University of Groningen
Faculty of mathematics and natural sciences
Computing Science
Advisor: Marco Aiello
Co-advisor: Alex Telea

RugCo: composing Web services from large repositories

Nico VAN BENTHEM
<nicovanbenthem@gmail.com>

December 1, 2009

Abstract

In a service oriented architecture, loosely coupled software components are orchestrated as service in order to implement a working information system. How services are orchestrated is decided by developers and is also referred to as service composition.

To stimulate research in the field of automatic service compositioning, an annual competition was started in 2005, called the Web Services Challenge (WSC). The RugCo system is the entry from the University of Groningen to the 2008 edition of the WSC.

RugCo automatically composes Web services from a repository based on the semantics of a domain ontology and a user composition query. Using a beam search algorithm with straight-forward ordering heuristics, the system was able to find compositions for all challenge queries well within the time limit. In addition, RugCo was recognized for its flexible architecture based on modifiability and integratebility.

This thesis decribes the RugCo approach and analyzes its results compared to other participating systems as a base for future participation and contribution to editions of the WSC.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Service-Oriented Architectures (SOA) are becoming more popular and software engineers are spending their time increasingly on developing services and making them interact [1]. The process of making different services cooperate to achieve some user goal is usually referred to as service composition. Currently, most SOA's are based on an XML based technology known as Web services [2]. Various services are considered as the building blocks of the software architecture and developers decide how to orchestrate them in order to have a working information system. The more automation one can bring to the process, the more efficient the production of SOA oriented software would be.

To stimulate research in the field of automatic service compositioning, the IEEE Conference on Electronic Commerce (CEC) has started an annual competition back in 2005 around Web service composition, the Web Services Challenge (WSC). Participants of the WSC are asked to create software that will create a Web service composition to fulfill a query, using a provided set of available Web services and a semantic annotation of of their in- and output elements.

The University of Groningen participated in the 2008 edition (WSC'08) with a team of four people: master student Nico van Benthem, bachelor student Jaap Bresser, PhD student Elie el Khoury, under the supervision of Marco Aiello from the Computer Science department. The team was awarded in both prize categories: first place for their system architecture and first runner up for the performance of their system.

This thesis describes the RugCo system which is the entry from the University of Groningen to the WSC'08 . The RugCo system automatically composes Web services from repository based on the semantics of a domain ontology and a user composition query.

1.1 Background

The main building blocks for the WSC'08 are Web services, OWL ontologies, semantic annotation using MECE and the Business Execution Process Language (BPEL).

A Web service can be described as a reusable software component which can be invoked over network like the Internet. An example of a Web service is an application that takes care of booking a room at a specific hotel or a service that books a flight at a specific airline company. Web services can be published using the Web Service Description Language (WSDL) which is a standard containing all details on how to technically invoke the service in a machine processable format (XML) [3]. Publishing WSDL allows efficient reuse of Web services and combining them for fast development of SOA applications.

By default WSDL does not include semantics in the description of Web services. Two services can have similar technical descriptions while the result of their invocation can be totally different. Such ambiguities can be resolved by giving semantic meaning to the inputs and outputs of a service, which is essential for the automation of service discovery and composition. One of the technologies available is the Mediation Contract Extension (MECE) for Web services [4]. MECE allows semantic annotation of XSD message elements inside WSDL documents linking service parameters to a concept of a knowledge domain.

Ontologies are formal representations of knowledge in a certain domain defined by a set of concepts and relationships between them which enables reasoning about properties of the domain. The Web Ontology Language (OWL) is a family of knowledge representation languages based on a RDF/XML syntax [5].

Finally, the Business Process Execution Language (BPEL) is a standard for specifying interactions with Web services [6]. BPEL is an orchestration language which centralizes control by exchanging information solely through loosely coupled Web services. Services within the process are unaware of the process responsible for their invocation and can therefor can be easily interchanged or reused. A BPEL process itself can also be published as a Web service.

1.2 The Web Service Challenge 2008

The main focus of this years edition of the challenge is on the semantic composition of Web services. Services available for composition are given semantic meaning by annotation using MECE and OWL. In addition, the BPEL solution format of WSC'08 allows participants to return compositions which include parallel and alternative invocation paths.

Given a repository of semantic Web services and an OWL ontology, users should be able to submit requests to the participating composition software for service composition. A composition request is expressed by a service description of a non existing service. By using the same semantic annotation the request describes the required semantic outputs of the composition and which semantic inputs are available for the invocation of the services in the composition.

Participating systems are evaluated in two categories. First the performance of each system is

compared to other entries, based on i) the time needed to return any composition ii) the minimal number of services in any of the compositions returned and iii) the use of parallel invocation to minimize the execution path length of the composition. The second evaluation category is the architecture of the system based on quality attributes like modularity, scalability, flexibility etc.

1.3 Related work

The Web Service Challenge is related to the Semantic Web Service Challenge (SWS)[1] which also uses semantics for the automation of composition and discovery for Web Services. Unlike WSC, the SWS challenge allows participants to provide additional semantic annotations of the WSDL in order to solve the composition problem. In addition, the SWS challenge focuses on the productivity of the developer and not the speed of the composition software. SWS provides a platform for researchers and industry to show what their Web service discovery and composition technology can do.

The Semantic Services Selection (S3) contest [2] is also related to the WSC. S3 evaluates the retrieval performance of semantic Web service matchmakers commonly used for selecting relevant semantic Web services in any application setting. WSC is complimentary to S3 because service discovery is a rather small part of WSC whereas service composition is untouched by the latter contest.

1.4 Contribution and thesis organization

This thesis presents the RugCo approach which combines an efficient search algorithm with an flexible implementation architecture. RugCo uses a beam search algorithm to optimize composition size while reducing memory requirements. In addition, the RugCo system provides the option to visualize compositions using an open XML standard for graph representation and a compatible graph editor tool.

The second chapter of this thesis describes the details of the WSC'08 rules and formats. The third chapter translates the WSC'08 rules into a formal definition of the composition problem. In the fourth chapter, the RugCo approach for service composition is presented. The fifth chapter explains the implementation details of the RugCo system. The sixth chapter presents the results of the RugCo system at the challenge and discusses its performance by comparing its strong and weak points with approaches from other entries to the challenge. The final chapter summarizes the findings of this thesis and gives recommendations for future implementations of RugCo and

[1] Details on the SWS challenge can be found at http://sws-challenge.org/, checked December 1, 2009

[2] Details on the S3 can be found at http://www-ags.dfki.uni-sb.de/ klusch/s3/, checked December 1, 2009

suggests possible focuses for future editions of the Web Service Challenge.

Chapter 2

Background

Given an ontology, a repository of web services, a set of semantic annotations and a composition query, participating systems of the Web Service Challenge of 2008 (WSC'08) are required to automatically create service compositions which follow the semantic evaluation of input and output parameters of their services.

This chapter starts with describing the structure of the ontologies used as a base for the semantic annotation of service parameters. After a brief description of the service repository, the chapter handles the semantic annotation of service input and output messages elements. The final section describes the solution format of the challenge.

2.1 The ontology

To enable reasoning about the semantics of service input and output paramaters, the WSC'08 makes use of the Web Ontology Language (OWL) [5]. The XML based language contains an extensive vocabulary for defining relations between semantic concepts and their semantic properties, but the challenge limits its use to the definition of classes, subclass relations and semantic individuals as shown in figure 2.1.

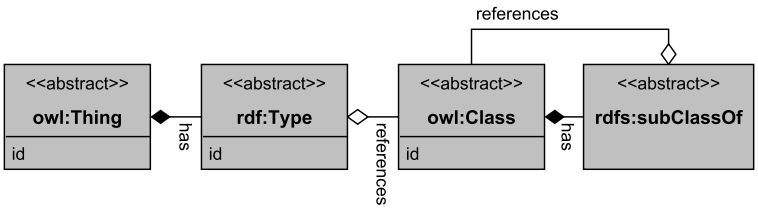


Figure 2.1: *Object model of OWL subset*

Each OWL class corresponds to a semantic concept and the subclass relations corresponds to a generalization of one concept by another concept. Semantic individuals represent instances or members of a class and the subsumption relation between their classes also holds for their

members.

For the remainder of this thesis, the term concept is used to refer to the underlying meaning of an OWL class.

2.2 The service repository

Each challenge set provides a repository of virtual web services, available for service compositions. The information about the fysical location of the service and the syntax of its input and output parameters is specified using the Web Service Description Language (WSDL) [3].

The WSDL file also contains an XML Schema Definition Language (XSD) [7] section in which the structures for service input and output parameters are described.

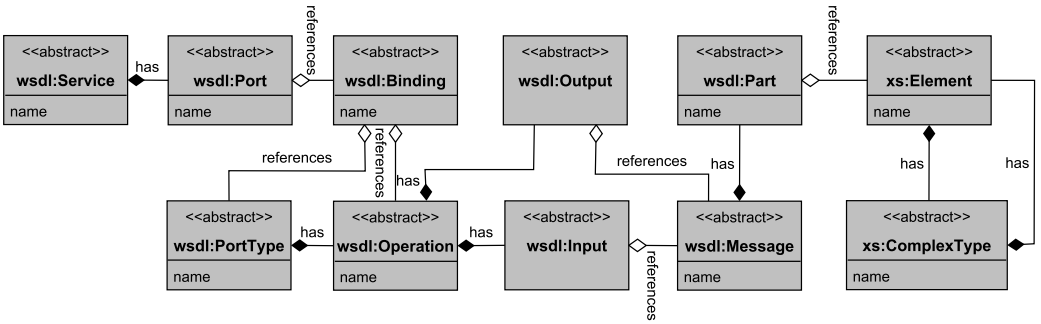


Figure 2.2: *Object model of WSDL service definition with XS elements*

All services of the challenge are synthetic and share the same simplified WSDL structure. Figure 2.2 shows the object model of a single WSC'08 service definition.

The service element contains a single port element which refers to a specific binding. Binding elements contain message format and protocol details for the port type element it refers to, mapping the abstract operation element to a concrete format.

The port type element contains a single request-response operation element consisting of a single input and a single output element. Both the operation input and the operation output element refer to a input and an output message element respectively. The message elements contain one or more message parts each refers to an schema element.

All schema element are either a simple type or a complex type where complex type elements may consist of other simple and/or complex type elements.

2.2.1 Semantic annotation

The Mediator Contract Extension (MECE) [4] for WSDL makes it possible to add semantics to the service messages structures. An MECE annotation links a message element to a semantic individual in the ontology, giving meaning to the element within its message context. The semantic information enables reasoning about service parameters as a base for semantic matching between an output element of one service and the input element of another.

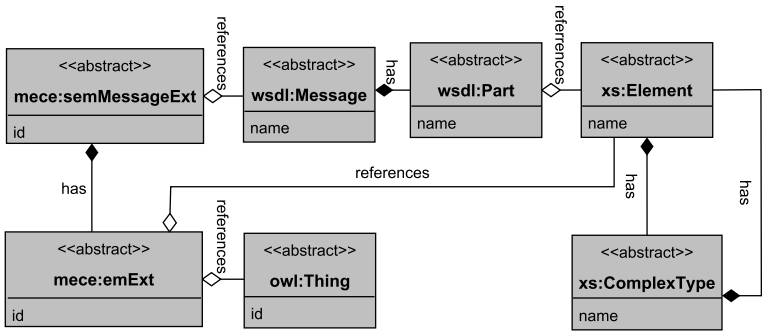


Figure 2.3: *Object model of MECE message annotation*

Figure 2.3 shows the object model of XML schema for the MECE extension of WSDL which consists of two elements. A message extension element refers to any input or output message of the service repository and contains one or more semantic extension. The semantic extension element refers to element within the annotated message linking them to a semantic individual of the OWL ontology.

For WSC'08 , only simple type elements within the message structure are annotated, so the structure of complex elements is actually ignored.

2.3 Solution format

A service composition is an executable process in which each step involves using message elements from previous steps to invoke the next service in the process. This orchestraction of services can be done using the Business Process Execution Language (BPEL) [6]. The WSC'08 solution format is a subset of BPEL simplifying the evaluation process. Using BPEL, participating systems are able to describe a composition with alternative and/or parallel sequences of service invocations.

Figure 2.4 shows the object model of the WSC'08 solution format. Each composition consist of a single process element. Within the process element, the main sequence defines the requirement to receive the composition query input concepts before executing one of the alternative

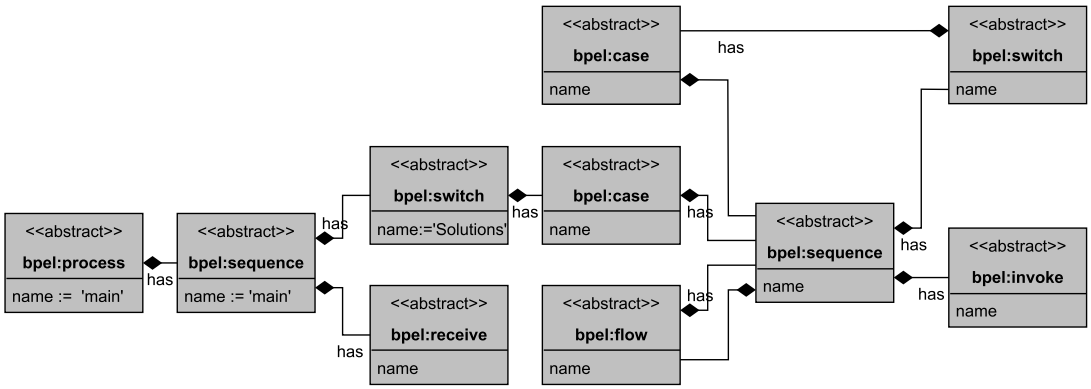


Figure 2.4: *Object of WSC'08 BPEL solution structure*

solutions in the main switch element. Each solution case element contains a sequence of invoke, flow and/or switch elements. An invoke element refers to a web service to be invoked and two or more sequences within a flow element indicate that these sequences can be executed in parallel. Finally, a switch element contains two or more alternative sequences.

Chapter 3

A formal look at the WSC'08

In this chapter, the description of the WSC'08 is translated into a formal definition of the composition problem. Based on the semantic annotion of its message elements, the chapter starts defining services as sets of input concepts and output concepts. Next, the chapter describes the properties of the subsumption relation between semantic concepts as a base for the matching of service parameters. The third section gives a formal definition of a service composition and discusses properties like composition size and execution length. At the end, the chapter concludes these sections into a complete formal definition of the 2008 web service challenge as a base for the RuGCo approach in the next chapter.

3.1 Web service

The matching of service parameters is based their semantics, rather than the structure of the messages and their elements. Through semantic annotation, each simple element within the message structure, whether or not part of a complex element, refers to a semantic individual in the ontology. A semantic individual in turn is a member of an ontology class, linking the simple element to a semantic concept. As a result, any service message m within the WSC'08 context is reduced a set of concepts $m \subseteq \mathbb{C}$, and any web service to a set input and output concepts.

Definition 3.1 (Service)**.** *A* service $s \in S$ *is a pair of input message* s^{in} *and output message* s^{out} *with* $s^{in}, s^{out} \subseteq \mathbb{C}$ *and service repository* S*:*

$$s = \left\langle s^{in}, s^{out} \right\rangle$$

For example, service $s_3 = \langle \{c_7\}, \{c_5, c_6\} \rangle$ requires concept c_7 as input for invocation and returns semantic concepts c_5 and c_6 .

Similarly, a composition request $q = \left\langle q^{in}, q^{out} \right\rangle$ has an input message $q^{in} \subseteq \mathbb{C}$ and an output message $q^{out} \subseteq \mathbb{C}$ containing the provided and required concepts of the challenge respectively.

For example, composition query $q = \langle \{c_1, c_2\}, \{c_3, c_4, c_5\} \rangle$ is a request for a composition

that provides output concepts c_1, c_2 and c_3. Query input concepts c_1 and c_2 are available for the invocation of the composition's services.

3.2 Parameter matching

The input and output concepts of the composition query and services can be matched when the set of semantic properties of the input concept is a subset of the set of semantic properties of the output concept. In other words, concept c_a subsumes concept c_b when concept c_a is equal to concept c_b or when concept c_a is a specialization of concept c_b.

In the OWL ontology, the subsumption relation between two concepts is defined by the subclass property between corresponding classes in the taxonomy.

Definition 3.2 (SubclassOf)**.**

$$\text{subclassOf}(A, B) \equiv \textit{The set of semantic properies of class} A \textit{ is a subset of the set of semantic properties of class } B$$

Figure 3.1 shows an example food taxonomy, each class representing a concept of the ontology. An apple is a subclass of fruit, which means that each member of the apple class is also a member of the fruit class.

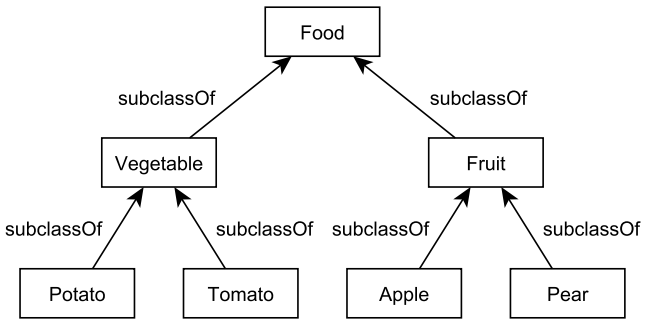


Figure 3.1: *Example taxonomy*

When class A represents concept c_A and class B represents concept c_B, a subclass relation between class A and B directly translates to the subsumption relation between concept c_A and concept c_B.

Definition 3.3 (Subsumes)**.** *Concept* c_A subsumes *concept* c_B *when either* c_A *is a subclass of* c_A *or when there exists a concept* c_S *such that* c_A *is a subclass of* c_S *and* c_S *subsumes* c_B *with*

$c_A, c_B, c_S \in \mathbb{C}$*:*

$$\text{subsumes}(c_A, c_B) = \{\text{subclassOf}(A, B) \ \vee \ \{\exists c_S \in \mathbb{C} : \text{subclassOf}(A, S) \ \wedge \ \text{subsumes}(c_S, c_B)\}\}$$

For example, based the taxonomy of figure 3.1, concept c_{apple} subsumes concepts c_{fruit} and c_{food}. In constract, the proposition $\text{subsumes}(c_{apple}, c_{vegetable})$ is false.

When looking for services to subsume an unmatched service input or query output concept, it is necessary be able to determine all concepts that are subsumed by a set of output concepts.

Definition 3.4 (Subsuming)**.** *The set of all concepts* subsuming *a concept in the set of concepts C are the subsuming concepts of C with $C \subseteq \mathbb{C}$:*

$$\text{subsuming}(C) = \bigcup_{c \in C} \{c_s \in \mathbb{C} \mid \text{subsumes}(c_s, c)\}$$

Let C be a set of concepts $\{c_{potato}, c_{tomato}\}$. The set of all subsuming concepts of C equals $\{c_{potato}, c_{tomato}, c_{vegetable}, c_{food}\}$.

Note that for any $c \in \mathbb{C}$ the concept c subsumes all of its own semantic properties and therefor $\{c\} \in \text{subsuming}(\{c\})$.

3.3 The composition

The focus of WSC'08 lies on the order in which web services may be invoked. Before a web service can be invoked, each of its input concepts must be subsumed by a concept provided by the composition query or by an output concept of a previously invoked service.

For some service pairs this dependency defines the order in which the services may be invoked with respect to each other. Other pairs may have no such dependency and the order is therefor undefined. In other words, the composition defines a partial order between participating services.

Definition 3.5 (Composition)**.** *A web service* composition $W = \langle S, < \rangle$ *is a partial ordered set of services $S \subseteq \mathbb{S}$ in which the binary relation $<$ defines the requirement of invoking service s_l before s_2 for one or multiple pairs of services $(s_1, s_2) \in S$.*

For example, when service $s_2 \in S$ depends on one or more output concepts of service $s_1 \in S$ then $s_1 < s_2$. Pairs for which no such relation exists (neither $s_1 < s_2$ or $s_2 < s_1$) can be invoked independently from each other.

For services $s_1, s_2, s_3 \in S$, the strict partial order relation $<$ has the following properties:

- $\neg(s_1 < s_1)$ (irreflexivity: a service cannot depend on its own output);

- if $s_1 < s_2$ then $\neg(s_2 < s_1)$ (antisymmetry: a service s_1 cannot depend on the output of a service s_2 when s_2 depends on output of s_1);

- if $s_1 < s_2$ and $s_2 < s_3$ then $s_1 < s_3$ (transitivity: if service s_2 depends on the output of s_1 and s_3 depends on output of s_2, then indirectly s_3 depends on the output of s_1);

3.3.1 Directed Acyclic Graph

The partial order relation $<$ of a composition W can be visualized by a directed acyclic graph (dag) $G_W = (S, E)$ in which there is a directed edge $(s_1 \rightarrow s_2) \in E$ between two services s_1 and s_2 exactly when $s_1 < s_2$. Graph G_W is acyclic by definition since any path from a service to itself would contradict either the irreflexivity or antisymmetry property of $<$.

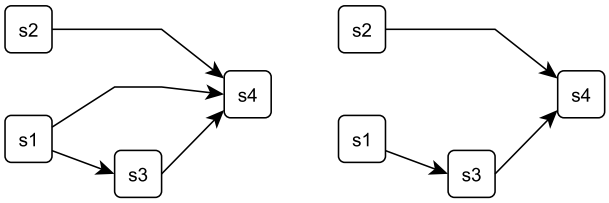


Figure 3.2: *Example: (a) composition graph G_W and (b) its transitive reduction G^*_W*

Figure 3.2(a) shows a dag $G_W = (S, E)$ consisting of five services $s_1 \ldots s_5 \in S$ with multiple dependencies between them. For example, service s_3 requires at least one output concept of service s_1, which means that s_3 may only be invoked after the successful execution of s_1. Similarly, service s_4 can only be invoked after service s_2 and s_3 have finished their execution.

In the composition dag, source nodes are services without predecessors and have no incoming edges. In figure 3.2(b) both service s_1 and s_2 are sources nodes and represent services for which each input concept is subsumed by one of the query input concepts.

In contrast, service s_4 has no services depending on its output. Nodes which have no outgoing edges are sink nodes and represent services that provide at least one concept subsuming a composition query output concept. The latter is based on the assumption that the composition does not include any redundant services.

Transitive reduction

The purpose of the directed acyclic graph representation is to record the order in which services can be executed. With respect to this purpose, graph G_W may contain edges that are redundant because they are already implied by the transitive property of the partial order $<$. For example, the edge $(s_1 \rightarrow s_4)$ implies that that service s_1 has to be invoked before service s_4. This order is also implied by the combination of $(s_1 \rightarrow s_3)$ and $(s_3 \rightarrow s_4)$.

Definition 3.6 (Transitive reduction)**.** *The* transitive reduction *of G_W is a graph G^*_W with such that: i) G^*_W is a subset of G_W ii) For every path between any two nodes in G_W there is also a path in G^*_W iii) G_W is minimal*

The transitive reduction of G_W is displayed in 3.2(b). For the remainder of this thesis the transitive reduction of the dag will be used to visualize service compositions.

3.3.2 Valid compositions

A composition is executable when all input concepts of each service in the composition are subsumed by either a query input concept or an output concept of a preceding service.

Definition 3.7 (Predeceding services)**.** *The* predeceding services *of service s are the services on which s depends on for the subsumption of one or multiple of its input concepts with $W = \langle S, < \rangle$ and $s \in S$:*

$$\text{predecessors}(W, s) = \{s_p \in S \mid s_p < s\}$$

In the composition of figure 3.2(b) the predecessors of service s_4 are $\{s_1, s_2, s_3\}$.

Definition 3.8 (Available concepts)**.** *The* concepts available *for the invocation of service s is the union of the subsuming concepts of the query input message q_{in} with the subsuming concepts of the output message of each preceding service of s in W with $W = \langle S, < \rangle$ and $s \in S$:*

$$\text{available}(W, s) = \text{subsuming}(q^{in}) \cup \bigcup_{s_p \in \text{predecessors}(W,s)} \text{subsuming}(s_p^{out})$$

For example, for the composition in figure 3.2(b) the concepts available for invoking service s_3 equals $\text{subsuming}(q^{in}) \cup \text{subsuming}(s_1^{out})$.

When all services in the composition have been invoked succesfully, the subsuming concepts of each service output message are available to subsume the query output concepts.

Definition 3.9 (Unsatisfied concepts)**.** *The set of* unsatisfied concepts *of a composition W is the union of: i) All query output concepts not subsumed by a service output concept in the composition ii) All service input concepts not subsumed by a concept in the set available concepts, with*

$W = \langle S, < \rangle$, $s \in S$ and composition query q:

$$\text{unsatisfied}(W) = \left(q^{out} \setminus \bigcup_{s \in S} \text{subsuming}(s^{out}) \right) \cup \left(\bigcup_{s \in S} \left(s^{in} \setminus \text{available}(s) \right) \right)$$

When the composition has no unsatisfied concepts it is a valid composition satisfying the composition query:

$$\text{unsatisfied}(W) = \emptyset$$

3.3.3 Parallel paths

In the composition dag, paths represent sequences of services that need to be invoked in a particular order. The longest path between any two services is called the critical path of the dag. The 2008 challenge solution format enables the specification of parallel paths, reducing the minimal response time of the composition to the sum of the response times of the services on the critical path. Parallel paths exist when there are pairs of services in W without dependency relation $<$.

Definition 3.10 (Execution length)**.** *The* execution length *of composition W is the number of services on the longest path between any two services in the composition graph G_W with $W = \langle S, < \rangle$.*

Since all challenge services are considered to have equal response times, the minimal response time of a composition equals the execution length of the composition.

In composition G^*_W displayed in figure 3.2(b), there is no partial order relation between service $s2$ and services s_1 and s_3. Therefor, service sequences $[s_2]$ and $[s_1, s_3]$ are parallel invocation paths in the composition and the longest path in the composition equals $[s_1, s_3, s_4]$.

3.4 WSC'08 challenge

For each composition query $q \notin \mathbb{S}$, service repository $\mathbb{S}$ and ontology classes $\mathbb{C}$ multiple solutions may exist:

$$\text{solutions}(\mathbb{S}, \mathbb{C}, q) = \{ W \in \mathbb{W} \mid \text{unsatisfied}(w) = \emptyset \}$$

Definition 3.11 (Challenge goal)**.** *The* goal of the challenge *is to find the set of solutions containing both the composition with the minimal number of services and the composition with the*

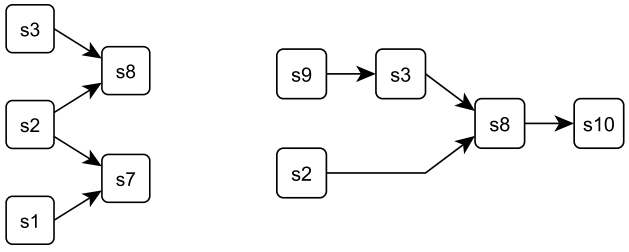
minimal height:

$$\begin{aligned}\text{goal}(\mathbb{S},\mathbb{C},q) = \{W \in \text{solutions}(\mathbb{S},\mathbb{C},q) \wedge W = \langle S, < \rangle \mid \\ \forall W_x \in \text{solutions}(\mathbb{S},\mathbb{C},q) \ \wedge \ W_x = \langle S_x, < \rangle : \\ \text{length}(W) \leq \text{length}(W_x) \ \vee \ |S| \leq |S_x|\}\end{aligned}$$

Let W_1 and W_2 be two alternatives for composition G_W displayed in figure 3.2(b) so $\text{solutions}(\mathbb{S},\mathbb{C},q) = \{W, W_1, W_2\}$. The corresponding composition dags G_{W_1} and G_{W_2} are displayed in figures 3.3(a) and 3.3(b) respectively. The execution lengths of and the number of services in the these compositions are:

$$\begin{aligned}\text{length}(G_W) = 3 \quad , \quad |W| = 4 \\ \text{length}(G_{W_1}) = 2 \quad , \quad |W_1| = 5 \\ \text{length}(G_{W_2}) = 4 \quad , \quad |W_2| = 5\end{aligned}$$

Since composition W contains the least number of services and composition W_1 has the shortest execution length, the goal of the challenge equals $\text{goal}(\mathbb{S},\mathbb{C},q) = \{W, W_1\}$.



(a) (b)

Figure 3.3: *Alternative composition dags a) G_{W_1} and b) G_{W_2}*

Chapter 4

RugCo Approach

This chapter describes the RugCo heuristic based approach for finding challenge solutions . The chapter starts with the introduction of a running example, to illustrate the algorithms and heuristics introducted in this chapter. Secondly, the chapter describes the topologic sort of a directed acyclic graph to represent compositions as a simple sequence of services. In the third section, the search space of the challenge is defined as a base for tree based searching algorithms. Next, the chapter explains the beam search algorithm as an optimized version of best-first to reduce memory requirements, together with the order heuristics used. Then the search algorithm as used for the WSC'08 is presented. The final section considers the correctness and complexity of the algorithm.

4.1 Running example

The running example is based on the formal definition of the challenge presented in the previous chapter and presents a challenge set consisting of an ontology $\mathbb{C}$, a service repository $\mathbb{S}$ and a composition query q.

Figure 4.1 shows ontology $\mathbb{C}$ which contains a taxonomy of thirtheen concepts. The subsumes relation between two concepts is visualized by the dashed generalization arrow. For example, concept c_1 is subsumed by c_5 and c_6.

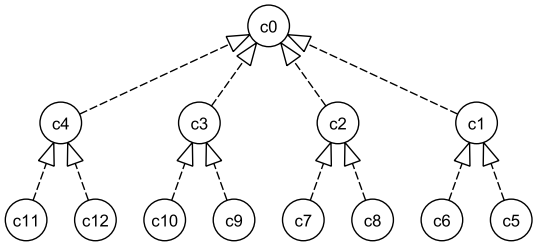


Figure 4.1: *Running example: ontology* $\mathbb{C}$

Repository $\mathbb{S}$ contains six services and is displayed in figure 4.2. The annotation of service messages with semantic individuals from the ontology is represented by linking each service directly to the concepts of the individuals. As a result each service in the repository is displayed with one or more input and one or more output concepts. For example, service s_4 requires input subsuming concepts c_6 and c_9 to provide concept c_{11}.

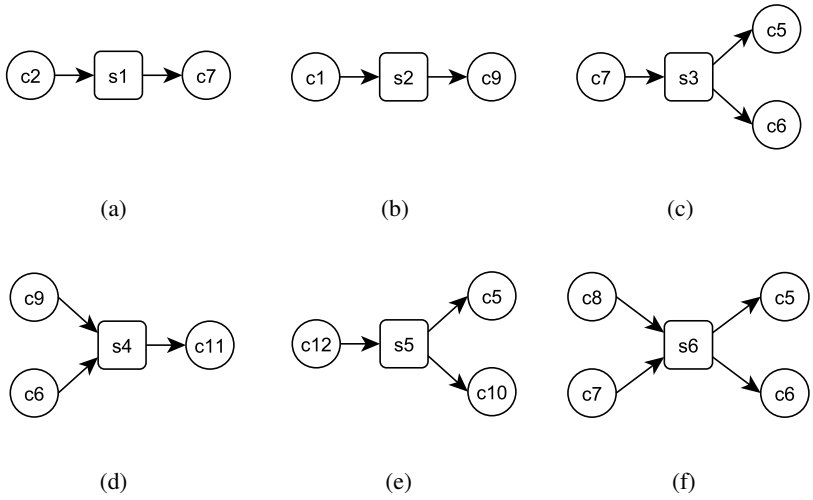


Figure 4.2: *Running example: service repository* $\mathbb{S} = \{s_1, s_2, s_3, s_4, s_5, s_6\}$

In figure 4.3, the composition query q is displayed which has two input concepts and three output concepts. The query calls for a composition that returns the set of concepts that subsume each of the query output concepts c_3, c_4 and c_5. To satisfy service inputs of the composition, concepts c_1, c_2 are available.

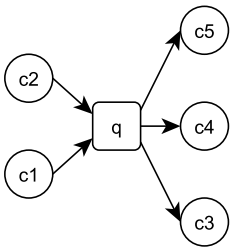


Figure 4.3: *Running example: composition query* q

Figure 3.2(a) in the previous chapter shows composition W which is a solution for composition query q based on ontology $\mathbb{C}$, service repository $\mathbb{S}$. The dag in figure 4.4 shows the same composition but includes the mappings between matched input and output concepts of its services based on the subsumes relation. Concepts c_1 and c_2 are query input concepts and are

mapped directly to the input concepts of service s_1 and s_2 respectively. The output concepts c_9 of service s_2 and c_7 of service s_1 subsume query output concepts c_3 and c_4. The third and final query output concept c_5 is provided by service s_3 directly.

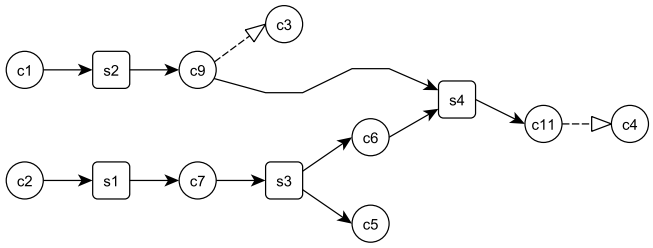


Figure 4.4: *Running example: composition dag G_W with parameter mappings*

4.2 Topological sort

Each composition can be executed as a single sequence in which each service is invoked after the previous service in the sequence has returned its output. This sequence is not unique when one or multiple pairs of services exist in the composition for which no such partial order is defined.

Definition 4.1 (Topological sort)**.** *A* topologic sort T_W *of composition* $W = \langle S, < \rangle$ *is a linear ordering of services in which each service* $s \in S$ *is listed after all services* $s_p \in predecessors(s)$ *it depends on.*

For example, for the execution of G_W in figure 3.2(b) it is irrelevant whether service s_2 or s_3 is invoked first, just as long as they are both finished before service s_4 is invoked. Figure 4.5 shows the two corresponding topological orderings of G_W.

Figure 4.5: *Running example: Two topological orderings of dag G_W*

4.3 Tree based search

The RugCo approach uses a tree based search algorithm to find compositions that satisfy the challenge set. The order in which each node in the search tree is visited depends on the specific

search algorithm that is used.

In the search tree, each node T corresponds to a topological sort of a services. The root of the tree T_r, is an empty list with $\text{unsatisfied}(T_r) = q^{out}$. Node T_g is a leaf node with $\text{unsatisfied}(T_g) = \emptyset$ and is the goal of our search. All other leaf nodes are nodes that cannot be expanded. The algorithm terminates as soon as the goal node is reached or when there are no more nodes left to expand.

4.3.1 Node expansion

When a search node T is visited and it is not identified as a goal node, the node has unsatisfied concepts. When expanding the node, one or more of these concepts are eliminated by adding a new service to the composition. For each child node a different service is introduced.

Definition 4.2 (Providing services)**.** *The set of* providing services *for a concept c is the set of services having one or more output concepts that subsume concept c with $c \in \mathbb{C}$:*

$$\text{providers}(c) = \left\{ s_p \in \mathbb{S} \mid c \in \text{subsuming}(s_p^{out}) \right\}$$

Any node T that has an unsatisfied concept c_u with $\text{providers}(c_u) \setminus T = \emptyset$ cannot be on the path to a goal node because this concept will never be satisfied and therefor cannot be expanded. To further reduce the branching factor of the search tree, this idea is extended for search nodes for which providers for all unsatisfied concepts exist. Instead of extending T with the complete set of providers for each of its unsatisfied concepts, the algorithm determines which concept c_u has the least number of providers and only extends the node with these concepts.

Definition 4.3 (Promising services)**.** *The set of* promising services *for a search node T is the smallest set of providing services for any unsatisfied concept $c_u \in \text{unsatisfied}(T)$ not already in T:*

$$\begin{aligned} \text{promising}(T) = \\ \{ s_p \in \text{providers}(c_u) \setminus T \mid \forall c_x \in \text{unsatisfied}(T) : \\ \mid \text{providers}(c_u) \setminus T \mid \leq \mid \text{providers}(c_x) \setminus T \mid \} \end{aligned}$$

Consider the root node $T_r = \emptyset$ of the running example. The set of unsatified concepts is equal to the set of query output concepts: $\text{unsatisfied}(T_r) = \{c_3, c_4, c_5\}$. The corresponding sets of providing services for these unsatisfied concepts are $\text{providers}(c_3) = \{s_2, s_5\}$, $\text{providers}(c_4) = \{s_4\}$ and $\text{providers}(c_5) = \{s_3, s_5\}$. Since there is only one provider for concept c_3, service s_4 has to be part of the composition. Therefor the set of promising services for T_r equals $\{s_4\}$.

Algorithm 1 below shows how the expansions for search node T are generated in which the function $\text{expand}(T, s_p)$ contains the logics of adding the promising service s_p to composition T.

Algorithm 1: $E = \text{expansions}(T)$

Input: Search node T
Output: A set E with expansions of T

```
1  begin
2      E ⟵ ∅
3      foreach s_p ∈ promising(T) do
4          T' ⟵ expand(T, s_p)
5          add T' to E;
6      return E
7  end
```

Adding a promising service to a composition eliminates unsatisfied concepts but can also introduce new unsatisfied concepts because the input concepts of the new service must now also be satisfied by either the query input concepts or output concepts of existing services in the composition.

Algorithm 2: $T' = \text{expand}(T, s_p)$

Input: node T, promising service s_p
Output: expansion T' of T with s_p
Data: composition query q

```
1  begin
2      T' ⟵ T
3      newEliminated ⟵ unsatisfied[T] ∩ acquired({s_p^out})
4      i ⟵ 0
5      dependencyFound ⟵ false
6      while not dependencyFound and i < size(T) do
7          s ⟵ T[i]
8          if s^in ∩ newEliminated ≠ ∅ then
9              dependencyFound ⟵ true
10         else
11             i++
12     insert service s_p at position i in T'
13     return T'
14 end
```

Algorithm 2 shows the algorithm for expanding a composition with a promising service. To preserve the topological ordering of services and to mininimize the new unsatisfied concepts, the promising service is inserted into the sequence in front of the first service having an unsatisfied input concept eliminated by the output of the new service. This maximises the number of services the newly introduced service can depend on for the satisfaction of its own inputs.

Let $T = [s_2, s_4]$ be a search node in the search tree of the running example with $\mathrm{unsatisfied}(T) = \{c_5, c_6\}$ and $s_p = s_3$. The set of new eliminated concepts now equals $\{c_5, c_6\} \cap \{c_5, c_6, c_1, c_0\} = \{c_5, c_6\}$. Since $s_4^{in} \cap C_E = \{c_6\}$ service s_3 will be added in front of service s_4 and therefor $T' = \mathrm{expand}(T, s_3) = [s_2, s_3, s_4]$.

4.4 Beam Search

The RugCo system uses an optimized version of a the best-first search. At each search step, the algorithm determines which unexplored node in the search tree is the most promising one and expands it by generating its child nodes. Typically the selection of the next candidate node is based on a priority queue in which nodes are ordered by some heuristic ordering function.

Because the total number of search nodes to be explored can rapidly increase with the expansion of nodes, memory often is a bottleneck when the search space is significantly large. The beam search used by the RugCo system, reduces this memory requirement by limiting the number of nodes that are kept in the priority queue. At each level of the search tree, only the b most promising nodes are saved.

Because there is a possibility that the nodes on the path to the goal node are not considered to be in the set of b most promising nodes, there is a risk that the algorithm might not find any solutions or may not find the best solution. Preliminary results showed that the beam search did find good solutions fast.

Algorithm 3 shows the beam search used for the challenge. The size of priority queues P_b and P_b' is limited by beamwidth b so only the most promising nodes are kept in the queue. The algorithm starts by inserting the root node T_r into priority queue P_b. At each level of the search, each node T in priority queue P_b is extended. For each child node $T' \in expansions(T)$ the algorithm checks whether it is the goal node T_W. If not, T' is inserted in the next level priority queue. The algorithm finishes as soon as the goal node T_w was found or when there are no nodes left to expand.

Algorithm 3: $T_W = beamSearch(q)$

Input: composition query q and root node T_r
Data: priority queues P_b and P'_b with limited size b
Output: goal node T_W or $null$ when no solution exists

```
1  begin
2      P_b ⟵ P'_b ⟵ ∅
3      insert T_r into P_b
4      while P is not empty do
5          pop T from P_b
6          foreach T' ∈ expansions(T) do
7              if unsatisfied(T') = ∅ then
8                  T_W ⟵ T'
9                  return T_W
10             else insert T' into P'_b
11         P_b ← P'_b
12 end
13 return null
```

4.4.1 Ordering heuristics

The ordering of search nodes is based on heuristics. The heuristic comparator function compares two search nodes to determine which of the two is more likely to be on the path of the goal node.

With the pending deadline of WSC'08 , little time was spend on determining and optimizing appropriate heuristics. Experimenting with these heuristics may reveal additional or more appropriate properties. For the same reason, the execution length of the composition defined in 3.10 is currently not included in the heuristics while minimizing the value of this property is part of the challenge.

The heuristic ordering of search nodes used for the WSC'08 is based on the following two composition properties:

- The number of unsatisfied concepts. The composition having the least number of unsatisfied concepts is more likely to be close to a solution since the tree search algorithm is based on eliminating these concepts.
- The number of acquired concepts. The composition that has acquired the most concepts is closer to the goal node since this composition will more likely have less new unsatisfied concepts when a new service is added.

Algorithm 1 shows the comparator function used by priorities queues P_b and P'_b in algorithm 3.

Algorithm 4: $r = compare(T_1, T_2)$

Input: Tree search nodes T_1 and T_2
Output: $r \in \mathbb{Z}$ with $r <= 0$ when T_1 is more or equal promising compared to T_2 and $r > 0$ when T_2 is more promising

```
1 begin
2   if | unsatisfied(T1) | ≠ | unsatisfied(T2) | then
3     return | unsatisfied(T1) | − | unsatisfied(T2) |
4   else return | acquired(T1) | − | acquired(T2) |
5 end
```

When the number of unsatisfied concepts of search nodes T_1 and T_2 are not equal, the composition with the least number of unsatisfied concepts will be placed before the other in the priority queue. If equal, the nodes are compared based on the number of of acquired concepts.

Let $T_1 = [s_2, s_3, s_4]$ and $T_2 = [s_2, s_6, s_4]$ with $unsatisfied(T_1) = \{c_7\}$ and $unsatisfied(T_2) = \{c_7, c_8\}$. Because $|\{c_7\}| \neq |\{c_7, c_8\}|$ and $|\{c_7\}| < |\{c_7, c_8\}|$ the result of the comparator function will be $r = compare(T_1, T_2) = -1$ and thus T_1 can be considered to be closer to the goal node then T_2.

4.5 Parallel services

After goal node $W = \{S, <\}$ is found, the RugCo system determines which services of the composition can be invoked in parallel before the composition can be exported to a BPEL process.

Definition 4.4 (Service depth)**.** *The* depth of a service s *in composition* W *is the longest path from any node in dag without predecessors to service* s *in composition dag* G_W *with* $W = \langle S, < \rangle$ *and* $s \in S$.

The services are grouped by their depth in the composition. Services in the same group can be invoked in parallel.

For example, the depth of the services in composition W of the running example are:

$$\begin{aligned} \text{depth}(G_W, s_1) &= 0 \\ \text{depth}(G_W, s_2) &= 0 \\ \text{depth}(G_W, s_3) &= 1 \\ \text{depth}(G_W, s_4) &= 2 \end{aligned}$$

Because the depths of services s_1 and s_2 are both 0, these services can be invoked in parallel.

4.6 Correctness and complexity

After each iteration of the while loop in algorithm 3, all candidate compositions in the priority queue have of an equal number of services. The heuristic ordering function determines which of these search nodes are more likely to be the path to the goal node. When beam size b is large enough the goal node will not be purged and therefor the algorithm will return a solution composition which has the least number of services.

Theorem 4.1 (Worst case complexity)
The *worst case complexity* of the RugCo search algorithm equals $\mathrm{O}(b \times |S|)$, with beam width b and service repository S.

Proof. The number of iterations of the while loop in algorithm 3 is limited by the maximum number of services in a composition. In the worst case scenario, the number of nodes in the priority queue exceeds beam width b and all node in the queue are on the path to a composition node containing all services of S. In each of the $|S|$ iterations, all b nodes are expanded. Therefor, the worst case complexity of the RugCo composition algorithm equals the beam width b times the number of services in repository S. □

Chapter 5

Implementation

This chapter describes the details of the RugCo system which was implemented entirely using the Java programming language. The first section gives a high level overview of the main components of the system where the following sections go into more detail of each component. The final section lists the tools used for the development of the RugCo system.

5.1 High level overview

The architecture of the RugCo system is designed to be highly flexible to enable future changes in technology standards, challenge requirements and search strategies.

Figure 5.1 shows a high level view of the RugCo system which is divided into five main components. The 1 components are loosely coupled and the arrows indicate the direction of control. The following list describes the functionality of each main component before going into detail in the following sections:

XML parsers The parsers required to process the WSC'08 input documents to load the ontology and the service definitions into appropriate Java class representations.

Service repository Keeps track of all services available for the composition of services and provides an index for the fast retrieval of service providers for an unsatisfied concept.

Composer The main component of the RugCo system providing the composition search capability. The composer encapsulates all composition logic including the beam search algorithm.

XML writers The document writers required to export a composition to the WSC'08 solution format or to an intermediate graph definition language which can be used to visualize compositions.

Web service interface A wrapper for the RugCo system which enables the composer to be invoked as a web service by the WSC'08 client software.

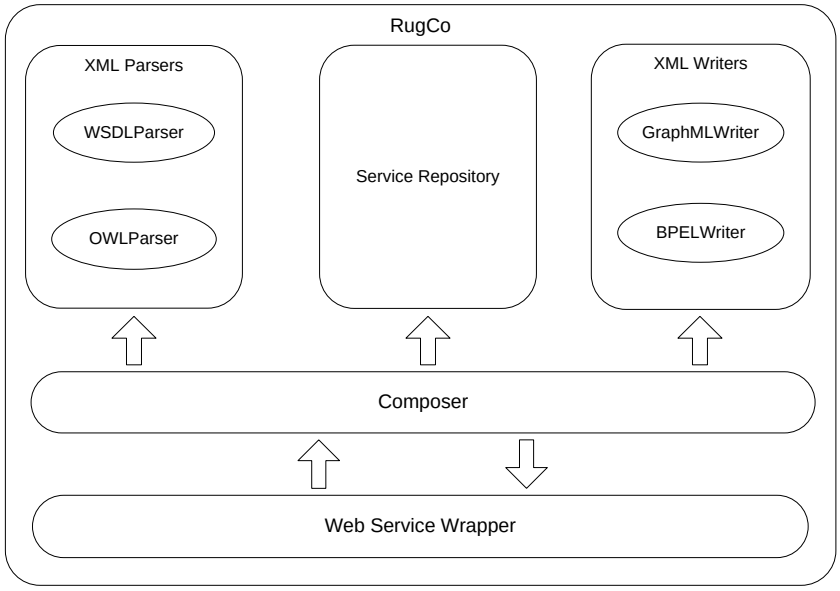


Figure 5.1: *High level overview of RugCo architecture*

5.2 XML parsers

The RugCo system uses SAX[1] for processing XML files, which is a popular parsing API for XML. SAX does not require reading the complete input file into the memory but reads an XML file sequentially and raises an event each time it encounters the start or the end of an XML tag. These events can then be handled by custom methods in which the required actions for processing the specific XML element are implemented.

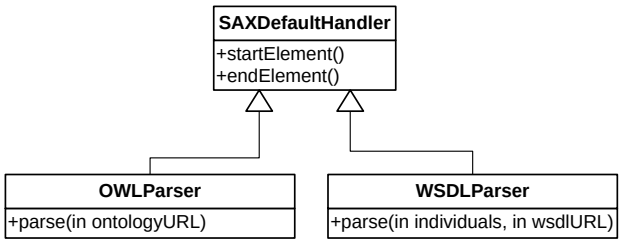


Figure 5.2: *Object model of XML parsers*

Figure 5.2 shows how the OWL parser and the WSDL parser both extend the default SAX handler. Both parsers are described in detail in the following two subsections.

[1]The SAX project can be found at http://www.saxproject.org, verified December 1, 2009

5.2.1 OWL parser

The OWL parser extends the SAX default handler to process the XML elements of concept, the subclass property and individual. When finished, the parser returns a set of individuals, each refering to a concept within the taxonomy.

The concept taxonomy is a tree like structure and validating the subsumption relation between two concepts holds would normally require searching for the first concept in the descendants of the second concept. To disable the need of traversing the taxonomy tree, RugCo uses the nested tree model [8]. Each tree element is labeled with a left and a right value depending on the number of child elements and the left value of its parent element. To determine whether or not a concept subsumes another concept, it now suffices to check whether or not the left and right value of the first concept lies in range of the left and right values of the second concept.

Figure 5.3 below shows the hierarchy of the example ontology of figure 2.1 in a nested sets representation.

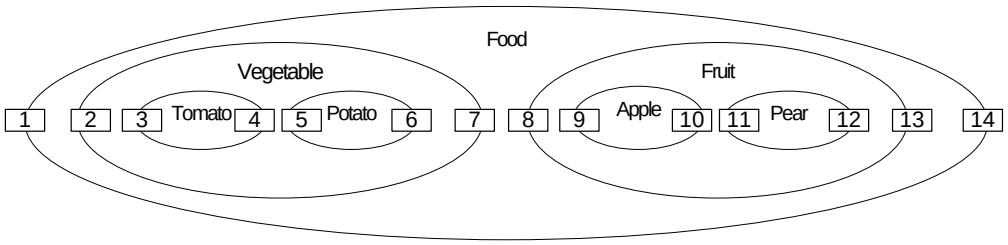


Figure 5.3: *Nested set model of example ontology*

In the example, the concept of *apple* has a left value of 9 and a right value of 10. The concept *food* has a left value of 1 and a right value 14. The subsumption relation between the two concepts holds because $1 \leq 9$ and $10 \leq 14$.

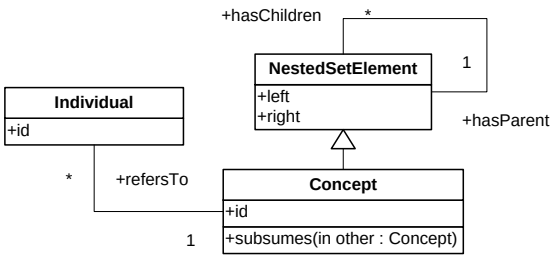


Figure 5.4: *Object model of ontology*

Figure 5.4 shows the object model of the ontology. Each individual is associated with a concept and the concept extends a nested set element. The nested set element has a left and right

value and keeps references to its parent and child elements in the hierarchy.

5.2.2 WSDL parser

The WSDL parser is used to process the service descriptions in the repository as well as the service decription of the composition query. When a WSDL file is offered to the parsed, the set of ontology individuals is included in the call. This enables the parser to map the annotations of the message service elements directly to the concept of the individual and return service objects as sets of input and output concepts.

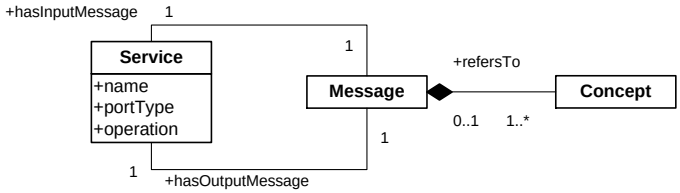


Figure 5.5: *Object model of service*

Figure 5.5 shows the object model of a single service. Each service has a single input and a single output message and each message refers to one or more concepts of the ontology. In addition, the service object holds invocation details through its name, porttype and operation attributes.

5.3 The service repository

The services returned by the WSDL parser are loaded into the service repository. The service repository provides access to all services available for creation compositions. A hashmap is used to provide a constant lookup time for services that provide a certain concept, independent of the number of services in the repository. When a service is loaded into to the repository, the set of its output concepts is extended to include all concepts subsumed by one of its members. Next, the service is added to the set of service providers for each concept in the resulting set.

concept	providers	concept	providers
c_0	$\{s_1, s_2, s_3\}$	c_1	$\{s_3\}$
c_2	$\{s_1\}$	c_3	$\{s_2\}$
c_5	$\{s_3\}$	c_6	$\{s_3\}$
c_7	$\{s_1\}$	c_9	$\{s_2\}$

Table 5.1: *Example index of service repository*

Table 5.1 shows an example index for services $s_1 \ldots s_3$ of the running example displayed in figure 4.2 of the previous chapter. It is easy to see that services s_1, s_2 and s_3 have an output concept that subsumes concept c_0.

5.4 Composer

The composer is the main component containing all composition logic and is the main control unit of the RugCo sytem. Next to the initialization and query methods, the composer has private methods implementing the composition logic of the previous chapter.

The composer makes use of two important subcomponents. A composition component which represents a candidate composition and a bounded priority queue to order the candidate compositions for the beam search algorithm.

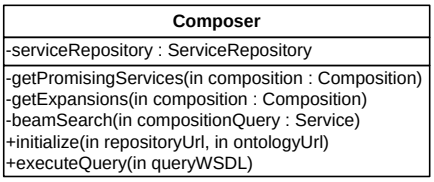


Figure 5.6: *Object model of composer*

Figure 5.4 shows the object model of the composer. Adding a different or additional search algorithm then the current beam search algorithm can be done relatively easy by implementing the new search method in the composer class and reusing the existing components.

The composition component is a simple topological ordered list of composition services and holds the corresponding set of unsatisfied concepts of the candidate solution. The composition component itself contains no composition logic but relies on the composer to add services and determine the unsatisfied concepts.

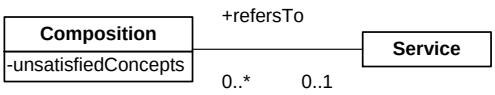


Figure 5.7: *Object model of composition*

Figure 5.7 shows the object model of a composition. Both services and unsatisfied concepts are stored by reference to reduce memory requirements.

5.4.1 Bounded priority queue

The bounded priority queue is an extension of a default priority queue implemented using the fast priority heap algorithm [9]. The worst case complexity of adding an element to priority queue with n elements equals $\mathrm{O}(\log n)$ and the complexity for retrieving the head of the queue is only $\mathrm{O}(1)$. Elements in the queue are compared using a comparator function with determines whether the priority of one element is less, equal or greater than the second element.

RugCo implements the bounded priority queue by extending the default Java priority queue. When an element is added, the number of elements in the queue is compared with the beam width which is the maximum size of the queue. When the queue size exceeds the beam width, the last element of the priority queue is deleted from the queue.

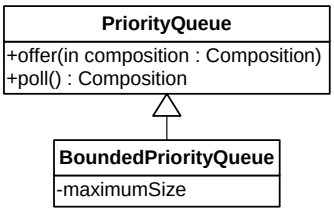


Figure 5.8: *Object model of the bounded priority queue*

Figure 5.8 shows the object model of the bounded priority queue. To compare two composition elements in the queue, RugCo uses the compare function defined in algorithm 1.

5.5 XML Writers

The RugGo system enables the export of composition to two different XML output format. By default, the composition is exported to a BPEL document, which is the solution format of the WSC'08 . In addition, compositions can be exported to GraphML which is an extensive XML language for defining graphs.

5.5.1 BPELWriter

To export the composition to a BPEL document, the BPELWriter needs to determine which services can be invoked in parallel, based on the depth of each service in the composition. Invocation elements of services with equal depth are nested into a single flow element and the order of flow elements is determined by a surrounding sequence element.

Figure 5.9 shows the object model of the BPELWriter which has private methods to create the

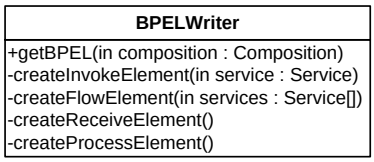


Figure 5.9: *Object model of the BPELWriter*

different types of BPEL elements. Figure 5.10 below shows the result of exporting composition graph G_w of the running example to a BPEL document.

```
<?xml version="1.0" encoding="UTF-8"?>
<bpel:process
  xmlns:bpel="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:service="http://www.ws-challenge.org/WSC08Services/"
  name="WSC08"
  targetNameSpace="http://www.ws-challenge.org/WSC08CompositionSolution/">
  <bpel:sequence name="main">
    <bpel:receive name="receiveQuery" portType="solutionProcess" variable="query"/>
    <bpel:flow>
      <bpel:invoke
        name="service:s1"
        operation="service:s1Operation"
        portType="service:s1PortType"/>
      <bpel:invoke
        name="service:s2"
        operation="service:s2Operation"
        portType="service:s2PortType"/>
    </bpel:flow>
    <bpel:invoke
      name="service:s3"
      operation="service:s3Operation"
      portType="service:s3PortType"/>
    <bpel:invoke
      name="service:s4"
      operation="service:s4Operation"
      portType="service:s4PortType"/>
  </bpel:sequence>
</bpel:process>
```

Figure 5.10: *Example BPEL output of running example composition* G_w

5.5.2 GraphMLWriter

To export to a GraphML document, the GraphMLWriter simply converts the composition nodes and edges of the composition graph to their GraphML element equivalents. The writer does not add any layout information to the output document since compatible graph editors tools exist, capable of visually arranging the nodes and edges of the graph in various layouts.

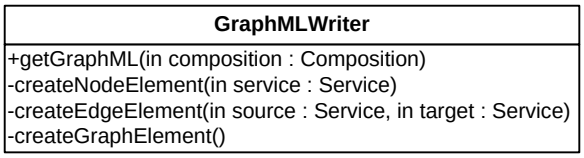


Figure 5.11: *Object model of the GraphMLWriter*

Figure 5.11 shows the object model of the GraphMLWriter which has private methods to create the different types of GraphML elements. Figure 5.12 below shows the result of exporting composition graph G_w of the running example to a GraphML document.

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml
  xmlns="http://graphml.graphdrawing.org/xmlns"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:y="http://www.yworks.com/xml/graphml"
  xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
  http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
  <graph edgedefault="directed" id="mygraph">
    <node id="s1"></node>
    <node id="s2"></node>
    <node id="s3"></node>
    <node id="s4"></node>
    <edge source="s1" target="s2"></edge>
    <edge source="s2" target="s4"></edge>
    <edge source="s3" target="s4"></edge>
  </graph>
</graphml>
```

Figure 5.12: *Example GraphML output of running example composition G_w*

5.6 Web service wrapper

The WSC'08 rules place no restrictions on how the composition systems are be implemented. The only requirement is that participating systems can be deployed as a webservice themselves

through a predefined interface (WSDL) provided by the WSC'08 . In addition, a Java class stub was provided as a ready to deploy service wrapper the composition systems.

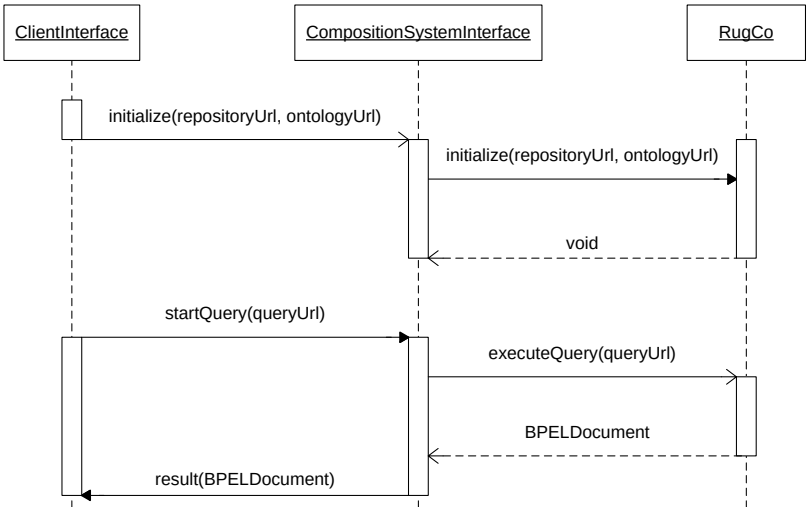


Figure 5.13: *Sequence diagram of interaction between WSC client and RugCo system*

Figure 5.13 shows the interaction of the RugCo system with the WSC'08 client software through the service wrapper. Using the client software the user can select the ontology OWL file and the repository WSDL file of the challenge set to be submitted to the composer system. When the RugCo system finishes initializing the user can select a composition query file and execute the query by submitting it to the RugCo system. As soon RugCo finds a solution the composition is exported to a BPEL document and send back to the client.

5.7 Tools

The following tools were used for the implementation of the RugCo system:

Eclipse An development platform which includes an extensive set of Java development tools. Eclipse is free and can be downloaded from the Eclipse project website[2].

yEd graph editor A free tool to create and edit graphs. yEd can be used to load GraphML files and automically arrange graph nodes and edges in the preferred layout. All graph figures in this thesis are created with this tool. The yED tool can found on the website of yWorks[3].

[2] http://www.eclipse.org
[3] http://www.yworks.com

Subversion (SVN) An open source version control system to maintain current and historical versions of source code and related documents. More information can be found on the SVN project page[4].

TortoiseSVN A popular free client for Subversion which integrates as a shell extension in Windows. The tool is free and can be downloaded from the TortoiseSVN project page[5].

[4] http://subversion.tigris.org
[5] http://tortoisesvn.net

Chapter 6

Results from WSC '08

This chapter describes the results of the participation of the RugCo system to the Web Service Challenge 2008. The challenge was held on site of the 10th IEEE Conference on E-Commerce Technology (CEC' 08) and the 5th IEEE Conference on Enterprise Computing, E-Commerce and E-Services (EEE ' 08) which took place from July 22nd until July 24th, 2008 in Washington, DC.

The first section describes the components used for the performance evaluation of the participating systems and presents the results. Next, the results are discussed and the final section gives a brief overview of other WSC'08 participants.

6.1 Results

Evaluation was done in two categories. First the performance of the participating systems was measured bases on predetermined evaluation rules. The second category was the architectural challenge were the systems were judged based on the overall structure of their implementation regarding modularity, scalability and flexibility.

6.1.1 Evaluation components

The performance evaluation of the composition systems was based on three components:

Composition size For all valid compositions generated by the system, the composition consisting of the least number of services is considered to be the best solution with respect to composition size.

Composition length For all valid compositions generated by the system, the composition with the shortest critical path is considered to be the best solution with respect to parallelization. The critical path in a directed acyclic graph is the longest path from any source node to any sink node.

Execution time The time measured from the moment the system receives the actual composition query until the moment the BPEL description of a valid solution is returned. When no results are returned after a maximum execution time of twenty minutes, the system is terminated and considered to have failed in finding any valid solutions.

Note that when multiple solutions exist, the composition with minimum length is not necessarily the same as the composition involving the least number of services. In this case, the composition system that returns both types of optimal compositions in their solution set is considered better than systems returning just one.

6.1.2 Challenge Sets

Before receiving the actual compositon query, the composition systems were presented with the challenge set containing the ontology, the service descriptions and annotations. Each participating system was evaluated using the same three sets and corresponding composition queries. Table 6.1 shows the main properties of the three sets.

Challenge	Number of services	Number of concepts	Number of individuals
set 1	1041	3135	6162
set 2	1090	3067	6258
set 3	2198	12468	24744

Table 6.1: *Properties of the WSC'08 challenge sets*

The test sets vary in terms of search space size and solution complexity but there exists at least one solution for each composition query. The time needed to parse and optionally index preprocess the challenge set data was not taken into account.

6.1.3 Performance evaluation results

Table 6.2 shows the results of the performance challenge. For each evaluation component participating teams received six points for the best, four points for the second best and two points for the third best score. The RugCo system found a solution for each challenge set and was rewarded second place.

6.2 Discussion

During the preperation for the challenge, the evalution rules were subject to changes. Initially the completeness of the solution was also an evalution component. Participating systems would

Challenge	Evaluation	Thingua University	University of Groningen	Pennysylvania State University	University of Kassel
set 1	size	10	10	10	10
	length	5	5	5	5
	time (ms)	312	219	28078	828
	points	12	18	12	12
set 2	size	20	20	20	21
	length	8	10	8	8
	time (ms)	250	14734	726078	300219
	points	18	10	12	6
set 3	size	46	37	-	-
	length	7	17	-	-
	time (ms)	306	241672	-	-
	points	12	10	0	0
total	**points**	**46**	**38**	**24**	**22**

Table 6.2: *Performance results for WSC'08*

be rewarded for returning the largest set of alternative compositions. Due to lack of a proper evalution method, this evaluation component was eventually ignored. The remaining evalution components are conflicting. Finding the optimal solution in size and/or length will most likely take longer than finding any composition that satisfies the query.

The search strategy of the RugCo system was to find the composition with the least number of services as fast as possible. With respect to the final WSC'08 evaluation rules this is probably not the best approach. The focus on finding the composition with the least number of services is already at cost of the optimal execution time. A better approach might be to focus on finding the composition with the shortest possible length as well, prior to optimizing execution time.

Memory optimization using beam search a large drawback. When the search space is too large, search nodes are pruned and an optimal solution can no longer be guaranteed. For future composition challenges this might impose a big problem, but one should also consider that the challenge is based on artifially generated concepts, services and composition queries. The solution set of third challenge set contains composition with more than thirty-five services which may be much larger than any practical composition in the near future.

The execution time of the RugCo system depends on the beam width used for the search algorithm and the depth of the solution. For the challenge, a fixed beam width was used regardless of the properties of the challenge sets. The execution time grows exponentially with the depth of the solution until the beam width limits the maximum number of nodes in the priority queue. After the beam width is reached the execution time is close to lineair with respect to the solution depth at the cost of pruning potential solution candidates.

6.3 Other WSC'08 approaches

The best performing composition system at the WSC'08 is based on an AND/OR Graph algorithm which finds parallel optimized compositions efficiently [10]. The algorithm constructs a services dependency graph (SDG) which links the input concepts of each service in the repository with each subsuming output concept and its providing service. Mapping concept nodes to OR nodes and service nodes to AND nodes, the algorithm finds a directed sub-graph of the SDR as solution to the query. The SDR efficiently stores dependencies between services in the repository during intialization, while the RugCo approach requires to determine these dependencies on the fly. In addition the current implementation of the RugCo system keeps a topological sort representation of each candidation solution in memory while the memory requirements of the AND/OR graph is strongly related to the number of concepts in the ontology. Finally, the AND/OR graph search optimizes composition length while RugCo optimizes the composition size.

A multi agents approach focusses on scalability and building from open standards [11]. The system extends the RDF triples of the ontology by translating service descriptions and annotations. The RDF triples are then stored in an AllegroGraph database which supports the SPARQL query language for extracting subgraphs from the RDF data. Although the multiagents approach is scalable, the components used require setup making the approach less portable than the Java approach of RugCo.

Similar to the RugCo system, a greedy search approach uses an heuristic function to determine which search node is to be expanded next [12]. Unlike beam search, all known candidate solutions are ordered in a single priority queue and only the most promising node is expanded. With appropriate heuristics the algorithm finds a solution fast but the solution is not guaranteed to be optimal in terms of size and or length. Although this approach will beat the beam search with respect to the time needed to return any solution, the RugCo system can find the size optimized composition when the beam width is large enough and/or the solution depth is limited.

Another participating system transforms the challenge into a reachability problem on a state-transition system [13]. Each state consists of a set of boolean variables for each concept of the ontology indicating whether or not the concept is available. Possible state transisitions are determined by the services of the repository. The resulting problem can be solved an of the shelf boolean satisfiability (SAT) solver [14] to benefit from years of research and development. Unlike beam search this approach currently does not make use of the knowledge of the search space. This makes search performance of the system less than informed search based systems like RugCo.

Chapter 7

Conclusions

This thesis describes the RugCo approach for automatic services composition based on sementic annotation of service parameters and some user request. Using beam search, RugCo is able to find size optimized compositions in $O(|S| \times b)$ complexity with service repository S and beam width b. With a highly modifyable and flexible architecture, the system can be easily altered in order to implement future requirements or alternative search strategies.

7.1 Future work

After expanding a search node, the RugCo system does not check if the expansion is not already in the priority queue. Checking for duplicates or similar compositions may decrease the branching factor in future implementations.

In the search algorithm, topological ordered set of services are used to represent composition candidates while this ordering is not necessarly unique for the composition dag. In future work, the topological should be replaced by a dag representation of the dependencies between composition services.

Nowadays more and more computer systems have multiple processor cores enabling multiple threads running at the same time. The algorithm used can easily be modified enable multithreading. Depending on the number of cores available, the ordering of the priority queues can be done in seperate threads with one or more additional worker threads that get the next node from the queue, expand it and insert the expansions in the next level priority queue.

The beam search algorithm used could be made complete by combining it with an algorithm that finds an suboptimal solution fast and use it to set a dynamic beam with based on the properties of the found solution [15]. The RugCO system could use the found composition to purge all candidates from the priority queue for which the size of the composition and the length of the composition is equal or greater than the size and length of the suboptimal solution.

7.2 Future of Web Service Challenge

The focus of WSC'08 was on based on semantic matching and the use of existing industry standards like OWL for semantic annotation and BPEL as a solution format.

Future editions of might focus on quality of services (QoS) in which non functional attributes of services are also considered. In the current challenge, composition services are only selected based on the output they provided and the input they require. By providing QoS information on services, composition software an also consider quality attributes like the availability or response time of a service or the costs involved to invoke a service from a third party.

Another focus might be to make the gap between the theory of the Web Service Challenge and practical application smaller. If services match semantically, how can a system automatically translate the output parameter structure of one service to an input parameter of another? How can the knowledge and experience of the Web Service Challenge be used to aid developers in their design tasks?

Bibliography

[1] Marco Aiello, Nico van Benthem, and Elie el Khoury, "Visualizing compositions of services from large repositories", in *CECANDEEE '08: Proceedings of the 2008 10th IEEE Conference on E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services*, Washington, DC, USA, 2008, pp. 359–362, IEEE Computer Society.

[2] Michael Papazoglou, *Web Services: Principles and Technology*, Prentice Hall, 1 edition, September 2007.

[3] E. Christensen, F. Curbera, G. Meredith, and Weerawarana S., "Web services description language (wsdl) 1.1", *World Wide Web Consortium*, March 2001.

[4] A. Bansal, M. B. Blake, S. Kona, S. Bleul, T. Weise, and M.C. Jaeger, "Wsc-08: Continuing the web services challenge", in *10th IEEE Conference on E-Commerce Technology and 5th Enterprise Computing, E-Commerce and E-Services. CEC 2008 and EEE 2008*, Washington, D.C., District of Columbia, USA, July 2008, pp. 351–354.

[5] D. L. McGuinness and Frank van Harmelen, "Owl web ontology language overview", *World Wide Web Consortium*, Februari 2004.

[6] "Web services business process execution language version 2.0", *OASIS*, April 2007.

[7] D. C. Fallside and Walmsley P., "Xml schema part 0: Primer second edition", *World Wide Web Consortium*, October 2004.

[8] Joe Celko, *Joe Celko's SQL for Smarties: Advanced SQL Programming*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995.

[9] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson, *Introduction to Algorithms*, McGraw-Hill Higher Education, 1999.

[10] Y. Yan, B. Xu, and Z. Gu, "Automatic service composition using and/or graph", in *10th IEEE Conference on E-Commerce Technology and 5th Enterprise Computing, E-Commerce and E-Services. CEC 2008 and EEE 2008*, Washington, D.C., District of Columbia, USA, July 2008, pp. 335–338.

[11] P. A. Buhler and R. W. Thomas, "Experiences building a standards-based service description repository", in *10th IEEE Conference on E-Commerce Technology and 5th Enterprise Computing, E-Commerce and E-Services. CEC 2008 and EEE 2008*, Washington, D.C., District of Columbia, USA, July 2008, pp. 243–246.

[12] T. Weise, S. Bleul, M. Kirchhoff, and K. Geihs, "Semantic web service composition for service-oriented architectures", in *10th IEEE Conference on E-Commerce Technology and 5th Enterprise Computing, E-Commerce and E-Services. CEC 2008 and EEE 2008*, Washington, D.C., District of Columbia, USA, July 2008, pp. 355–358.

[13] W. Nam, H. Kil, and Lee D., "Type-aware web service composition using boolean satisfiability solver", in *10th IEEE Conference on E-Commerce Technology and 5th Enterprise Computing, E-Commerce and E-Services. CEC 2008 and EEE 2008*, Washington, D.C., District of Columbia, USA, July 2008, pp. 331–334.

[14] Lin Xu, Frank Hutter, Holger Hoos, and Kevin Leyton-Brown, "Satzilla2009: an automatic algorithm portfolio for sat", in *SAT Competition 2009*, 2009.

[15] R. Zhou and E.A. Hansen, "Beam-stack search: Integrating backtracking with beam search", in *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 2005, pp. 90–98.