftarrow \phi^{\mathcal{D}_W}$	if $\tau \in \{\tau\}_W$ then $\tau \in \{\tau\}_G$
intersect ($\mathcal{D}_G, \mathcal{D}_W$)	$\Omega_A \models \exists \beta. \phi^{\mathcal{D}_G} \wedge \phi^{\mathcal{D}_W}$	there is a τ such that $\tau \in \{\tau\}_G$ and $\tau \in \{\tau\}_W$
disjoint ($\mathcal{D}_G, \mathcal{D}_W$)	$\Omega_A \models \neg \exists \beta. \phi^{\mathcal{D}_G} \wedge \phi^{\mathcal{D}_W}$	there is no τ such that $\tau \in \{\tau\}_G$ and $\tau \in \{\tau\}_W$

Goal Instance Level Discovery.

A goal instance $GI(\mathcal{G})$ is created by defining an input binding β for the IF -variables in the functional description $\mathcal{D}_{\mathcal{G}}$ of the corresponding goal template \mathcal{G} . Formally, an input binding $\beta : \{i_1, \dots, i_n\} \rightarrow \mathcal{U}_{\mathcal{A}}$ is a total function that defines a variable assignment over the universe $\mathcal{U}_{\mathcal{A}}$ for the input variables IF defined in a functional description \mathcal{D} (cf. Definition 1.2). We therewith obtain an assignment of concrete values v for all inputs required in \mathcal{D} , i.e. $\beta = \{i_1|v_1, \dots, i_n|v_n\}$. Given such a β , we can instantiate \mathcal{D} by substituting all IF -variables that occur as free variables in ϕ^{pre} and ϕ^{eff} by the concrete values defined in β . We obtain $[\mathcal{D}]_{\beta}$ as the functional description that is instantiated for the context of β ; this can be evaluated as it longer contains any free variables. By instantiating $\mathcal{D}_{\mathcal{G}}$ with the input binding β defined in $GI(\mathcal{G})$, we obtain $[\mathcal{D}_{\mathcal{G}}]_{\beta}$ as the functionality requested by $GI(\mathcal{G})$; for the functional description \mathcal{D}_W of the Web service W , we obtain $[\mathcal{D}_W]_{\beta}$ as the functionality that can be provided by W when it is invoked with β .

Recalling from clause (ii) of Definition 1.1, a match on the goal instance level is given if there exists a $\tau = (s_0, \dots, s_m)$ in \mathcal{A} that is a solution for $GI(\mathcal{G})$ and can be provided by a Web service W when it is invoked with the concrete input values defined in $GI(\mathcal{G})$. To determine this on basis of the given descriptions, it must hold that – with respect to the domain knowledge – there exists a Σ -interpretation I that is a common model for $\phi^{\mathcal{D}_{\mathcal{G}}}$ and $\phi^{\mathcal{D}_W}$ when both functional descriptions are instantiated with the input binding β defined in $GI(\mathcal{G})$. Formally, this means that the union of the formulae $\Omega_{\mathcal{A}} \cup \{[\phi^{\mathcal{D}_{\mathcal{G}}}]_{\beta}, [\phi^{\mathcal{D}_W}]_{\beta}\}$ must be satisfiable, i.e. that there exists a Σ -interpretation that is a model for the extended domain knowledge $\Omega_{\mathcal{A}}$ and for the instantiated goal description $[\phi^{\mathcal{D}_{\mathcal{G}}}]_{\beta}$ and for the instantiated Web service description $[\phi^{\mathcal{D}_W}]_{\beta}$. In accordance to Definition 1.3, this I represents a τ that is a solution for $GI(\mathcal{G})$ and can be provided by W if it is invoked with β .

Definition 1.4 (Semantic Matchmaking on the Goal Instance Level). Let $\mathcal{D}_{\mathcal{G}} = (\Sigma, \Omega, IF_{\mathcal{G}}, \phi^{\mathcal{D}_{\mathcal{G}}})$ be a functional description of a goal template \mathcal{G} . Let $GI(\mathcal{G})$ be a goal instance that instantiates \mathcal{G} with the input binding $\beta : IF_{\mathcal{G}} \rightarrow \mathcal{U}_{\mathcal{A}}$. Let $\mathcal{D}_W = (\Sigma, \Omega, IF_W, \phi^{\mathcal{D}_W})$ be a functional description, and let $W = (IF, \iota)$ be a Web service with $W \models_{\mathcal{A}} \mathcal{D}_W$.

$match(GI(\mathcal{G}), W)$ is given if there exists a Σ -interpretation I such that:

$$I \models \Omega_{\mathcal{A}} \quad \text{and} \quad I \models [\phi^{\mathcal{D}_{\mathcal{G}}}]_{\beta} \quad \text{and} \quad I \models [\phi^{\mathcal{D}_W}]_{\beta}.$$

Integration for Two-Phase Discovery.

To attain an integrated matchmaking framework for our two-phase Web service discovery, we finally combine the semantic matchmaking techniques for the goal template and the goal instance level. We therefore extend matchmaking degrees from Table 1 with the matchmaking condition for the goal instance level. Due to their definition, we can simplify the matching condition from Definition 1.4 for the distinct matchmaking degrees as follows.

Definition 1.5 (Integrated Matchmaking for Two-Phase Web Service Discovery). Let $\mathcal{D}_{\mathcal{G}}$ describe the requested functionality in a goal template \mathcal{G} . Let $GI(\mathcal{G})$ be a goal instance of \mathcal{G} that defines an input binding β . Let W be a Web service, and let \mathcal{D}_W be a functional description such that $W \models_{\mathcal{A}} \mathcal{D}_W$.

W is usable for solving $GI(\mathcal{G})$ if and only if:

- (i) $exact(\mathcal{D}_{\mathcal{G}}, \mathcal{D}_W)$ or
- (ii) $plugin(\mathcal{D}_{\mathcal{G}}, \mathcal{D}_W)$ or
- (iii) $subsume(\mathcal{D}_{\mathcal{G}}, \mathcal{D}_W)$ and $\bigwedge \Omega_{\mathcal{A}} \wedge [\phi^{\mathcal{D}_W}]_{\beta}$ is satisfiable, or
- (iv) $intersect(\mathcal{D}_{\mathcal{G}}, \mathcal{D}_W)$ and $\bigwedge \Omega_{\mathcal{A}} \wedge [\phi^{\mathcal{D}_{\mathcal{G}}}]_{\beta} \wedge [\phi^{\mathcal{D}_W}]_{\beta}$ is satisfiable.

This specifies the minimal matchmaking conditions for determining the usability of a Web service for solving a concrete client request that is described by a goal instance. Under both the *exact* and the *plugin* degree, W can be used for solving any goal instance $GI(\mathcal{G})$ because $\{\tau\}_{GI(\mathcal{G})} \subset \{\tau\}_{\mathcal{G}} \subseteq \{\tau\}_W$ and $\tau \in \{\tau\}_{GI(\mathcal{G})} \Leftrightarrow \tau \in \{\tau\}_{W(\beta)}$. Under the *subsume* degree it holds that $\{\tau\}_{\mathcal{G}} \supseteq \{\tau\}_W$, i.e. every execution of W can solve \mathcal{G} but there can be solutions of \mathcal{G} that cannot be provided by W . Hence, W is only usable for solving $GI(\mathcal{G})$ if the input binding β defined in $GI(\mathcal{G})$ allows to invoke W . This is given if there is a Σ -interpretation that is a model for $[\phi^{\mathcal{D}_W}]_{\beta}$ and the conjunction of the axioms in $\Omega_{\mathcal{A}}$. Under *intersect* as the weakest degree, the complete matchmaking condition for the goal instance level must hold because there can be solutions for \mathcal{G} that can not be provided by W and vice versa. The *disjoint* degree denotes that W is not usable for solving the goal template and thus neither for any of its instantiations. We refer to [39] for the formal proof of this definition.

2 Requirements Analysis

This section determines the requirements that arise for the Semantic Discovery Caching technique (SDC). We commence the discussion with general aspects on Web service discovery that are relevant in this context, and then discuss the requirements for achieving a computationally efficient and scalable discovery procedure. On the basis of this, we determine the requirements on the constructs and operations for goal-based discovery caching with respect to the approach undertaken by the SDC technique. Finally, Table 2 summarizes the determined requirements in a concise overview.

2.1 Web Service Discovery

Figure 4 shows the overall procedure for solving a client request that is formulated as a goal by the use of Web services. In particular, it shows the central reasoning steps for automatically detecting and executing Web service on the basis of comprehensive descriptions. This procedure reflects the abstract architecture for Semantic Web services proposed in [29], and, in particular, denotes an abstraction of the workflow supported by WSMO-enabled environments for Semantic Web services such as WSMX [50] and IRS [6].

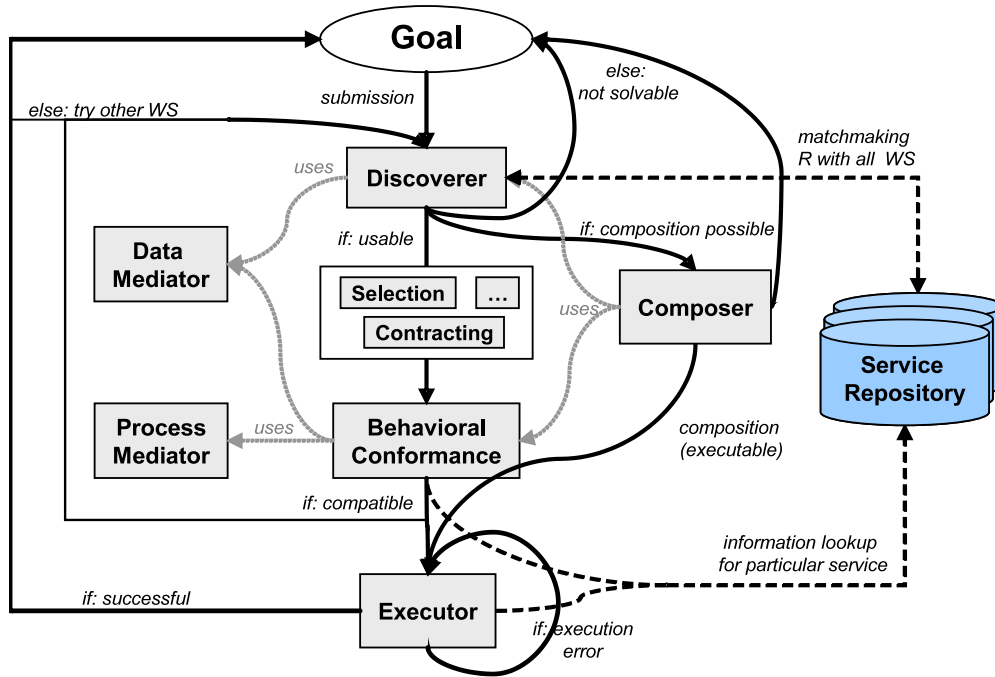


Figure 4: Abstract Architecture for Semantic Web Services

At first, potentially usable Web services are detected out of the available ones. This is performed by discovery or by composition in case no directly usable Web service exists. This is followed by optional steps for refining the detection result, such as selecting the most appropriate Web service out of the usable ones or contracting for determining details on the provided functionality. Then, the behavioral compatibility for successful interaction between the request and the discovered or composed Web services is tested. Mediation techniques for handling possibly occurring mismatches can be utilized as auxiliary facilities [37]. Finally, automated execution of the detected or composed Web services results in resolution of the goal.

The procedure in Figure 4 considers the matchmaking of formally described requested and provided functionalities as the primary aspect for Web service discovery; other aspects on the usability and of a Web service are detected in subsequent test. This conception is widely accepted – within the WSMO discovery framework [17] as well as other approaches (e.g. [28, 23, 4, 16]) – because it allows to more precisely determine the usability of a Web service than keyword-based matchmaking. Throughout this work, we focus on discovery of *directly usable Web services*, i.e. to find one Web service that can solve a given goal. We therefore apply the description for goals and Web service as well as the semantic matchmaking techniques as specified in Section 1.2. Another scenario that might apply matchmaking of formal functional descriptions is the candidate detection for Web service composition (i.e. to find those Web services out of the available ones out of which a composition shall be constructed). However, this merely requires slightly different definitions of the matchmaking techniques (for further discussion see [35, 44, 9]).

At run time, i.e. for solving a concrete goal, the discovery only needs to find one directly usable Web service (i.e. that satisfies the matching conditions). Once such a Web service has been found, the subsequent steps from Figure 4 can be performed; the discovery of further usable Web services can be continued in the background. This is a central difference to search techniques in other areas: for example in data bases, usually the answer to for a query is only considered to be complete if it contains all knowledge items out of the stored ones that satisfy the query statement [45]. In contrast, for Web service discovery we can continue the goal resolution procedure as soon as one usable Web service has been found. Hence, the first requirement for the SDC technique is to interleave Web service discovery with the subsequent goal resolution steps.

Requirement 1 (Interleaved Web Service Discovery). *The discovery of directly usable Web services at runtime only needs to find ONE Web service W that satisfies the matching condition for a given goal G . Once such a W has been found, the subsequent reasoning steps for usability determination and execution can be performed for solving G ; the discovery of further usable Web services can be performed orthogonal to the resolution of G by W .*

Although we primarily consider Web service discovery as the suitability of the provided functionality for solving the requested one, also other aspects are relevant. In particular, these are (1) weighting and selection of Web services with respect to non-functional aspects such as quality-of-service [47], financial and locality [19]; (2) dynamic details on the provided functionality that is not covered by the overall functional description, e.g. whether a retailer provides the specific product that the client asks for; this is commonly referred to as contracting [29, 22]; (3) the behavioral conformance, i.e. whether the client can provide a compatible counterpart for the interface of the Web service for invoking and consuming its functionality [34]. While we consider all these aspects to be checked after the Web service discovery on functional aspects, the relevant information should be available in the goal and Web service descriptions.

Requirement 2 (Support for Non-Functional Discovery). *Next to the formally described requested and provided functionality, the goal and Web service descriptions should contain information for other aspects relevant for determining the usability of a Web service for solving a goal, in particular quality-of-service, financial, and locality aspects, support for contracting, and behavioral aspects.*

We shall not further elaborate on this requirement, as it is not primarily relevant in the context of SDC. However, it is met by our approach as follows. Goal templates carry a formally described requested functionality (cf. Definition 1.2), client policies and preferences on quality-of-service, financial, and locality aspects, and client interfaces for automated invocation and consumption of Web services [42]. For Web services, we adopt the WSMO description model that consists of a capability (functional description), non-functional properties, and the choreography interface [30].

Moreover, the distinction of goal templates and goal instances that underlies this work allows to define a resolution procedure for goal instances that encompasses all aspects for Web service discovery. This differentiates two branches as shown in Figure 5. The first one encompasses the discovery operations for goal templates that are performed *orthogonal to runtime*: at first, matchmaking of formal functional descriptions is performed; then, the set of discovered Web services is weighted and reduced with respect to non-functional aspects, and finally the set is again reduced with respect to behavioral conformance test. The runtime branch encompasses the operations for creation and resolution of a goal instance. At first, the client browses existing goal templates, chooses one for that is appropriate for formulating the objective that shall be achieved, and creates a goal instance by instantiating the input values required in the goal template. Then – in our two-phase discovery approach (cf. Section 1.2) – the Web service discovery on the goal instance is performed for the set of Web services that have been discovered for the corresponding goal template. Once a usable Web service has been found, it is invoked and executed for solving the goal instance.

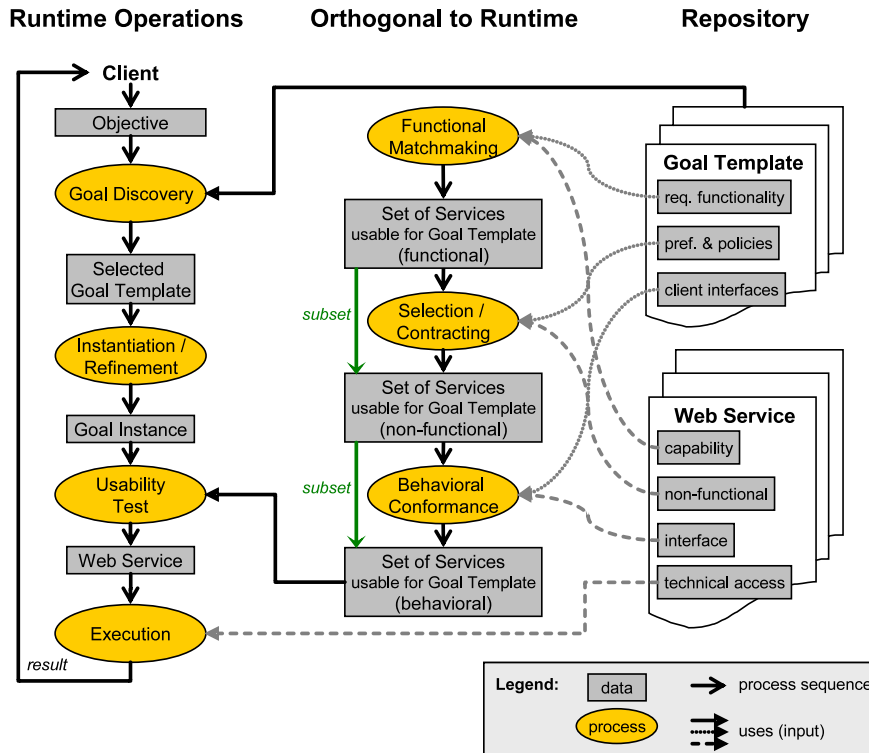


Figure 5: Goal Instance Resolution Procedure

2.2 Efficiency and Scalability

We now turn towards the *computational efficiency* (the costs of a system for performing an operation, in this context: solving of a client request) and *scalability* (the operational reliability with respect to the expected amount of resources in its designated area, here: the ability to deal with a very large number of Web services). These are important factors for technology adaptation in real world applications. Especially, these can be considered to be critical for the success of semantically enabled SOA technology that expectably needs to deal with several millions of Web services.

Examining the overall procedure for automatically detecting and executing Web service for solving a given goal from Figure 4 reveals that the bottleneck for both efficiency and scalability is Web service discovery. As the first processing step, this needs to take all available Web services into consideration. All subsequent steps need to deal with a significantly smaller amount of Web services – in fact those determined by matchmaking of formal functional descriptions as illustrated in Figure 5. Hence, if we can increase the efficiency and scalability of Web service discovery, we consecutively can increase the efficiency and scalability of semantically enabled SOA technology; this is the overall aim of the author’s PhD work [35]. The following discusses the arising requirements for achieving an efficient and scalable Web service discovery.

2.2.1 Computational Efficiency for Web Service Discovery

In computational theory (e.g. [32]), efficiency is concerned with desirable properties of algorithms or computer systems apart from functionality and technical design. It is described by two properties: the *speed* refers to the time it takes for an operation to complete, which is commonly described by the *Big-O* notation as a time complexity measurement [21]; the *space* property refers to the memory or non-volatile storage used up by the algorithm or system, measured in terms of the amount of persistent and working memory required at compile time as well as at runtime. Naturally, adequate optimization techniques for efficiency are highly dependent on the system design and functionality.

In our context, the computational efficiency is mainly related to the runtime branch in Figure 5. As the most critical aspect of efficiency for technology acceptance by end-users, we understand the speed of Web service discovery as the *time needed for finding one Web service that can be directly used for solving a given goal instance*. The first two steps in the runtime branch – goal template discovery and instantiation – require interaction of the client with the system. Thus, their speed is not only dependent on the computational efficiency of the supporting technology. The critical operation is the automated detection of a Web service that can be used for solving the created goal instance, which is performed by semantic matchmaking as specified in Section 1.2. While the computational costs for individual matchmaking operations can – in theory – be optimized to a negligible extend [12, 15], the aspect that hampers the time efficiency of Web service discovery is the size of the search space, i.e. the number of available Web services that need to be taken into consideration for matchmaking.

The efficiency of Web service discovery is proportional to the size of the search space: the smaller the number of Web services that need to be matched with the given goal description, the faster Web service discovery can be completed. Hence, the efficiency for discovery on the goal instance level is proportional to the number of usable Web services for the corresponding goal template. We can confer the same principle to the discovery of the goal template level that is performed at design time, respective orthogonal to runtime: the speed of Web service discovery for a new goal template can be increased if we can infer from the semantic similarity degree between an existing goal template and the new one that only a subset of the available Web services is relevant. While this conception is mainly related to the speed property of efficiency, we shall consider the space property in more detail below in the context of scalability.

Requirement 3 (Efficiency of Web Discovery). *The efficiency of Web service discovery denotes the time needed for finding one Web service that can be directly used for solving a given goal. It is proportional to the size of the search space, i.e. the number of Web services that need to be matched with the goal description. The efficiency for Web service discovery on both the goal template level (design time) and on the goal instance level (runtime) is optimal if the search space is minimal.*

2.2.2 Scalability for Web Service Discovery

Scalability is another desirable property of algorithms or computer systems, concerned with the ability to handle the large, growing number of available resources in a graceful manner [5]. This is a pre-requisite for the operational reliability of a system: if it can not handle the amount of resources in its designated application area, then it can not be considered to be functional for its purpose. However, due to the high dependence of a system's design and its usage environment, commonly accepted measurement and analysis techniques do not exist.

In the context of SOA, scalability refers to the ability to handle the very potentially large number of Web service that are distributed among the Web and that change dynamically [13]. For Web service discovery as the first processing step that needs to consider all available Web services, scalability relates to the programmatic management of semantic matchmaking. The critical aspect is the scalability of the used reasoning infrastructure. As analyzed in [48], this is hampered by (1) the general complexity of logical reasoning in comparison to conventional technologies, and (2) that most reasoner implementations keep all relevant knowledge in the working memory, which limits the number of processable resources tremendously.

In order to perform a matchmaking operation between a goal and a Web service, all related knowledge must be made available to the underlying reasoning infrastructure. In particular, this is the formal functional descriptions of the goal and the Web service as well as all background ontologies that are used in the functional descriptions. Thus, in order to maintain the operational reliability of Web service discovery for a large, dynamically changing number of available Web services, the invocation of the matchmaker should be decoupled from the resource management such that for each matchmaking operation only the minimal knowledge is loaded into the working memory of the matchmaker.

Requirement 4 (Scalability of Web Service Discovery). *A Web service discovery is scalable if it maintains its operational reliability for a large, dynamically changing number of available Web services. To achieve this, the programmatic management must ensure that only the minimal knowledge needed to perform the matchmaking (i.e. functional descriptions and background ontologies) is loaded into the working memory.*

2.3 Goal-Based Discovery Caching

While the preceding elaborations have discussed general requirements, we now turn towards specific ones that arise for the planned realization of the SDC technique. As outlined above, the approach is to capture knowledge on discovery results on the goal template level and utilize this to enhance the efficiency of Web service discovery. This technique reveals two properties:

1. it provides an index of Web services that is constituted by the similarity of goal templates. In contrast to existing approaches that cluster Web services with respect to the provided functionalities (e.g. [46, 8]), this index organizes Web services with respect to the goal templates that can be solved by them.
2. on the basis of this index, it reduces the search space for Web service discovery. For discovery on the goal instance level (runtime) only those Web services need to be considered that have been discovered for the corresponding goal instance; for the goal template level (design time), in most cases the same Web services are usable for adjacent goal templates in the index.

Therewith, SDC allows to perform Web services in terms of a caching technique that finds answers to requests from an intermediate store of answers to previous, similar request. The concept of caching is a optimization technique that is successfully applied in several areas that need to deal with a large amount of

information, e.g. in hardware optimization [14], in databases for efficient query answering [27], and for efficient traffic management on the Web [49]. Under certain circumstances – if there are many similar requests – caching can achieve the best efficiency in comparison to other performance optimization techniques.

The following discusses the requirements for the indexing structure while we address the requirements for the respective operations in the next section.

2.3.1 Similarity of Goals

The first requirement for creating a sophisticated index of available Web services with respect to the goals that can be solved by them is the definition of an appropriate measurement for the semantic similarity of goal templates. As the constituting concept for Web service indexing, the purpose is to declare goal templates to be similar such that the set of Web services that are usable for solving them is overlapping to a high degree. If this is given, then the search space for Web service discovery on the goal template level can be reduced by inferring the usability of a Web services with respect to the semantic similarity of adjacent goal templates. As we shall elaborate below, this similarity of goal templates can most adequately be expressed in terms of the matching degrees between their formal functional descriptions.

We do not need to declare the similarity of goal instances because only those Web services that are usable for the corresponding goal template are potential candidates. We therewith already have a sophisticated pre-filter for Web service discovery on the goal template level as discussed above in Section 1.2.3.

Requirement 5 (Goal Similarity Measure). *The notion of semantic similarity of goal templates is the constituting concept for creating an index of Web services with respect to the goals that can be solved by them. Given two goal templates \mathcal{G}_1 and \mathcal{G}_2 with $\{W\}_{\mathcal{G}_1}$ and $\{W\}_{\mathcal{G}_2}$ as the set of usable Web services for them, the similarity between \mathcal{G}_1 and \mathcal{G}_2 should be defined such that $\{W\}_{\mathcal{G}_1} \cap \{W\}_{\mathcal{G}_2}$ is maximal.*

2.3.2 Goal-based Index of Web Services

The second requirement is concerned with the formal properties of the indexing structure. This consists of the goal templates and the Web services that are usable for them. The goal templates are organized with respect to their similarity. The connections between goal templates can be defined as directed edges, so that we obtain a tree of goal templates as the foundation of the indexing structure. Each goal template is connected to the Web services that can be used to solve it, so that Web services denote the leaf nodes of the goal tree. We shall refer to this structure as the *SDC graph*.

The intended use of the SDC graph is to serve as an efficient search tree for the goal discovery phase (cf. Figure 5). In particular, it should allow to efficiently find the most adequate goal template for an incoming goal instance. This means if a new goal instance is created for which usable Web services shall be discovered, then it should be associated with the goal template out of the existing ones that most precisely fits to the goal instance. The reason is that the closer the corresponding goal template fits to the goal instance, the smaller is the number of Web services that are usable for the goal template but not for the goal instance. Given that the existing goal templates are organized in a tree with respect to their semantic similarity, then the most adequate goal template in this tree should be detectable with the minimal computational efficiency. Hence, the following requirements arise for the SDC graph.

Requirement 6 (Properties of the SDG Graph). *The SDC graph is the indexing structure for Web services with respect to the goals that can be solved by them. It consists of a tree of goal templates that are connected by directed edges with respect to their semantic similarity, and the Web services usable for each*

goal template as the leaf nodes in the SDC graph. In order to serve as an efficient search tree for the most adequate goal template for an incoming goal instance, the SDC graph must satisfy the following properties:

1. the goal template tree should be constituted by a subsumption hierarchy with respect to the functionalities requested in the goal templates. The reason is that if a goal template \mathcal{G}_2 requested a more specific functionality than a goal template \mathcal{G}_1 , then the set of possible solutions for \mathcal{G}_2 is a subset of those for \mathcal{G}_1 ; using the notation from Definition 1.1, this means that $\{\tau\}_{\mathcal{G}_1} \supseteq \{\tau\}_{\mathcal{G}_2}$. In consequence, it holds that the set of Web services usable for \mathcal{G}_2 is a subset of those usable for \mathcal{G}_1 because $\neg \text{match}(\mathcal{G}_1, W) \Rightarrow \neg \text{match}(\mathcal{G}_2, W)$.
2. the child nodes in the goal template tree must be disjoint. The reason is that for the detection of the most adequate goal template only one branch of the subsumption hierarchy needs to be investigated. If the goal template tree is balanced, then this allows to achieve a logarithmic search time.
3. only minimal knowledge on the usability of a Web service should be captured in order to ensure a scalable management of the SDC graph.

2.4 Operations and Technical Integration

We conclude the analysis by discussing the requirements on the operations and technical integration of the SDC technique into complete architectures for Semantic Web services.

The first requirement in this context concerns the technical integration. The SDC technique is not a single, detached technology that can solve a given goal by automated detection and usage of Web services. Rather, it provides a component for efficient Web service discovery that must be integrated into a system that provides the other components for automated goal solving by Web services as shown above in Figure 4. In such an overall architecture, the SDC graph provides an intermediate cache for performing Web service discovery. To realize this integration, a discovery component must be provided that properly utilizes the SDC graph to perform efficient Web service discovery at runtime.

Requirement 7 (Integration into SWS Architecture). *The SDC technique is a component for automated solving of goals by the use of Web services. It must be integrated into an overall architecture such that it serves as an intermediate during the discovery procedure for accessing the Web service repository. The discovery component should properly use the SDC graph in order to perform efficient Web service discovery, and it should be integrated with the other system components for automated resolution of a goal instance in an interleaved manner.*

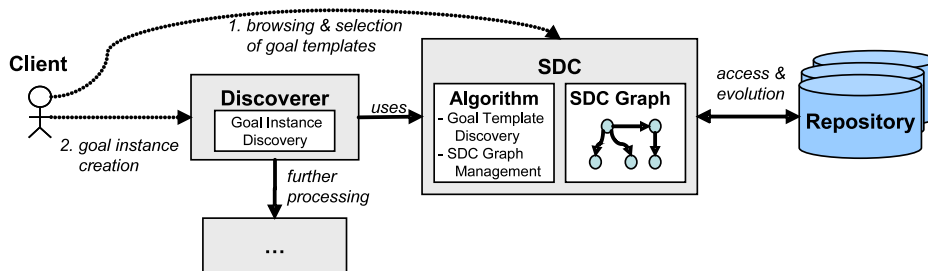


Figure 6: SDC Allocation in SWS Environments

Figure 6 shows the aspired allocation of the SDC technique in an overall architecture for Semantic Web services and illustrates the workflow of formulating and solving a goal instance. At first, the client browses the repository of existing goal templates and selects one for formulating the objective that shall be achieved. This can be supported by the SDC graph, which serves as a taxonomy for browsing goal templates. Then, the client creates a goal instance by instantiating the inputs required in the chosen goal template. The discoverer performs matchmaking on the goal instance level at runtime, using the SDC graph as a pre-filter; once a usable Web service has been discovered, the subsequent processing steps for solving the goal instance are performed. The SDC component manages the SDC graph and, orthogonal to runtime, performs Web service discovery on the goal template level.

In order to remain operational in the dynamically changing environment (i.e. the provision of new or changed Web services), the SDC technique needs to provide support for the changes of relevant aspects in the world. This denotes the final requirement.

Requirement 8 (Evolution Support). *The SDC technique must support the addition, removal, and updating of goal template and Web service descriptions in order to stay operational in its dynamically changing environment.*

Summary. In order to provide a concise overview of the determined requirements for reference in the subsequent elaborations, Table 2 summarizes the requirements determined above

Table 2: Overview of Requirements for Semantic Discovery Caching

Number	Name	Description
1	interleaved Web service discovery	only one Web Service needs to be found at runtime
2	non-functional discovery support	goal and Web service descriptions should contain information on non-functional aspects relevant for discovery, esp.: quality-of-service / financial / locality aspects, support for contracting, and behavioral aspects.
3	efficiency of discovery	reduce the search space to a minimum for Web service discovery on both the goal template and the goal instance level
4	scalability of discovery	decouple resource management and matchmaking such that for each single matchmaking operation during discovery only the minimal knowledge is loaded into the working memory
5	goal similarity measurement	define the semantic similarity of two goal templates such that the overlap between their usable Web services is maximal
6	SDC Graph Properties	the indexing structure should (1) specify a tree of goal templates as subsumption hierarchy of the requested functionalities such that (2) child nodes in the tree are disjoint, and (3) only the minimal knowledge on the usability of Web services is captured
7	integration into SWS architecture	provide a discovery component that (1) properly utilizes the SDC graph for efficient Web service discovery, and (2) is integrated with a system for solving a goal by automated Web service usage
8	evolution support	support addition, removal, and updating of goal template and Web service descriptions

3 The SDC Graph

This section specifies the SDC graph, i.e. the knowledge structure for capturing Web services for goal templates. It consists of a tree of goal templates and, as the leaf nodes, the Web services that are usable for each goal template. The SDC graph serves two purposes: (1) as an taxonomy of existing goal templates that supports goal formulation by clients, and (2) as the indexing structure whereupon specialized algorithms can perform efficient Web service discovery. The following specifies the elements and structure of the SDC graph and discusses its formal properties with respect to its application purpose. The operations for Web service discovery and management of the SDC graph are specified in the subsequent sections.

Throughout the specification, we will explain the definitions by means of a running example for illustration and clarification. We consider the following scenario: goal templates describe the objective of finding the best restaurant in a city that is to be provided for instantiation, and Web services provide search facilities for the best restaurant in a city that is to be provided as input for invocation. We can define specialized goal and Web service descriptions with respect to two dimensions that are described in the background ontology: the geographic location of a city (e.g. the continent, country, or state), and the type of the restaurant (e.g. French, Italian, Chinese, etc.). This allows to provide easy to understand examples for the different matching situations between goals and Web services, and has been exhaustively discussed within the specification of the two-phase Web service discovery in [41] and [39].

3.1 Definition

We commence with the definition of the central concepts and elements of the SDC Graph. At first, we define the similarity measurement for goal templates as the constituting notion for the goal template tree. Upon this, we specify the elements and structure of the SDC graph.

3.1.1 Goal Template Similarity

The similarity of goal templates is the constituting notion for creating the indexing structure for Web services with respect to the goals that can be solved by them. To meet requirement 5, the measurement should define two goal templates to be similar such that the set of usable Web services overlaps to the maximal extent.

With respect to the focus on requested and provided functionalities in our Web service discovery approach (*cf.* Section 1.2), we consider two goal templates to be similar if they have at least one common solution. If there is a Web service that can provide this common solution, then it is usable for both goal templates. The more common solutions exist for the goal templates, the higher is the overlap between the set of usable Web services for each one. For constructing the goal tree, we are particularly interested in goal templates whose solutions denote proper subset relationships. We discuss this below in more detail.

One could also consider other aspects for describing the similarity of goal templates, such as that they are described by semantically related keywords or have been defined in the same application area. However, the primary purpose of the goal template similarity measurement in the context of SDC is to organize goal templates in a way that allows to efficiently determine usable Web services for them. Other, non-functional aspects of goal similarity may be used in the client interface for browsing existing goal templates.

Definition 3.1 (Meaning of Goal Template Similarity). *Let \mathcal{G}_1 be goal template with $\{\tau\}_{\mathcal{G}_1}$ as the set of its possible solutions, and let \mathcal{G}_2 be goal template with $\{\tau\}_{\mathcal{G}_2}$ as the set of its possible solutions.*

We say that \mathcal{G}_1 and \mathcal{G}_2 are semantically similar if and only if $\exists \tau. \tau \in (\{\tau\}_{\mathcal{G}_1} \cap \{\tau\}_{\mathcal{G}_2})$.

In our model, the objective that is described by a goal template is specified in terms of a functional description. Recalling from Definition 1.2, a functional description is a 4-tuple $\mathcal{D} = (\Sigma, \Omega, IF, \phi^{\mathcal{D}})$ such that Σ is the extended signature, Ω is the background ontology, $IF = (i_1, \dots, i_n)$ are the input variables, and $\phi^{\mathcal{D}} = [\phi^{pre}]_{\Sigma_D^{pre} \rightarrow \Sigma_D} \Rightarrow \phi^{eff}$ is a formula that specifies meaning of \mathcal{D} as an implication between the precondition ϕ^{pre} and the effect ϕ^{eff} , wherein IF occur as the only free variables. Such a functional description formally describes all possible solutions of a goal template with respect to the possible start- and end-states and their explicit dependency.

We express the similarity of goal templates in terms of matching degrees between their formal functional descriptions. Four degrees – *exact*, *plugin*, *subsume*, *intersect* – distinguish situations wherein the similarity measure from Definition 3.1 is satisfied; the *disjoint* denotes that this is not given. The degrees and their conditions are the same as the ones we have identified for Web service discovery on the goal template level (cf. Table 1 in Section 1.2.3). Similar to the usability of a Web service for a goal template, the matching conditions for each degree are defined over the functional descriptions of goal templates along with an explicit quantification of the input bindings. The condition for the *subsume* degree $\Omega_{\mathcal{A}} \models \forall \beta. \phi^{\mathcal{D}_{\mathcal{G}_1}} \Leftarrow \phi^{\mathcal{D}_{\mathcal{G}_2}}$ specifies that under the consideration of the background ontology $\Omega_{\mathcal{A}}$ all solutions for \mathcal{G}_2 are also solutions for \mathcal{G}_1 . It is to remark that the matching conditions encompass the compatibility of the input variables of $\mathcal{D}_{\mathcal{G}_1}$ and $\mathcal{D}_{\mathcal{G}_2}$: the matching condition is satisfied only if there is an input binding $\beta : (i_1, \dots, i_n) \rightarrow \mathcal{U}_{\mathcal{A}}$ that defines concrete values for all IF -variables in both $\mathcal{D}_{\mathcal{G}_1}$ and $\mathcal{D}_{\mathcal{G}_2}$; otherwise, the models for a functional description can not be determined because their might be free variables after the instantiation. [39] discuss the input compatibility of functional descriptions in more detail.

Table 3 provides a concise overview of the goal similarity degree definitions and their meaning. In the following, we distinguish between *similarity degrees* that denote the similarity of goal templates from Definition 3.1, and *usability degrees* that denote the usability of a Web service for a goal template; both are expressed in terms of the matching degree in order to distinguish the situations that are relevant for SDC.

Table 3: Definition and Meaning of Goal Similarity Degrees

Denotation $\mathcal{D}_{\mathcal{G}_1} = (\Sigma, \Omega, IF, \phi^{\mathcal{D}_{\mathcal{G}_1}})$ $\mathcal{D}_{\mathcal{G}_2} = (\Sigma, \Omega, IF, \phi^{\mathcal{D}_{\mathcal{G}_2}})$	Definition $\beta : IF \rightarrow \mathcal{U}_{\mathcal{A}}$ $\phi^{\mathcal{D}} = [\phi^{pre}]_{\Sigma_D^{pre} \rightarrow \Sigma_D} \Rightarrow \phi^{eff}$ $\Omega_{\mathcal{A}} = \Omega \cup [\Omega]_{\Sigma_D^{pre} \rightarrow \Sigma_D}$	Meaning for $\{\tau\}_{\mathcal{G}_1}, \{\tau\}_{\mathcal{G}_2}$
exact ($\mathcal{D}_{\mathcal{G}_1}, \mathcal{D}_{\mathcal{G}_2}$)	$\Omega_{\mathcal{A}} \models \forall \beta. \phi^{\mathcal{D}_{\mathcal{G}_1}} \Leftrightarrow \phi^{\mathcal{D}_{\mathcal{G}_2}}$	$\tau \in \{\tau\}_{\mathcal{G}_1}$ if and only if $\tau \in \{\tau\}_{\mathcal{G}_2}$. all Web services that are usable for \mathcal{G}_1 are also usable \mathcal{G}_2 under the same usability degree.
plugin ($\mathcal{D}_{\mathcal{G}_1}, \mathcal{D}_{\mathcal{G}_2}$)	$\Omega_{\mathcal{A}} \models \forall \beta. \phi^{\mathcal{D}_{\mathcal{G}_1}} \Rightarrow \phi^{\mathcal{D}_{\mathcal{G}_2}}$	if $\tau \in \{\tau\}_{\mathcal{G}_1}$ then $\tau \in \{\tau\}_{\mathcal{G}_2}$. all Web services usable for \mathcal{G}_1 are also usable for \mathcal{G}_2 but not vice versa.
subsume ($\mathcal{D}_{\mathcal{G}_1}, \mathcal{D}_{\mathcal{G}_2}$)	$\Omega_{\mathcal{A}} \models \forall \beta. \phi^{\mathcal{D}_{\mathcal{G}_1}} \Leftarrow \phi^{\mathcal{D}_{\mathcal{G}_2}}$	if $\tau \in \{\tau\}_{\mathcal{G}_2}$ then $\tau \in \{\tau\}_{\mathcal{G}_1}$. all Web services usable for \mathcal{G}_2 are also usable for \mathcal{G}_1 but not vice versa.
intersect ($\mathcal{D}_{\mathcal{G}_1}, \mathcal{D}_{\mathcal{G}_2}$)	$\Omega_{\mathcal{A}} \models \exists \beta. \phi^{\mathcal{D}_{\mathcal{G}_1}} \wedge \phi^{\mathcal{D}_{\mathcal{G}_2}}$	there is a τ such that $\tau \in \{\tau\}_{\mathcal{G}_1}$ and $\tau \in \{\tau\}_{\mathcal{G}_2}$. a Web service that can provide this τ is hence usable for both goals.
disjoint ($\mathcal{D}_{\mathcal{G}_1}, \mathcal{D}_{\mathcal{G}_2}$)	$\Omega_{\mathcal{A}} \models \neg \exists \beta. \phi^{\mathcal{D}_{\mathcal{G}_1}} \wedge \phi^{\mathcal{D}_{\mathcal{G}_2}}$	there is no τ such that $\tau \in \{\tau\}_{\mathcal{G}_1}$ and $\tau \in \{\tau\}_{\mathcal{G}_2}$; we can not make any statement on the Web services usable for solving the goals.

The purpose of the goal similarity degrees is to enable efficient determination of usable Web services for similar goal templates. While defining the inference rules for this in Section 3.2, let us consider an example for illustration. Let $\mathcal{D}_{\mathcal{G}_1}$ be the functional description of a goal template \mathcal{G}_1 , and let $\mathcal{D}_{\mathcal{G}_2}$ be the functional description of a goal template \mathcal{G}_2 . Let the similarity degree be $\text{subsume}(\mathcal{D}_{\mathcal{G}_1}, \mathcal{D}_{\mathcal{G}_2})$, such that $\{\tau\}_{\mathcal{G}_1} \supseteq \{\tau\}_{\mathcal{G}_2}$. Then, it holds that every Web service that is usable for \mathcal{G}_2 is also usable for \mathcal{G}_1 , because if $\exists \tau. \tau \in (\{\tau\}_{\mathcal{G}_2} \cap \{\tau\}_W)$, then this τ is also an element of $\{\tau\}_{\mathcal{G}_1}$. Here, the overlap between the sets of usable Web services for similar goal templates is maximal. Hence, specifying the goal template similarity in terms of matching degree between their formal functional descriptions satisfies requirement 5.

3.1.2 Elements and Structure

On the basis of the goal similarity measure we can now define the structure of the SDC graph. The following specifies the elements and the basic structure, while we shall discuss its formal properties in the detail below in Section 3.3.

Definition 3.2 (SDC Graph). *An SDC Graph consists of four elements: goal templates \mathcal{G} , Web services W , GG mediators GGM and WG mediators WGM . It is defined such that:*

- (i) *every inner node is a goal template \mathcal{G}*
- (ii) *every leaf node is a Web service W*
- (iii) *a GG mediator is a triple $GGM = (\text{source}, \text{target}, d_{\text{similarity}})$ with:*
 - *a goal template \mathcal{G}_1 as the source,*
 - *another a goal template \mathcal{G}_2 as the target, and*
 - *$d_{\text{similarity}}$ denoting the similarity degree between \mathcal{G}_1 and \mathcal{G}_2*
- (iv) *a WG mediator is a triple $WGM = (\text{source}, \text{target}, d_{\text{usability}})$ with:*
 - *a goal template \mathcal{G} as the source,*
 - *a Web service W as the target, and*
 - *$d_{\text{usability}}$ denoting the usability degree between \mathcal{G} and W .*

In this definition, we consider all elements to be associated with a complete description that is needed for solving a goal by the automated usage of Web services. This means that for clause (i) a goal template \mathcal{G} carries a functional description, requirements non-functional aspects, and optionally a desired workflow as an orchestration of goals (i.e. the complete description model of goal templates as defined in [42]), and for clause (ii) a Web service is described by a capability (overall provided functionality), non-functional aspects, and behavioral interfaces for consumption and aggregation of other Web services. Therewith, requirement 2 is satisfied as information on all other aspects are available in the goal and Web service descriptions.

In the clauses (iii) and (iv), we use *mediators* for describing the edges of the SDC graph. The concept of mediators is promoted by the WSMO framework as a means for handling potentially arising heterogeneities that hamper the interoperability of goals and Web services [30]; GG mediators and WG mediators are specialized mediator types defined in WSMO [26]. As specified in [36], a mediator is an intermediate that connects a source and a target component and utilizes respective mediation facilities for resolving mismatches of the data and the process level. The main merit from the use of such mediators for defining the edges in the SDC graph is that we obtain a *directed* relationship between the source and the target element. This allows to precisely describe the similarity degree between goal templates (in a GG Mediator) as well as the usability of a particular Web service for a goal template (in a WG mediator). Definition 3.3 specifies the relationship of inverse mediators. Besides, with respect to requirement 7, mediation facilities for handling and resolving potentially occurring heterogeneities are implicitly incorporated in the SDC graph.

Definition 3.3 (Inverse Mediators). Let a mediator be a triple $M = (s, t, d)$ such that s is the source element, t is the target element, and d is the matching degree between s and t .

An *inverse mediator* M' defines the relationship between s and t with the inverse direction. We can derive M' from M with the following relationship between M' and M for distinct values of d :

- (i) $M = (s, t, exact) \Leftrightarrow M' = (t, s, exact)$
- (ii) $M = (s, t, plugin) \Leftrightarrow M' = (t, s, subsume)$
- (iii) $M = (s, t, subsume) \Leftrightarrow M' = (t, s, plugin)$
- (iv) $M = (s, t, intersect) \Leftrightarrow M' = (t, s, intersect)$
- (v) $M = (s, t, disjoint) \Leftrightarrow M' = (t, s, disjoint)$

Essentially, Definition 3.2 defines an SDC graph to consist to two layers. The upper one are the goal templates that are connected by GG mediators with respect to their similarity degree as defined above in Table 3. A GG mediator defines a directed edge between two goal templates (from the source goal to the target goal, see above). Hence, we obtain a *directed graph* that constitutes the indexing structure for available Web services. The lower level are the Web services that are connected to goal templates via WG mediators. This constitutes the discovery cache for capturing knowledge on Web service discovery on the goal template level. For this, a WG mediator defines the usability degree of a Web service for solving a goal template. This can explicate any matching degree under which the Web service is usable for solving the source goal template (i.e. all but *disjoint*). With respect to our primary focus on functional aspects, this is the minimal knowledge for capturing Web service discovery results on the goal template level and thus satisfies the third aspect of requirement 6. However, the graph of goal templates may contain cycles so that this structure does not yet satisfy the first and second aspect of requirement 6. For achieving this, we will specify the resolution of intersect-arcs and such cycles below in Section 3.3.

To illustrate the definition, Figure 7 shows an SDC graph for your running example with two goal templates \mathcal{G}_1 and \mathcal{G}_2 , and two Web services W_1 and W_2 . Let \mathcal{G}_1 specify the objective of finding the best restaurant in an European city, and let \mathcal{G}_2 request to find the best restaurant in an Austrian city. As every Austrian city is also a European city but not vice versa, the similarity degree of the goal templates is *subsume*($\mathcal{G}_1, \mathcal{G}_2$) that is explicated in the GG mediator. Let W_1 offer a search facility fir the best French restaurant in a French city. W_1 is usable for a goal instance of \mathcal{G}_1 if the city provided as input is in located in France and if the best restaurant in this city is of type French. Thus, the usability degree is *intersect*(\mathcal{G}_1, W_1) as discussed in [41]. Obviously, W_1 is not usable for \mathcal{G}_2 because French and Austrian cities are disjoint. Finally, let W_2 provide a search facility for the best restaurant in every city of the world. It is usable for both \mathcal{G}_1 and \mathcal{G}_2 under the *plugin* degree because $\{\tau\}_{W_2} \supseteq \{\tau\}_{\mathcal{G}_1} \supseteq \{\tau\}_{\mathcal{G}_2}$. Moreover, because of this relationship we can infer the usability degree of W_2 for \mathcal{G}_2 if the usability degree of W_2 for \mathcal{G}_1 is known.

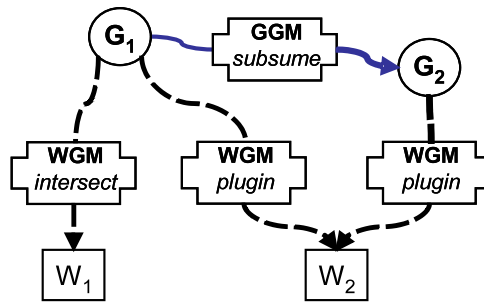


Figure 7: Example of an SDC graph

3.2 Inference Rules for Web Service Usability Degree

The main merit of capturing Web service discovery results for goal templates in the SDC Graph is that we can infer knowledge on the usability of a Web service for similar goal templates. We express this in terms of inference rules for the arcs in the SDC graph. For two goal templates $\mathcal{G}_1, \mathcal{G}_2$ and a Web service W , the general form of these rules is $d_{usability}(\mathcal{G}_2, W) \leftarrow d_{similarity}(\mathcal{G}_1, \mathcal{G}_2) \wedge d_{usability}(\mathcal{G}_1, W)$. As we shall discuss below, these inferences provide the foundation for several operations for Web service discovery and management of the SDC Graph.

Theorem 3.1 specifies these inference rules for all possible combinations of the usability of W for \mathcal{G}_1 , and the similarity of \mathcal{G}_1 and \mathcal{G}_2 . We refer to the definition of the Web service usability degrees in Table 6 in Section 1.2.3 and the definition of the goal template similarity degrees from Table 3 in Section 3.1.1. For convenience, we provide a comprehensive overview of the matching degree definitions as well as their meaning for goal template similarity and the usability of a Web service for solving a goal template in Appendix A.

From the theorem, we can make the following observations that are relevant in the context of SDC:

- there are four types of inferences for the usability degree of W for \mathcal{G}_2 (separated by horizontal lines):
 1. *directly inferable*, i.e. the usability degree can be determined without matchmaking (clauses: 1, 2.1, 2.2, 3.1, 3.2, 3.11, 4.1, 5.1, 5.2)
 2. *guaranteed usability* but the degree must be determined via matchmaking of $\mathcal{D}_{\mathcal{G}_2}$ and \mathcal{D}_W (clauses: 2.3 - 2.6; 2.7 - 2.8; 4.2 - 4.3)
 3. *possible usability*, usability and degree must be determined via matchmaking of $\mathcal{D}_{\mathcal{G}_2}$ and \mathcal{D}_W (clauses: 3.3 - 3.7; 3.8 - 3.10; 4.4 - 4.6; 4.7 - 4.11)
 4. *not inferable* so that matchmaking between $\mathcal{D}_{\mathcal{G}_2}$ and \mathcal{D}_W is required (clauses: 2.9, 4.12, 5.3)
- under the *plugin* similarity degree, all Web services that are usable for \mathcal{G}_1 are also usable for \mathcal{G}_2 , but we can not make any statement about Web services that are usable for \mathcal{G}_2 but not for \mathcal{G}_1
- under the *subsume* similarity degree, only those Web services that are usable for \mathcal{G}_1 are potentially usable for \mathcal{G}_2 but no others can be (*cf.* clause 2.10)
- under the *disjoint* similarity degree, Web services that are usable for \mathcal{G}_1 under the *exact* or the *subsume* are not usable for \mathcal{G}_2 ; for Web services with other usability degrees for \mathcal{G}_1 , we can not make any statement about their usability for \mathcal{G}_2 .

Theorem 3.1 (Inference Rules for Web Service Usability Degrees). *Let W be a Web service, and let \mathcal{G}_1 and \mathcal{G}_2 be goal templates. Let $d(\mathcal{G}, W)$ denote the usability degree of W for a goal template, and let $d(\mathcal{G}_i, \mathcal{G}_j)$ denote the similarity degree between \mathcal{G}_1 and \mathcal{G}_2 .*

Given $d(\mathcal{G}_1, W)$ and $d(\mathcal{G}_1, \mathcal{G}_2)$, we can infer knowledge about $d(\mathcal{G}_2, W)$ by the following rules.

1. $exact(\mathcal{G}_1, \mathcal{G}_2)$: $d(\mathcal{G}_1, W) = d(\mathcal{G}_2, W)$.
2. $plugin(\mathcal{G}_1, \mathcal{G}_2)$:

(1)	$exact(\mathcal{G}_1, W) \Rightarrow subsume(\mathcal{G}_2, W)$.
(2)	$subsume(\mathcal{G}_1, W) \Rightarrow subsume(\mathcal{G}_2, W)$.
(3)	$plugin(\mathcal{G}_1, W) \Rightarrow exact(\mathcal{G}_2, W)$ or
(4)	$plugin(\mathcal{G}_1, W) \Rightarrow plugin(\mathcal{G}_2, W)$ or
(5)	$plugin(\mathcal{G}_1, W) \Rightarrow subsume(\mathcal{G}_2, W)$ or
(6)	$plugin(\mathcal{G}_1, W) \Rightarrow intersect(\mathcal{G}_2, W)$.
(7)	$intersect(\mathcal{G}_1, W) \Rightarrow subsume(\mathcal{G}_2, W)$ or
(8)	$intersect(\mathcal{G}_1, W) \Rightarrow intersect(\mathcal{G}_2, W)$.
(9)	$disjoint(\mathcal{G}_1, W)$: no statement possible.
3. $subsume(\mathcal{G}_1, \mathcal{G}_2)$:

(1)	$exact(\mathcal{G}_1, W) \Rightarrow plugin(\mathcal{G}_2, W)$.
(2)	$plugin(\mathcal{G}_1, W) \Rightarrow plugin(\mathcal{G}_2, W)$.
(3)	$subsume(\mathcal{G}_1, W) \Rightarrow exact(\mathcal{G}_2, W)$ or
(4)	$subsume(\mathcal{G}_1, W) \Rightarrow plugin(\mathcal{G}_2, W)$ or
(5)	$subsume(\mathcal{G}_1, W) \Rightarrow subsume(\mathcal{G}_2, W)$ or
(6)	$subsume(\mathcal{G}_1, W) \Rightarrow intersect(\mathcal{G}_2, W)$ or
(7)	$subsume(\mathcal{G}_1, W) \Rightarrow disjoint(\mathcal{G}_2, W)$.
(8)	$intersect(\mathcal{G}_1, W) \Rightarrow plugin(\mathcal{G}_2, W)$ or
(9)	$intersect(\mathcal{G}_1, W) \Rightarrow intersect(\mathcal{G}_2, W)$ or
(10)	$intersect(\mathcal{G}_1, W) \Rightarrow disjoint(\mathcal{G}_2, W)$.
(11)	$disjoint(\mathcal{G}_1, W) \Rightarrow disjoint(\mathcal{G}_2, W)$.
4. $intersect(\mathcal{G}_1, \mathcal{G}_2)$:

(1)	$exact(\mathcal{G}_1, W) \Rightarrow intersect(\mathcal{G}_2, W)$.
(2)	$plugin(\mathcal{G}_1, W) \Rightarrow plugin(\mathcal{G}_2, W)$ or
(3)	$plugin(\mathcal{G}_1, W) \Rightarrow intersect(\mathcal{G}_2, W)$.
(4)	$subsume(\mathcal{G}_1, W) \Rightarrow subsume(\mathcal{G}_2, W)$ or
(5)	$subsume(\mathcal{G}_1, W) \Rightarrow intersect(\mathcal{G}_2, W)$ or
(6)	$subsume(\mathcal{G}_1, W) \Rightarrow disjoint(\mathcal{G}_2, W)$.
(7)	$intersect(\mathcal{G}_1, W) \Rightarrow exact(\mathcal{G}_2, W)$ or
(8)	$intersect(\mathcal{G}_1, W) \Rightarrow plugin(\mathcal{G}_2, W)$ or
(9)	$intersect(\mathcal{G}_1, W) \Rightarrow subsume(\mathcal{G}_2, W)$ or
(10)	$intersect(\mathcal{G}_1, W) \Rightarrow intersect(\mathcal{G}_2, W)$ or
(11)	$intersect(\mathcal{G}_1, W) \Rightarrow disjoint(\mathcal{G}_2, W)$.
(12)	$disjoint(\mathcal{G}_1, W)$: no statement possible.
5. $disjoint(\mathcal{G}_1, \mathcal{G}_2)$:

(1)	$exact(\mathcal{G}_1, W) \Rightarrow disjoint(\mathcal{G}_2, W)$.
(2)	$subsume(\mathcal{G}_1, W) \Rightarrow disjoint(\mathcal{G}_2, W)$.
(3)	$d(\mathcal{G}_1, W)$ and $d \neq subsume$: no statement possible.

Proof. The formal proof is provided in Appendix B of this document. □

3.3 Formal Properties and Refinement

We complete the definition of the SDC Graph with the necessary refinements for establishing the desirable properties as identified in requirement 6. Recalling from Section 2.3.2, these are (1) a tree structure that allow to efficiently search the most appropriate goal template for a new, incoming goal instance, and (2) to capture only the minimal knowledge on Web service discovery results on the goal template level.

The following first analyzes the properties of the initial SDC graph that is obtained from creating the GG and WG mediators for a given set of goal template and Web service descriptions. Then, we present the resolution of intersect arcs between goal templates as the means for establishing the desirable properties and illustrate this in our running example. Finally, we show the formal properties of the refined SDC graph. For the discussion, we apply the conventional terminology from graph theory as defined in [11].

3.3.1 Initial SDC Graph

Let us consider the following to be given: a set of goal templates $\mathcal{G}_1, \dots, \mathcal{G}_n$, and a set of Web services W_1, \dots, W_m . Furthermore, let many of the goal templates be similar, and let many of the Web services are usable for the distinct goal templates (e.g. if they are allocated in the same application domain).

Now, let us examine the initial SDC graph that is obtained by determining the matching degrees for goal similarity and Web service usability and then explicating this knowledge in GG and WG mediators in accordance to Definition 3.2. Recalling from above, an SDC graph consists of two layers: the upper one are the goal templates that are connected by GG mediators with respect to their similarity degree; we shall refer to this as the **goal graph**. The lower layer are the Web services whose usability degree for goal templates is explicated in WG mediators that are allocated as leaf nodes to the goal graph; we shall refer to this as the **discovery cache**. In the following, we focus on the properties of the goal graph. In particular, we analyze its structure and identify the reasons why the initially obtained goal graph does not satisfy requirement 6, i.e. providing a sophisticated knowledge structure for searching the most appropriate goal template for a new, incoming goal instance. In this respect, we make the following observations.

Observation 1 – Types of Similarity Degrees in Goal Graph. The result of performing matchmaking in order to determine the similarity degree of goal templates as well as the usability degree of Web services can of course result in any of the five matching degrees. However, we do not keep knowledge about *disjoint* degrees in the SDC graph: this degree neither provides useful information for inferring the usability of a Web service between two goal templates, nor is a Web service usable for solving a goal template or any of its instantiations (*cf.* Appendix A). We also do not have to keep two goal templates in the goal graph whose similarity is *exact*: the same Web services are usable for both under the same usability degree (*cf.* Theorem 3.1), so we just need to keep one of them.

If there are two goal templates \mathcal{G}_1 and \mathcal{G}_2 whose possible solutions denote a proper subset relationship – e.g. $\{\tau\}_{\mathcal{G}_1} \supset \{\tau\}_{\mathcal{G}_2}$ – then we can either define a $GGM = (\mathcal{G}_2, \mathcal{G}_1, \text{plugin})$ or its inversion $GGM' = (\mathcal{G}_1, \mathcal{G}_2, \text{subsume})$ (*cf.* Definition 3.3). We prefer to keep GG mediators with a *subsume* similarity degree because this allows to utilize the goal similarity as a pre-filter for Web service discovery on the goal template level: if $\text{subsume}(\mathcal{G}_1, \mathcal{G}_2)$ then all Web services that are not usable for \mathcal{G}_1 are also not usable for \mathcal{G}_2 (*cf.* clause 3.11 of Theorem 3.1). If the similarity degree of \mathcal{G}_1 and \mathcal{G}_2 is *intersect*, then we can define the connecting GG mediator with any direction without loosing or gaining any important information. However, this similarity degree appears to not be very valuable for inferring the usability degree of a Web service: if $\text{intersect}(\mathcal{G}_1, \mathcal{G}_2)$, then a Web service W is only usable for both \mathcal{G}_1 and \mathcal{G}_2 if it can provide a solution that is allocated in the intersection of possible solutions of \mathcal{G}_1 and \mathcal{G}_2 (*cf.* clauses 4.1 - 4.12 of Theorem 3.1).

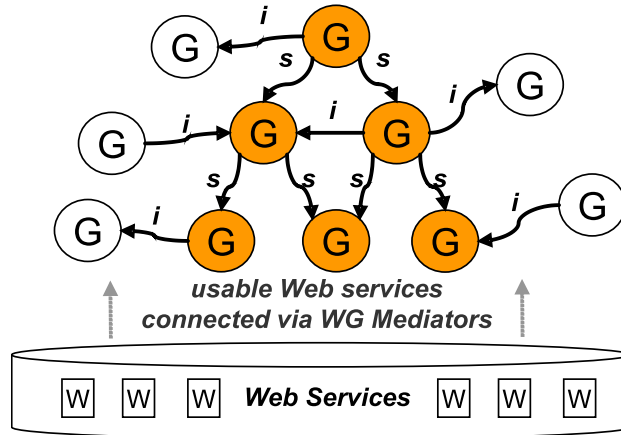


Figure 8: Initial Structure of the Goal Graph

In conclusion, there are only two types of similarity degrees that occur in the goal graph: *subsume* and *intersect*. For the sake of comprehensibility, in the following we shall refer to a GG mediator that explicates a *subsume* similarity degree as an *s-arc*, and to one that explicates an *intersect* similarity degree as an *i-arc* (*arc* = a directed edge in a graph [11]). The hierarchy that is constituted by *s-arcs* denotes the core of the goal graph: the set of usable Web services for a child node in this hierarchy is a subset of the set of Web services that are usable for its parent node. Goal templates that are connected by *i-arcs* can occur everywhere in the goal graph; in particular, there can be goal templates that are child nodes in the *s-arc* hierarchy but have a common solution. Figure 8 illustrates the structure of an initial goal graph. It is to remark that in the discovery cache any matching degree can appear under which the target Web service is usable for solving the source goal template (i.e. all but *disjoint*). Besides, the creation of such an initial SDC graph requires management operations; we will address this in detail in Section 5.

Observation 2 – Cycles and Concatenations of Intersect-Arcs in the Goal Graph. Two types of undesirable situations can occur in the initial goal graph as illustrated in Figure 9. The first one is a cycle that occurs if $\text{intersect}(\mathcal{G}_1, \mathcal{G}_2)$, $\text{intersect}(\mathcal{G}_2, \mathcal{G}_3)$, and $\text{intersect}(\mathcal{G}_3, \mathcal{G}_1)$. Such a cycle can only occur in a sequence of goal templates that are connected by *i-arcs* but not for *s-arcs*: if $\text{subsume}(\mathcal{G}_1, \mathcal{G}_2)$ and $\text{subsume}(\mathcal{G}_2, \mathcal{G}_3)$, then also $\text{subsume}(\mathcal{G}_1, \mathcal{G}_3)$ and thus there can not be a cyclic relation between $\mathcal{G}_1, \mathcal{G}_2$, and \mathcal{G}_3 ; we discuss this below in more detail. With such a cycle, we might run into an infinite loop when searching for the most appropriate goal template for an incoming goal instance. This contradicts requirement 6 and thus must be resolved.

The second construct is an acyclic concatenations of *i-arcs*, i.e. if $\text{intersect}(\mathcal{G}_1, \mathcal{G}_2)$ and $\text{intersect}(\mathcal{G}_2, \mathcal{G}_3)$ such that $\neg \exists \tau. \tau \in (\{\tau\}_{\mathcal{G}_1} \cap \{\tau\}_{\mathcal{G}_2} \cap \{\tau\}_{\mathcal{G}_3})$ – i.e. there is at least one common solution for each pair of goal templates but there is no common solution for all three. In this situation, there can not be any Web service that is usable for all three goal templates. This does not provide valuable information for inferring the usability of a Web service for adjacent goal templates, and hence should be resolved.

Observation 3 – Disconnected Nodes in the Goal Graph. There might be goal templates that are not connected in the goal graph. In particular, it can occur that there are several disconnected sub-graphs, i.e. separate collections of connected goal templates. This occurs when there are goal templates that do not have

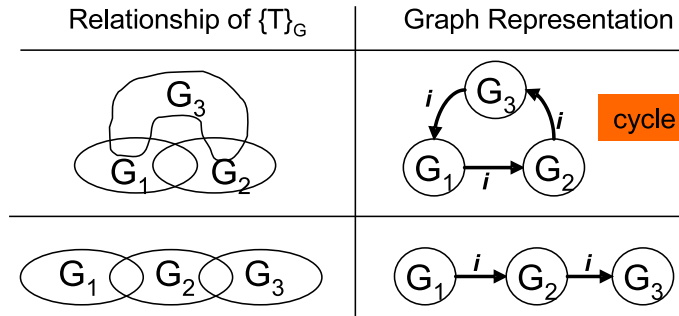


Figure 9: Examples for Cycles and I-arc Concatenations in Initial Goal Graph

any common solution - e.g. one for searching restaurants and another one for booking flight tickets. In fact, we can expect that distinct application areas form separate sub-graphs of goal templates. Nevertheless, it can be that the same Web service is usable for goal templates in disconnected sub-graphs. Figure 10 illustrates this situation. Note that this does not hamper the formal properties of the SDC graph - it merely reflects different application contexts for which for the same Web service might be usable.

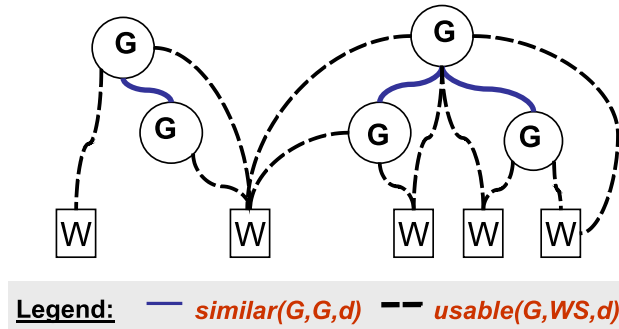


Figure 10: Disconnected Sub-Graphs in Goal Graph

The discussion reveals that the initial goal graph does not yet satisfy the requirements for serving as a sophisticated knowledge structure for efficient goal template search. Examining the observations (1) and (2) reveals that those deficiencies that contradict the required formal properties result from i-arcs, i.e. when the similarity degree of two connected goal templates is *intersect*. On the other hand, s-arcs appear to be the most desirable goal template similarity degree as it allows to utilize the goal graph as a pre-filter for Web service discovery on the goal template level. In conclusion, we can identify the following aspects to the necessary for establishing the desired properties of the SDC graph in accordance to requirement 6:

1. remove all i-arcs such that only the knowledge on the intersection of the solutions for the connected goal templates remains; then, cycles in the goal graph can no longer occur
2. ensure that the child nodes in the goal graph are disjoint; if this is given, only one branch needs to be followed for finding goal templates
3. skip inferable WG mediators: if the usability of a Web service for a parent node in the goal graph can be inferred directly from the usability degree of the Web service for a child node, then omit the WG mediator between the Web service and the parent node

3.3.2 Resolution of Intersect Arcs in Goal Graph

The following defines the handling and resolution of i-arcs in the goal graph, which is the central technique for establishing the desirable formal properties of the SDC Graph. As examined above, the situations where the similarity degree of goal templates is *intersect* cause undesired properties of the goal graph, such as cycles, deficiencies as a sophisticated knowledge structure for goal template search, and insufficient support for inferring the usability degree of Web services among similar goal templates.

The approach is as follows. Given two goal templates \mathcal{G}_1 and \mathcal{G}_2 such that $\text{intersect}(\mathcal{G}_1, \mathcal{G}_2)$, we define a new goal template that precisely describes the intersection of the possible solution of \mathcal{G}_1 and \mathcal{G}_2 . We refer to this new goal template $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}$ as an *intersection goal template*. Formally, $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}$ is defined as the logical conjunction of the functional descriptions of \mathcal{G}_1 and \mathcal{G}_2 . We then replace the i-arc between \mathcal{G}_1 and \mathcal{G}_2 in the goal graph by two s-arcs $\text{subsume}(\mathcal{G}_1, \mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)})$ and $\text{subsume}(\mathcal{G}_2, \mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)})$ so that $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}$ becomes a child node of \mathcal{G}_1 and \mathcal{G}_2 . Therewith, we can remove all i-arcs from the initial SDC Graph, and establish a hierarchy of s-arcs that does not longer reveal any undesirable properties. Figure 11 illustrates the approach that we formally specify in the following.

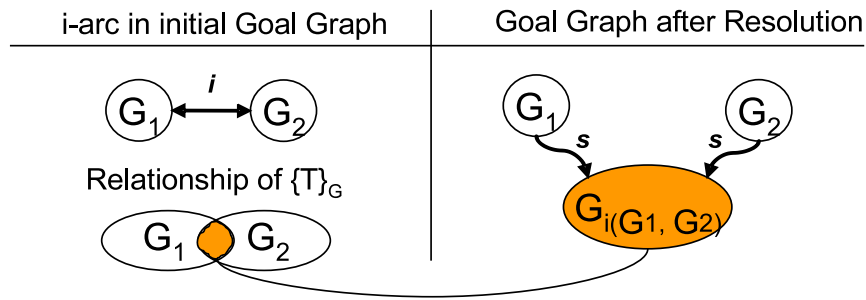


Figure 11: Resolution of Intersect Arcs in the Goal Graph

Definition 3.4 (Intersection Goal Template). Let \mathcal{G}_1 be a goal template that is formally described by $\mathcal{D}_{\mathcal{G}_1} = (\Sigma, \Omega, IF_{\mathcal{G}_1}, \phi^{\mathcal{D}_{\mathcal{G}_1}})$ with $\phi^{\mathcal{D}_{\mathcal{G}_1}} : [\phi_{\mathcal{G}_1}^{pre}]_{\Sigma_D^{pre} \rightarrow \Sigma_D} \Rightarrow \phi_{\mathcal{G}_1}^{eff}$, and let \mathcal{G}_2 be a goal template that is formally described by $\mathcal{D}_{\mathcal{G}_2} = (\Sigma, \Omega, IF_{\mathcal{G}_2}, \phi^{\mathcal{D}_{\mathcal{G}_2}})$ with $\phi^{\mathcal{D}_{\mathcal{G}_2}} : [\phi_{\mathcal{G}_2}^{pre}]_{\Sigma_D^{pre} \rightarrow \Sigma_D} \Rightarrow \phi_{\mathcal{G}_2}^{eff}$.

We define $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}$ as the *intersection goal template* of \mathcal{G}_1 and \mathcal{G}_2 such that $\mathcal{D}_{\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}} = (\Sigma, \Omega, IF, \phi^{\mathcal{D}_{\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}}})$ with $\phi^{\mathcal{D}_{\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}}} : ([\phi_{\mathcal{G}_1}^{pre}]_{\Sigma_D^{pre} \rightarrow \Sigma_D} \Rightarrow \phi_{\mathcal{G}_1}^{eff}) \wedge ([\phi_{\mathcal{G}_2}^{pre}]_{\Sigma_D^{pre} \rightarrow \Sigma_D} \Rightarrow \phi_{\mathcal{G}_2}^{eff})$.

This defines the intersection goal template $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}$ of two goal templates \mathcal{G}_1 and \mathcal{G}_2 as the logical conjunction of their formal functional descriptions (cf. Definition 1.2 in Section 1.2.2 for the elements and structure of formal functional descriptions that we apply in our model). Such an intersection goal template can be defined independently of the similarity degree between \mathcal{G}_1 and \mathcal{G}_2 ; however, it only provides useful information if the similarity degree is *intersect*($\mathcal{G}_1, \mathcal{G}_2$).

In Definition 3.4, we assume that the functional descriptions $\mathcal{D}_{\mathcal{G}_1}$ and $\mathcal{D}_{\mathcal{G}_2}$ use the same signature Σ and the same or at least compatible background ontologies Ω . Moreover, for constructing $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}$ it must hold that \mathcal{G}_1 and \mathcal{G}_2 define compatible input variables. This is given if there is a bijection $\pi : IF_{\mathcal{G}_1} \rightarrow IF_{\mathcal{G}_2}$ such that for each input variable $i_{\mathcal{G}_1} \in IF_{\mathcal{G}_1}$ there exists a compatible input variable $i_{\mathcal{G}_2} \in IF_{\mathcal{G}_2}$. We refer to [39] for a more detailed discussion of the compatibility of functional descriptions with respect to the signature, the background ontology, and the input variables.

Theorem 3.2 specifies the meaning of intersection goal templates. Essentially, for two goal templates \mathcal{G}_1 and \mathcal{G}_2 the intersection goal template $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}$ formally describes the set of possible solutions that are common for \mathcal{G}_1 and \mathcal{G}_2 . Recalling the formal semantics from Definition 1.3, a functional description $\mathcal{D}_{\mathcal{G}}$ formally describes $\{\tau\}_{\mathcal{G}}$ as set of all possible solutions of a goal with respect to their start- and end-states such that $\tau \in \{\tau\}_{\mathcal{G}}$ if τ can be represented by a Σ -interpretation that is a model of $\mathcal{D}_{\mathcal{G}}$.

Theorem 3.2 (Meaning of an Intersection Goal Template). *Let \mathcal{G}_1 be a goal template described by $\mathcal{D}_{\mathcal{G}_1}$, let \mathcal{G}_2 be a goal template described by $\mathcal{D}_{\mathcal{G}_2}$, and let $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}$ be the intersection goal template of \mathcal{G}_1 and \mathcal{G}_2 that is described by $\mathcal{D}_{\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}}$. Let $\{\tau\}_{\mathcal{G}}$ be all possible solutions of a goal template \mathcal{G} such that $\tau \in \{\tau\}_{\mathcal{G}}$ if and only if τ is represented by a Σ -interpretation I with $I \models \mathcal{D}_{\mathcal{G}}$. It holds that:*

$$\tau \in \{\tau\}_{\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}} \quad \text{if and only if} \quad \tau \in (\{\tau\}_{\mathcal{G}_1} \cap \{\tau\}_{\mathcal{G}_2})$$

Proof. A functional description $\mathcal{D} = (\Sigma, \Omega, IF, \phi^{\mathcal{D}})$ describes a set of sequences of states $\{\tau\}$ with $\tau \in \{\tau\}$ if τ can be represented by a Σ -interpretation I such that I is a model of $\phi^{\mathcal{D}}$ under an input binding $\beta : IF \rightarrow \mathcal{U}$, formally: $I, \beta \models \phi^{\mathcal{D}}$. Under the implication semantics with $\phi^{\mathcal{D}} : [\phi^{pre}]_{\Sigma_D^{pre} \rightarrow \Sigma_D} \Rightarrow \phi^{eff}$, this is given if $I, \beta \models \phi^{pre}$ and $I, \beta \models \phi^{eff}$; if $I \not\models \phi^{pre}$, then we can not precisely determine whether I is a model of $\phi^{\mathcal{D}}$ or not.

A Σ -interpretation I under an input binding β represents a $\tau \in (\{\tau\}_{\mathcal{G}_1} \cap \{\tau\}_{\mathcal{G}_2})$ if and only if $I, \beta \models \mathcal{D}_{\mathcal{G}_1}$ and $I, \beta \models \mathcal{D}_{\mathcal{G}_2}$. Such an I is also a model of $\mathcal{D}_{\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}}$ under β because $\phi^{\mathcal{D}_{\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}}} = \phi^{\mathcal{D}_{\mathcal{G}_1}} \wedge \phi^{\mathcal{D}_{\mathcal{G}_2}}$. If there is a Σ -interpretation I under an input binding β such that $I, \beta \models \mathcal{D}_{\mathcal{G}_1}$ and $I, \beta \not\models \mathcal{D}_{\mathcal{G}_2}$, then this I is not a model of $\mathcal{D}_{\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}}$ under β because $\phi^{\mathcal{D}_{\mathcal{G}_1}} \wedge \text{false} \Leftrightarrow \text{false}$ under I, β ; accordingly, $I, \beta \not\models \mathcal{D}_{\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}}$ if $I, \beta \not\models \mathcal{D}_{\mathcal{G}_1}$ and $I, \beta \models \mathcal{D}_{\mathcal{G}_2}$. Thus, under all input bindings β every Σ -interpretation I with $I, \beta \models \mathcal{D}_{\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}}$ represents a $\tau \in (\{\tau\}_{\mathcal{G}_1} \cap \{\tau\}_{\mathcal{G}_2})$. \square

Resolving Undesirable Situations in the Goal Graph. Intersection goal templates serve as the central construct for resolving undesirable situations in the initial goal graph. To achieve this, we construct the intersection goal template for two goal templates whose similarity degree is *intersect*, and insert this into the goal graph by replacing the i-arc with s-arcs from the original goal templates to the intersection goal template (cf. Figure 11). This is performed iteratively until all i-arcs in the goal graph are removed. We therewith can transform connected sub-graphs of the initial goal graph into directed trees of goal templates whose arcs are only s-arcs without exception. We refer to this as *goal trees* that satisfy all desirable properties that have been identified in requirement 6.

The following specifies the structure of goal trees that is obtained from the resolution of undesirable situations in the initial goal graph by constructing and inserting intersection goal templates. In particular, we address the following situations: concatenations of i-arcs, the disjoint representation of child nodes in the goal tree, and cycles in the initial goal graph. We here specify the resolution patterns for collections of three goal templates; however, they are generally applicable for paths in the goal graph of any length. For programmatic realization, the resolution procedures require additional operations that need to be performed iteratively after each construction and insertion of intersection goal templates: (1) check for i-arcs in newly created levels of the s-arc hierarchy, and (2) remove redundant s-arcs and goal templates with the similarity degree *exact*. While the following merely addresses the resulting structure of the SDC graph, we shall specify the algorithms for the SDC graph refinement in Section 5. To illustrate the resolution procedures, we discuss them in our running example in the next section.¹

¹relevant terms from graph theory [11, 2]: *path* = walk with no double occurring vertices; *length* = number of visited vertices

Proposition 3.1 (Goal Graph I-Arc Concatenation Handling). *Let there be three goal templates $\mathcal{G}_1, \mathcal{G}_2$, and \mathcal{G}_3 with the similarity degrees $\text{intersect}(\mathcal{G}_1, \mathcal{G}_2)$ and $\text{intersect}(\mathcal{G}_2, \mathcal{G}_3)$. As implicitly $\text{disjoint}(\mathcal{G}_1, \mathcal{G}_3)$, we refer to this as a concatenation of i-arcs in the initial goal graph.*

The resolution pattern denotes a 2-level goal tree such that

- (i) *the goal templates at level 1 are: $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3$*
- (ii) *the goal templates at level 2 are: $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}, \mathcal{G}_{i(\mathcal{G}_2, \mathcal{G}_3)}$*
- (iii) *$\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}$ and $\mathcal{G}_{i(\mathcal{G}_2, \mathcal{G}_3)}$ are disjoint*

Proposition 3.2 (Representation of Non-Disjoint Child Nodes in Goal Tree). *Let there be three goal templates $\mathcal{G}_1, \mathcal{G}_2$, and \mathcal{G}_3 with the similarity degrees $\text{subsume}(\mathcal{G}_1, \mathcal{G}_2)$, $\text{subsume}(\mathcal{G}_1, \mathcal{G}_3)$, and $\text{intersect}(\mathcal{G}_2, \mathcal{G}_3)$. This denotes a 2-level goal graph wherein \mathcal{G}_2 and \mathcal{G}_3 are child nodes of \mathcal{G}_1 in the s-arc hierarchy; \mathcal{G}_2 and \mathcal{G}_3 are not disjoint but have a common solution.*

The resolution pattern denotes a 3-level goal tree such that \mathcal{G}_1 is at level 1, \mathcal{G}_2 and \mathcal{G}_3 are at level 2, and $\mathcal{G}_{i(\mathcal{G}_2, \mathcal{G}_3)}$ is at level 3.

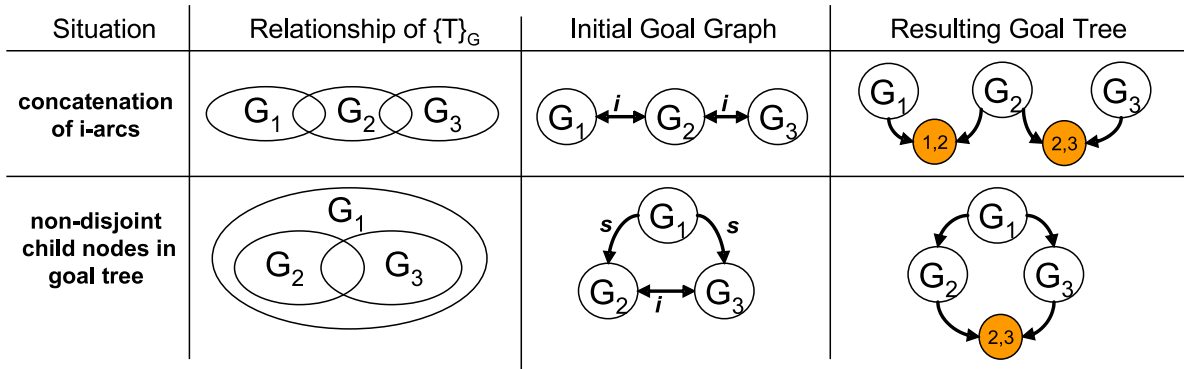


Figure 12: Resolution of Undesirable Situations in the Goal Graph

Figure 12 illustrates the resulting goal trees for both situations. When resolving a concatenation of i-arcs, the intersection goal templates at the second level describe those solutions that are common for both parent goal templates. To represent non-disjoint child nodes in the same level in a goal tree, their i-arc is replaced by an intersection goal template that is allocated on the next level of the goal tree. For both situations, the original goal templates are not changed and thus still have common solutions; merely the representation of their similarity is changed in the resulting goal tree.

The purpose of the resulting goal tree is two-fold: on the one hand, it shall enable efficient search of a goal template; on the other hand, we want to make use of the pre-filtering function of the *subsume* similarity degree for Web service discovery on the goal template level. For the latter, let $\{W\}_{\text{match}(\mathcal{G})}$ denote the set of usable Web services W for a goal template \mathcal{G} . It holds that $\text{subsume}(\mathcal{G}_1, \mathcal{G}_2) \Rightarrow \{W\}_{\text{match}(\mathcal{G}_1)} \subseteq \{W\}_{\text{match}(\mathcal{G}_2)}$ (cf. clause 3.11 of Theorem 3.1). Hence, in a goal tree wherein the only s-arcs occur, it thus holds that $\{W\}_{\text{match}(\mathcal{G}_{l+1})}$ for every goal template at level $l+1$ is always a subset of $\{W\}_{\text{match}(\mathcal{G}_l)}$ for every goal template at level l . In consequence, the deeper a goal template is allocated in the goal tree, the smaller is the set of Web services that are usable for solving it.

in a walk; *tree* = connected graph without cycles; *level* = (number of edges from the root to a node in a tree) + 1; *height* = length of the path of the longest branch / maximal level in a tree

For the search aspect, let us consider the resulting goal tree for the situation from Proposition 3.2. Imagine that we receive a goal instance GI whose closest goal template is $\mathcal{G}_{i(\mathcal{G}_2, \mathcal{G}_3)}$; the purpose of finding the closest goal template is that the lower its is allocated in the goal tree, the fewer Web services are potential candidates for solving GI (cf. Definition 1.5 from Section 1.2.3). When starting the search from level 1, we would need to perform three matchmaking operations: determine whether GI is a proper instance of \mathcal{G}_1 , then the same for \mathcal{G}_2 , and finally for $\mathcal{G}_{i(\mathcal{G}_2, \mathcal{G}_3)}$. The same number of operations would be required when following the branch of \mathcal{G}_2 . One could also define the resolution of i-arcs such that the child nodes in the resulting goal tree are truly disjoint. In this example, we could define three new goal templates such that $\mathcal{D}_{\mathcal{G}_2^-} = \mathcal{D}_{\mathcal{G}_2} \wedge \neg \mathcal{D}_{\mathcal{G}_2}$, $\mathcal{D}_{\mathcal{G}_{i(\mathcal{G}_2, \mathcal{G}_3)}} = \mathcal{D}_{\mathcal{G}_2} \wedge \mathcal{D}_{\mathcal{G}_3}$, and $\mathcal{D}_{\mathcal{G}_3^-} = \mathcal{D}_{\mathcal{G}_3} \wedge \neg \mathcal{D}_{\mathcal{G}_2}$; all three goal templates would be disjoint and allocated at level 2 such that \mathcal{G}_1 is the direct parent for \mathcal{G}_2^- , $\mathcal{G}_{i(\mathcal{G}_2, \mathcal{G}_3)}$, and \mathcal{G}_3^- . To find $\mathcal{G}_{i(\mathcal{G}_2, \mathcal{G}_3)}$ as the closest goal template for GI , the minimal amount of matchmaking operations is 2 (first for \mathcal{G}_1 and then for $\mathcal{G}_{i(\mathcal{G}_2, \mathcal{G}_3)}$), and the maximal number is 4 (first for \mathcal{G}_1 , then for \mathcal{G}_2^- and then for \mathcal{G}_3^- (or vice versa; both are negative), and finally for then $\mathcal{G}_{i(\mathcal{G}_2, \mathcal{G}_3)}$). Here, the efficiency of goal template search is dependent on the number of child nodes on each level of the goal tree – which is artificially increased by defining three disjoint goal templates for resolving two that have a common solution. In contrast, the computational costs for searching goal templates in the goal tree structure as defined above is only dependent on the height of the goal tree, which in most cases reveals a better search efficiency.

While above we have investigated the resolution of relatively easy situations, we now turn towards the resolution of cycles in the initial goal graph. An analysis reveals that only three types of cycles can occur in the initial goal graph: (1) if there is a cycle of i-arcs and there is no common solution for all involved goal templates, (2) if there is a cycle of i-arcs and there is at least one common solution for all involved goal templates, and (3) if there are three goal templates with at two i-arcs that constitute a cycle; Theorem 3.3 shows this formally. The resolution of each type results in a specific pattern of the obtained goal tree. We define these below along with an illustration in Figure 13.

Theorem 3.3 (Types of Cycles in Initial Goal Graph). *Given three goal templates $\mathcal{G}_1, \mathcal{G}_2$, and \mathcal{G}_3 , there can only be three types of cycles in the initial goal graph.*

Type 1: $intersect(\mathcal{G}_1, \mathcal{G}_2)$, $intersect(\mathcal{G}_2, \mathcal{G}_3)$, and $intersect(\mathcal{G}_3, \mathcal{G}_1)$ and $\neg \exists \tau. \tau \in (\{\tau\}_{\mathcal{G}_1} \cap \{\tau\}_{\mathcal{G}_2} \cap \{\tau\}_{\mathcal{G}_3})$

Type 2: $intersect(\mathcal{G}_1, \mathcal{G}_2)$, $intersect(\mathcal{G}_2, \mathcal{G}_3)$, and $intersect(\mathcal{G}_3, \mathcal{G}_1)$ and $\exists \tau. \tau \in (\{\tau\}_{\mathcal{G}_1} \cap \{\tau\}_{\mathcal{G}_2} \cap \{\tau\}_{\mathcal{G}_3})$

Type 3: $intersect(\mathcal{G}_1, \mathcal{G}_2)$, $intersect(\mathcal{G}_2, \mathcal{G}_3)$, and $subsume(\mathcal{G}_3, \mathcal{G}_1)$.

Proof. The following implications hold:

For type 1: $\langle 1 \rangle$ if one of the i-arcs is missing, then the situation between $\mathcal{G}_1, \mathcal{G}_2$, and \mathcal{G}_3 is a concatenation of i-arcs (cf. Proposition 3.1). $\langle 2 \rangle$ if all three i-arcs are given but $\neg \exists \tau. \tau \in (\{\tau\}_{\mathcal{G}_1} \cap \{\tau\}_{\mathcal{G}_2} \cap \{\tau\}_{\mathcal{G}_3})$ is false, then there is a cycle of type 2.

For type 2: $\langle 3 \rangle$ if all three i-arcs are given but $\exists \tau. \tau \in (\{\tau\}_{\mathcal{G}_1} \cap \{\tau\}_{\mathcal{G}_2} \cap \{\tau\}_{\mathcal{G}_3})$ is false, then there is a cycle of type 1. $\langle 4 \rangle$ if at least one of the arcs between $\mathcal{G}_1, \mathcal{G}_2$, and \mathcal{G}_3 does not define an *intersect* similarity degree, then any possible cycle is of type 3.

For type 3: $\langle 5 \rangle$ if only at least one s-arc exists between $\mathcal{G}_1, \mathcal{G}_2$, and \mathcal{G}_3 , then there can be a cycle of type 1 or 2. $\langle 6 \rangle$ if more than one s-arc exists between $\mathcal{G}_1, \mathcal{G}_2$, and \mathcal{G}_3 , then there is a goal tree with two non-disjoint child nodes at level 2 (cf. Proposition 3.2).

Clauses $\langle 1 \rangle - \langle 6 \rangle$ define under which conditions a certain type of cycles is not given. As these conditions cover all possible situations of similarity between $\mathcal{G}_1, \mathcal{G}_2$, and \mathcal{G}_3 , the three identified cycle types are the only possible ones that can occur in an initial goal graph. \square

Proposition 3.3 (Resolution Patterns for Cycles in Initial Goal Graph). *Let there be three goal templates $\mathcal{G}_1, \mathcal{G}_2$, and \mathcal{G}_3 such that their similarity degrees form a cycle of type 1, type 2, or type 3.*

The structural patterns of the goal trees that result from resolving the i-arcs are:

for type 1 cycle: a goal tree of height 2 such that

- (i) the original goal templates $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3$ are allocated at level 1
- (ii) all three intersection goal templates are allocated at level 2
- (iii) all goal templates at level 2 are disjoint because $\neg \exists \tau. \tau \in (\{\tau\}_{\mathcal{G}_1} \cap \{\tau\}_{\mathcal{G}_2} \cap \{\tau\}_{\mathcal{G}_3})$

for type 2 cycle: a goal tree of height 3 such that

COMMENT: there can also be another sub-type: pattern = type 1 + common intersection at level 3

- (i) the original goal templates $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3$ are allocated at level 1
- (ii) two of the three intersection goal templates are allocated at level 2
- (iii) the third intersection goal template is allocated at level 3; this describes the common solutions of $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3$, i.e. all $\tau \in (\{\tau\}_{\mathcal{G}_1} \cap \{\tau\}_{\mathcal{G}_2} \cap \{\tau\}_{\mathcal{G}_3})$

for type 3 cycle: let the initial situation be $\text{subsume}(\mathcal{G}_3, \mathcal{G}_1)$, $\text{intersect}(\mathcal{G}_1, \mathcal{G}_2)$, and $\text{intersect}(\mathcal{G}_2, \mathcal{G}_3)$.

The resulting goal tree is of height 3 such that

- (i) two of the three original goal templates are allocated at level 1:
 \mathcal{G}_3 (the source of the subsume-arc), and \mathcal{G}_2 that is only connected by i-arcs
- (ii) level 2 contains \mathcal{G}_1 (the target of the subsume-arc), and the intersection goal template $\mathcal{G}_{i(\mathcal{G}_2, \mathcal{G}_3)}$
- (iii) the second intersection goal template $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}$ is allocated at level 3;
this describes all $\tau \in (\{\tau\}_{\mathcal{G}_1} \cap \{\tau\}_{\mathcal{G}_2} \cap \{\tau\}_{\mathcal{G}_3})$ as the common solutions of $\mathcal{G}_1, \mathcal{G}_2$, and \mathcal{G}_3 .

Situation	Relationship of $\{\mathcal{T}\}_{\mathcal{G}}$	Initial Goal Graph	Resulting Goal Tree
cycle (type 1)			
cycle (type 2)			
cycle (type 3)			

Figure 13: Resolution of Cycles in the Goal Graph

3.3.3 Illustrative Example

In order to illustrate the preceding definitions, the following illustrates the resolution of a cycle in the initial goal graph within our best restaurant research example. We consider an example for a cycle of type 3 (cf. Theorem 3.3). For this, we construct three goal templates whose requested functionalities differ with respect to the locality of the input city and the type of the requested restaurant.

Table 4 shows the goal templates, their similarity degrees, and the intersection goal templates that are relevant for discussion. For the sake of simplicity, we use the same numbering of the goal templates as in Proposition 3.3. Below, we explain the stepwise resolution of the cycle as defined above. For better traceability, Figure 14 illustrates the structure of the goal graph in each step.

Table 4: Goal Templates, Similarity Degree, and Intersection Goal Templates in Example

original goal templates	similarity degree
\mathcal{G}_1 : find best restaurant in an Austrian city	$intersect(\mathcal{G}_1, \mathcal{G}_2)$
\mathcal{G}_2 : find best French restaurant in any city of the world	$intersect(\mathcal{G}_2, \mathcal{G}_3)$
\mathcal{G}_3 : find best restaurant in a European city	$subsume(\mathcal{G}_3, \mathcal{G}_1)$
intersection goal templates	
$\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}$: find best French restaurant in an Austrian city	
$\mathcal{G}_{i(\mathcal{G}_2, \mathcal{G}_3)}$: find best French restaurant in a European city	

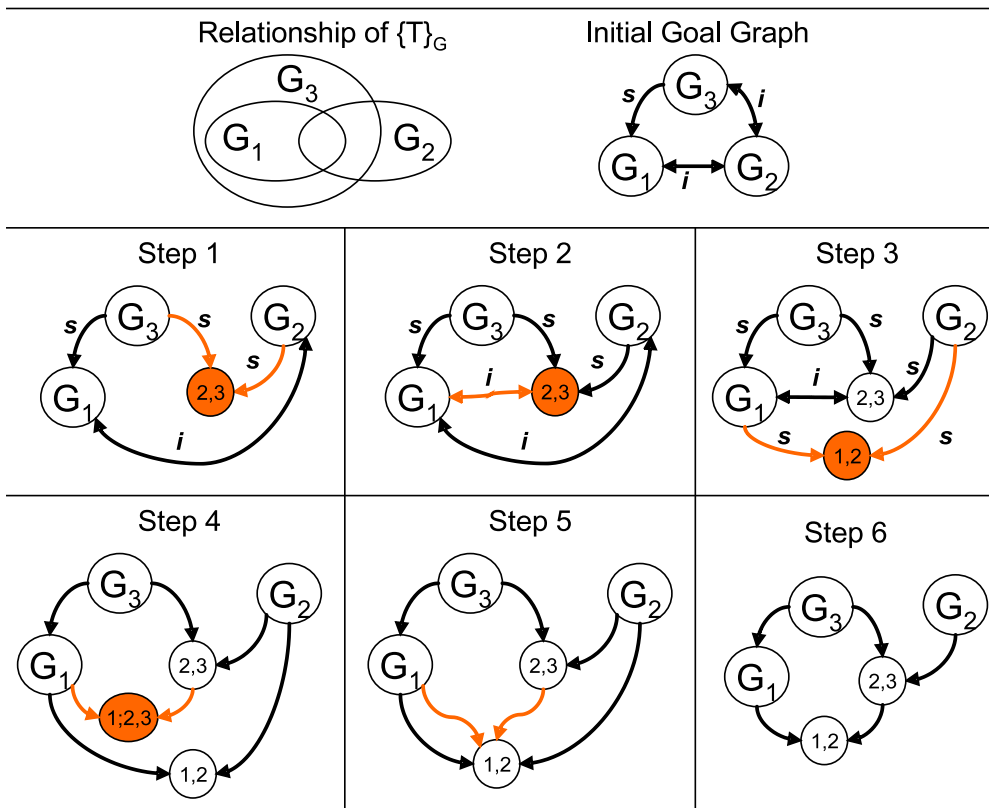


Figure 14: Example for Resolving a Cycle in the Initial Goal Graph

We perform the resolution of i-arcs in a top-down manner – a bottom-up approach could require redundant iterations. Thus, at first we resolve the i-arc between \mathcal{G}_2 and \mathcal{G}_3 as the most top-level i-arc in the initial goal graph. We obtain the intersection goal template $\mathcal{G}_{i(\mathcal{G}_2, \mathcal{G}_3)}$, and insert it into the goal graph (i.e. remove the i-arc and insert to s-arcs; cf. Step 1 in Figure 14). We obtain a goal graph of height 2. Next, we need to check the similarity degree between the newly inserted goal template. We observe that there is a new i-arc between \mathcal{G}_1 and $\mathcal{G}_{i(\mathcal{G}_2, \mathcal{G}_3)}$ (cf. Step 2). However, the most top level i-arc is the one between \mathcal{G}_1 and \mathcal{G}_2 . So, we resolve this next and obtain a goal graph of height 3 with $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}$ at the lowest level (cf. Step 3).

Next, we address the newly created i-arc between \mathcal{G}_1 and $\mathcal{G}_{i(\mathcal{G}_2, \mathcal{G}_3)}$. We obtain another intersection goal template that could be inserted into the goal graph as shown in Step 4. We now have resolved all i-arcs, so that we obtain a goal tree whose edges are s-arcs without exception; we thus omit the arc-labels in the figure. However, when checking the similarity degree of the new intersection goal template and $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}$ – the only goal template that exists at the same level – we observe that their similarity degree is *exact*: $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}$ describes the common solutions of \mathcal{G}_1 and \mathcal{G}_2 , and the new intersection goal template describes the solutions that are common for all three original goal templates. Hence, we do not keep the newly created intersection goal template but merely connect the new s-arcs to $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}$ (cf. Step 5).

As the final step, we remove redundant arcs from the goal tree. The first redundant arc is one of those that connect \mathcal{G}_1 and $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}$; both are s-arcs with the same source and target, so we can omit one of them. The second one is the s-arc between \mathcal{G}_2 and $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}$. This arc is redundant because (1) we can reach the target via $\mathcal{G}_{i(\mathcal{G}_2, \mathcal{G}_3)}$, and (2) it may decrease the efficiency of goal template search. Imagine that we receive a goal instance whose closest goal template is $\mathcal{G}_{i(\mathcal{G}_2, \mathcal{G}_3)}$. When commencing the search at \mathcal{G}_2 and we first follow the direct arc to $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}$ (for which the matchmaking is not successful), then we have to start again from \mathcal{G}_2 before reaching $\mathcal{G}_{i(\mathcal{G}_2, \mathcal{G}_3)}$. Without the direct arc between \mathcal{G}_2 and $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}$, we reach the search target with one matchmaking step less. Hence, we remove that s-arc as well and finally reach the goal tree structure shown in Step 6 of the figure – which is the same as defined in Proposition 3.3.

3.3.4 Formal Properties of Refined SDC Graph

We conclude the specification of the SDC graph by summarizing the refinements that have been defined above. In particular, we address the properties that result from transforming the initial goal graph into goal trees by the resolution of i-arcs. We also address the omittance of redundant WG mediators in the discovery cache, which is the last open aspect from the analysis of the initial SDC graph above in Section 3.3.1.

Definition 3.5 (Structure of Refined SDC Graph). *The refined SDC graph consists of two layers. The upper one is an **unconnected set of goal trees** wherein goal templates are connected by GG mediators such that the only occurring similarity degree is *subsume*. The lower layer is the **discovery cache** wherein WG mediators connect goal templates with their usable Web services. The possible usability degrees of a Web service for solving a goal template are *exact*, *plugin*, *subsume*, or *intersect*.*

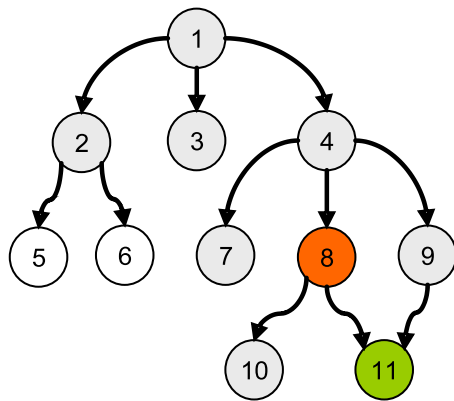
This states that in the refined SDC graph, the initial goal graph is refined into a set of goal trees. A goal tree is a collection of goal templates that are connected by GG mediators such that the only occurring similarity degree is *subsume*. Such a goal tree does not contain any cycles (cf. Proposition 3.3) or any other undesirable situations (cf. Proposition 3.1 and Proposition 3.2), thus satisfies aspects (i) and (ii) of requirement 6. Each connected sub-graph of goal templates from the initial SDC graph is refined into a goal tree. However, there still can be unconnected goal templates in the refined SDC graph, so that we obtain a set of unconnected goal trees (cf. Figure 10). One could artificially connect these by defining a goal template \mathcal{G}_0 with $\phi^{\mathcal{D}\mathcal{G}_0} = \text{true}$ such that every consistent goal template becomes a child node in one goal tree. However, this does not appear to be necessary for the application purpose of the SDC graph.

The first application purpose of the SDC graph is to enable efficient search of the most appropriate goal template for an new incoming goal instance. The aim of this search is to detect the goal template \mathcal{G} whereof the incoming goal instance $GI(\mathcal{G})$ is a proper instantiation such that out of the possible goal templates \mathcal{G} is allocated at the deepest level in the goal tree. The reason is that the deeper \mathcal{G} is located in the goal tree, the smaller is the set of Web services that are usable for \mathcal{G} , and, in consequence, the smaller is the number of possible candidates for solving $GI(\mathcal{G})$ (cf. Definition 1.5. While we shall specify the algorithm below in Section 4, let us here examine the computational costs for the goal template search as an important property of the refined SDC graph. The search is performed in a depth-first manner as discussed above: it commences at the root of a goal tree, checks for this and every child node whether $GI(\mathcal{G})$ is a proper instantiation, and terminates at the goal template for whose child nodes this is not given. In the worst case, we need to inspect all goal templates on every level of the goal tree that is located on the path from the root node to search target \mathcal{G} , as well as all child nodes of \mathcal{G} .

Proposition 3.4 (Efficiency for Goal Template Search). *Let T be a goal tree wherein the goal template \mathcal{G} is the search target. Let $n(l)$ denote the number of goal templates on a level of the goal tree. Let $p(\mathcal{G})$ be the path in T from the root to \mathcal{G} , and let $n(l, p(\mathcal{G}))$ denote the number of goal templates on each level of $p(\mathcal{G})$. Let $n(l_{\mathcal{G}+1})$ denote the number of goal templates that are child nodes of \mathcal{G} in T .*

The computational costs for finding \mathcal{G} in T is $O(n(l, p(\mathcal{G})) + n(l_{\mathcal{G}+1}))$.

Figure 15 illustrates the search for two target goal templates in a goal tree. If the target is \mathcal{G}_{11} , we start from the root and then inspect all goal templates at the second level. For the third level, apart from \mathcal{G}_7 we only need to inspect \mathcal{G}_8 or \mathcal{G}_9 because they are not disjoint (as they have \mathcal{G}_{11} as a common child node – one can image this to be derived from Proposition 3.2). If we follow the path via \mathcal{G}_8 , we may also need to investigate \mathcal{G}_{10} , while for the path via \mathcal{G}_9 this step can be omitted. If the target is \mathcal{G}_8 , we follow the same path for the first two levels of the goal tree. On the third level, we may first investigate \mathcal{G}_7 , then \mathcal{G}_9 , and finally \mathcal{G}_8 . We also need to ensure that none of the child nodes of \mathcal{G}_8 is a possible target. Hence, the search terminates after determining that both \mathcal{G}_{10} and \mathcal{G}_{11} are not possible targets.



if target = \mathcal{G}_{11} :

$p(\mathcal{G}_{11})_1 = \mathcal{G}_1, \mathcal{G}_4, \mathcal{G}_8, \mathcal{G}_{11}$ or $p(\mathcal{G}_{11})_2 = \mathcal{G}_1, \mathcal{G}_4, \mathcal{G}_9, \mathcal{G}_{11}$

$l_1, (p(\mathcal{G}_{11})) = \mathcal{G}_1$ $n(l_1, p(\mathcal{G}_{11})) = 1$

$l_2, (p(\mathcal{G}_{11})) = \mathcal{G}_2, \mathcal{G}_3, \mathcal{G}_4$ $n(l_2, p(\mathcal{G}_{11})) = 3$

$l_3, (p(\mathcal{G}_{11})) = \mathcal{G}_7, \mathcal{G}_8$ $n(l_3, p(\mathcal{G}_{11})) = 2$

$l_4, (p(\mathcal{G}_{11})_1) = \mathcal{G}_{10}, \mathcal{G}_{11}$ $n(l_4, p(\mathcal{G}_{11})_1) = 2$

$l_4, (p(\mathcal{G}_{11})_2) = \mathcal{G}_{11}$ $n(l_4, p(\mathcal{G}_{11})_2) = 1$

$l_{\mathcal{G}_{11}+1} = \emptyset$ $n(l_{\mathcal{G}_{11}+1}) = 0$

costs for finding \mathcal{G}_{11} : minimal = 7, maximal = 8

if target = \mathcal{G}_8 :

$p(\mathcal{G}_8) = \mathcal{G}_1, \mathcal{G}_4, \mathcal{G}_8$

$l_1, (p(\mathcal{G}_8)) = \mathcal{G}_1$ $n(l_1, p(\mathcal{G}_8)) = 1$

$l_2, (p(\mathcal{G}_8)) = \mathcal{G}_2, \mathcal{G}_3, \mathcal{G}_4$ $n(l_2, p(\mathcal{G}_8)) = 3$

$l_3, (p(\mathcal{G}_8)) = \mathcal{G}_7, \mathcal{G}_8, \mathcal{G}_9$ $n(l_3, p(\mathcal{G}_8)) = 3$

$n(l_{\mathcal{G}_8+1}) = \mathcal{G}_{10}, \mathcal{G}_{11}$ $n(l_{\mathcal{G}_8+1}) = 2$

costs for finding \mathcal{G}_8 = 9

Figure 15: Illustration of Goal Template Search

The second application purpose of the SDC graph is to serve as the basis for efficient Web service discovery. For this, the *discovery cache* captures knowledge on the usability of Web services for goal templates in WG mediators that define a directed arc with the usability degree of the target Web service for the source goal template (cf. Definition 3.2). We recall from Definition 1.5 that for properly performing Web service discovery on the goal instance level at runtime, we must know the precise usability degree of a Web service for the corresponding goal template. However, on the basis of the rules for inferring the usability of a Web service with respect to the similarity degree of adjacent goal templates as specified in Section 3.2, we can omit certain WG mediators in the SDC graph. In particular, we can omit all WG mediators at child nodes in the goal tree whose usability degree can be inferred directly from a WG mediator with the same target that is defined at the parent node.

Let us clarify this by the example illustrated in Figure 16. Let there be a goal tree with three goal templates $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3$ such that $\text{subsume}(\mathcal{G}_1, \mathcal{G}_2)$ and $\text{subsume}(\mathcal{G}_1, \mathcal{G}_3)$. We consider four Web services such that each one is usable for the parent node \mathcal{G}_1 under a different usability degree. For $\text{exact}(\mathcal{G}_1, W_1)$, we can directly infer that $\text{plugin}(\mathcal{G}_2, W_1)$ and $\text{plugin}(\mathcal{G}_3, W_1)$ (cf. clause 3.1 in Theorem 3.1). Hence, we can omit the WG mediators that connect W_1 with \mathcal{G}_2 and with \mathcal{G}_3 . The same holds under $\text{plugin}(\mathcal{G}_1, W_1)$, cf. clause 3.2. However, we must explicitly define the WG mediators at child nodes in the goal tree if the usability degree of a Web service for the parent node is *subsume* or *intersect*. If $\text{subsume}(\mathcal{G}_1, W_3)$ as illustrated in the figure, the usability degree of W_3 for a child node of \mathcal{G}_1 can be any of the five degrees, even *disjoint* (cf. clauses 3.3 - 3.7). Also under $\text{intersect}(\mathcal{G}_1, W_4)$, this Web service might not be usable for a child node of \mathcal{G}_1 (cf. clauses 3.8 - 3.10). Besides, as discussed above, under the *subsume* similarity degree the set of usable Web services for every child node in the goal tree is a subset of the usable Web services for the parent goal template (cf. clause 3.11).

This allows to reduce the discovery cache to consists of only those WG mediators that are necessary for enabling efficient Web service discovery on both the goal template and the goal instance level. Therewith, the refined SDC graph finally satisfies the third aspect of requirement 6.

Proposition 3.5 (Minimality of Discovery Cache). *Let \mathcal{G} be a goal template such that there exists at least one \mathcal{G}_2 that is a child node in the goal tree. For all Web services whose usability degree for \mathcal{G} is exact or plugin, only WG mediators with \mathcal{G} as the source are defined but none with \mathcal{G}_2 as the source.*

This is the minimal tree on the usability of available Web services for the existing goal templates such that:

- (i) every WG mediator that is removed disconnects the SDC graph, and
- (ii) any additional WG mediator is redundant.

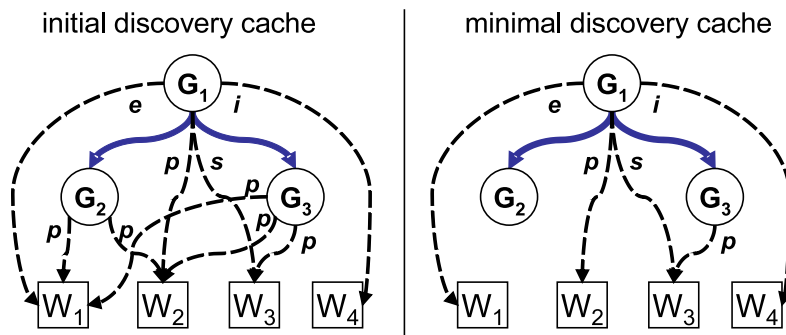


Figure 16: Omittance of WG Mediators in Discovery Cache

4 Web Service Discovery with SDC

This section specifies Web service discovery with the Semantic Discovery Caching technique. While we have summarized the approach for Web service discovery in the introduction (*cf.* Section 1.2), we here define the algorithms that utilize the SDC graph for realizing an efficient Web service discovery. The operations for management and evolution support of the SDC graph are specified in Section 5; the complete algorithm for the SDC technique that integrates discovery and management operations is provided in Appendix C.

We have outlined in the introduction that the author's work presents a refinement of the Web service discovery framework that has been proposed for WSMO (*cf.* Section 1.1). This distinguishes three operations: (1) the formulation of a client desire as a goal, (2) the discovery of usable Web services for solving goals on an abstract level, and (3) the refinement of discovery results for consuming a real world service. In this work, we specify all three operations such that each one uses the knowledge kept in the SDC graph to the maximal extent. We therewith satisfy requirement 7 on the integration of the SDC technique into an overall architecture for Semantic Web services, and realize an efficient and scalable Web service discovery process with respect to requirements 3 and 4. Illustrated in Figure 17, the operations are defined as follows.

- 1. Goal Formulation:** this is concerned with the formulation of the objective that a client wants to achieve as a goal description. For this, the client browses the goal templates existing in the SDC graph, and chooses one for creating a goal instance. When the goal instance is received by the system, we internally assign it to the closest goal template in order to minimize the search space for Web service discovery. This operation is performed at runtime, i.e. when a client specifies a new objective and, for automated solving, submits this to the system in form of a goal instance.
- 2. Web Service Discovery on the Goal Template Level:** this determines the usability degree of the available Web services for goal templates by matchmaking of their formal functional descriptions. The result is captured as the discovery cache in the SDC graph. This operation is performed at design time – respectively orthogonal to runtime – i.e. whenever a goal template or a Web service description is added, modified, or removed.
- 3. Web Service Discovery on the Goal Instance Level:** this determines the usability of Web service for solving a goal instance. For this, the matchmaker checks whether the functional descriptions of the corresponding goal template and the Web service are satisfiable under the input binding that is defined in the goal instance. This operation is performed at runtime. The corresponding goal template is the one determined by the goal formulation procedure; for matchmaking, only those Web services need to be taken into account that are declared as usable for that goal template in the SDC graph.

In the following, we specify each operation in detail. For human understandability, we specify the algorithms in Java-style pseudo code; this reflects their realization in the planned prototype implementation. We also define the semantic matchmaking procedures that are related to each discovery operation. We specify these in a first-order logic framework on the basis of the formal functional descriptions and the approach for semantic matchmaking as outlined in the introduction (*cf.* Section 1.2). Moreover, we specify the algorithms for a **refined** SDC graph – i.e. its goal graph consists of goal trees with *subsume* as the only occurring similarity degree, and the discovery cache does not contain any redundant WG mediators (*cf.* Definition 3.5) – and in a stable environment (i.e. no updates or changes of the goal template and Web service descriptions take place). These aspects are ensured by the integration of the respective management and evolution algorithms (*cf.* Section 5) into the overall SDC algorithm that is specified in Appendix C.

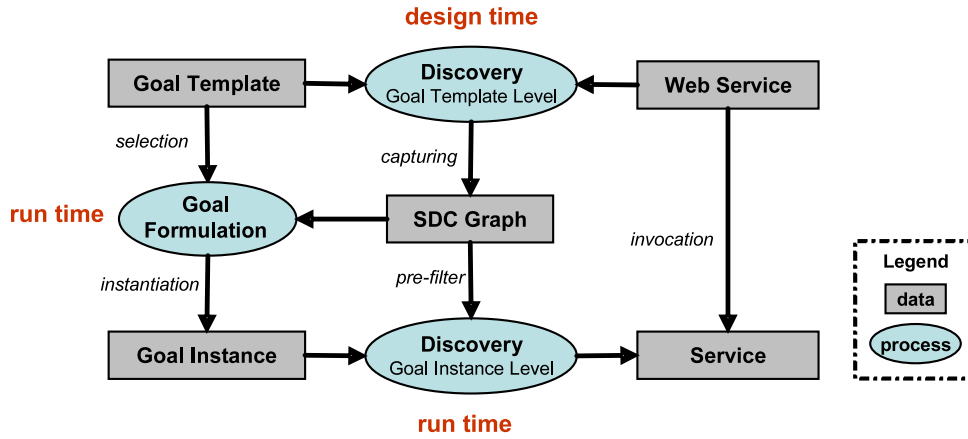


Figure 17: Operations for Web Service Discovery with SDC

4.1 Goal Instance Formulation and Goal Template Discovery

We commence with the goal formulation procedure. This covers the creation of a goal instance that describes the concrete objective that the client wants to achieve, and the assignment of this goal instance to the closest existing goal template. The aim of the goal formulation procedure is to declare a goal template \mathcal{G} as the corresponding one for a newly created goal instance $GI = (\mathcal{G}, \beta)$ such that \mathcal{G} is located as deep in the goal tree as possible while GI is still a proper instantiation of \mathcal{G} . The reason is that the lower \mathcal{G} is allocated in the goal tree, the fewer Web service are usable for \mathcal{G} and hence are possible candidates for GI . The following first explains the goal formulation procedure in our framework, and then specifies the matchmaking and algorithm for search the most proper goal template in the SDC graph.

Let us consider a scenario from our running example. A client wants to find the best restaurant in Vienna (the Austrian capital). Let us assume that there are some goal templates for finding the best restaurant in a city that is provided as input. Let the goal templates differ with respect to the locality of the input city, so that the SDC graph contains the goal tree shown in Figure 18: the root node is \mathcal{G}_1 that describes the objective of finding best restaurant in any city of the world, the second level is differentiates continents, and the third level distinguishes countries that are located in continents. The figure also illustrates the steps for goal formulation: at first the client browses the goal tree and chooses a goal template \mathcal{G}_c that appears to be suitable for describing the objective. Then, the client creates a goal instance $GI = (\mathcal{G}_c, \beta)$ for the chosen goal template. Imagine that in this example the client chooses $\mathcal{G}_c = \mathcal{G}_2$ (for Europe), and defines $\beta = \{city | Vienna\}$ as the input binding. However, the most appropriate goal template is \mathcal{G}_5 (best restaurant in Austrian cities). Hence, as the final step, we need to search for the most appropriate goal template and, for further processing, declare this to be the corresponding goal template for the goal instance.

The first two steps of the goal formulation procedure are not SDC-specific, but are required for any system that realizes goal-based Web service usage. For technical realization, the SDC graph browsing can best be supported by a graphical user interface. One of the most promising tools for this is WSMT, the graphical user interface of the WSMX system that provides adequate browsing support for repositories of WSMO elements (ontologies, Web services, goals, mediators) [20]. For the second step of defining a goal instance for a chosen goal template, form-based editors for defining concrete values for the inputs required in chosen goal template appear to be a sophisticated solution. Such graphical support for goal instance creation by clients are provided by related system implementation such as IRS [6] or SWF [43].

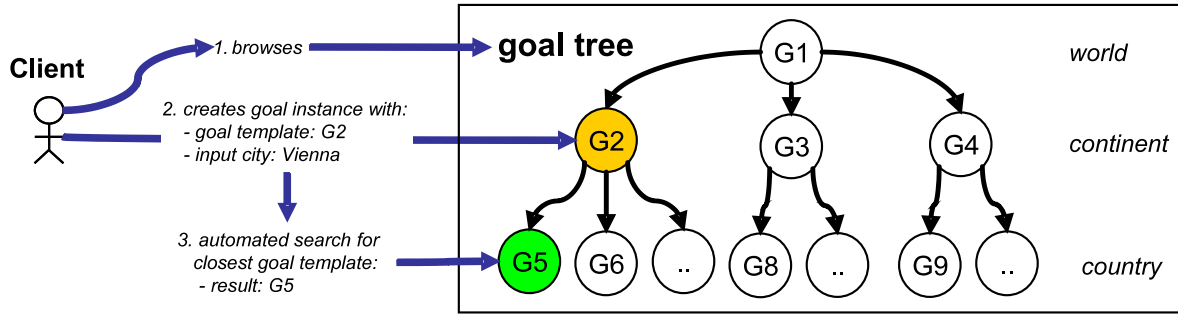


Figure 18: Illustration of Goal Formulation Procedure

In the context of SDC, we thus concentrate on the third step of finding the closest goal template \mathcal{G} for a given goal instance $GI = (\mathcal{G}_c, \beta)$. We shall refer to this automated operations as *goal template search*. Essentially, it commences at the goal template \mathcal{G}_c that has been chosen for goal formulation by the client, and tests whether GI is a proper instantiation of the child nodes of \mathcal{G}_c . This is repeated in a depth-first manner until the closest goal template \mathcal{G} has been found. The result is a revision of the corresponding goal template for the goal instance such that $GI = (\mathcal{G}_c, \beta) \rightarrow GI' = (\mathcal{G}, \beta)$. The following first defines required matchmaking, and then specifies the algorithm.

4.1.1 Matchmaking for Goal Template Search

The matchmaking required for goal template search is to determine whether the goal instance GI defined by the client is a proper instantiation of a goal template \mathcal{G} . This is given if the precondition ϕ^{pre} of $\mathcal{D}_{\mathcal{G}}$ as the functional description of \mathcal{G} is satisfiable under the input binding β that is defined in GI .

To formally define this, we recall the definition of goal templates and goal instances in our framework from Section 1.2. A goal template describes an objective by a functional description of the form a functional description $\mathcal{D}_{\mathcal{G}} = (\Sigma, \Omega, IF, \phi^{\mathcal{D}_{\mathcal{G}}})$. Therein, $IF = (i_1, \dots, i_n)$ are the input variables that occur as free variables in the precondition ϕ^{pre} and the effect ϕ^{eff} , and $\phi^{\mathcal{D}_{\mathcal{G}}} = [\phi^{pre}]_{\Sigma^{pre} \rightarrow \Sigma_D} \Rightarrow \phi^{eff}$ defines the implication semantics. An input binding $\beta : (i_1, \dots, i_n) \rightarrow \mathcal{U}_{\mathcal{A}}$ is a total function that assigns concrete values of the universe $\mathcal{U}_{\mathcal{A}}$ to the IF -variables (cf. Definition 1.2). We consider all functional descriptions to be *consistent*, i.e. there must be an input binding β under which there exists at least one Σ -interpretation I that is a model of \mathcal{D} , i.e. $\exists I, \beta. I, \beta \models \phi^{\mathcal{D}}$. If this is not given, then a Web service implementation that provides the functionality described by \mathcal{D} is not realizable [18]; in consequence, there cannot be any Web service that is usable for a goal template with an inconsistent functional description.

A goal instance $GI = (\mathcal{G}, \beta)$ is created by defining an input binding β for functional description $\mathcal{D}_{\mathcal{G}}$ of a goal template \mathcal{G} . We call β *complete* for $\mathcal{D}_{\mathcal{G}}$ if it defines a concrete value assignment for at least each IF -variable in $\mathcal{D}_{\mathcal{G}}$. For example, if $IF = (i_1, i_2, i_3)$, then $\beta = \{i_1|v_1, i_2|v_2, i_3|v_3\}$ is complete because it defines a concrete value for each of the three input variables. It may also contain more value assignments (e.g. also $i_4|v_4$). Given a complete input binding β , we can instantiate the functional description of a goal template by replacing every occurrence of all IF -variables in $\mathcal{D}_{\mathcal{G}}$ with the concrete values defined in β . From this, we can obtain an instantiated functional description $[\mathcal{D}_{\mathcal{G}}]_{\beta}$ that formally describes the objective that is formulated in the goal instance. $[\phi^{\mathcal{D}_{\mathcal{G}}}]_{\beta}$ does not contain any free variables, so that its truth value for this formula can be determined under every Σ -interpretation.

A goal instance $GI = (\mathcal{G}, \beta)$ properly instantiates a goal template \mathcal{G} if β is complete for $\mathcal{D}_{\mathcal{G}}$ and if

$[\mathcal{D}_{\mathcal{G}}]_{\beta}$ is satisfiable. This means that β must assign a concrete value for at least every *IF*-variable in $\mathcal{D}_{\mathcal{G}}$ – otherwise the truth value of $[\mathcal{D}_{\mathcal{G}}]_{\beta}$ might not be determined – and there must be a Σ -interpretation I that is a model for $\phi^{\mathcal{D}_{\mathcal{G}}}$ under β . Then, this I represents a sequence of states $\tau = (s_0, \dots, s_n)$ that is a solution for \mathcal{G} under the input binding β defined in GI , and thus τ is also a solution for GI . Because $\mathcal{D}_{\mathcal{G}}$ is consistent (see above), with this definition of proper instantiation it is ensured that the possible solution for a goal instance $GI(\mathcal{G})$ are always a subset of those for its corresponding goal template, i.e. $\{\tau\}_{GI(\mathcal{G})} \subseteq \{\tau\}_{\mathcal{G}}$ (cf. Definition 1.1).

Definition 4.1 (Goal Instantiation). *Let GI be a goal instance that defines an input binding β . Let \mathcal{G} be a goal template that has a functional description $\mathcal{D}_{\mathcal{G}} = (\Sigma, \Omega, IF_{\mathcal{G}}, \phi^{\mathcal{D}_{\mathcal{G}}})$ with $\phi^{\mathcal{D}_{\mathcal{G}}} = [\phi^{pre}]_{\Sigma_D^{pre} \rightarrow \Sigma_D} \Rightarrow \phi^{eff}$. Let $[\phi^{\mathcal{D}_{\mathcal{G}}}]_{\beta}$ denote the formula that is derived by replacing all occurrences of every *IF*-variable in $\phi^{\mathcal{D}_{\mathcal{G}}}$ by the concrete values that are assigned in β .*

*We say that GI **properly instantiates** \mathcal{G} , denoted by the predicate $instantiates(GI, \mathcal{G})$, if and only if:*

- (i) β is complete for $\mathcal{D}_{\mathcal{G}}$, and
- (ii) $[\phi^{\mathcal{D}_{\mathcal{G}}}]_{\beta}$ is satisfiable.

This defines matchmaking condition for determining whether a goal instance is a proper instantiation of a goal template. We illustrate this in our running example below in the context of the goal template search algorithm. It is to note that if β is complete for a goal template \mathcal{G}_1 then it is also complete for every goal template \mathcal{G}_2 that is a child node of \mathcal{G}_1 in the goal tree. This holds because of the matchmaking condition for the *subsume* similarity degree that implicitly requires the compatibility of the *IF*-variables of $\mathcal{D}_{\mathcal{G}_1}$ and $\mathcal{D}_{\mathcal{G}_2}$: $\Omega_{\mathcal{A}} \models \forall \beta. \phi^{\mathcal{D}_{\mathcal{G}_1}} \Leftarrow \phi^{\mathcal{D}_{\mathcal{G}_2}}$ (cf. Table 3). If there is a β such that $[\mathcal{D}_{\mathcal{G}_2}]_{\beta}$ is satisfiable then β must be complete for $\mathcal{D}_{\mathcal{G}_2}$; if $subsume(\mathcal{G}_1, \mathcal{G}_2)$, then also $[\mathcal{D}_{\mathcal{G}_1}]_{\beta}$ must be satisfiable and therefore β must be complete for $\mathcal{D}_{\mathcal{G}_1}$.

Proposition 4.1 (Transitive Completeness of Input Bindings in a Goal Tree). *Let $\mathcal{G}_i, \mathcal{G}_j$ be goal templates such that $subsume(\mathcal{G}_i, \mathcal{G}_j)$. An input binding β is complete for a functional description $\mathcal{D} = (\Sigma, \Omega, IF, \phi^{\mathcal{D}})$ with $IF = (i_1, \dots, i_n)$ if β assigns a concrete value for at least every $i \in IF$.*

It holds that: β is complete for $\mathcal{D}_{\mathcal{G}_i}$ if and only if β is complete for $\mathcal{D}_{\mathcal{G}_j}$.

4.1.2 Algorithm for Goal Template Search

We now can define the algorithm for goal template search. For a newly defined goal instance GI this performs a depth-first search for the goal template \mathcal{G} whereof GI denotes a proper instantiation so that \mathcal{G} is allocated deepest in the goal tree. The procedure of goal formulation as illustrated above is an approximation towards a real world setting, assuming that the client selects a goal template at least from the correct goal tree in the SDC graph. We here specify the goal template search algorithm for the more general case: finding of the closest template for a goal instance for which we do not know a corresponding goal template.

Listing 1 below provides the algorithm for this. The *main* part defines the overall control for the *goalTemplateSearch* method. Initially, the *target* (i.e. the goal template that we search) is defined to be empty. At first, we invoke the *findRootNode* method that searches for a goal template that is a root node of one of the goal trees in the SDC graph. A root node is a goal template that does not have any parents: $root(\mathcal{G}) \Leftrightarrow \neg \exists \mathcal{G}_2. subsume(\mathcal{G}_2, \mathcal{G})$; isolated goal templates are also considered as root nodes – merely their respective goal tree is of height 1. The search tests iteratively for all root nodes if the incoming goal instance is a proper instantiation of the goal template; this is given if $instantiates(GI, \mathcal{G})$ holds for the currently inspected goal template (cf. Definition 4.1). By definition, the similarity of each two root goal templates is

disjoint – otherwise the goal templates would be connected in the SDC graph. Hence, the halting condition of the *forall* loop in the *findRootNode* method is reached if a matching root goal template has been found. If the incoming goal instance *GI* is not a proper instantiation of any root goal template, then the result of the goal template search is empty, meaning that *GI* is not a proper instantiation of any existing goal template.

If we have found a root goal template whereof *GI* is a proper instantiation, this is the intermediate *target* of the search. We then search the respective goal tree for the closest goal template, which is performed by the *findChildNode* method. This tests whether *GI* is a proper instantiation of any of the child nodes of the current target; if yes, this child node is the intermediate *target*, and the *findChildNode* method is iteratively invoked for the new *target*. The halting condition in the *else*-part of the *forall* loop returns the current *target* as the search result if there does not exist any child node whereof *GI* is a proper instantiation; the halting condition outside the *forall* loop returns the root goal template as the search result.

We therewith realize a depth-first search whose computational complexity we have already discussed above in Proposition 3.4. Note that the iteration in the *findChildNode* method allows to skip unnecessary matchmaking steps for non-disjoint goal templates \mathcal{G}_1 and \mathcal{G}_2 on the same level of the goal tree: regardless of which one is investigated at first, the iteration will directly proceed with examining the next level of the goal where the intersection goal template $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}$ is allocated (cf. Proposition 3.2).²

```

// type declarations
GI := goalinstance;
G1,G2 := goaltemplate;
target := goaltemplate;
// main
goalTemplateSearch(GI);
target = null;
findRootNode(GI);
if (! target = null) then
  findChildNode(target);
return target;
// find root of goal tree
findRootNode(GI){
  forall (root(G)) {
    if instantiates (GI,G) then
      target = G;
      return target;
  }
  return target;
}
// find child node in goal tree
findChildNode(G1) {
  forall (subsume(G1,G2)) {
    if instantiates (GI,G2) then
      target = G2;
      findChildNode(target);
    else
      return target;
  }
  return target;
}

```

Listing 1: Goal Template Search Algorithm

²Syntax for the pseudo code used for algorithm specifications:

$:=$ is a data type declaration, *name(input)* is the name and the input value of a method; *forall(condition)* defines a loop that is iterated for all objects for which the condition is satisfied until the halting condition is reached; *if (condition) then (action) else (action)* defines a conventional guarded action; *return(value)* is the halting condition that returns the value; *null* denotes that the value of an object is empty, $=$ defines a value assignment for an object, $!$ defines the negation of the subsequent condition.

For illustration, let us recall the example from Figure 18. If we have already given a corresponding goal template within the definition of the incoming goal instance (i.e. as in the above procedure), we can (1) skip the *findRootNode* method, and (2) commence the *findChildNode* method directly from the already known goal template. This allows to obtain a better efficiency at runtime.

In the above example, the goal instance GI created by the client defines $\beta = \{city|vienna\}$, and it specifies \mathcal{G}_2 as the initial corresponding goal template. \mathcal{G}_2 defines $IF = \{?x\}$, the precondition $\phi^{pre} = city(?x) \wedge locatedIn(?x, europe)$, and the effect $\phi^{eff} = \forall ?y. out(?y) \Leftrightarrow restaurant(?y) \wedge locatedIn(?y, ?x) \wedge \neg(\exists ?z. restaurant(?z) \wedge locatedIn(?z, ?x) \wedge better(?z, ?y))$. GI is a proper instantiation of \mathcal{G}_2 because β satisfies ϕ^{pre} and the output object $?y$ is the best restaurant in Vienna as requested. So, $target = \mathcal{G}_2$ for the first iteration of the *findChildNode* method. Imagine that the *forall* loop first chooses \mathcal{G}_6 that is for finding the best restaurant in a German city with $\phi^{pre} = city(?x) \wedge locatedIn(?x, germany)$. GI is not a proper instantiation of \mathcal{G}_6 because $\forall ?y. locatedIn(?y, austria) \Rightarrow \neg locatedIn(?y, germany)$. Hence, the intermediate target remains to be \mathcal{G}_2 . Next, we try \mathcal{G}_5 that defines $\phi^{pre} = city(?x) \wedge locatedIn(?x, austria)$ with the same input variable as all other goal templates in the goal tree. Obviously, GI is a proper instantiation of this goal template. Hence, the intermediate *target* is changed to \mathcal{G}_5 , and the *findChildNode* method is invoked again. Because \mathcal{G}_5 does not have any child nodes in the goal tree, the outer halting condition holds and the result of the search is \mathcal{G}_5 – which is the closest goal template for GI .

4.2 Web Service Discovery – Goal Template Level

We now turn towards Web service discovery. This section defines the matchmaking and algorithm for discovery on the goal template level that is performed at design time, respectively orthogonal to runtime (cf. Figure 17); we address discovery on the goal instance level below in Section 4.3.

The aim of the algorithms is to realize an efficient and scalable procedure for Web service discovery on the goal template level (cf. requirements 3 and 4). We therefore make extensive use of the inference rules for usability degrees as defined in Theorem 3.1 (cf. Section 3.2). We here specify the algorithm for determining the usability degree of all usable Web services for a new goal template that is inserted in the SDC graph. The result of this operation constitutes the discovery cache for the new goal template. As we shall discuss below in detail, this algorithm contains all methods that are relevant for Web service discovery on the goal template level during the evolution of the SDC graph (cf. Section 5). The following first explains the matchmaking for determining the usability degree of a Web service for a goal template, and then specifies the algorithm for discovering all usable Web services for a new goal template.

4.2.1 Matchmaking for Web Service Usability Degree Determination

Web service discovery on the goal template level is concerned with determining the usability degree of Web services for a specific goal template. The usability degree provides sufficiently rich information for our discovery approach, as it allows to perform efficient Web service discovery on the goal instance level at runtime (cf. Definition 1.5). Hence, this is the only information stored in the discovery cache of the SDC graph (cf. Definition 3.2).

The usability degree of a Web service W for a goal template \mathcal{G} is determined by matchmaking of their functional descriptions \mathcal{D}_W and $\mathcal{D}_{\mathcal{G}}$, as defined in Section 1.2.3. Relevant for the following discussion, Table 5 recalls the definition of the usability degrees and shows the logical relationship between them. For technical realization of a matchmaker, the conditions for each degree as defined here can be implemented straight forward in a inference engine for the chosen specification language – e.g. as proof obligations for a first-order logic theorem prover as done in [40].

Table 5: Definition and Relationship of Usability Degrees

Degree Definition	Formal Relationships
exact (\mathcal{G}, W): $\Omega_{\mathcal{A}} \models \forall \beta. \phi^{\mathcal{D}_{\mathcal{G}}} \Leftrightarrow \phi^{\mathcal{D}_W}$	$plugin \wedge subsume \Rightarrow exact$
plugin (\mathcal{G}, W): $\Omega_{\mathcal{A}} \models \forall \beta. \phi^{\mathcal{D}_{\mathcal{G}}} \Rightarrow \phi^{\mathcal{D}_W}$	$plugin \Rightarrow intersect$
subsume (\mathcal{G}, W): $\Omega_{\mathcal{A}} \models \forall \beta. \phi^{\mathcal{D}_{\mathcal{G}}} \Leftarrow \phi^{\mathcal{D}_W}$	$subsume \Rightarrow intersect$
intersect (\mathcal{G}, W): $\Omega_{\mathcal{A}} \models \exists \beta. \phi^{\mathcal{D}_{\mathcal{G}}} \wedge \phi^{\mathcal{D}_W}$	$intersect \Rightarrow \neg disjoint$
disjoint (\mathcal{G}, W): $\Omega_{\mathcal{A}} \models \neg \exists \beta. \phi^{\mathcal{D}_{\mathcal{G}}} \wedge \phi^{\mathcal{D}_W}$	

The matching conditions for the degrees are not precise in the sense that if the condition for a degree is satisfied, the actual relationship \mathcal{D}_W and $\mathcal{D}_{\mathcal{G}}$ can maybe also be expressed as a different degree. For example, if $plugin(\mathcal{G}, W)$, then maybe also $exact(\mathcal{G}, W)$ holds – in the case that \mathcal{D}_W and $\mathcal{D}_{\mathcal{G}}$ are logically equivalent. For our application purpose, we want to dispose this ambiguity: we must know the precise usability degree in order to beneficially apply the inference rules as well as for realizing an efficient runtime discovery. The right column of Table 5 imposes the following preference order: $exact > plugin, subsume > intersect$. We always prefer the degree with the higher preference because the higher the preference, the more beneficially we can deal with the usability degrees among adjacent goal templates in the SDC graph.

To achieve this, we do not need to modify the matchmaking conditions – we merely need to define a suitable control algorithm for invoking the matchmaker. Listing 2 shows our solution for this. Initially, we set the usability degree to *disjoint*. We first check whether the condition for the *plugin* degree is satisfied; if yes, we update the usability degree to *plugin*. Then, we do the same for the *subsume* degree. If the conditions for both are satisfied – the information is kept in boolean constants – then we update the usability degree to *exact*. If this is not given, we check the condition for the *intersect* degree and update the usability degree accordingly. The *matchmakingUsability* method returns the determined degree; in case that the Web service is not usable for the goal template, the resulting degree remains *disjoint*. Although this is just a minor issue towards an efficient Web service discovery, this algorithm requires maximal 3 matchmaking operations and hence is more efficient than respective algorithms defined in related works (e.g. [28]).

```

// type declarations
G := goaltemplate;
W := webservice;
d := usabilityDegree;
plugin, subsume := boolean;
// main
matchmakingUsability(G,W){
  plugin = false;
  subsume = false;
  d = disjoint;
  if ( plugin(G,W) ) then {
    plugin = true;
    d = plugin; }
  if ( subsume(G,W) ) then {
    subsume = true;
    d = subsume; }
  if ( (plugin = true) and (subsume = true)) then {
    d = exact; }
  else {
    if ( intersect(G,W) ) then d = intersect; }
  return d;
}

```

Listing 2: Algorithm for Unambiguous Usability Degree Determination

Another relevant aspect is that only the minimal knowledge needed to perform the matchmaking is loaded into the matchmaker. As discussed in the context of requirement 4 (*cf.* Section 2.2.2), this is an essential pre-requisite for maintaining the scalability of the discovery engine under a large amount of available Web services. To guarantee this, for every invocation of the *matchmakingUsability* method we only load the minimal required knowledge into the matchmaker. This is the functional descriptions \mathcal{D}_G of the goal template and \mathcal{D}_W of the Web service, and the background ontologies that are used in \mathcal{D}_G and \mathcal{D}_W . These can be several distinct ontologies – i.e. $\Omega_1, \dots, \Omega_n$ – and they might be heterogeneous. We consider all mismatches between the ontologies to be resolved *before* the matchmaker is invoked. This can be achieved by respective data level mediation techniques, e.g. those developed in the context of the WSMO mediation framework [26]. Moreover, we can expect to make use of the work from Francois Scharffe on an algorithm for providing a global view on an integrated ontology for a particular application purpose as proposed in [31]. Let $\Omega_{i(G,W)}$ be such an integrated, global ontology for all background knowledge used in the \mathcal{D}_G and \mathcal{D}_W , we merely need to extend this with the additions for dynamic symbols and then utilize this as the minimal relevant background knowledge for the matchmaking.

4.2.2 Algorithm for Discovering All Usable Web Services for a New Goal Template

We now define the algorithm for SDC-enabled Web service discovery for a new goal template that has been inserted into the SDC graph. This covers all aspects relevant for discovery on the goal template level whereof parts can be reused in other situations, e.g. if an existing one is modified or removed (we shall address this in detail in the context of SDC graph evolution management, *cf.* Section 5).

We consider the following situation as the context of the goal template discovery algorithm. There is a **refined** SDC graph (*cf.* Definition 3.5) given so that (1) there are existing goal templates that are organized in a set of goal trees wherein the only occurring similarity degree is *subsume*, and (2) the discovery cache of the SDC graph is minimized, i.e. WG mediators for child nodes in a goal tree whose usability degree is directly inferable from the parent node are omitted. A new goal template \mathcal{G}_{new} is defined and has already been inserted into the goal graph (i.e.: the GG mediators that connect \mathcal{G}_{new} to its neighbors in the SDC graph are defined, and all possibly occurring i-arcs have already been resolved). So, we have a refined SDC graph wherein there is a new goal template \mathcal{G}_{new} for which there does not yet exist a discovery cache.

The purpose of the Web service discovery algorithm that we specify here is to detect all Web services that are usable for the new goal template \mathcal{G}_{new} along with the respective usability degree. The aim is to make extensive use of the inference rules for determining usability degrees in the SDC graph as defined in Theorem 3.1 (*cf.* Section 3.2). The new goal template \mathcal{G}_{new} can be allocated at three different position in the SDC graph. We need to differentiate the Web service discovery for each possible position:

1. **\mathcal{G}_{new} is a new child node in an existing goal tree:** \mathcal{G}_{new} is allocated in an existing goal tree at any position but not as the root node. We shall denote this situation as $child(\mathcal{G}_{new})$ such that \mathcal{G}_{new} has at least one parent: $child(\mathcal{G}_{new}) : \exists \mathcal{G}. subsume(\mathcal{G}, \mathcal{G}_{new})$.

Here, we know that only those Web services can be usable for \mathcal{G}_{new} that are usable for its parents (the afore mentioned filter functionality of the *subsume* similarity degree). For any Web service W that is usable for a parent of \mathcal{G}_{new} under the *exact* or *plugin* usability degree, we can directly infer that its usability degree is $plugin(\mathcal{G}_{new}, W)$ (*cf.* clauses 3.1 and 3.2 of Theorem 3.1). However, we do not need to inspect these Web services in our algorithm, because the respective WG mediators will be removed afterwards in order to maintain the minimality of the discovery cache (*cf.* Proposition 3.5). For the Web services that are usable for a parent of \mathcal{G}_{new} under the *subsume* or *intersect* degree, we can apply the respective inference rules from clauses 3.3 - 3.10 in Theorem 3.1.

2. \mathcal{G}_{new} is a new root node of an existing goal tree: \mathcal{G}_{new} is a root node of an existing goal tree so that it only has outgoing s-arcs. We shall denote this situation as $root(\mathcal{G}_{new})$ such that \mathcal{G}_{new} does not have any parent but one or more child nodes: $root(\mathcal{G}_{new}) : \forall \mathcal{G}. \neg subsume(\mathcal{G}, \mathcal{G}_{new}) \wedge \exists \mathcal{G}_2. subsume(\mathcal{G}_{new}, \mathcal{G}_2)$.

In this situation, every Web service that is usable for any child node of \mathcal{G}_{new} is usable for \mathcal{G}_{new} . For this, we can make use of the inference rules under the *plugin* similarity degree (cf. clauses 2.1 - 2.8 of Theorem 3.1). However, there can be Web services that are usable for \mathcal{G}_{new} but not for any of its child nodes (cf. clause 2.9 of Theorem 3.1). Hence, we also need to perform matchmaking for all other available Web services. Redundant WG mediators that might be created in this operation must be removed afterwards (we shall cover this later in Section 5).

3. \mathcal{G}_{new} is new disconnected node in the SDC graph: there does not exist any goal template in the SDC graph that has a common solution with \mathcal{G}_{new} , so that \mathcal{G}_{new} appears as a disconnected node in the SDC graph. We shall denote this situation as $disconnected(\mathcal{G}_{new})$ such that $disconnected(\mathcal{G}_{new}) : \forall \mathcal{G}. disjoint(\mathcal{G}_{new}, \mathcal{G})$. Here, we can not make beneficial use of any inference rules. Thus, we need to perform matchmaking with all other available Web services.

Listing 3 shows the algorithm that covers all three situations. The result of the main method *discovery*(G) is a set of triples $d_{usability}(\mathcal{G}, W)$ that constitutes the discovery cache for the goal template that the algorithm is invoked with. The operator $+$ denotes the addition of an element to the discovery cache. The method *childNodesDiscovery*(G) performs Web service discovery for the first situation identified above. It considers all parent nodes of the new goal template, and determines the usability degree for every Web service under consideration of the inference rules for the *subsume* similarity degree. As discussed above, it omits all Web services that are usable under the *exact* or *plugin* degree. The method *rootNodeDiscovery*(G) performs Web service discovery for the second situation. It first inspects the Web services that are usable for all child nodes of the new goal template, thereby considering the inference rules for the *plugin* similarity degree. Secondly, it performs matchmaking for all Web services that are not usable for any child node. For this, the operator *in* checks whether an element is existing in a set, and the method *matchmakingUsability*(G, W) invokes the algorithm specified above in Listing 2. For the third situation, matchmaking is performed for all available Web services.

This algorithm performs the minimal number of matchmaking operations that is needed for determining the usability degree of Web services that are usable for a newly defined goal template. This is performed orthogonal to runtime, so that the computational efficiency of algorithm does not influence the runtime efficiency of the overall discovery procedure. We hence omit a computational costs analysis of this algorithm.

```

// type declarations
G, G2 := goaltemplate;
W := webservice;
d := usabilityDegree;
discoverycache := {d(G, W)};
// main
discovery(G){
  discoverycache = {};
  if child(G) then childNodeDiscovery(G);
  if root(G) then rootNodeDiscovery(G);
  if disconnected(G) then {
    forall (W){
      matchmakingUsability(G, W);
      if (! d = disjoint) then discoverycache = discoverycache + d(G, W);
    }
  }
  return discoverycache; }

```

```

// discovery for G if it is a child node in an existing goal tree
childNodeDiscovery(G){
  forall ( G2 and subsume(G2,G) ) {
    forall ( W and subsume(G2,W) ) {
      matchmakingUsability(G,W);
      if (! d = disjoint ) then
        discoverycache = discoverycache + d(G,W);
    }
    forall ( W and intersect(G2,W) ) {
      if ( plugin(G,W) ) then d = plugin;
      if ( intersect (G,W) ) then d = intersect;
      discoverycache = discoverycache + d(G,W);
    }
  }
  return discoverycache;
}

// discovery for G if it is a root node of an existing goal tree
rootNodeDiscovery(G){
  forall ( G2 and subsume(G,G2) ) {
    forall ( W and exact(G2,W) ) {
      d = subsume;
      discoverycache = discoverycache + d(G,W);
    }
    forall ( W and subsume(G2,W) ) {
      d = subsume;
      discoverycache = discoverycache + d(G,W);
    }
    forall ( W and plugin(G2,W) ) {
      plugin, subsume := boolean;
      d = intersect;
      if ( plugin(G,W) ) then {
        plugin = true;
        d = plugin; }
      if ( subsume(G,W) ) then {
        subsume = true;
        d = subsume; }
      if ( (plugin = true) and (subsume = true)) then {
        d = exact; }
      discoverycache = discoverycache + d(G,W);
    }
    forall ( W and intersect(G2,W) ) {
      if ( subsume(G,W) ) then d = plugin;
      if ( intersect (G,W) ) then d = intersect;
      discoverycache = discoverycache + d(G,W);
    }
  }

  forall ( W and !(W in discoverycache) ) {
    matchmakingUsability(G,W);
    if (! d = disjoint ) then
      discoverycache = discoverycache + d(G,W);
  }

  return discoverycache;
}

```

Listing 3: Algorithm for Web Service Discovery for a new Goal Template

4.3 Web Service Discovery – Goal Instance Level

The final operation in the SDC-enabled Web service discovery framework is the discovery on the goal instance level. Performed at runtime, the purpose is to determine those Web services that are actually usable for a given goal instance that represents a concrete client objective. For this, we perform matchmaking on the goal instance level for those Web services that are usable for the corresponding goal template. As soon as a usable Web service has been detected, the subsequent steps for resolving the client request are performed. The following recalls the matchmaking technique for discovery on the goal instance level, then specifies the algorithm for runtime Web service discovery in the SDC framework, and finally analyzes the computational efficiency of the algorithm.

4.3.1 Matchmaking and Procedure

The central matchmaking technique for runtime Web service discovery is to determine whether a Web service is usable for solving the concrete objective described in a goal instance. This is given if the execution of a Web service provides a solution for the goal instance when it is invoked with the concrete input values that are defined in the goal instance. We briefly recall the formal matchmaking approach for this as explained in the introduction (*cf.* Section 1.2.3).

A goal instance $GI = (\mathcal{G}, \beta)$ defines an input binding β for the functional description \mathcal{D}_G of its corresponding goal template \mathcal{G} . Given a β , we can instantiate the functional descriptions by replacing every occurrence of each *IF*-variable with the respective value defined in β . We obtain $[\phi^{\mathcal{D}_G}]_\beta$ as the instantiated functional description of the goal template; in fact this formally describes the concrete objective that is represented in GI . Accordingly, $[\phi^{\mathcal{D}_W}]_\beta$ describes the subset of possible solutions of the Web service W when it is invoked with the concrete input values defined in β . Because of the formal relationship of a goal instance and its corresponding goal template, we have defined a matchmaking approach that requires the minimal number of matchmaking operations for determining the usability of a Web service for solving a goal instance. Relevant in our context, we recall this from Definition 1.5.

Let \mathcal{D}_G describe the requested functionality in a goal template \mathcal{G} . Let $GI(\mathcal{G})$ be a goal instance of \mathcal{G} that defines an input binding β . Let W be a Web service, and let \mathcal{D}_W be a functional description such that $W \models_{\mathcal{A}} \mathcal{D}_W$. W is usable for solving $GI(\mathcal{G})$ if and only if:

- (i) *exact*($\mathcal{D}_G, \mathcal{D}_W$) *or*
- (ii) *plugin*($\mathcal{D}_G, \mathcal{D}_W$) *or*
- (iii) *subsume*($\mathcal{D}_G, \mathcal{D}_W$) *and* $\bigwedge \Omega_{\mathcal{A}} \wedge [\phi^{\mathcal{D}_W}]_\beta$ is satisfiable, *or*
- (iv) *intersect*($\mathcal{D}_G, \mathcal{D}_W$) *and* $\bigwedge \Omega_{\mathcal{A}} \wedge [\phi^{\mathcal{D}_G}]_\beta \wedge [\phi^{\mathcal{D}_W}]_\beta$ is satisfiable.

Essentially, this defines that if a Web service W is usable for the corresponding goal template under the *exact* or *plugin* degree, then it is also usable for the goal instance. If the usability degree of W for the goal template is *subsume* or *intersect*, then we need to perform additional matchmaking on the instantiated functional descriptions in order to determine the usability of W for solving the goal instance. Moreover, only those Web services that are usable for the goal template can be usable for the goal instance while no others can be (*cf.* Definition 1.1). The matching conditions under the *subsume* and *intersect* usability degrees can be implemented as a conventional satisfiability test in the chosen reasoner. For our first-order logic approach, we have defined this as a proof obligation of the form $\exists O. \text{output}(O, [\phi^{\mathcal{D}_G}]_\beta) \wedge \text{output}(O, [\phi^{\mathcal{D}_W}]_\beta)$: this tests whether there exists a Σ -interpretation that is a common model of the instantiated functional descriptions and defines a common output object [41].

Above, we have recalled the matchmaking technique that allows to determine whether a Web service is usable for solving a goal instance. However, the matchmaking alone does not yet satisfy all the requirements for runtime discovery that we have identified in Section 2. To achieve this, we integrate the matchmaking in the overall discovery procedure as illustrated in Figure 19.

At first, we perform the goal template search algorithm specified above in Section 4.1 in order to find the closest goal template \mathcal{G} for the goal instance $GI = (\mathcal{G}_c, \beta)$ that the client has defined. We obtain a revision of the goal instance $GI = (\mathcal{G}, \beta)$ such that \mathcal{G} is a proper goal template for GI that is located the deepest in the goal tree. The number of Web services that are usable for \mathcal{G} is minimal in comparison to all other existing goal templates whereof GI is a proper instantiation. Thus, we ensure that the search space for matchmaking the minimal and therewith satisfy requirement 3 on the computational efficiency. Secondly, we realize the interleaved Web service discovery from requirement 1: whenever the matchmaking has detected a usable Web service for the goal instance, the subsequent reasoning steps for automatically solving the goal instance are invoked; the search continues orthogonal to runtime. Thirdly, with respect to requirement 4 on the scalability, we only load the minimal knowledge into the matchmaker for each single matchmaking operation (the same as explained above for matchmaking on the goal template level, cf. Section 4.2).

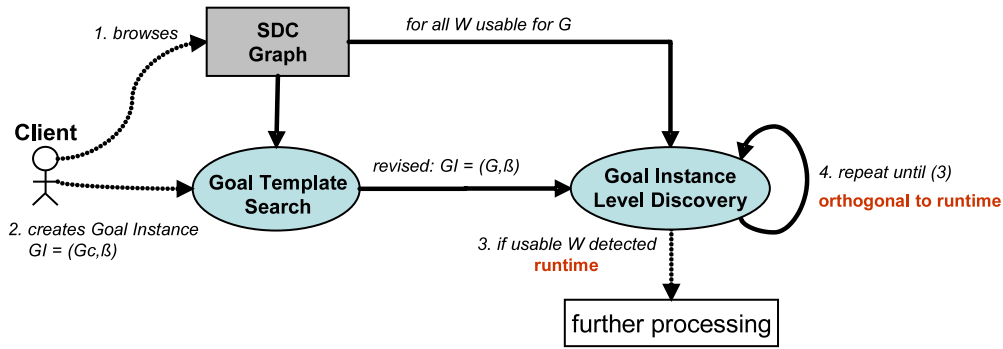


Figure 19: Runtime Operations for SDC-enabled Web Service Discovery

4.3.2 Algorithm for Runtime Web Service Discovery

Listing 4 provides the algorithm for the integrated runtime Web service discovery as explained above. The main method *discovery*(GI) first invokes the *goalTemplateSearch* algorithm that we have specified in Listing 1 (cf. Section 4.1). This returns \mathcal{G} as the most appropriate goal template whereof the goal instance GI (the one for which *discovery*(GI) is invoked) is a proper instantiation. We then define $GI = (\mathcal{G}, \beta)$.

The first subroutine is *lookup*(G) that is only invoked if the corresponding goal template is allocated as a child node in the SDC graph (see possible positions of goal templates above). This finds usable Web services by inspecting the omitted WG mediators in the SDC graph. Recalling from Proposition 3.5 (cf. Section 3.3.4), WG mediators whose usability degree can be directly inferred are omitted in the SDC graph in order to maintain the minimality of the discovery cache. In particular, all WG mediators that connect a child node in a goal tree to a Web service that is usable under the *exact* or *plugin* degree for a parent node. For example, if *subsume*($\mathcal{G}_1, \mathcal{G}_2$) and *plugin*(\mathcal{G}_1, W), then we only store the arc $WGM_1 = (\mathcal{G}_1, W, plugin)$ but omit $WGM_2 = (\mathcal{G}_2, W, plugin)$ because WGM_2 can be directly inferred from WGM_1 . This omission of WG mediators is iteratively applied throughout a goal tree. Thus, non-redundant WG mediators are always allocated at the highest possible level in a goal tree – i.e. at the root node in the most general case.

It holds that if a Web service W is usable for a goal template \mathcal{G}_1 under the *exact* or *plugin* degree, then it is always usable under the *plugin* degree for every child node: $\forall \mathcal{G}_2. \text{plugin}(\mathcal{G}_2, W) \leftarrow \text{subsume}(\mathcal{G}_1, \mathcal{G}_2) \wedge (\text{exact}(\mathcal{G}_1, W) \vee \text{plugin}(\mathcal{G}_1, W))$, cf. clauses 3.1 and 3.2 of Theorem 3.1. Coincidentally, if a Web service W is usable under the *plugin* degree for a goal template \mathcal{G} that is referenced in a goal instance $GI((\mathcal{G}, \beta))$, then we know that W is usable for solving GI without the need of invoking a matchmaker.

These relationships are utilized within the method $\text{lookup}(G)$ to efficiently detect usable Web services for the given goal instance without matchmaking. It commences at the goal template that is referenced in the goal instance description, and checks if there is a Web service that is usable under the *exact* or *plugin* degree for any of its parent nodes. As soon as such a Web service has been detected, this is returned as the result of the overall algorithm. This is repeated in an inverse depth-first manner: it is iteratively invoked for each parent node before considering another parent node at the same level. This ensure to find usable Web services via omitted WG mediators with the minimal computational costs.

The second method $\text{goalInstanceMatching}(GI)$ performs matchmaking on the goal instance level. This is invoked whenever $\text{lookup}(G)$ is not usable or if it did not return a usable Web service. As explained above, we know without matchmaking that a Web service W is usable for $GI = (\mathcal{G}, \beta)$ if $\text{exact}(\mathcal{G}, W)$ or $\text{exact}(\mathcal{G}, W)$. Under the other two usability degrees, we need to perform matchmaking. For this, the method $\text{satisfiable}(W, \text{inputs})$ tests the additional matching condition if $\text{subsume}(\mathcal{G}, W)$, and $\text{satisfiable}(G, W, \text{inputs})$ tests the one for the case of $\text{intersect}(\mathcal{G}, W)$. If the latter is not given, then there does not exists any Web service that can solve the goal instance.

```

// type declarations
GI := goalinstance;
inputs := inputbinding;
G := goaltemplate;
W := webservice;
// main
discovery(GI) {
  G = goalTemplateSearch(GI);
  GI = (G, inputs);
  lookup(G);
  goalInstanceMatching(GI);
}
// usability lookup for inferable usability degrees
lookup(G) {
  if ( child(G) ) then {
    forall ( G2 and subsume(G2, G) ) {
      forall ( W and ( exact(G2, W) or plugin(G2, W) ) ) {
        return W;
      }
      lookup(G2);
    } } }
// goal instance level matchmaking
goalInstanceMatching(GI) {
  forall ( W and exact(G, W) or plugin(G, W) ) {
    return W; }
  forall ( W and subsume(G, W) ) {
    if ( satisfiable (W, inputs) ) then
      return W; }
  forall ( W and intersect(G, W) ) {
    if ( satisfiable (G, W, inputs) ) then
      return W;
    else
      return systemout = 'goal instance can not be solved';
  } }
}

```

Listing 4: Algorithm for Runtime Web Service Discovery

With respect to the importance of this algorithm for the use of the SDC-enabled Web service discovery in real-world applications, we illustrate it in our running example. For this, we recall the scenario discussed above in the context of goal formulation and discuss the two situations shown in Figure 20. The goal instance GI requests to find the best restaurant in Vienna. There are three goal templates for finding the best restaurants in cities with different locality restrictions: \mathcal{G}_2 for Europe, \mathcal{G}_5 for Germany, and \mathcal{G}_6 for Austria. Obviously, and as discussed above, the closest goal template is \mathcal{G}_6 so that the goal instance is defined as $GI = (\mathcal{G}_6, \{i_1 | Vienna\})$.

As the first situation, let there be a Web service W_1 that allows to find the best restaurant in any city of the world (left hand side in the figure). We easily see that its usability degree for \mathcal{G}_2 is *plugin* and thus the same for \mathcal{G}_5 and \mathcal{G}_6 (as both are child nodes of \mathcal{G}_2). Here, the WG mediators to connect W_1 with \mathcal{G}_5 or with \mathcal{G}_6 are omitted. In this situation, we can find W_1 to be usable for the goal instance by the *lookup(G)* method: in the first iteration, it considers the level where \mathcal{G}_2 is located at. Because there is W_1 as a Web service that is usable under the *plugin* degree, we know that it is usable for solving GI . As said above, no matchmaking is needed in this method.

In the second situation, let there be a different Web service W_2 for searching the best Italian restaurant in a European city. The usability degree of W_2 for all three goal templates is *intersect*: it only provides the best restaurant if, by accident, the best restaurant in the input city is of type Italian [41]. Here, the *lookup(G)* method will not find a usable Web service because it does not exist any Web service that is usable under the *exact* or *plugin* degree for any parent node of \mathcal{G}_6 . When executing the *goalInstanceMatching(GI)* method, we enter the *forall* loop for the *intersect* degree. If we assume that the best restaurant in Vienna is not of type Italian, then the additional matching condition for W_2 to be usable for GI is not satisfied: W_2 would provide the best Italian restaurant in Vienna – this is not identical to the best restaurant in Vienna that is requested by GI .

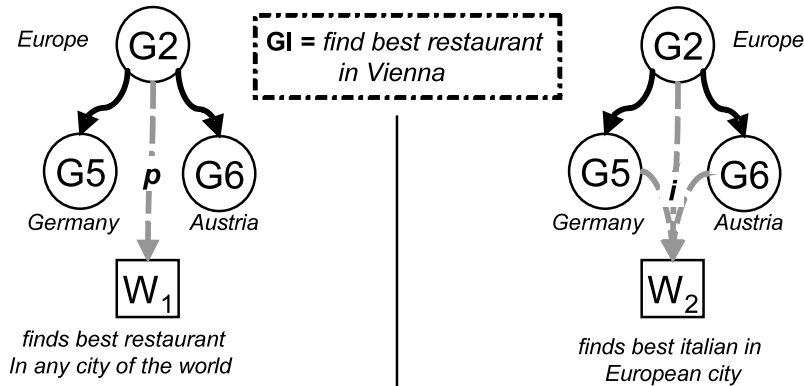


Figure 20: Illustrative Example for Runtime Web Service Discovery

4.3.3 Computational Efficiency Analysis

The runtime Web service discovery defines the complete procedure that is processed for SDC-enabled Web service discovery at runtime. In fact, the increase in computational efficiency for Web service discovery that is aspirated with SDC technique is primarily dependent on the efficiency of the runtime Web service discovery. With respect to this, we conclude the specification with a computational cost analysis of the algorithm specified in Listing 4.

As we have shown in the example discussion above, the algorithm can be very efficient. In particular, if there is a Web service that is usable under the *exact* or *plugin* degree for the corresponding goal template, then it can determine the usability of Web service for solving a goal instance without the need of invocation a matchmaker. However, in order to express the computational costs in terms of the Big-O-notation, we must consider the worst case scenario. There are two such scenarios: (1) if there does not exist any usable Web service for solving the goal instance, or (2) if the usability degree of the only Web services that are usable for the corresponding goal template is *subsume* or *intersect*.

In both situations, the *goalInstanceMatching(GI)* method needs to perform matchmaking for the additional conditions; in the worst case, every Web service that is usable under either the *subsume* or the *intersect* degree must be inspected. Moreover, if the corresponding goal template is located as a child node in the SDC graph, then the *lookup(G)* method will inspect every path from the corresponding goal template up to the root node without success. However, discovery with the *lookup(G)* method as well as for Web services under the *exact* or *plugin* degree does not require any matchmaking. This can be realized with conventional technologies – e.g. in a database system that allows to process large amounts of such simple lookup operations in a time range of milliseconds. This is not the case for discovery steps that require matchmaking: although the actual matchmaking can (in theory) be performed fast, the invocation and communication with the matchmaker requires much time. Because of this, for our analysis we only consider the operations that require matchmaking while neglecting the lookup operations.

Analyzing the two worst case scenarios, we observe that the computational costs of the *goalInstanceMatching(GI)* method is proportional to the number of Web services that are usable for \mathcal{G} that is corresponding goal template for the goal instance $GI = (\mathcal{G}, \beta)$. If there is a Web service that is usable for \mathcal{G} under the *exact* or *plugin* degree, then the algorithm does not need to perform matchmaking. If the only Web services for \mathcal{G} are usable under the *subsume* or the *intersect* degree, then matchmaking needs to be performed for them. Hence, when $\{W\}_{match(\mathcal{G})}$ is the set of Web services that is usable for \mathcal{G} , then we can express the computational complexity of the *goalInstanceMatching(GI)* method as the size of this set: $|\{W\}_{match(\mathcal{G})}|$. We then can express the efficiency of the overall runtime discovery algorithm by extending this with the complexity of the goal template search algorithm (cf. Proposition 3.4).

Proposition 4.2 (Efficiency of Runtime Web Service Discovery). *Let $GI = (\mathcal{G}, \beta)$ be a goal instance that is described by its corresponding goal template \mathcal{G} and an input binding β . Let \mathcal{G} be an element of a refined SDC graph. The computational costs of Web service discovery for GI is*

$O((n(l, p(\mathcal{G})) + n(l_{\mathcal{G}+1})) + |\{W\}_{match(\mathcal{G})}|)$ with:

- (i) $O(n(l, p(\mathcal{G})) + n(l_{\mathcal{G}+1}))$ is the computational complexity for finding \mathcal{G} with
 - $n(l, p(\mathcal{G}))$ is the number of goal templates on each level of the path p from the root node of the goal tree to \mathcal{G}
 - $n(l_{\mathcal{G}+1})$ is the number of goal templates that are child nodes of \mathcal{G}
- (ii) $|\{W\}_{match(\mathcal{G})}|$ is the number of Web services that are usable for \mathcal{G} .

This defines the maximal computational costs for finding a Web service for a goal instance at runtime. The elements of the computational complexity are dependent of each other: the lower the corresponding goal template is allocated in the goal tree, the longer the goal template search can take – but the lower is the number of Web services. Furthermore, whenever there is a Web service that is usable under the *exact* or *plugin* degree, no matchmaking is required for discovery at the goal instance level. We shall discuss the efficiency increase that is achievable with SDC-enabled Web service discovery in the context of an extensive applicability study.

5 SDC Graph Maintenance

This section completes the specification of the SDC technique with the operations and algorithms for management and evolution of the SDC graph. We except the SDC technique to work in dynamic environments wherein goals and Web services are continuously created, removed, or modified. The SDC graph needs to support this and should stay operational in the dynamic environment. We therefore distinguish three aspects relevant for the maintenance of the SDC graph:

1. **iterative creation of the SDC graph:** this is concerned with the creation of an SDC graph for given goal templates and Web services. We realize an iterative procedure that commences with a single goal template and then successively adds other or new goal templates. This must ensure that the result always reveals the properties of a refined SDC graph, i.e. that it is a set of goal trees with *subsume* as the only occurring similarity degree and with a minimal discovery cache (*cf.* Definition 3.5).
2. **evolution support:** this covers the operations for removal or updates of goal templates, and the addition, removal, or update of Web service descriptions. These operations are necessary to maintain the operational functionality of the SDC technique in its dynamic environment.
3. **advanced management:** this is concerned with techniques for increasing the quality of the SDC-enabled Web service discovery. In particular, we discuss possibilities for automatically creating new goal templates that provide the backbone of the SDC technique. While the former two aspects are mandatory, the advanced management technique are optional extensions.

The following specifies the techniques and algorithms for each aspect. All of these operations are performed at design time, respectively orthogonal to runtime (i.e. only if a goal template or a Web service description is added, removed, or updated). Because this does not influence the runtime efficiency of SDC-enabled Web service discovery, we omit a computational cost analysis of the algorithms specified here. In order to maintain the consistency of the SDC graph, we define the overall control of the SDC algorithms such that it disallows parallel execution of operations (*cf.* Appendix C).

5.1 Iterative Creation of the SDC Graph

We commence with the first aspect on the creation of the SDC graph. The aim is to ensure that the resulting SDC graph always as exposes the properties of a *refined* SDC graph, i.e. that the goal templates are organized as a set of goal trees with *subsume* as the only occurring similarity degree and the discovery cache does not contain any redundant WG mediators (*cf.* Definition 3.5). The reason is the algorithms for Web service discovery specified in Section 4 only work properly on the basis of such a refined SDC graph.

To achieve this with minimal computational costs, we specify the algorithm to subsequently build up the SDC graph and to resolve undesirable situations en-route. This means that we start with one goal template and then subsequently add other existing ones, respectively newly created goal templates. In each iteration, we first compute the goal graph and then add the discovery cache. During the constructing of the goal graph, we resolve all possibly occurring intersection arcs between goal templates. As discussed above (Section 3.3.2), such i-arcs are the reason for undesirable situations in the SDC graph (cycles, insufficient information, etc.), and their resolution allows to create the goal graph to become a set of goal trees. The following first defines the algorithms for constructing such a refined goal graph, then for constructing the minimal discovery cache, and finally illustrates the algorithms in our running example. We here concentrate

on the functional correctness of the distinct operations; the overall algorithm that integrates all operations is provided in Appendix C.

5.1.1 Algorithm for Goal Template Insertion

The following specifies the operations for the iterative creation of the SDC graph. For this, we define the overall procedure for inserting a new goal template into the SDC graph. To ensure that the constructed goal graph exposes the desired properties of a refined SDC graph, the sub-routines resolve all potentially occurring i-arcs in the goal graph, and remove redundant edges are resolved during the creation procedure.

The creation commences with an empty SDC graph (i.e. no goal templates are stored). Then, the goal template insertion algorithm is iteratively invoked for all existing goal templates, respectively when a new one is added. The matchmaking technique required for creating the goal graph is the determination of the similarity degree between goal templates (*cf.* Section 3.1.1). For this, we perform matchmaking of the formal functional descriptions of goal templates with the matching conditions as defined in Table 3; the technical realization is similar to the determination of Web service usability degrees (*cf.* Section 4.2). In order to minimize the computational costs of the SDC graph management algorithms, we only apply matchmaking when necessary.

The complete algorithm for inserting a new goal template becomes complicated because we must take all possible situations into consideration. We thus discuss each method of the algorithm separately.

Overall Procedure for Goal Template Insertion. We commence with the overall control procedure for the insertion of a new goal template \mathcal{G}_{new} . Listing 5 shows the algorithm for this. We define the following data types that are relevant for this operation. The *goalStore* is the set of all existing goal templates, a *goalTree* is the set of s-arcs that connect all goal templates in a goal tree, and the *goalGraph* is the set of all goal templates in the goal store together with all existing goal trees. The *discoveryCache* captures the WG mediators that connect usable Web services to the goal templates in the goal graph. All these elements are kept in a persistent memory (e.g. a data base). The function *position*(\mathcal{G}) internally keeps the position of a goal template \mathcal{G} in the goal graph. This function can only have 2 values: *root* denotes that \mathcal{G} is either the root node of a goal tree, or it is disconnected in the goal graph; *child* denotes that \mathcal{G} is located at any position but not as the root node in a goal tree.

The overall procedure for the insertion of a new goal template \mathcal{G}_{new} is as follows. Initially – when there is no goal template stored – the new template is added and defined to be a root node. If there are already elements in the goal store, the insertion commences with the investigation of the existing goal templates whose position is *root*. We then distinguish the insertion actions for the distinct similarity degrees. If this is *exact* between \mathcal{G}_{new} and an existing root node, then we do not add \mathcal{G}_{new} ; for the similarity degree *plugin*, *subsume*, or *intersect*, the insertion is handled by sub-routines that we shall discuss below in more detail. If the similarity degree of \mathcal{G}_{new} with all existing root nodes is *disjoint*, we add \mathcal{G}_{new} as a disconnected root node in the goal graph. We then have inserted \mathcal{G}_{new} at the appropriate position in the goal graph. As the second step, we create the discovery cache for \mathcal{G}_{new} by performing Web service discovery.

```
// type declarations
G, G2 := goaltemplate;
goalStore := {goaltemplate};
goalTree := {s(goaltemplate, goaltemplate)};
goalGraph := (goalStore, {goalTree});
discoveryCache := {d(goaltemplate, webservice)};
// function declarations
position(goaltemplate) := (root | child);
```

```

// main
insert(G){
  if ( goalStore != {} ) then {
    forall ( G2 and position(G2) = root) {
      if ( exact(G2,G) ) then return goalStore;
      if ( plugin(G2,G) ) then rootNodeInsertion(G,G2);
      if ( subsume(G2,G) ) then childNodeInsertion(G,G2);
      if ( intersect(G2,G) ) then {
        position(G) = root;
        iArcResolution(G,G2); }
    else {
      position(G) = root;
      return goalStore = goalStore + G;
    } }
  discoveryCacheCreation(G);
}

```

Listing 5: Algorithm for Goal Template Insertion

Insertion of a New Root Node. We now specify the methods for inserting a new goal template into an existing goal graph. We commence with the insertion of \mathcal{G}_{new} as a new root node of an existing goal tree. This is given if the similarity degree between \mathcal{G}_{new} and an existing root node is *plugin*. In this situation, we must replace the existing root node by \mathcal{G}_{new} and define the necessary s-arcs of the goal tree.

Figure 21 shows the possible allocation of \mathcal{G}_{new} after the insertion: in case (a), \mathcal{G}_{new} becomes the new root node of a goal tree that only had one root node beforehand; this also covers the case when \mathcal{G}_{new} becomes the parent of a before disconnected goal template. As case (b), \mathcal{G}_{new} becomes the common root node of two goal trees that were separated beforehand. Here, the prior root nodes are disjoint, and thus the child nodes in the each goal tree are disjoint. For instance, imagine that in the figure \mathcal{G}_1 is for finding the best restaurant in a European city, \mathcal{G}_2 for an American city, and \mathcal{G}_{new} is for any city of the world. In the remaining two possibilities, \mathcal{G}_{new} becomes a new root node of an existing goal tree that has had two or more root nodes before. Such a goal tree can only occur as the result from resolving an i-arc: in the figure, the similarity degree between \mathcal{G}_1 and \mathcal{G}_2 is *intersect*, and \mathcal{G}_3 denotes the intersection goal template (cf. Section 3.3.2). \mathcal{G}_{new} will only become a new root node only if it is a parent node of at least one of the prior root nodes, i.e. if and only if either *plugin*($\mathcal{G}_1, \mathcal{G}_{new}$) or *plugin*($\mathcal{G}_2, \mathcal{G}_{new}$) in the figure. If this is given for one of the prior root nodes, then \mathcal{G}_{new} replaces this one; this is case (c). Then, the similarity degree between \mathcal{G}_{new} with the other existing root node can only be *plugin* or *intersect*: in the former case \mathcal{G}_{new} replaces both prior root nodes (i.e. situation (d)); in the latter case, the i-arc between \mathcal{G}_{new} and the other root node must be resolved (we discuss the resolution of i-arcs below).

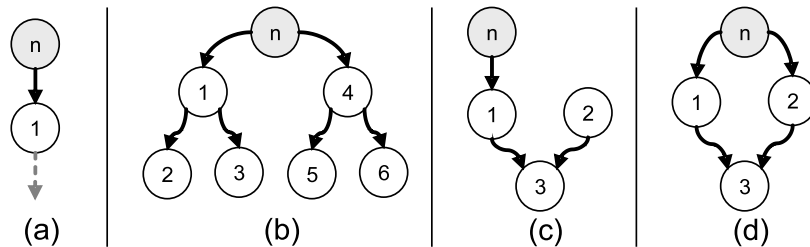


Figure 21: Possible Situations for New Root Node Insertion

The principles discussed for the four cases are the same when we consider more complex goal graphs,

i.e. with more nodes and edges. Besides, the lower levels of existing goal trees are not changed if \mathcal{G}_{new} becomes a new root node; thus we do not need to investigate them. For the latter cases (c) and (d), every goal tree that has more than one root node must have an intersection goal template as an element, and, because of this, all its root nodes are non-disjoint. Therewith, Figure 21 shows the only possibilities for the insertion of \mathcal{G}_{new} as a new root node in the goal graph.

Listing 6 shows the algorithm for the goal template insertion in these situations. It is relatively simple: essentially, we only need to investigate all goal templates that are currently declared to be root nodes, replace the current one with \mathcal{G}_{new} , and add the new s-arc into the goal graph (instead of $plugin(\mathcal{G}_{oldRoot}, \mathcal{G}_{new})$ we store the inverse GG mediator $subsume(\mathcal{G}_{new}, \mathcal{G}_{oldRoot})$, cf. Definition 3.3). All potential cases discussed above are handled by the method *rootNodeInsertionG* because it is invoked as a sub-routine of the *forall* loop over all root nodes in Listing 5 (it also inherits the type declarations). For case (a), the prior root node whereof \mathcal{G}_{new} becomes the new parent node will be investigated at some point in the *forall* loop while no other root node with a similarity other than *disjoint* exists. In case (b), the root node of the second, beforehand separated goal tree will be replaced by \mathcal{G}_{new} (and accordingly for all other goal trees that will be joint under \mathcal{G}_{new}). The same will happen in case (d). In case (c) the *forall* loop over all root nodes will eventually detect the other root nodes of the respective goal tree and resolve the occurring i-arcs. As the result of the insertion, the method *rootNodeInsertionG* returns the updated goal graph.

```

rootNodeInsertion{G,G2} {
  goalStore = goalStore + G;
  position(G2) = child;
  position(G) = root;
  goalTree = goalTree + s(G,G2);
  return goalGraph;
}

```

Listing 6: Algorithm for Insertion of a new Root Node

Insertion of a New Child Node in a Goal Tree. Next, we specify the algorithm for insertion \mathcal{G}_{new} as a new child node into a goal graph. This situation is given if the similarity degree between \mathcal{G}_{new} and an existing root node \mathcal{G} is *subsume*. Here, \mathcal{G} is a root node of the goal tree wherein \mathcal{G}_{new} shall be inserted as a new child node, and we have to successively traverse this goal tree in order to properly insert \mathcal{G}_{new} .

Listing 7 below shows the algorithm for this that handles all possibly occurring situations as illustrated in Figure 22. In case (a), \mathcal{G}_{new} becomes a new, disjoint child node at the lowest level of an existing goal tree. This also covers the situation when \mathcal{G}_{new} becomes a child node of a goal template that has been disconnected in the goal graph beforehand. For this, we merely need to add the new s-arc to the goal graph. The other situations are handled by the *forall* loop of the *childNodeInsertion* method that investigates the child nodes of the current root node. If the similarity degree between \mathcal{G}_{new} and the a child node is *exact*, we do not add \mathcal{G}_{new} to the SDC graph. If the similarity degree is *plugin*, then \mathcal{G}_{new} becomes an intermediate parent in the existing goal graph (cases (b) and (c); these are successively determined throughout the iterations of the *forall* loop). In this situation, the s-arcs between the current parent and its prior child nodes become redundant, thus we remove the respective s-arcs (cf. Section 3.3.3). In case (d), the similarity degree between \mathcal{G}_{new} and the currently inspected child node is *subsume*. Here, we can inspect the next lower level for the goal tree in a depth-first manner. For this, we invoke the *childNodeInsertion* method for \mathcal{G}_{new} and the current node. As the last possible situation, the similarity degree between \mathcal{G}_{new} and the currently inspected child node is *intersect*. In this case, we invoke the *iArcResolution* method for \mathcal{G}_{new} and the currently inspected child node in order to resolve the occurring i-arc.

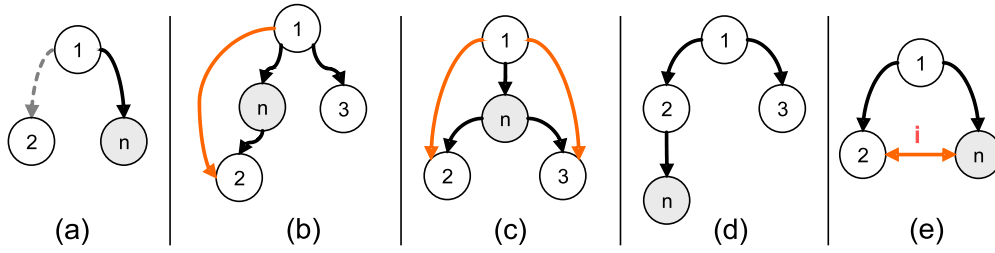


Figure 22: Possible Situations for Insertion of a New Child Node

```

childNodeInsertion{G,G2}{
  goalStore = goalStore + G;
  position(G) = child;
  goalTree = goalTree + s(G2,G);
  forall ( G3 and s(G2,G3) in goalGraph ) {
    if ( exact(G3,G) ) then {
      goalStore = goalStore - G;
      goalTree = goalTree - s(G2,G);
      return goalGraph; }
    if ( plugin(G3,G) ) then {
      goalTree = goalTree - s(G2,G);
      goalTree = goalTree + s(G2,G) + s(G,G3);
      goalTree = goalTree - s(G2,G3); }
    if ( subsume(G3,G) ) then {
      goalTree = goalTree - s(G2,G);
      childNodeInsertion{G,G3}; }
    if ( intersect(G3,G) ) then {
      goalTree = goalTree - s(G2,G);
      iArcResolution{G,G3}; }
  }
  return goalGraph;
}

```

Listing 7: Algorithm for Insertion of a new Child Node in a Goal Tree

En-Route Resolution of Intersection Arcs in the Goal Graph. The next sub-routine of the goal template insertion algorithm is concerned with the insertion of \mathcal{G}_{new} if its similarity degree with the currently inspected node is *intersect*. As discussed above, the occurrence of i-arcs causes undesirable situations in the goal graph. The aim of the sub-routine for insertion a new goal template under the *intersect* similarity degree is to resolve all occurring i-arcs during the insertion procedure. For explaining the algorithm for this, we briefly recall the approach for i-arc resolution as specified in detail in Section 3.3.2.

An i-arc occurs if the similarity degree between two goal template \mathcal{G}_1 and \mathcal{G}_2 is *intersect*. If we would keep such i-arcs in the goal graph, there could occur cycles that hamper the goal template search as well as other undesirable situations like concatenations of i-arcs or non-disjoint child nodes in a goal tree that hamper the operational efficiency of SDC-enabled Web service discovery. The approach for resolving i-arcs is to define a so-called *intersection goal template* $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}$; its functional description is the conjunction of the original goal templates, so that its possible solutions are exactly those that are common to \mathcal{G}_1 and \mathcal{G}_2 . The similarity degree between $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}$ and either of the original goal templates is *subsume*, i.e. $subsume(\mathcal{G}_1, \mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)})$ and $subsume(\mathcal{G}_2, \mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)})$. Because of this, we do not store the i-arc but only the two new s-arcs so that $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2)}$ becomes a child node of both \mathcal{G}_1 and \mathcal{G}_2 . However, the insertion of an intersection goal template results in new relationships in the goal graph that need to be handled.

The approach for handling the insertion of a new goal template \mathcal{G}_{new} under the *intersect* similarity degree is that we stepwise resolve each occurring i-arc along with possible implications that can result from this. Listing 8 shows the algorithm for this. In the above algorithms, the method $iArcResolution(G1, G2)$ is invoked for two goal templates: the first one is \mathcal{G}_{new} as the new goal template that shall be inserted, and another one \mathcal{G}_2 for with the similarity degree with \mathcal{G}_{new} is *intersect*. At first, we add \mathcal{G}_{new} as well as new intersection goal template $\mathcal{G}_{i(\mathcal{G}_{new}, \mathcal{G}_2)}$ into the goal store along with the respective s-arcs, and perform Web service discovery for $\mathcal{G}_{i(\mathcal{G}_{new}, \mathcal{G}_2)}$. Then, we need to handle the implications that may result from this insertion. From the discussion in Section 3.3.2, we observe that such implications can only occur at the level of the goal tree where $\mathcal{G}_{i(\mathcal{G}_{new}, \mathcal{G}_2)}$ has been inserted. In particular, any similarity degree is possible between the inserted intersection goal template and a before existing child node of its parents (i.e. \mathcal{G}_{new} and \mathcal{G}_2).

In order to ensure that the goal graph maintains its desired properties (as a tree of goal template with s-arcs as the only occurring edge type), we need to resolve undesired similarity on the level where the new intersection goal template has been inserted. This is done by the *forall* loop of the $iArcResolution$ method that iteratively checks the similarity degree between $\mathcal{G}_{i(\mathcal{G}_{new}, \mathcal{G}_2)}$ and all of the child nodes of \mathcal{G}_{new} and \mathcal{G}_2 . If the similarity degree is *exact*, then we do not store $\mathcal{G}_{i(\mathcal{G}_{new}, \mathcal{G}_2)}$ but merely re-direct the new s-arcs to the currently inspected child node. If the similarity degree is *plugin*, then $\mathcal{G}_{i(\mathcal{G}_{new}, \mathcal{G}_2)}$ becomes an intermediate parent of the currently inspected node; as the opposite situation, $\mathcal{G}_{i(\mathcal{G}_{new}, \mathcal{G}_2)}$ becomes a child node of the currently inspected node if their similarity degree is *subsume*. If the degree is *intersect*, then we resolve the new i-arc by invoking the $iArcResolution$ method with $\mathcal{G}_{i(\mathcal{G}_{new}, \mathcal{G}_2)}$ and the currently inspected goal template. We keep the initially created goal graph only if the similarity degree of $\mathcal{G}_{i(\mathcal{G}_{new}, \mathcal{G}_2)}$ and all child nodes of \mathcal{G}_{new} and \mathcal{G}_2 is *disjoint*.

```

// function declarations
intersectionGoalTemplate(goaltemplate,goaltemplate) := goaltemplate;
// main
iArcResolution(G1,G2) {
  iG1_G2 = intersectionGoalTemplate(G1,G2);
  goalStore = goalStore + G1 + iG1_G2;
  goalTree = goalTree + s(G1,iG1_G2) + s(G2,iG1_G2);
  discoveryCacheCreation(iG1_G2);
  forall (G3 and ( (s(G1,G3) or s(G2,G3)) in goalGraph) and G3 != iG1_G2 ) {
    if ( exact(G3, iG1_G2) ) then {
      goalStore = goalStore - iG1_G2;
      goalTree = goalTree - s(G1,iG1_G2) - s(G2,iG1_G2);
      if ( ! (s(G1,G3) in goalGraph) ) then {
        goalTree = goalTree + s(G1,G3); }
      if ( ! (s(G2,G3) in goalGraph) ) then {
        goalTree = goalTree + s(G2,G3); } }
    if ( plugin(G3, iG1_G2) ) then {
      goalTree = goalTree - s(G1,G3) - s(G2,G3);
      goalTree = goalTree + s(iG1_G2,G3); }
    if ( subsume(G3, iG1_G2) ) then {
      if ( s(G1,G3) in goalGraph ) then {
        goalTree = goalTree - s(G1,G3);
        goalTree = goalTree + s(iG1_G2,G3); }
      if ( s(G2,G3) in goalGraph ) then {
        goalTree = goalTree - s(G2,G3);
        goalTree = goalTree + s(iG1_G2,G3); } }
    if ( intersect(G3, iG1_G2) ) then
      iArcResolution(iG1_G2, G3);
  }
  return goalGraph();
}

```

Listing 8: Algorithm for Dynamic Resolution of Intersection Arcs in the Goal Tree

The main merit of this algorithm is that it prevents the emergence of all undesirable situations in the goal graph because all i-arcs are resolved at the time when they occur during the insertion of a new goal template. Moreover, this algorithm resolves every undesirable situation that can occur due to the existence of i-arcs into the pattern that we have defined in Section 3.3.2. We omit the formal proof here and refer to the respective discussion for the resolution of cycles (*cf.* Proposition 3.3), for concatenations of i-arcs (*cf.* Proposition 3.1), and the representation of non-disjoint child in the initial goal graph (*cf.* Proposition 3.2). We shall demonstrate the algorithm in our running example below in Section 5.1.2.

Discovery Cache Creation. Above we have specified the algorithms for inserting a goal template such that the resulting goal graph exposes the desirable properties of a refined SDC graph. To complete the SDC graph creation, we need to determine the Web services that are usable for the newly inserted goal template. For this, we perform Web service discovery on the goal template level as specified in Section 4.2.

After the insertion of a new goal template \mathcal{G}_{new} into the goal graph, the *insert*(G) method from Listing 5 invokes the *discoveryCacheCreation*(G). Specified in Listing 9, this performs Web service discovery for \mathcal{G}_{new} and ensures that the resulting discovery cache is minimal, i.e. that it does not contain any redundant WG mediators (*cf.* Proposition 3.5). Because we have stored knowledge about the position of \mathcal{G}_{new} in the goal graph, we can directly invoke the respective methods from the Web service discovery algorithm specified in Listing 3 (*cf.* Section 4.2). If \mathcal{G}_{new} has been inserted as a new child node into the goal graph, then *childNodesDiscovery*(G) determines the usable Web services. With respect to the inference rules from Theorem 3.1 (*cf.* Section 3.2), this only considers Web services that are usable for a parent nodes of \mathcal{G}_{new} under the *subsume* or *intersect* degree; under the *exact* or *plugin* degree, the WG mediators for \mathcal{G}_{new} are omitted in the SDC graph. If \mathcal{G}_{new} is a new root node or if it is disconnected in the goal graph, then the *rootNodeDiscovery*(G) method performs the Web service discovery. In order to maintain the minimality of the discovery cache, we must remove redundant WG mediators that may result from this operation. An existing WG mediator from a child node of \mathcal{G}_{new} to W is redundant if W has been discovered to be usable for \mathcal{G}_{new} under the *exact* or *plugin* degree. Finally, we can remove \mathcal{G}_{new} in case that no Web service has been found that can be used to solve \mathcal{G}_{new} . Therewith, we obtain a minimal discovery cache for the SDC graph, and for each goal template there is at least one usable Web service.

```

// function declaration
discoveryCache(G) := {d(G,webservice)};
// main
discoveryCacheCreation(G) {
  if ( position(G) = child ) then
    childNodeDiscovery(G);
  else {
    rootNodeDiscovery(G);
    forall (G2 and (s(G,G2) in goalGraph) ) {
      if ( ( d(G2,W) = (exact or plugin) ) and d(G,W) ) in discoverycache ) then
        discoverycache = discoverycache - d(G,W);
    }
  }
  if ( discoveryCache(G) = {} ) then remove(G);
  return discoverycache;
}

```

Listing 9: Algorithm for Discovery Cache Creation

In the preceding elaborations we have shown that the goal insertion algorithm ensures that after every run the resulting SDC graph exposes the desirable properties of a refined SDC graph (*cf.* Definition 3.5). This is essential because the Web service discovery operations specified in Section 4 only work efficiently on a SDC graph with these properties.

Proposition 5.1 (Properties of Iteratively Constructed SDC Graph). *The SDC graph that results from an execution of the goal template insertion algorithm always exposes the properties of a refined SDC graph:*

- (i) *the goal graph is a set of **unconnected goal trees** wherein the only occurring similarity degree is subsume and all occurring i-arcs are properly resolved, and*
- (ii) *the discovery cache is minimal such that there are no redundant WG mediators.*

5.1.2 Example

The iterative goal insertion is one of the central algorithms of the SDC technique. Because of this, we illustrate the overall procedure within our running example for searching the best restaurant in a city. We consider the iterative insertion of three goal templates: G_1 for finding the best restaurant in a city located in country that is member of the European Union, G_2 for a German city, and G_3 for finding the best Italian restaurant in any city of the world. Furthermore, we discuss the discovery cache creation for a Web service W for finding the best restaurant in a European city. Remember that the set of European countries is a superset of member countries of the European Union (e.g. Switzerland and Norway are geographically located in Europe but are not members of the EU). Figure 23 illustrates the setting as well as steps for the iterative goal template insertion.

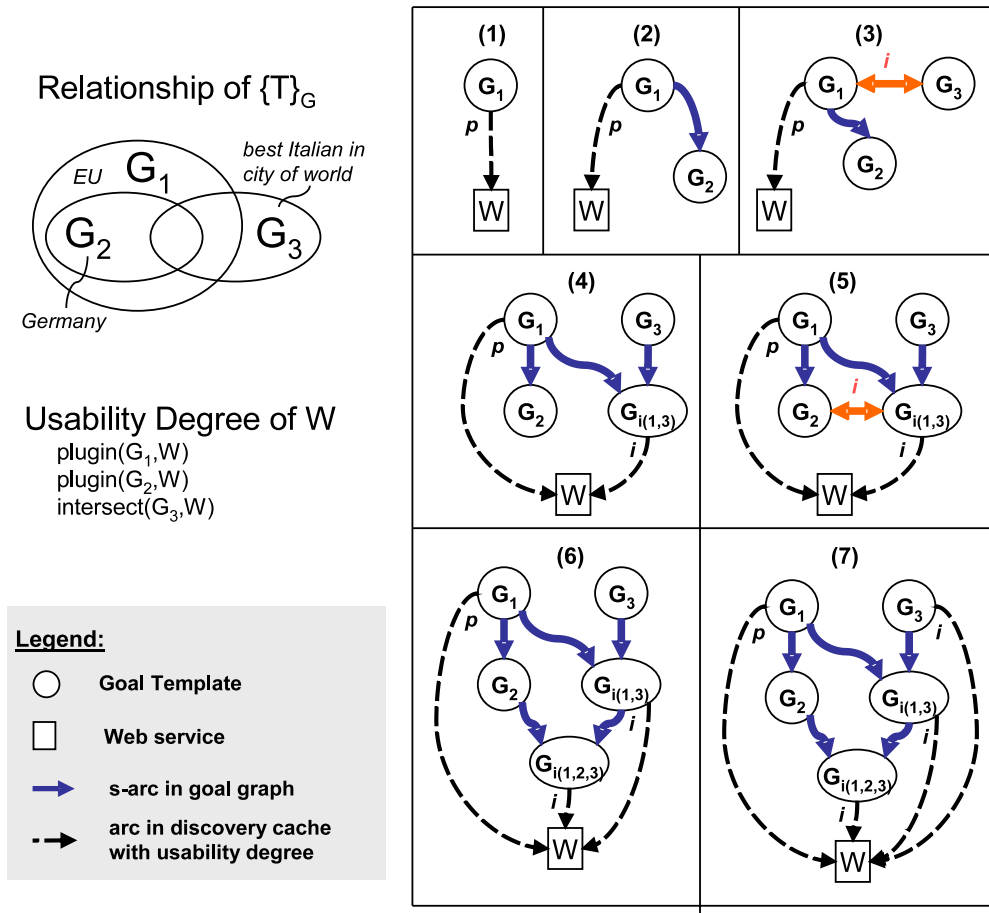


Figure 23: Example for Iterative SDC Graph Creation

We commence with the insertion of \mathcal{G}_1 . The SDC graph is empty at this point, so that the *insert* method from Listing 5 will allocate \mathcal{G}_1 as a single root node in the goal graph. Then, we perform Web service discovery for \mathcal{G}_1 , which determines the usability degree of W for \mathcal{G}_1 to be *plugin* and adds the respective WG mediator. This completes the first insertion and results in a SDC graph as shown in step (1) of the figure. Next, we insert \mathcal{G}_2 . The *insert* method will find \mathcal{G}_1 and determine the similarity degree to be *subsume*($\mathcal{G}_1, \mathcal{G}_2$) and invoke the *childNodeInsertion* from Listing 7. As there are no further nodes in the goal graph, \mathcal{G}_2 is allocated as a new child node of \mathcal{G}_1 . For Web service discovery, the *discoveryCacheCreation* method from Listing 9 will not add a new WG mediator because the usability degree between \mathcal{G}_1 and W is *plugin*. Thus, the insertion of \mathcal{G}_2 results in the SDC graph shown in step 2.

When we now insert \mathcal{G}_3 , the *insert* method will find the similarity degree of \mathcal{G}_3 and \mathcal{G}_1 as the root node to be *intersect*, cf. step 3, and thus invoke the *iArcResolution* algorithm from Listing 8. This creates the intersection goal template $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_3)}$, and inserts this into the goal graph. Following Definition 3.4 (cf. Section 3.3.2), an intersection goal template is defined as the conjunction of the functional descriptions of the original goal templates. This means that here $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_3)}$ describes the objective of finding the best restaurant in a city in the European Union if and only if the best restaurant in this city is of type Italian – not to find the best Italian restaurant in a city of an EU member country. Nevertheless, the following Web service discovery for the intersection goal template will determine the usability degree of W for $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_3)}$ to be *intersect* (this performed before the *forall* loop in the *iArcResolution* algorithm, cf. Listing 8). Step 4 in the figure shows the intermediate SDC graph after this operation.

Now, we enter the *forall* loop of the *iArcResolution* algorithm in order to resolve possible undesired implications from the insertion of $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_3)}$. Indeed, we will find the similarity degree between $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_3)}$ and \mathcal{G}_2 to be *intersect*, see step 5. Hence, the *iArcResolution* method is invoked again for resolving the new i-arc. This creates another intersection goal template $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3)}$, inserts this into the goal graph, and performs Web service discovery that determines the usability degree of W for $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3)}$ to also be *intersect*. As there are no other nodes on the same level as $\mathcal{G}_{i(\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3)}$, the *iArcResolution* algorithm terminates and returns the SDC graph shown as step 6 of the figure. However, the insertion operation for \mathcal{G}_3 is not yet completed: as the last step, the *insert* method invokes the *discoveryCacheCreation* for \mathcal{G}_3 . The position of \mathcal{G}_3 has been set to *root*, and hence the *rootNodeDiscovery* method is used that will determine the usability degree of W for \mathcal{G}_3 to be *intersect*. This completes the insertion operation for \mathcal{G}_3 that results in the SDC graph shown as step 7 in the figure.

5.2 Evolution in Dynamic Environment

We now turn towards the maintenance of the SDC graph. In this section we define the operations for evolution of the SDC graph, that is when goal template or Web service descriptions are added, removed, or updated. The algorithms for handling such changes in the environment are mandatory in order to maintain the operational reliability of the SDC technique in its dynamic environment. The following first discusses the removal and updating of goal templates that are already existing in an SDC graph, and then discusses changes to the available Web services.

5.2.1 Goal Template Removal and Update

We have already covered the insertion of new goal templates as the foundational operation for creating the SDC graph above. Hence, we here merely need to specify the algorithms for handling the deletion or an update of a goal description from the SDC graph.

Such changes on existing goal templates can occur in the process of maintaining the SDC graph during the lifetime of the system. They can only be performed for original goal templates but not for intersection goal templates because these are automatically generated during the SDC graph creation. Moreover, changes on existing goal templates may seriously derogate the quality of SDC-enabled Web service discovery as they provide the foundation of the SDC graph. Because of this, removal and in particular updates of goal templates should only be performed if they are absolute necessary in the application context. Nevertheless, we must support them in such a way that the properties of the resulting SDC graph are maintained after the removal or update of a goal template.

Goal Template Removal. This is concerned with removing an existing goal template from the SDC graph. Possible scenarios for this are the clearing of the SDC graph in the context of an application scenario (cf. Section 5.3.3), or the removal of a goal template if there does not exist any usable Web service for solving it (e.g. in Listing 9). Independent of the reason why a goal template is removed, the properties of the SDC graph must be maintained after a goal template removal.

Listing 10 shows the algorithm for removing a single goal template from an SDC graph such that the properties of the SDC graph are maintained. Let \mathcal{G}_{rm} be the goal template that shall be removed. In general, we only need to consider the impacts on the direct neighbors of \mathcal{G}_{rm} that can result from its deletion. We therefore must distinguish three cases: (1) if \mathcal{G}_{rm} is a parent of an intersection goal template, (2) if the position of \mathcal{G}_{rm} on the SDC graph is *root*, and (3) if the position of \mathcal{G}_{rm} is *child*. The proper handling of the latter two cases requires that case (1) is not given. Hence, we first check whether \mathcal{G}_{rm} is a parent of an intersection goal template; in this case, we first remove the intersection goal template and then remove \mathcal{G}_{rm} (the first *if*-clause in Listing 10); this is independent of the position of \mathcal{G}_{rm} .

We then require two methods for the removal of a goal template. The first one is *rootNodeRemoval* that handles the removal of \mathcal{G}_{rm} in case (2). It first removes all outgoing s-arcs, so that \mathcal{G}_{rm} becomes disconnected while the similarity relationship between all other goal templates remains. Secondly, we must adjust the discovery cache. In order to maintain the minimality of the discovery cache, we re-direct those WG mediators that start from \mathcal{G}_{rm} and define *exact* or *plugin* as the usability degree of the target Web service such that a new WG mediator is defined for each child node of \mathcal{G}_{rm} to the target Web service with the usability degree *plugin*. Therewith, we re-materialize the previously omitted WG mediators. Then, we can remove all WG mediators that start \mathcal{G}_{rm} without losing relevant knowledge. For case (3), i.e. if \mathcal{G}_{rm} is a child node in a goal tree, we redirect the outgoing a-arcs of \mathcal{G}_{rm} to each parent node of \mathcal{G}_{rm} . This also covers the case if \mathcal{G}_{rm} has been an intersection goal template with two or more parents (i.e. the first removal step in case (1)). The discovery cache handling is the same as in the *rootNodeRemoval* method explained above. Finally, we remove \mathcal{G}_{rm} from the goal store and obtain the updated SDC graph that still captures the relationship of the remaining goal templates in the goal graph and has a minimal discovery cache.

```
// type declarations
G,G2,G3 := goaltemplate;
goalStore := {goaltemplate};
goalTree := {s(goaltemplate,goaltemplate)};
goalGraph := (goalStore,{goalTree});
discoveryCache := {d(goaltemplate,webservice)};
sdgGraph := (goalGraph,discoveryCache);
// function declarations
position(goaltemplate) := (root | child);
outgoingGGArcs(G) := {s(G,goaltemplate)};
incomingGGArcs(G) := {s(goaltemplate,G)};
discoveryCache(G) := {d(G,webservice)};
```

```

// main
remove(G) {
  if ( G2 and (s(G,G2) in goalGraph) and G3 and (s(G3,G2) in goalGraph) ) then {
    remove(G2);
    remove(G);
  }
  if ( position(G) = root ) then {
    rootNodeRemoval(G); }
  else {
    childNodeRemoval(G);
  }
  goalStore = goalStore - G;
  return sdcGraph;
}
// removing a root node
rootNodeRemoval(G) {
  goalTree = goalTree - outgoingGGArcs(G);
  forall ( G2 and (s(G,G2) in goalGraph) ) {
    if ( W and (d1(G,W) in discoveryCache(G)) and (d1 = exact or d1 = plugin) ) then {
      d2 = plugin;
      discoveryCache = discoveryCache + d2(G2,W); }
  }
  discoveryCache = discoveryCache - discoveryCache(G);
  return sdcGraph;
}
// removing a child node
childNodeRemoval(G) {
  forall ( G2 and (s(G,G2) in goalGraph) ) {
    forall ( G3 and (s(G3,G) in goalGraph) ) {
      goalTree = goalTree + s(G3,G2); }
    if ( W and (d1(G,W) in discoveryCache(G)) and (d1 = exact or d1 = plugin) ) then {
      d2 = plugin;
      discoveryCache = discoveryCache + d2(G2,W); }
  }
  goalTree = goalTree - outgoingGGArcs(G);
  return sdcGraph;
}

```

Listing 10: Algorithm for Removal of a Goal Template from the SDC Graph

Goal Template Update. This is concerned with the update of the definition of a goal template that exists in the SDC graph. A possible scenario for such an update is the weakening or strengthening of the objective description (e.g. in our running example, if the locality constraint for the input city is changed from “European Union” to “Europe”, respectively vice versa). This may result from refinement operations on a goal template that has not been solvable by any available Web service before; techniques for such goal refinements are presented in [7, 38]. Another possibility for the need of a goal template update is if the existing goal template definition is inconsistent or inappropriate in the application context. The straight forward solution for supporting such goal template updates would be to remove the old version and then insert the new version of the description. However, in certain cases we can omit the execution of the removal and insertion operation (which both are computationally expensive).

In general, the update of a goal template results in a change of the goal description. We can express this change in terms of the similarity degree between the old and the new version of the goal template. If the similarity degree is *exact*, we merely replace the old version of the goal template with the new one. We do not need to perform any changes in the SDC graph because the position of the updated goal template in the goal graph as well as its usable Web services will be the same after the update. Furthermore, we can efficiently handle two other situations: (1) if the updated goal template is a single root node in the goal

graph and the similarity degree between the old and the new version is *plugin*, and (2) if the updated goal template is a child node at the lowest level of a goal tree with only one parent and the similarity degree between the old and the new version is *subsume*. In both cases, the position of the updated goal template in the goal tree will remain the same; we thus merely replace the old version with the new one. However, we must perform a Web service discovery for the new version because there might be more usable Web services (case (1)), respectively fewer (case (2)). In all other situations, we perform the default update operation that first removes the old version (calling the *remove* method from Listing 10) and then inserts the new version with the goal template insertion algorithm from Listing 5.

Listing 11 shows the algorithm for handling updates of goal templates in the goal graph as explained. Therein, the *update(G1,G2)* takes two goal templates as input: the first one denotes the old version of the updated goal template, and the second one is the new version.

```

// type declarations
G,G2,G3,G4,G5 := goaltemplate;
goalStore := {goaltemplate};
goalTree := {s(goaltemplate,goaltemplate)};
goalGraph := (goalStore,{goalTree});
discoveryCache := {d(goaltemplate,webservice)};
sdcGraph := (goalGraph,discoveryCache);
// function declarations
position(goaltemplate) := (root | child);
singleRoot(G) := boolean;
lowestChildWithSingleParent(G) := boolean;
// main
update(G1,G2) {
  if ( exact(G1,G2) ) then {
    goalStore = goalStore - G1;
    goalStore = goalStore + G2;
  }
  if ( position(G1) = root and (s(G1,G3) in goalGraph) and ! (s(G4,G3) in goalGraph) )
    then singleRoot(G1) = true;
  else singleRoot(G1) = false;
  if ( position(G1) = child and ! (s(G1,G3) in goalGraph) and (s(G4,G3) in goalGraph) and ! (s(G5,G3) in goalGraph) )
    then lowestChildWithSingleParent(G1) = true;
  else lowestChildWithSingleParent(G1) = false;
  if ( (singleRoot(G1) and plugin(G1,G2)) or (lowestChildWithSingleParent(G1) and subsume(G1,G2)) ) then {
    position(G2) = position(G1);
    goalStore = goalStore - G1;
    goalStore = goalStore + G2;
    discoveryCacheCreation(G2);
  }
  else {
    remove(G1);
    insert(G2);
  }
  return sdcGraph;
}

```

Listing 11: Algorithm for Update of a Goal Template in the SDC Graph

5.2.2 Changes on Available Web Services

The second aspect for the maintenance of the SDC graph during the life time of the system is the handling of changes on the available Web services. Such changes occur if a Web service provider publishes a new Web service, removes an existing Web service from the registry, or provides an updated description for an existing Web service. Of course, the discovery cache of the SDC graph must reflect the currently available

Web service in order to provide useful discovery results. For this, the following specifies the operations for the insertion, removal, and update of Web service descriptions. We recall that the SDC-enabled Web service discovery component is decoupled from the Web service registry in the overall system architecture; the maintenance operations for the SDC graph specified here are invoked when a respective change occurs in the Web service registry (*cf.* Figure 6 in Section 2.4).

Web Service Insertion. This handles the insertion of the a new Web service that has been deployed into the repository. Let W_{new} be this new Web service. To properly incorporate this into the SDC graph, we merely need to add W_{new} to discovery cache – that is, to define the minimal number of new WG mediators that describe the usability degree of W_{new} for every goal template that exists in the goal graph.

Listing 12 shows the algorithm for this. We start with inspecting the usability degree of W_{new} for all goal templates whose position is *root*. If W_{new} is usable for a root node, we proceed with determining its usability degree for the child nodes in the goal graph. The sub routine *childNodeWSInsertion* does this in a depth-first manner, thereby taking the inference rules from Theorem 3.1 into account.

```

// type declarations
G,G2,G3,G4,G5 := goaltemplate;
W := webservice;
goalStore := {goaltemplate};
goalTree := {s(goaltemplate,goaltemplate)};
goalGraph := (goalStore,{goalTree});
discoveryCache := {d(goaltemplate,webservice)};
sdcGraph := (goalGraph,discoveryCache);
// main
insert (W) {
  forall ( G and position(G) = root) {
    matchmakingUsability(G,W);
    if (! d = disjoint ) then {
      discoverycache = discoverycache + d(G,W);
      childNodeWSInsertion(G,W);
    }
  }
  return discoverycache;
}
//
childNodeWSInsertion(G,W) {
  if ( (d(G,W) = exact) or (d(G,W) = plugin) or (d(G,W) = disjoint) ) then return discoverycache;
  else {
    forall ( G2 and (s(G,G2) in goalGraph) ) {
      if ( d(G,W) = subsume ) then {
        matchmakingUsability(G2,W);
        if (! d = disjoint ) then
          discoverycache = discoverycache + d(G2,W);
      }
      if ( d(G,W) = intersect ) then {
        if ( plugin(G2,W) ) then d = plugin;
        if ( intersect (G2,W) ) then d = intersect;
        discoverycache = discoverycache + d(G2,W);
      }
      childNodeWSInsertion(G2,W);
    }
  }
  return discoverycache;
}

```

Listing 12: Algorithm for Insertion of a Web Service Description

Web Service Removal. When a Web service W_{rm} is removed from the registry, it is no longer available for solving a goal and hence must be removed from the SDC graph. The algorithm for this is straight forward: we delete all WG mediators whose target is W_{rm} . As an additional step, we can afterwards remove those goal templates for which W_{rm} has been the only usable Web service – because then there does no longer exists a Web service that can be used to solve this goal template or any of its instantiations. Listing 14 provides the algorithm for this; we here omit the type declarations as they are the same as above.

```
// function declarations
discoveryCache(W) := {d(goaltemplate,W)}
// main
remove(W) {
  discoveryCache(W) = {d(G,W)};
  discoverycache = discoverycache - discoveryCache(W);
  return discoverycache;
}
// optional removal of goal templates
removeGAfterWSDeletion(W) {
  discoveryCache(W) = {d(G,W)};
  forall ( G and (d(G,W) in discoveryCache(W)) ) {
    remove(G); }
  return sdcGraph;
}
```

Listing 13: Algorithm for Deletion of a Web Service Description

Web Service Update. The final operation handles the change of the description of a Web service W that is an element of the SDC graph. This occurs when the functionality provided by W is changed, or as a correction of the description of W . Similar to the update of goal templates discussed above, the straight forward solution for this is to remove the old version and then add the new version of the Web service description. However, here there is also a situation that can be handled more efficiently.

If the description update result in a matching degree $plugin(W_{old}, W_{new})$ – i.e. that the new functionality completely covers the old one – and the usability degree of W_{old} for a root node G_{root} in the goal graph has been *exact* or *plugin*, then the usability degree of W_{new} for G_{root} and all its child nodes remains the same. We also do not need to add additional WG mediators for the child nodes of G_{root} as these would be redundant. This might be a quite regular situation in real world settings, e.g., if in our running example the provider extends the locality coverage of a best-restaurant-search Web service from the member states of the European Union to all countries located in Europe. In all other situations, the potential impacts of the update are too complex to be handled individually so that we apply the default update procedure.

```
update(W1,W2) {
  if ( plugin(W1,W2) ) then {
    forall ( G and position(G) = root ) {
      if ( (d(G,W1) = exact) or (d(G,W1) = plugin) ) then {
        forall ( d(G,W1) in discoverycache ) {
          discoverycache = discoverycache + d(G,W2);
          discoverycache = discoverycache - d(G,W1);
          return discoverycache;
        } } } }
    else {
      remove(W1);
      insert (W2);
    }
  }
}
```

Listing 14: Algorithms for Deletion and Update of a Web Service Description

To conclude, we observe that the algorithms specified above ensure that the SDC graph maintains its formal properties under any possible change that can occur in during its life time (i.e. the goal graph as a set of goal trees and a minimal discovery cache). This is essential because therewith the SDC-enabled Web service discovery stays operational in its dynamically evolving environment. We therewith satisfy requirement 8 – which has been the only remaining one from those discussed in Section 2.

Proposition 5.2 (Properties of Evolving SDC Graph). *The algorithms for the insertion, removal, and updating of goal templates and Web services ensure that the SDC graph exposes the properties of a refined SDC graph at any point in time. This ensures the operational reliability of the SDC-enabled Web service discovery in its dynamically evolving environment.*

5.3 Advanced Management

The preceding elaborations have defined the mandatory algorithms for creation and maintenance of the SDC graph in its dynamic environment. We now discuss possible extensions for increasing the quality of the SDC technology. The following addresses ontology-based learning of goal templates, in the integration of selection and ranking techniques of usable Web services, and the maintenance of the SDC graph in an application context. These advanced management techniques for the SDC graph are optional extensions, meaning that they are not mandatory in order to maintain the operational reliability of SDC-enabled Web service discovery. We thus abandon the definition of algorithms but merely discuss the obtainable benefits.

5.3.1 Goal Template Learning

The first possibility is the automated learning of additional goal templates on the basis of the background ontology. Goal templates are the backbone of the SDC graph, serving as the indexing structure of available Web services and as the basis for goal formulation by clients. The more goal templates exist that can be organized into a fine grained goal tree, the better becomes the quality of the SDC graph for both its purposes. This quality can be enforced by automated generation of additional goal templates, in particular of new goal templates that establish a more fine grained goal graph.

Let us illustrate this in our running example. Imagine that there exists a goal template \mathcal{G}_1 for finding the best restaurant in a Germany city. The background ontology Ω distinguishes the two dimensions of the geographic locality of a city and the type of the restaurant. On the basis of the locality taxonomy described in Ω , we can generate a new goal template \mathcal{G}_2 for finding the best restaurant in a European city. When inserting this into the SDC graph, \mathcal{G}_2 becomes a parent node of \mathcal{G}_1 because their similarity degree is $\text{subsume}(\mathcal{G}_2, \mathcal{G}_1)$. Therewith, we have artificially expanded the SDC graph so that now searches for the best restaurant throughout Europe is supported, and we can make use of the properties of goal trees for SDC-enabled Web service discovery as discussed above. This procedure can of course be repeated for every dimension of the description of \mathcal{G}_1 , e.g. for other countries in Europe as well as for the distinct restaurant types. So, we can eventually generate an extensive goal tree whose elements cover every possible goal instance of the best restaurant search that can be expressed by the domain knowledge described in Ω . One step further, we can also generate the goal template from a concrete goal instance. For example, if a client specifies a goal instance for finding the best restaurant in Berlin (and “Berlin” is specified to be an instance of city with respect to Ω), we can generalize this towards the goal template \mathcal{G}_1 from above by lifting the instance declaration to its corresponding concept in the domain ontology. Then, we can commence with the generation of the extensive goal tree as explained above. These operations would be performed at design time, and thus do not hamper the run time efficiency of the SDC-enabled Web service discovery.

This technique applies the idea of ontology learning and especially concept extraction [24]. However, respective techniques like taxonomy extraction need to be adopted into the context of goal-based Web service usage. Therein, the foundational principle is that the application context (i.e. client requests formalized as goals) is explicitly separated from the functionalities provided by Web services [10]. In general, the functional description of a goal can consist of an arbitrary number of predicates. Each of these predicates denotes one dimension for which new, semantically similar goal templates can be created. Nevertheless, the ontology-based generation of more extensive goal trees appears to be a suitable technique for increasing the quality of SDC-enabled Web service discovery.

5.3.2 Integration with Non-Functional Discovery

A second possibility is the integration with non-functional discovery aspects. Because of the primary focus on functional usability of a Web service for solving a goal, our framework allocates the testing of quality-of-service requirements and behavioral compatibility after the functional Web service discovery (*cf.* Figure 5 in Section 2.1). However, these aspects are relevant for the usability of a Web service. Thus, we could integrate them into the SDC graph in order to increase its quality as a pre-filter for Web service discovery.

Essentially, there are two possibilities that can result non-functional discovery: (1) that a Web service is not usable for solving a goal, and (2) a ranking of the set of usable Web services. The first case occurs when the Web service violates some quality requirements specified in the goal description (e.g. the use of a trusted payment method), or if not resolvable behavioral mismatches occur. This can be handled by removing the Web service from the discovery cache for the goal template, respectively to not consider the Web service as a potential candidate for solving a goal instance. For the second case, we can incorporate the result of a Web service ranking component by organizing the usable Web services for a goal template accordingly. For example, let there be five Web services that are functionally usable for a goal template \mathcal{G} , and let these be stored in the SDC graph as a set $\{W_1, W_2, W_3, W_4, W_5\}$. We then perform a ranking the Web services in this set with respect to the requirements in \mathcal{G} . Let this return the preference order $\langle W_2, W_4, W_1, W_5, W_3 \rangle$. We can store this as an ordered list in the SDC graph, and for a new goal instance we inspect the candidates in the sequence of this ordering. With such an integration, the SDC graph captures Web service discovery results for all aspects that are considered to be relevant for the usability of a Web service for solving a goal. However, the constituting aspect of the SDC graph remains the functional usability.

5.3.3 SGC Graph Clearing

The final aspect to be discussed here is the maintenance of the SDC graph for a specific application. In particular, we here refer to maintenance of the goal templates in the SDC graph; the consistency with the available Web services is automatically maintained by the algorithms for changes in the Web service repository (*cf.* Section 5.2.2). Changes on existing goal templates may become necessary if a goal template is not solvable by the available Web services, or if the objective described in a goal templates is no longer relevant for the application context (e.g. buying a no computer model that is not available any more). We expect such maintenance operations to be performed manually. The consistency of the SDC graph is ensured by the respective algorithms for goal template removal and updates (*cf.* Section 5.2.1). As discussed above, such maintenance operations may derogate the quality of the SDC-enabled Web service discovery.

The SDC graph is stored in a persistent memory, and only loaded partially into the working memory at runtime. Because of this, the size of the SDC graph is not critical for the operational reliability at runtime. This is an essential difference between SDC and caching techniques in other areas wherein cache clearing is critical, e.g. in caching techniques for Web traffic [49].

6 Summary and Future Work

This technical report has specified the Semantic Discovery Caching technique, short SDC. This captures Web service discovery results on the goal template level, and utilizes this knowledge to enable efficient Web service discovery for concrete goal instances at runtime.

We have recalled the approach for Web service discovery on formal functional descriptions that has been presented in early works. Extending the Web service discovery approach defined in the WSMO framework, this distinguishes goal templates as generic objective descriptions and goal instances that describe a concrete client objective by instantiating a goal template with concrete inputs. The purpose of the SDC technique is to reduce the search space for Web service discovery. Its central construct is the SDC graph that organizes existing goal templates in tree structures. Therein, the possible solutions of a goal template that is a child node always denote a subset of those for its parent node. On the basis of this, efficient runtime Web service discovery can be realized.

The central aspects that have discussed in detail in this report are:

- discussion of the requirements for the SDC technique, in particular the terminology clarification (efficiency and scalability), the requirements on the SDC graph, and the technical integration of SDC-enabled Web service discovery into semantically enabled SOA systems (*cf.* Section 2)
- the definition of the SDC graph (*cf.* Section 3), including its *elements* (goal templates, Web services, and mediators as directed arcs), the notion of *goal similarity* described as the matching degree between functional descriptions of goal templates, and the concept of *intersection goal templates* and their usage for resolving undesirable situations in the SDC graph
- the iterative creation of an SDC graph such that it exposes the following properties:
 1. its inner nodes are goal templates that are organized in trees wherein the only occurring similarity degree is *subsume*, and
 2. its leaf nodes are Web services that are connected by the minimal number of arcs such that the usability degree of each Web service for every goal template is captured
- algorithms that maintain the structure of the SDC graph in its dynamic environment, i.e. for adding, removing, and updating goal templates as well as Web service descriptions
- algorithms for SDC-enabled Web service discovery at runtime, including the goal formulation process and the determination of the usability of a Web service for a goal instance.

This report merely presents the detailed specification of the SDC-enabled Web service discovery. As the next steps in this research, the SDC technique will be implemented as a component in the WSMX system (the WSMO reference implementation), and, on the basis of this, its applicability for real world scenarios will be evaluated. We shall also discuss related work in more detail and therewith explicate the research contributions of the presented approach.

References

- [1] R. Akkiraju, J. Farrell, J. Miller, M. Nagarajan, M.-T. Schmidt, A. Sheth, and K. Verma. Web Service Semantics - WSDL-S. W3C Member Submission 7 November 2005, 2005. online: <http://www.w3.org/Submission/WSDL-S/>.
- [2] J. Bang-Jønsen and G. Gutin. *Digraphs: Theory, Algorithms and Applications*. Monographs in Mathematics. Springer, London, 2000.
- [3] S. Battle, A. Bernstein, H. Boley, B. Grosz, M. Gruninger, R. Hull, M. Kifer, Martin. D., McIlraith. S., D. McGuinness, J. Su, and S. Tabet. Semantic Web Services Framework (SWSF). W3C Member Submission 9 September 2005, 2005. online: <http://www.w3.org/Submission/SWSF/>.
- [4] B. Benatallah, M.-S. Hacid, A. Leger, C. Rey, and F. Toumani. On Automating Web Services Discovery. *VLDB Journal*, 14(1):84–96, 2005.
- [5] A. B. Bondi. Characteristics of Scalability and their Impact on Performance. In *Proceedings of the 2nd International Workshop on Software and Performance, Ottawa, Ontario, Canada*, pages 195–203, 2000.
- [6] L. Cabral, J. Domingue, S. Galizia, A. Gugliotta, B. Norton, V. Tanasescu, and C. Pedrinaci. IRS-III – A Broker for Semantic Web Services based Applications. In *In Proc. of the 5th International Semantic Web Conference (ISWC 2006), Athens(GA), USA*, 2006.
- [7] S. Colucci, T. Di Noia, E. Di Sciascio, F. M. Donini, and M. Mongiello. Concept abduction and contraction for semantic-based discovery of matches and negotiation spaces in an e-marketplace. *Electronic Commerce Research and Applications*, 4:345–361, 2005.
- [8] 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.
- [9] I. Constantinescu, B. Faltings, and W. Binder. Large Scale, Type-Compatible Service Composition. In *Proceedings of the IEEE International Conference on Web Services (ICWS'04), San Diego, California, USA*, 2004.
- [10] J. de Bruijn, D. Fensel, H. Lausen, A. Polleres, D. Roman, and M. Stollberg. *Enabling Semantic Web Services. The Web Service Modeling Ontology*. Springer, 2006. (to appear).
- [11] R. Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer, Heidelberg, 3. edition, 2005.
- [12] O. M. Duschka and M. R. Genesereth. Query planning in Infomaster. In *In Proc. of the ACM Symposium on Applied Computing*, 1997.
- [13] D. Fensel and F. van Harmelen. Unifying Reasoning and Search into Something that Scales up to Frillions of Triples. Technical Report DERI-TR-2007-01-11, DERI, 2007.
- [14] J. Handy. *The Cache Memory Book*. Series in Computer Architecture and Design. Morgan Kaufmann, 2 edition, 1998.

- [15] I. Horrocks and S. Tobies. Optimisation of Terminological Reasoning. In *Proceedings of the International Workshop in Description Logics 2000 (DL2000)*, 2000.
- [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, R. Lara, H. Lausen, and D. Fensel. Semantic Web Service Discovery in the WSMO Framework. In J. Cardoses, editor, *Semantic Web: Theory, Tools and Applications*. Idea Publishing Group, 2006. (to appear).
- [18] U. Keller, H. Lausen, and M. Stollberg. On the Semantics of Functional Descriptions of Web Services. In *Proceedings of the 3rd European Semantic Web Conference (ESWC 2006)*, Montenegro, 2006.
- [19] M. Kerrigan. Web Service Selection Mechanisms in the Web Service Execution Environment (WSMX). In *Proceedings of the 21st Annual ACM Symposium on Applied Computing (SAC)*, 2006.
- [20] M. Kerrigan. WSMOViz: An Ontology Visualization Approach for WSMO. In *Proc. of the 10th International Conference on Information Visualization (IV)*, London, England *Proceedings of the 21st Annual ACM Symposium on Applied Computing (SAC)*, 2006.
- [21] D. Knuth. Big Omicron and big Omega and big Theta. *ACM SIGACT News*, 8(2), 1976.
- [22] R. Lara, M. A. Corella, and P. Castells. A Flexible Model for Web Service Siscovery. In *Proc. of the 1st International Workshop on Semantic Matchmaking and Resource Retrieval: Issues and Perspectives*, Seoul, South Korea, 2006.
- [23] L. Li and I. Horrocks. A software framework for matchmaking based on semantic web technology. In *Proceedings of the 12th International Conference on the World Wide Web*, Budapest, Hungary, 2003.
- [24] A. D. Maedche. *Ontology Learning for the Semantic Web*. Springer, Berlin, Heidelberg, 2002.
- [25] D. Martin. OWL-S: Semantic Markup for Web Services. W3C Member Submission 22 November 2004, 2004. online: <http://www.w3.org/Submission/OWL-S/>.
- [26] A. Mocan, E. Cimpian, M. Stollberg, F. Scharffe, and J. Scicluna. WSMO Mediators. WSMO deliverable D29 final draft 21 Dec 2005, 2005. available at: <http://www.wsmo.org/TR/d29/>.
- [27] P O'Neil and E. O'Neil. *Database - Principles, Programming, Performance*. Morgan Kaufmann, 2 edition, 1999.
- [28] M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic matching of web services capabilities. In *Proceedings of the First International Semantic Web Conference*, Springer, 2002.
- [29] C. Preist. A Conceptual Architecture for Semantic Web Services. In *Proc. of the Int. Semantic Web Conf. (ISWC 2004)*, 2004.
- [30] D. Roman, H. Lausen, and U. Keller. Web Service Modeling Ontology (WSMO). Working Draft D2, WSMO Working Group, 2005. final version v1.2, 13 April 2005, online at: <http://www.wsmo.org/TR/d2/v1.2/>.

- [31] F. Scharffe. Schema Mappings for the Web. In Daniel Schwabe, editor, *Abstract in Proceedings of the International Semantic Web Conference*. International Semantic Web Conference, Springer, 2006. Poster at the PhD Symposium.
- [32] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 2 edition, 2005.
- [33] R. M. Smullyan. *First Order Logic*. Springer, 1968.
- [34] M. Stollberg. Reasoning Tasks and Mediation on Choreography and Orchestration in WSMO. In *Proceedings of the 2nd International WSMO Implementation Workshop (WIW 2005)*, Innsbruck, Austria, 2005.
- [35] M. Stollberg. Semantic Caching for Service Oriented Architectures. In *Serviceware – DERI PhD Seminar, November, 2006*. online: <http://members.deri.at/~michaels/publications/mstollberg-PhDthesisOverview.pdf>.
- [36] M. Stollberg, E. Cimpian, A. Mocan, and D. Fensel. A Semantic Web Mediation Architecture. In M. T. Kon and D. Lemire, editors, *Canadian Semantic Web*, Semantic Web and Beyond: Computing for Human Experience. Springer, 2006.
- [37] M. Stollberg, E. Cimpian, A. Mocan, and D. Fensel. A Semantic Web Mediation Architecture. In *Proceedings of the 1st Canadian Semantic Web Working Symposium (CSWWS 2006)*, Quebec, Canada, 2006.
- [38] M. Stollberg and U. Keller. Delta Relations – Semantic Difference of Functional Descriptions. Technical Report DERI-TR-2006-08-18, DERI, 2006.
- [39] M. Stollberg and U. Keller. Semantic Web Service Discovery: Matchmaking for Goal Templates and Goal Instances on Rich Functional Descriptions. Technical Report DERI-2006-10-20, DERI, 2006. available at: <http://members.deri.at/~michaels/publications/stollberg-keller-discovery-deriTR.pdf>.
- [40] M. Stollberg and U. Keller. Semantic Web Service Discovery with Rich Functional Descriptions. *International Journal of Electronic Commerce (IJECE)*, Special Issue on: *Semantic Matchmaking and Resource Retrieval*, 2007. (unpublished).
- [41] M. Stollberg, U. Keller, H. Lausen, and S. Heymans. Two-phase Web Service Discovery based on Rich Functional Descriptions. In *4th European Semantic Web Conference (ESWC 2007)*, 2007. (submitted).
- [42] M. Stollberg and B. Norton. A Refined Goal Model for Semantic Web Services. In *International Conference on Internet and Web Applications and Services (ICIW 2007)*, Mauritius, 2007. (submitted).
- [43] M. Stollberg, D. Roman, I. Toma, U. Keller, R. Herzog, P. Zugmann, and D. Fensel. Semantic Web Fred – Automated Goal Resolution on the Semantic Web. In *Proc. of the 38th Hawaii International Conference on System Science*, 2005.
- [44] P. Traverso and M. Pistore. Automatic Composition of Semantic Web Services into Executable Processes. In *Proc. 3rd International Semantic Web Conference (ISWC 2004)*, Hiroshima, Japan, 2004.
- [45] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume I. Computer Science Press, 1988.

- [46] 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. Special Issue on Universal Global Integration.
- [47] L.-H. Vu, M. Hauswirth, and K. Aberer. QoS-Based Service Selection and Ranking with Trust and Reputation Management. In *Proc. of the OTM Confederated International Conferences CoopIS, DOA, and ODBASE 2005, Agia Napa, Cyprus*, pages 466–483, 2005.
- [48] H. Wache, L. Serafini, and R. García-Castro. Survey of Scalability Techniques for Reasoning with Ontologies. Deliverable D2.1.1, Knowledge Web, 2004.
- [49] D. Wessels. *Web Caching*. O'Reilly & Associates Inc, 2001.
- [50] M. Zaremba and C. Bussler. Towards Dynamic Execution Semantics in Semantic Web Services. In *Proc. of the Workshop on Web Service Semantics: Towards Dynamic Business Integration, International Conference on the World Wide Web (WWW2005)*, Chiba, Japan, 2005.

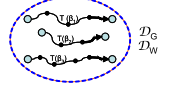
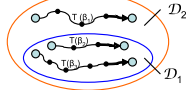
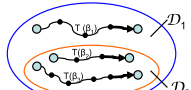
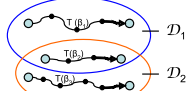
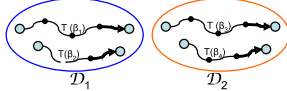
APPENDIX

A Matching Degrees Overview

The following provides a concise overview of the matching degrees definition, defined over functional descriptions as specified in Section 1.2.2.

In the SGC technique, we use the matchmaking degrees not only for denoting the usability of a Web service for solving a goal template but also for denoting the degree of similarity of goal templates. With respect to this, Table 6 provides a concise overview of the degree definitions, Table 7 explains their meaning for the usability of Web services, and Table 8 explains their meaning for the similarity of goal templates.

Table 6: Definition of Matching Degrees for $\mathcal{D}_1, \mathcal{D}_2$

Denotation $\mathcal{D}_1 = (\Sigma, \Omega, IF, \phi^{\mathcal{D}_1})$ $\mathcal{D}_2 = (\Sigma, \Omega, IF, \phi^{\mathcal{D}_2})$	Definition $\beta : IF \rightarrow \mathcal{U}_{\mathcal{A}}$ $\phi^{\mathcal{D}} = [\phi^{pre}]_{\Sigma_D^{pre} \rightarrow \Sigma_D} \Rightarrow \phi^{eff}$ $\Omega_{\mathcal{A}} = \Omega \cup [\Omega]_{\Sigma_D^{pre} \rightarrow \Sigma_D}$	Meaning for $\{\tau\}_1 \models_{\mathcal{A}} \mathcal{D}_1$, and $\{\tau\}_2 \models_{\mathcal{A}} \mathcal{D}_2$	Visualization $\bigcirc = \mathcal{D}_1$ $\bigcirc = \mathcal{D}_2$
exact ($\mathcal{D}_1, \mathcal{D}_2$)	$\Omega_{\mathcal{A}} \models \forall \beta. \phi^{\mathcal{D}_1} \Leftrightarrow \phi^{\mathcal{D}_2}$	if and only if $\tau \in \{\tau\}_1$ then $\tau \in \{\tau\}_2$	
plugin ($\mathcal{D}_1, \mathcal{D}_2$)	$\Omega_{\mathcal{A}} \models \forall \beta. \phi^{\mathcal{D}_1} \Rightarrow \phi^{\mathcal{D}_2}$	if $\tau \in \{\tau\}_1$ then $\tau \in \{\tau\}_2$	
subsume ($\mathcal{D}_1, \mathcal{D}_2$)	$\Omega_{\mathcal{A}} \models \forall \beta. \phi^{\mathcal{D}_1} \Leftarrow \phi^{\mathcal{D}_2}$	if $\tau \in \{\tau\}_2$ then $\tau \in \{\tau\}_1$	
intersect ($\mathcal{D}_1, \mathcal{D}_2$)	$\Omega_{\mathcal{A}} \models \exists \beta. \phi^{\mathcal{D}_1} \wedge \phi^{\mathcal{D}_2}$	there is a τ such that $\tau \in \{\tau\}_1$ and $\tau \in \{\tau\}_2$	
disjoint ($\mathcal{D}_1, \mathcal{D}_2$)	$\Omega_{\mathcal{A}} \models \neg \exists \beta. \phi^{\mathcal{D}_1} \wedge \phi^{\mathcal{D}_2}$	there is no τ such that $\tau \in \{\tau\}_1$ and $\tau \in \{\tau\}_2$	

Because of their matching conditions, there is a formally relationship between the matching degrees. In particular, it holds that:

- (i) $\neg \text{disjoint} \Rightarrow \text{intersect}$ (iii) $\text{plugin} \Rightarrow \text{intersect}$
- (ii) $\text{subsume} \Rightarrow \text{intersect}$ (iv) $\text{plugin} \wedge \text{subsume} \Rightarrow \text{exact}$.

With respect to this, we can define an order to the matching degrees such that

$$\text{exact} > \text{plugin}, \text{subsume} > \text{intersect}$$

To properly separate the distinct situations, we apply a strict use of the matching degrees by always using the degree with the highest ordering.

Table 7: Meaning of Matching Degrees for the Usability of a Web service for solving a Goal

exact ($\mathcal{D}_G, \mathcal{D}_W$)	W can be used for solving any goal instance $GI(\mathcal{G})$
plugin ($\mathcal{D}_G, \mathcal{D}_W$)	all possible solutions for \mathcal{G} can be provided by W but there can exist a $\tau \in \{\tau\}_W$ such that $\tau \notin \{\tau\}_G$. As every possible input binding β for \mathcal{G} that defined for $GI(\mathcal{G})$ triggers such an execution of W that $\{\tau\}_{W(\beta)} \in \{\tau\}_G$, under this degree W can be used for solving any goal instance of \mathcal{G} .
subsume ($\mathcal{D}_G, \mathcal{D}_W$)	all executions of W can satisfy \mathcal{G} , but there are possible solutions for \mathcal{G} that cannot be provided by W . In consequence, for W to be usable for a goal instance $GI(\mathcal{G})$, it has to hold that $\{\tau\}_{GI(\mathcal{G})} \subseteq \{\tau\}_W$. This is given if the input binding β defined for $GI(\mathcal{G})$ allows to invoke W .
intersect ($\mathcal{D}_G, \mathcal{D}_W$)	under this degree there are possible solutions for \mathcal{G} that cannot be provided by W , as well as executions of W that do not solve \mathcal{G} . Hence, for W to be usable for a goal instance $GI(\mathcal{G})$, it has to hold that the input binding β defined for $GI(\mathcal{G})$ instantiates \mathcal{G} in a way such that $\{\tau\}_{GI(\mathcal{G})} \subseteq \{\tau\}_{W(\beta)}$.
disjoint ($\mathcal{D}_G, \mathcal{D}_W$)	W can not be used for solving \mathcal{G} or any of its instantiations.

Table 8: Meaning of Matching Degrees for Semantic Similarity of Goals

exact ($\mathcal{D}_{G_1}, \mathcal{D}_{G_2}$)	Both goals are semantically equivalent. Hence, all τ that satisfy G_1 also satisfy G_2 and vice versa. All Web services that are usable for G_1 are also usable G_2 under the same usability degree.
plugin ($\mathcal{D}_{G_1}, \mathcal{D}_{G_2}$)	Each τ that satisfies G_1 also satisfies G_2 . Hence, all Web services usable for G_1 are also usable for G_2 but not vice versa.
subsume ($\mathcal{D}_{G_1}, \mathcal{D}_{G_2}$)	Each τ that satisfies G_2 also satisfies G_1 . Hence, all Web services usable for G_2 are also usable for G_1 but not vice versa.
intersect ($\mathcal{D}_{G_1}, \mathcal{D}_{G_2}$)	There is at least one τ that resolves both goals. A Web service that can provide this τ is hence usable for both goals.
disjoint ($\mathcal{D}_{G_1}, \mathcal{D}_{G_2}$)	A τ that resolves both goals does not exists. We can not make any statement between the Web services usable for solving the goals.

B Proof of Theorem 3.1: Inference Rules for Usability Degrees

The following provides the proof for theorem 3.1 in Section 3.2 that defines the inference rules for determining the usability degree of a Web service W for a goal template \mathcal{G}_2 on the basis of the similarity degree between \mathcal{G}_2 and another goal template \mathcal{G}_1 and the usability degree of W for \mathcal{G}_1 . For each of the five possible similarity degree, it distinguishes the possible usability degrees of W for \mathcal{G}_2 . To proof the theorem, we need to show that the enlisted inferable usability degrees for W for \mathcal{G}_2 are correct and the only possible ones.

We use the following symbols. \mathcal{G}_1 is the goal template for which the usability degree of a Web service W is known, and \mathcal{G}_2 is the goal template for which the usability degree of W shall be determined. $\mathcal{D}_{\mathcal{G}_1}$ is the functional description of \mathcal{G}_1 that formally describes all its possible solutions $\{\tau\}_{\mathcal{G}_1}$, $\mathcal{D}_{\mathcal{G}_2}$ the one of \mathcal{G}_2 describing the set of solutions $\{\tau\}_{\mathcal{G}_2}$, and \mathcal{D}_W a functional description such that $W \models_{\mathcal{A}} \mathcal{D}_W$, i.e. \mathcal{D}_W formally describes $\{\tau\}_W$ as the set of all possible executions of W . All functional descriptions are defined as a 4-tuple $\mathcal{D} = (\Sigma, \Omega, IF, \phi^{\mathcal{D}})$, cf. Definition 1.2, with implication semantics such that $\phi^{\mathcal{D}} = [\phi^{pre}]_{\Sigma_D^{pre} \rightarrow \Sigma_D} \Rightarrow \phi^{eff}$ (cf. Definition 1.3). We further use the definitions from Appendix A on the matching degrees (cf. Table 6), and their meaning for the usability of a Web service for a goal template and its corresponding goal instances (cf. Table 7), respectively the meaning for denoting the similarity degree of goal templates (cf. Table 8). To properly separate the distinct situations, we apply a strict use of the matching degrees by always using the degree with the highest ordering of *exact* > *plugin*, *subsume* > *intersect*.

Proof. We commence with the first part on *exact*($\mathcal{G}_1, \mathcal{G}_2$). This is defined as $\Omega_{\mathcal{A}} \models \forall \beta. \phi^{\mathcal{D}_{\mathcal{G}_1}} \Leftrightarrow \phi^{\mathcal{D}_{\mathcal{G}_2}}$, such that for all possible input bindings β it holds that $\tau \in \{\tau\}_{\mathcal{G}_1}$ if and only if $\tau \in \{\tau\}_{\mathcal{G}_2}$ so that $\{\tau\}_{\mathcal{G}_1} = \{\tau\}_{\mathcal{G}_2}$. Because of this equivalence, it trivially holds that the usability degree of a Web service W is the same for \mathcal{G}_1 and for \mathcal{G}_2 .

We now discuss the second part that is concerned with the *plugin*($\mathcal{G}_1, \mathcal{G}_2$) similarity degree. This is formally defined as $\Omega_{\mathcal{A}} \models \forall \beta. \phi^{\mathcal{D}_{\mathcal{G}_1}} \Rightarrow \phi^{\mathcal{D}_{\mathcal{G}_2}}$, so that $\{\tau\}_{\mathcal{G}_1} \subseteq \{\tau\}_{\mathcal{G}_2}$ and if $\tau \in \{\tau\}_{\mathcal{G}_1}$ then $\tau \in \{\tau\}_{\mathcal{G}_2}$. The following holds under this similarity degree:

⟨1⟩ Each Web service W that is usable for \mathcal{G}_1 is also usable for \mathcal{G}_2 : if $\exists \tau. \tau \in (\{\tau\}_{\mathcal{G}_1} \cap \{\tau\}_W)$ then also $\exists \tau. \tau \in (\{\tau\}_{\mathcal{G}_2} \cap \{\tau\}_W)$ because $\{\tau\}_{\mathcal{G}_1} \subseteq \{\tau\}_{\mathcal{G}_2}$. Thus, whenever the usability degree of W for \mathcal{G}_1 is either *exact*, *plugin*, *subsume*, or *intersect* – each of these satisfies the basic matching condition of the goal template level (cf. Definition 1.1) – then the usability degree of W for \mathcal{G}_2 can not be *disjoint*. This applies to clauses 2.1 to 2.8.

⟨2⟩ *exact*(\mathcal{G}_1, W) defines that $\{\tau\}_{\mathcal{G}_1} = \{\tau\}_W$. Under the assumption that not *exact*($\mathcal{G}_1, \mathcal{G}_2$) (handled above), it holds that $\{\tau\}_{\mathcal{G}_1} \subset \{\tau\}_{\mathcal{G}_2}$. Hence $\{\tau\}_{\mathcal{G}_2} \supset \{\tau\}_W$, so that *subsume*(\mathcal{G}_2, W) is a possible usability degree of W for \mathcal{G}_2 . There can not be any τ with $\tau \in \{\tau\}_W$ and $\tau \notin \{\tau\}_{\mathcal{G}_2}$ such that *intersect*(\mathcal{G}_2, W) can not hold; also, *disjoint*(\mathcal{G}_2, W) can not hold because of ⟨1⟩. This shows clause 2.1.

⟨3⟩ *subsume*(\mathcal{G}_1, W) defines that $\{\tau\}_{\mathcal{G}_1} \supseteq \{\tau\}_W$. Under the assumption that not *exact*($\mathcal{G}_1, \mathcal{G}_2$), it holds that $\{\tau\}_W \subseteq \{\tau\}_{\mathcal{G}_1} \subset \{\tau\}_{\mathcal{G}_2}$, so that the only possible usability degree of W for \mathcal{G}_2 is *subsume*(\mathcal{G}_2, W).

⟨4⟩ *plugin*(\mathcal{G}_1, W) defines that $\{\tau\}_{\mathcal{G}_1} \subseteq \{\tau\}_W$. Here, the usability degree of W for \mathcal{G}_2 can be either *exact* if $\{\tau\}_{\mathcal{G}_2} = \{\tau\}_W$, *plugin* if $\{\tau\}_{\mathcal{G}_2} \subset \{\tau\}_W$, *subsume* if $\{\tau\}_{\mathcal{G}_2} \supset \{\tau\}_W$, or *intersect* if there exists a τ such that $\tau \in \{\tau\}_W$ and $\tau \notin \{\tau\}_{\mathcal{G}_2}$; it can not be *disjoint* because of ⟨1⟩. This shows clauses 2.3 - 2.6.

⟨5⟩ *intersect*(\mathcal{G}_1, W) defines that $\exists \tau \in (\{\tau\}_{\mathcal{G}_1} \cap \{\tau\}_W)$, so that also $\exists \tau \in (\{\tau\}_{\mathcal{G}_2} \cap \{\tau\}_W)$ because of $\{\tau\}_{\mathcal{G}_1} \subseteq \{\tau\}_{\mathcal{G}_2}$. Hence, *intersect*(\mathcal{G}_2, W) is one possible usability degree of W for \mathcal{G}_2 . It can also be *subsume*(\mathcal{G}_2, W) if $\{\tau\}_{\mathcal{G}_2} \supset \{\tau\}_W$. It can not be *exact* or *plugin* because then it would hold that $\tau \in (\{\tau\}_{\mathcal{G}_1} \subseteq \{\tau\}_W)$ – which contradicts the matching condition for *intersect*(\mathcal{G}_1, W). This proves clauses 2.7 - 2.8.

$\langle 6 \rangle$ *disjoint*(\mathcal{G}_1, W) defines that $\neg \exists \tau \in (\{\tau\}_{\mathcal{G}_1} \cap \{\tau\}_W)$. Under this similarity degree, we only know that W can not provide any solution for \mathcal{G}_1 . However, W might be able to provide a solution for \mathcal{G}_2 such that $\exists \tau \in \{\tau\}_W \cap (\{\tau\}_{\mathcal{G}_2} \cap \{\tau\}_{\mathcal{G}_1})$, but we can not infer the usability degree of such a W for \mathcal{G}_2 . This relates to clause 2.9.

Next, we discuss the third part for the similarity degree *subsume*($\mathcal{G}_1, \mathcal{G}_2$). This is formally defined as $\Omega_{\mathcal{A}} \models \forall \beta. \phi^{\mathcal{D}_{\mathcal{G}_1}} \Leftarrow \phi^{\mathcal{D}_{\mathcal{G}_2}}$, so that $\{\tau\}_{\mathcal{G}_1} \supseteq \{\tau\}_{\mathcal{G}_2}$ and if $\tau \in \{\tau\}_{\mathcal{G}_2}$ then $\tau \in \{\tau\}_{\mathcal{G}_1}$. The following holds under this similarity degree:

$\langle 1 \rangle$ if *exact*(\mathcal{G}_1, W) such that $\{\tau\}_{\mathcal{G}_1} = \{\tau\}_W$ and under the assumption not *exact*($\mathcal{G}_1, \mathcal{G}_2$) (handled above), it holds that $\{\tau\}_W \supset \{\tau\}_{\mathcal{G}_2}$. Hence, the only possible usability degree of W for \mathcal{G}_2 is *plugin*(\mathcal{G}_2, W); it can not be *exact*, *subsume*, or *intersect* because $\{\tau\}_W = \{\tau\}_{\mathcal{G}_1} \supset \{\tau\}_{\mathcal{G}_2}$, and not *disjoint* because every possible solution for \mathcal{G}_1 can be provided by W . Similar, if *plugin*(\mathcal{G}_1, W) then $\{\tau\}_W \supseteq \{\tau\}_{\mathcal{G}_1} \supset \{\tau\}_{\mathcal{G}_2}$ so that the only possible usability degree of W for \mathcal{G}_2 is *plugin*(\mathcal{G}_2, W). This proves clauses 3.1 and 3.2.

$\langle 2 \rangle$ *subsume*(\mathcal{G}_1, W) defines that $\{\tau\}_{\mathcal{G}_1} \supseteq \{\tau\}_W$. Because $\{\tau\}_W$ can be any subset of $\{\tau\}_{\mathcal{G}_1}$, here all five usability degrees are possible for W and \mathcal{G}_2 . In particular, W can be not usable for \mathcal{G}_2 if $\neg \exists \tau \in (\{\tau\}_{\mathcal{G}_2} \cap \{\tau\}_W)$. This relates to clauses 3.3 - 3.7.

$\langle 3 \rangle$ *intersect*(\mathcal{G}_1, W) defines that $\exists \tau_1 \in (\{\tau\}_{\mathcal{G}_1} \cap \{\tau\}_W)$ but there can be a $\tau_2 \in \{\tau\}_{\mathcal{G}_1}$ but $\tau_2 \notin \{\tau\}_W$ as well as a $\tau_3 \notin \{\tau\}_{\mathcal{G}_1}$ but $\tau_3 \in \{\tau\}_W$. The possible degrees under which W is usable for \mathcal{G}_2 are *plugin*(\mathcal{G}_2, W) if $\{\tau\}_{\mathcal{G}_2} \subset \{\tau\}_W$, or *intersect*(\mathcal{G}_2, W) if $\exists \tau \in (\{\tau\}_{\mathcal{G}_2} \cap \{\tau\}_W)$. W might also be not usable, i.e. *disjoint*(\mathcal{G}_2, W), if the condition for the *intersect* degree is not satisfied. However, the usability can neither be *subsume* and hence not *exact* because this would require $\{\tau\}_W \subseteq \{\tau\}_{\mathcal{G}_2} \subseteq \mathcal{G}_1$ – which contradicts the condition of *intersect*(\mathcal{G}_1, W) under *subsume*($\mathcal{G}_1, \mathcal{G}_2$). This proves clauses 3.8 - 3.10.

$\langle 4 \rangle$ if *disjoint*(\mathcal{G}_1, W) then also *disjoint*(\mathcal{G}_2, W) because if W can not provide a $\tau \in \{\tau\}_{\mathcal{G}_1}$ then it also can not provide a $\tau \in \{\tau\}_{\mathcal{G}_2}$ with $\{\tau\}_{\mathcal{G}_1} \supseteq \{\tau\}_{\mathcal{G}_2}$. This proves clause 3.11.

We now turn towards the fourth part on the similarity degree *intersect*($\mathcal{G}_1, \mathcal{G}_2$). Its matching condition is $\Omega_{\mathcal{A}} \models \exists \beta. \phi^{\mathcal{D}_{\mathcal{G}_1}} \wedge \phi^{\mathcal{D}_{\mathcal{G}_2}}$, so that $\exists \tau_1 \in (\{\tau\}_{\mathcal{G}_1} \cap \{\tau\}_{\mathcal{G}_2})$ but there can be a $\tau_2 \in \{\tau\}_{\mathcal{G}_1}$ but $\tau_2 \notin \{\tau\}_{\mathcal{G}_2}$ as well as a $\tau_3 \notin \{\tau\}_{\mathcal{G}_1}$ but $\tau_3 \in \{\tau\}_{\mathcal{G}_2}$. Here, the following holds:

$\langle 1 \rangle$ if *exact*(\mathcal{G}_1, W) such that $\{\tau\}_{\mathcal{G}_1} = \{\tau\}_W$ and under the assumption not *exact*($\mathcal{G}_1, \mathcal{G}_2$) (handled above), the only possible usability degree of W for \mathcal{G}_2 is *intersect*(\mathcal{G}_2, W) because $\exists \tau. \tau \in (\{\tau\}_{\mathcal{G}_1} \cap \{\tau\}_{\mathcal{G}_2})$ and $\{\tau\}_{\mathcal{G}_1} = \{\tau\}_W$. This proves clause 4.1.

$\langle 2 \rangle$ if *plugin*(\mathcal{G}_1, W) such that $\{\tau\}_{\mathcal{G}_1} \subseteq \{\tau\}_W$, then the only possible usability degrees of W for \mathcal{G}_2 are *plugin*(\mathcal{G}_2, W) if $\{\tau\}_{\mathcal{G}_2} \subseteq \{\tau\}_W$ or *intersect*(\mathcal{G}_1, W) otherwise. It can not be *subsume*(\mathcal{G}_2, W) and hence not *exact*(\mathcal{G}_2, W) because this would require $\{\tau\}_{\mathcal{G}_2} \supseteq \{\tau\}_W$ which contradicts $\{\tau\}_{\mathcal{G}_1} \subseteq \{\tau\}_W$ under *intersect*($\mathcal{G}_1, \mathcal{G}_2$) because there exists a τ_2 such that $\tau_2 \in \{\tau\}_{\mathcal{G}_1}$ but $\tau_2 \notin \{\tau\}_{\mathcal{G}_2}$. This proves clauses 4.2 and 4.3.

$\langle 3 \rangle$ if *subsume*(\mathcal{G}_1, W) such that $\{\tau\}_{\mathcal{G}_1} \supseteq \{\tau\}_W$, then W can be usable for \mathcal{G}_2 under the *subsume* degree if $\{\tau\}_{\mathcal{G}_2} \supseteq \{\tau\}_W$, or under the *intersect* degree if there is a τ such that $\tau \in \{\tau\}_W$ but $\tau \in \{\tau\}_{\mathcal{G}_2}$; otherwise, W is not usable so that *disjoint*(\mathcal{G}_2, W). However, the usability degree of W for \mathcal{G}_2 can not be *plugin* and hence not *exact* because this would require that $\{\tau\}_{\mathcal{G}_2} \subseteq \{\tau\}_W$ which contradicts *subsume*(\mathcal{G}_1, W) under the *intersect* similarity degree. This proves clauses 4.3 - 4.6.

$\langle 4 \rangle$ under the *intersect* degree for both the similarity of \mathcal{G}_1 and \mathcal{G}_2 as well as for the usability of W for \mathcal{G}_1 , all five usability degrees are possible for W and \mathcal{G}_2 . In particular, W might not be usable for \mathcal{G}_2 if $\neg \exists \tau \in (\{\tau\}_{\mathcal{G}_2} \cap \{\tau\}_W)$. This relates to clauses 4.7 - 4.11.

$\langle 5 \rangle$ *disjoint*(\mathcal{G}_1, W) defines that $\neg \exists \tau \in (\{\tau\}_{\mathcal{G}_1} \cap \{\tau\}_W)$. However, W might be able to provide a solution for \mathcal{G}_2 such that $\exists \tau \in \{\tau\}_W \cap (\{\tau\}_{\mathcal{G}_2} \cap \{\tau\}_{\mathcal{G}_1})$, but we can not infer the usability degree of such a W for \mathcal{G}_2 . This relates to clause 4.12.

We finally discuss the implications of similarity degree $disjoint(\mathcal{G}_1, \mathcal{G}_2)$. This defines that \mathcal{G}_1 and \mathcal{G}_2 do not have a common solution with the condition $\Omega_{\mathcal{A}} \models \neg \exists \beta. \phi^{\mathcal{D}_{\mathcal{G}_1}} \wedge \phi^{\mathcal{D}_{\mathcal{G}_2}}$. If the usability of W for \mathcal{G}_1 is either *subsume* or *exact*, then $\{\tau\}_{\mathcal{G}_1} \supseteq \{\tau\}_W$. As $disjoint(\mathcal{G}_1, \mathcal{G}_2)$ defines that $\{\tau\}_{\mathcal{G}_1} \cap \{\tau\}_{\mathcal{G}_2} = \emptyset$, W can not be usable in these cases. This proves clauses 5.1 and 5.2. For all other usability degrees of W for \mathcal{G}_1 , we can not make any statement about the usability of W for \mathcal{G}_2 : regardless whether $plugin(\mathcal{G}_1, W)$ or $intersect(\mathcal{G}_1, W)$ or $disjoint(\mathcal{G}_1, W)$, there might always be a τ such that $\tau \in (\{\tau\}_{\mathcal{G}_2} \cap \{\tau\}_W)$. In particular, W might be usable for \mathcal{G}_2 if it is not usable for \mathcal{G}_1 . This relates to clause 5.3 and completes the proof. \square

C Complete SDC Algorithm

This appendix provides all algorithms that have been specified in this report in a concise manner, including the algorithms for creation and maintenance of the SDC graph (*cf.* Section 5), and those for SDC-enabled Web service discovery (*cf.* Section 5).

Table 9: Syntax for Pseudo Code used in Algorithm Definitions

<code>:=</code>	data type declaration
<code>=</code>	value assignment
<code>null</code>	denotes that the value of an object is empty
<code>and, or</code>	logical operators that connect conditions
<code>!</code>	defines the negation of the subsequent condition
<code>{e}</code>	set with elements of type <i>e</i>
<code>e in {e}</code>	denotes that <i>e</i> is an element of a set
<code>name(input,...)</code>	name and the input value of a method
<code>name(input)</code>	denotes a function
<code>forall(condition)</code>	loop that is iterated for all objects for which the condition is satisfied until the halting condition is reached
<code>if (condition) then (action) else (action)</code>	defines a conventional guarded action
<code>return(value)</code>	the halting condition that returns the value

```
// type declarations
G,G2,G3,G4,G5 := goaltemplate;
GI = goalinstance;
W := webservice;
goalStore := {goaltemplate};
goalTree := {s(goaltemplate,goaltemplate)};
goalGraph := (goalStore,{goalTree});
discoveryCache := {d(goaltemplate,webservice)};
sdcGraph := (goalGraph,discoveryCache);

// function declaration for supported entry points
action := ( new | remove | update )
item := ( goaltemplate | goalinstance | webservice );
event(action,item) := boolean;
// main control
SDCcontrol {
  if ( event(new,GI) ) then discovery(GI) ;
  if ( event(new,G) ) then insert(G);
  if ( event(remove,G) ) then remove(G);
  if ( event(update,G) ) then update(G);
  if ( event(new,W) ) then insert(W);
  if ( event(remove,W) ) then remove(W);
  if ( event(update,W) ) then update(W);
}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% runtime Web service discovery for a new goal instance
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
discovery(GI) {
  G = goalTemplateSearch(GI);
  GI = (G,inputs);
  lookup(G);
}
```

```

    goalInstanceMatching(GI);
}
// finding the most appropriate goal template
target := goaltemplate;
goalTemplateSearch(GI);
target = null;
findRootNode(GI);
if (! target = null) then
    findChildNode(target);
return target;
// find root of goal tree
findRootNode(GI){
    forall (root(G)) {
        if instantiates (GI,G) then
            target = G;
            return target;
        }
    }
return target;
}
// find child node in goal tree
findChildNode(G1) {
    forall (subsume(G1,G2)) {
        if instantiates (GI,G2) then
            target = G2;
            findChildNode(target);
        else
            return target;
        }
    }
return target;
}
// usability lookup for inferable usability degrees
lookup(G) {
    if ( child(G) ) then {
        forall ( G2 and subsume(G2,G) ) {
            forall ( W and ( exact(G2,W) or plugin(G2,W) ) ) {
                return W;
            }
        }
        lookup(G2);
    } } }
// goal instance level matchmaking
goalInstanceMatching(GI) {
    forall ( W and exact(G,W) or plugin(G,W) ) {
        return W; }
    forall ( W and subsume(G,W) ) {
        if ( satisfiable (W,inputs) ) then
            return W; }
    forall ( W and intersect(G,W) ) {
        if ( satisfiable (G,W,inputs) ) then
            return W;
        else
            return systemout = 'goal instance can not be solved';
    } }
}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% insertion of a new goal template
% (for iterative SDC graph creation)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// function declarations
position(goaltemplate) := (root | child);
intersectionGoalTemplate(goaltemplate,goaltemplate) := goaltemplate;
discoveryCache(G) := {d(G,webservice)};
// main
insert(G){
    if ( goalStore != {} ) then {

```

```

forall ( G2 and position(G2) = root) {
  if ( exact(G2,G) ) then return goalStore;
  if ( plugin(G2,G) ) then rootNodeInsertion(G,G2);
  if ( subsume(G2,G) ) then childNodeInsertion{G,G2};
  if ( intersect (G2,G) ) then {
    position (G) = root;
    iArcResolution{G,G2}; }
  else {
    position (G) = root;
    return goalStore = goalStore + G;
  } }
discoveryCacheCreation(G);
}
// inserting a new goal template as a root node
rootNodeInsertion{G,G2} {
  goalStore = goalStore + G;
  position (G2) = child;
  position (G) = root;
  goalTree = goalTree + s(G,G2);
  return goalGraph;
}
// inserting a new goal template as a child node
childNodeInsertion{G,G2}{
  goalStore = goalStore + G;
  position (G) = child;
  goalTree = goalTree + s(G2,G);
  forall ( G3 and s(G2,G3) in goalGraph ) {
    if ( exact(G3,G) ) then {
      goalStore = goalStore - G;
      goalTree = goalTree - s(G2,G);
      return goalGraph; }
    if ( plugin(G3,G) ) then {
      goalTree = goalTree - s(G2,G);
      goalTree = goalTree + s(G2,G) + s(G,G3);
      goalTree = goalTree - s(G2,G3); }
    if ( subsume(G3,G) ) then {
      goalTree = goalTree - s(G2,G);
      childNodeInsertion{G,G3}; }
    if ( intersect (G3,G) ) then {
      goalTree = goalTree - s(G2,G);
      iArcResolution{G,G3}; }
  }
  return goalGraph;
}
// en-route resolution of i-arcs
iArcResolution(G1,G2) {
  iG1_G2 = intersectionGoalTemplate(G1,G2);
  goalStore = goalStore + G1 + iG1_G2;
  goalTree = goalTree + s(G1,iG1_G2) + s(G2,iG1_G2);
  discoveryCacheCreation(iG1_G2);
  forall ( G3 and ( s(G1,G3) or s(G2,G3)) in goalGraph and G3 != iG1_G2 ) {
    if ( exact(G3, iG1_G2) ) then {
      goalStore = goalStore - iG1_G2;
      goalTree = goalTree - s(G1,iG1_G2) - s(G2,iG1_G2);
      if ( ! (s(G1,G3) in goalGraph) ) then {
        goalTree = goalTree + s(G1,G3); }
      if ( ! (s(G2,G3) in goalGraph) ) then {
        goalTree = goalTree + s(G2,G3); } }
    if ( plugin(G3, iG1_G2) ) then {
      goalTree = goalTree - s(G1,G3) - s(G2,G3);
      goalTree = goalTree + s(iG1_G2,G3); }
    if ( subsume(G3, iG1_G2) ) then {
      if ( s(G1,G3) in goalGraph ) then {
        goalTree = goalTree - s(G1,G3);

```

```

        goalTree = goalTree + s(iG1_G2,G3); }
    if ( s(G2,G3) in goalGraph ) then {
        goalTree = goalTree - s(G2,G3);
        goalTree = goalTree + s(iG1_G2,G3); } }
    if ( intersect (G3, iG1_G2) ) then
        iArcResolution(iG1_G2, G3);
}
return goalGraph();
}
// discovery cache creation (= Web service discovery for a goal template)
discoveryCacheCreation(G) {
    if ( position(G) = child ) then
        childNodeDiscovery(G);
    else {
        rootNodeDiscovery(G);
        forall ( G2 and (s(G,G2) in goalGraph) ) {
            if ( ( d(G2,W) = (exact or plugin) ) and d(G,W) ) in discoverycache ) then
                discoverycache = discoverycache - d(G,W);
        }
        if ( discoveryCache(G) = {} ) then remove(G);
    }
    return discoverycache;
}
// discovery for G if it is a child node in an existing goal tree
childNodeDiscovery(G){
    forall ( G2 and subsume(G2,G) ) {
        forall ( W and subsume(G2,W) ) {
            matchmakingUsability(G,W);
            if (! d = disjoint ) then
                discoverycache = discoverycache + d(G,W);
        }
        forall ( W and intersect(G2,W) ) {
            if ( plugin(G,W) ) then d = plugin;
            if ( intersect (G,W) ) then d = intersect;
            discoverycache = discoverycache + d(G,W);
        }
    }
    return discoverycache;
}
// discovery for G if it is a root node of an existing goal tree
rootNodeDiscovery(G){
    forall ( G2 and subsume(G,G2) ) {
        forall ( W and exact(G2,W) ) {
            d = subsume;
            discoverycache = discoverycache + d(G,W);
        }
        forall ( W and subsume(G2,W) ) {
            d = subsume;
            discoverycache = discoverycache + d(G,W);
        }
        forall ( W and plugin(G2,W) ) {
            plugin, subsume := boolean;
            d = intersect;
            if ( plugin(G,W) ) then {
                plugin = true;
                d = plugin; }
            if ( subsume(G,W) ) then {
                subsume = true;
                d = subsume; }
            if ( (plugin = true) and (subsume = true)) then {
                d = exact; }
            discoverycache = discoverycache + d(G,W);
        }
        forall ( W and intersect(G2,W) ) {
            if ( subsume(G,W) ) then d = plugin;

```

[illegible]

```

position(goaltemplate) := (root | child);
singleRoot(G) := boolean;
lowestChildWithSingleParent(G) := boolean;
// main
update(G1,G2) {
    if ( exact(G1,G2) ) then {
        goalStore = goalStore - G1;
        goalStore = goalStore + G2;
    }
    if ( position(G1) = root and (s(G1,G3) in goalGraph) and ! (s(G4,G3) in goalGraph) )
    then singleRoot(G1) = true;
    else singleRoot(G1) = false;
    if ( position(G1) = child and ! (s(G1,G3) in goalGraph) and (s(G4,G3) in goalGraph) and ! (s(G5,G3) in goalGraph) )
    then lowestChildWithSingleParent(G1) = true;
    else lowestChildWithSingleParent(G1) = false;
    if ( (singleRoot(G1) and plugin(G1,G2) ) or (lowestChildWithSingleParent(G1) and subsume(G1,G2)) ) then {
        position(G2) = position(G1);
        goalStore = goalStore - G1;
        goalStore = goalStore + G2;
        discoveryCacheCreation(G2);
    }
    else {
        remove(G1);
        insert(G2);
    }
}
return sdcGraph;
}

```

%%
 % insertion of a new Web service

%%

```

insert(W) {
    forall ( G and position(G) = root ) {
        matchmakingUsability(G,W);
        if (! d = disjoint ) then {
            discoverycache = discoverycache + d(G,W);
            childNodeWSInsertion(G,W);
        }
    }
    return discoverycache;
}
//
childNodeWSInsertion(G,W) {
    if ( (d(G,W) = exact) or (d(G,W) = plugin) or (d(G,W) = disjoint) ) then return discoverycache;
    else {
        forall ( G2 and (s(G,G2) in goalGraph) ) {
            if ( d(G,W) = subsume ) then {
                matchmakingUsability(G2,W);
                if (! d = disjoint ) then
                    discoverycache = discoverycache + d(G2,W);
            }
            if ( d(G,W) = intersect ) then {
                if ( plugin(G2,W) ) then d = plugin;
                if ( intersect(G2,W) ) then d = intersect;
                discoverycache = discoverycache + d(G2,W);
            }
        }
        childNodeWSInsertion(G2,W);
    }
}
return discoverycache;
}

```

%%

% removal of a Web service


```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// function declarations
discoveryCache(W) := {d(goaltemplate,W)}
// main
remove(W) {
  discoveryCache(W) = {d(G,W)};
  discoverycache = discoverycache - discoveryCache(W);
  return discoverycache;
}
// optional removal of goal templates
removeGAfterWSDeletion(W) {
  discoveryCache(W) = {d(G,W)};
  forall ( G and (d(G,W) in discoveryCache(W)) ) {
    remove(G); }
  return sdcGraph;
}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% update of a Web service
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
update(W1,W2) {
  if ( plugin(W1,W2) ) then {
    forall ( G and position(G) = root ) {
      if ( (d(G,W1) = exact) or (d(G,W1) = plugin) ) then {
        forall ( d(G,W1) in discoverycache ) {
          discoverycache = discoverycache + d(G,W2);
          discoverycache = discoverycache - d(G,W1);
          return discoverycache;
        } } } }
    else {
      remove(W1);
      insert (W2);
    }
  }
}

```
