THE WEB SERVICE CHALLENGE 2009, PARTICIPATION AND LESSONS LEARNED.

AUTHOR: PATRICK RATELBAND, RATELBAND [AT] GMAIL.COM
ADVISOR: MARCO AIELLO, M.AIELLO [AT] RUG.NL
CO-ADVISOR: ALEXANDER LAZOVIK, A.LAZOVIK [AT] RUG.NL
ACADEMIC YEAR 2008-2009

ABSTRACT

During the academic year of 2008 - 2009 I participated in the Web Services Challenge (WSC) 2009 as part of the team from the Rijksuniversiteit Groningen (RuG). The goal of the WSC is to create an algorithm that can solve the web service composition problem in a fast and efficient way.

The web service composition problem consist of creating a chain of services, regarded as black boxes with specified input and output, that will satisfy a requested set of output concepts with given input concepts. When no single service exists that can satisfy such a request, services will have to composed so that the output of one or more services serves as the input of a next service. This chaining of services is called composing and the challenge of the WSC is to solve this NP-hard problem in an as short amount of time as possible.

In this thesis the RuGQoS system, the entry of the RuG in this year's WSC, is described in detail. It uses a priority queue and adds a single service in every iteration while searching for the optimal solution of any given challenge. Algorithms presented by the other participating teams are analyzed and four general approaches for solving the web service composition problem are derived. Using the results from the WSC 2009 guidelines for future web service composition software and recommendations for future Web Service Challenges are proposed.

1. Introduction

With the amount of information available on the internet growing ever more, the need for automation becomes larger. Only a small portion of the internet is available on the surface web, whereas the lion's share of the information and functionality is hidden in the deep web. The deep web is that part of the internet that does not have its information or function available to the user until the user submits data or otherwise interacts with it. A good example would be your favorite search engine: although the main page is available at all times, the power of the search engine only becomes apparent when the user enters a query.

Not only search engines are available for invocation on the internet, many types of services can be accessed through the internet. Any service that offers an interface (machine or human accessible) that is available through the internet for use by third parties is considered a web service. Availability of services through the internet is becoming a well accepted and even an expected feature of many companies, greatly increasing the number of web services available. With this increase in web services, the possibility of not using one, but more than one web service to satisfy a request is desirable. By using information or functionality one service provides in conjunction with another service, a greater functionality can be achieved. This stringing together of services is called web service composition.

Web service composition will play a big role in the future of the internet and is even focal point of Service Oriented Architecture (SOA). SOA attempts to create frameworks that can satisfy requests by using different services in compositions. With this re-use of services, large frameworks can be created quickly and cost-effectively. The problem with composition creation is that the amount of web services is large, and their input and output parameters have no defined relation or markup. These circumstances make the problem of web service composition a complex issue of much interest for the future.

To stimulate research in this field, the IEEE Conference on Electronic Commerce (CEC) has started an annual competition back in 2005 around web service composition, the Web Services Challenge (WSC). The 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 description of their in- and outputs.

I have participated in this year's challenge as part of a team from the Rijksuniversiteit Groningen (RuG). In this thesis, I will explain the details of the WSC '09 and give a detailed overview of the RuGQoS system (1) which has been awarded a second place in the architectural competition and a fourth place in the computational competition. It uses an iterative algorithm that adds services one by one and uses a priority queue to determine the next action. This is an adaption of the software from the RuG from last year's challenge, the RuGCo (2) system, which used a limited beam search to optimize speed while losing some accuracy. The algorithms of the other teams are then analyzed to show there is a different approach that can achieve these guarantees for large data sets. The thesis finished with some guidelines for architectures of future solutions.

In chapter 2 the WSC 09 is discussed in detail. In chapter 3 the solution of the RuG, the RuGQoS system is presented in detail. In chapter 4 the results of the RuGQoS solution are presented along with an analysis of the algorithms used by the other competitors of the WSC 09. Finally in chapter 5 conclusions are drawn from the gathered data and observations.

2. THE WEB SERVICES CHALLENGE

The web service challenge started in 2005 to stimulate research into web service composition, growing and evolving each year it now consists of two challenges, the architectural challenge and the composition challenge. In the architectural challenge, participants are rated based on the architecture of their software solution, on the small presentation they give on this matter and on an interview per team with the jury. The composition challenge consists of running the software package of the teams individually with a common set of problems; teams are awarded points for finding the best solution in the least amount of time.

The WSC is not fully true to life as it is a semantic challenge and the issue of data representation has been omitted. The challenge uses artificially generated concepts and instances; a concept is the description of a piece of information, for instance a phone number. Instances are combinations of concepts in ways that will appear in web services, e.g. an instance of information will out of a phonebook contains the concepts phone number, name and address.

Data can be described and stored in a great number of ways, each with its own advantages and drawbacks, opening up a venue of great debate. However the organizers of the WSC 09 wish to avoid debates on data representation languages like OWL-S (3), WSML (4) and WSDL-S. This focuses the challenge on composition creation and starting from the WSC in 2008, data concepts and their instances are described in a subset of the Web Ontology Language (OWL) (5). OWL is a very powerful tool and will allow for many types of relations between concepts. However, the WSC '09 only uses the super- and subclass relations. An example of such a relationship would be ISBN-10 **is an** ISBN. This super- and subclass relationship will be denoted further in this thesis using the <: binary operator.

Definition 1, subclass operator: $t_1 <: t_2, t_1$ is a subclass of t_2, with $t_1, t_2 \in$ services.

This relation is transitive and reflexive (6).

- transitivity: $t_1 <: t_2, t_2 <: t_3 \Rightarrow t_1 <: t_3$
- reflexivity: $t <: t \mid \forall t$

In the 5th WSC, non-functional attributes have been added to the services. These non-functional attributes consist of the response time and the throughput of a service. Both metrics are artificially generated and they are uncorrelated across services. These metrics are supplied in a file using the WSLA format (7). Response time is given as an integer number, defined as the interval between receipt of the end of a query transmission to the beginning of the transmission of a response to that query. The throughput is also an integer number, defined as the maximum number of successful invocations of the service per minute (8).

In the context of the challenge, a web service can now be described as a four-tupel, combining the four given properties: input, output, response time and throughput (9).

Definition 2, web service:
$s(\{t_{in}\}, \{t_{out}\}, Re, Th)$, with $t_{in}, t_{out} \subseteq$ all concepts and Re, Th are positive numbers.

In this definition, $\{t_{in}\}$ and $\{t_{out}\}$ are the set of input types and the set of output types of the service respectively, Re denotes the response time and Th its throughput.

It is worth noting at this time, that the above definition does not only apply to a single web service, but also a composition of services. A composition of services consists of a chain of services, where (part of) the output of one service is used as (part of) the input of another service. This chain will have required inputs, provided outputs, a total response time and a maximal throughput. With this approach, we can achieve more functionality than with only a single service.

A composition is always an answer to a request. If we wish to formally define how a composition relates to a request, we must first define a request. To define a request, we must define the total concept space $T = \cup_{\forall t}\, t$ as the unison of all known concepts. We may then see a request for a composition as a 2-tupel. $r(I_{in}, I_{out})$, with $I_{in}, I_{out} \subseteq T$. Here I_{in} and I_{out} respectively denote the given input instances and the requested output instances of data concepts.

We can increase the size of this set, without losing semantic equivalence, by applying the previously defined super- and subclass relations recursively on both in- and output instances. $I_{in}^* = \{i_{in}^* | i_{in}^* <:^* i_{in}, i_{in} \in I_{in}\}$ and $I_{out}^* = \{i_{out}^* | i_{out} <:^* i_{out}^*, i_{out} \in I_{out}\}$, resulting in the semantically equivalent request $r(I_{int}^*, I_{out}^*)$.

We see here that we can satisfy a single input from a service by having an instance of any subclass of the required concept. We satisfy a requested output by having that specific output concept or one of its subclasses as output somewhere in our composition. The set of compositions satisfying a given request can now be defined as a two-tupel relating request and composition.

Definition 3, <u>challenge composition relation</u>:

$$(r(I_{in}, I_{out}), \{s\}) = \{s(\{t_{in}\}, \{t_{out}\}, Re, Th) | \{t_{in}\} \subseteq I_{in}^*, \{t_{out}\} \subseteq I_{out}^*\}$$

The set of compositions $\{s\}$ satisfying a given request r, is made up of all compositions that:

- can use the given input concepts or any their super class concepts as input and
- provide all requested output concepts or their subclasses as output.

Finding this complete set is an NP-hard problem (10) and would require searching the entire search space. We therefore limit ourselves to searches with given characteristics. In the WSC '09, this limitation is achieved by only looking for the composition with the lowest response time and the composition with the highest throughput.

We know that when we compose a web service composition, we need to chain services together. This chaining is done by using (part of) the output of one service as (part of) the input of another. If one service depends on the other in such a way, we denote this using the binary operator $\prec$.

Definition 4: <u>depends on operator</u>: $s_1 \prec s_2, s_2$requires at least one instance of the output of s_1.

This operator has several important characteristics: it is irreflexive, anti-symmetric and transitive, imposing a partial ordering on items. (11)

- irreflexivity: $\neg(s_1 \prec s_1)$, a service may not depend on itself.

- anti-symmetry: $s_1 \prec s_2 \Longrightarrow \neg(s_2 \prec s_1)$, a service s_1 cannot depend on the output of a service s_2 if that service s_2 depends on output of s_1. Note here, that this is also true for longer chains. $s_1 \prec s_2 \prec \cdots \prec s_n \Longrightarrow \neg(s_n \prec s_i), i \in (1..n)$.
- transitivity: $s_1 \prec s_2, s_2 \prec s_3 \Longrightarrow s_1 \prec s_3$

Pairs of services that do not have a "depends on" relationship $(\neg(s_1 \prec s_2) \land \neg(s_2 \prec s_1))$, can be executed independently of one another and may thus be eligible for parallel execution in the composition. This is an important feature, as it allows for compositions with lower execution times by exploiting parallelism where possible.

If we map all of the "depends on" relations of a given composition in a graph, we get a directed acyclic graph (DAG). In this graph, a node represents a service and an edge represents a "depends on" relationship. This graph is directed, since the "depends on" relation is ordered, and acyclic as the relation is anti-symmetric. In this graph, we can express the request to which this composition is tailored by two separate special nodes. The first node is the input node, which will is a service with no input, but generates the given input instances of the query as its output. The second node is the output node, requiring all of the requested instances as input. This node will not have any outgoing edges.

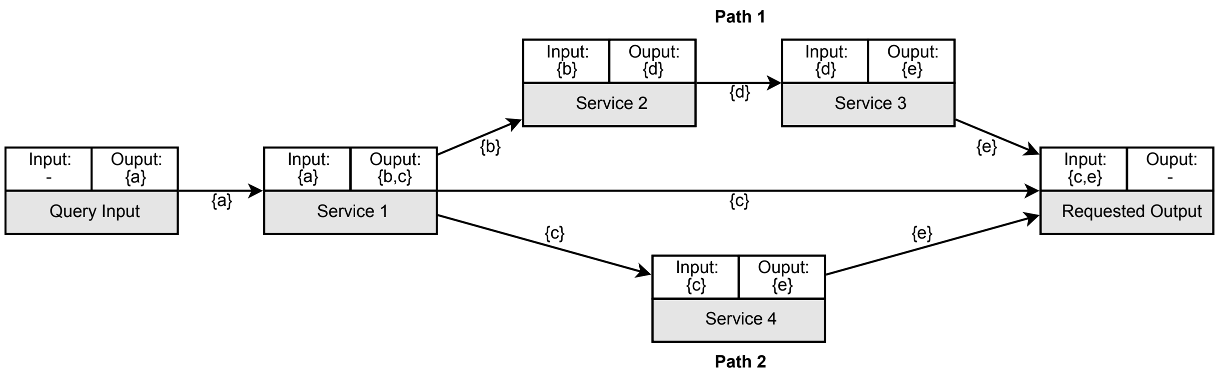


Figure 1: A simple composition.

In the above example composition we can see the input node on the left and the output node on right. Each service is labeled with its input and output concepts. We see that with the given services and request we can form two paths in our composition. Both will lead to a composition that will satisfy our request, but their aggregated attributes might differ as we will show later. We also see here that we can split the graph into three types of building blocks, invocations, sequences and flows.

- An invocation is the main building block of a composition. It is the invocation of a single service, s_1.
- A sequence is the execution of a chain of services that depend on each other for output, but on any other services, $s_1 \prec s_2 \prec \cdots \prec s_n$.
- In a flow, the paths of the graph split into two or more separate paths, which are independent and can be executed in parallel, $s_1 \prec s_2, s_1 \prec s_3, \neg(s_2 \prec s_3 \lor s_3 \prec s_2)$.

The case statement is also present in the challenge solution format. This statement allows for a choice between different paths but this is only used at the top level to indicate different solution possibilities. It can also be used at other points in a composition, but this will not be considered in this thesis as it would only compress the graph and not add functionality or possibilities that could not be expressed by using several different compositions.

If we wish to perform Quality of Service (QoS) aggregation on a composition to obtain its total QoS attributes, we must know how the different building blocks affect the different attributes of our QoS. Within the WSC '09 we concern ourselves with two metrics: the response time and throughput that are given for each individual service. We can aggregate these metrics with the following formulae.

Let $Re(x)$ denote the response time of building block type x, then

- $Re(\text{invocation}) = \text{the response time of the invoked service.}$
- $Re(\text{sequence}) = \sum(\text{response times of all blocks in the sequence})$
- $Re(\text{flow}) = \ max\ (\text{response time of all blocks in the flow})$

Let $Th(x)$ denote the throughput of building block type x, then

- $Th(\text{invocation}) = \text{the throughput of the invoked service.}$
- $Th(\text{sequence}) = \ min\ (\text{the throughput of all blocks in the sequence})$
- $Th(\text{flow}) = \ max\ (\text{the throughput of all blocks in the flow})$

Using these formulae, we can now annotate our previous figure of the composition graph with the QoS attributes at each level.

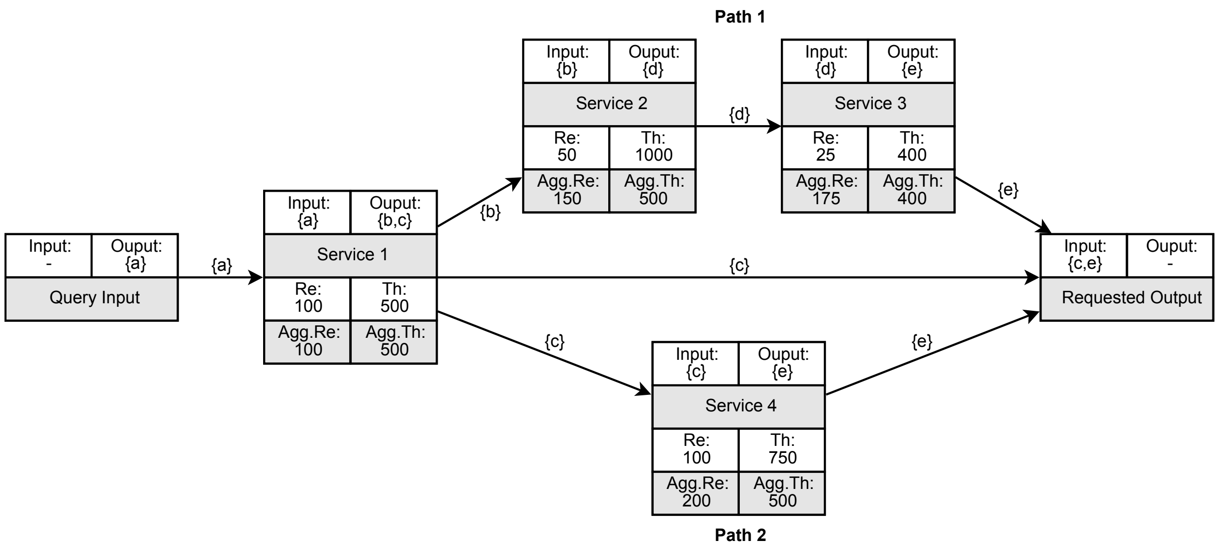


Figure 2: Simple composition with aggregate QoS attributes.

In the above example, which is an extension of our previous example, we build the aggregate QoS attributes from left-to-right. Here, the response time of a service is indicated with Re and its throughput with Th. The aggregate attributes are shown below in the grey areas. We can see that path 1 yields a total response time of 175 ms and a throughput of 400 invocations per second. Path 2 yields a higher response time, 200 ms, but also a higher throughput of 500 invocations per second. We would prefer path 1 if we are looking for the composition with the shortest response time and path 2 if we wish to maximize our throughput.

Note that throughput formula for a flow is counter-intuitive if examined closely. If several services are executed in parallel, one would expect the possible throughput to grow. If this approach were to be used in the challenge, all possible compositions satisfying the given request would have to be checked. This would render the problem NP-hard again and thus the abstraction was made to define the throughput as the maximal throughput of all blocks in a flow.

The challenges for the composition challenge part of the WSC '09 consist of four different files, supplying the search space in three files and the query in one file.

- a WDSL file with all of the services.
- a WSLA file with the QoS attributes of the services.
- a OWL file containing the ontology relating the different concepts.
- a WSDL file with the challenge, a provided and requested set of concepts.

Each query file contains only one query, and each query is solvable. The participants are required to find two solutions, one with the best response time and one with the best throughput. The services with their metrics and the ontology are provided and may be preprocessed; the time taken for preprocessing of this data is not counted towards the processing time of the challenge. The timer on the challenge starts when the query file is transmitted to the participant client and stops as soon as the answer file is received. The returned solution is the most important: if several participants have a solution with equal metrics, their processing times are compared. Points are awarded separately for the composition with the smallest response time and with the highest throughput. First place scores 6 points, second 4 and third place 2. These are added to obtain the final results of the challenge. A solution with better metrics will always score more, even if it has taken longer to create. However, the time limit for solving a query is 5 minutes. Any answer returned after those 5 minutes is disqualified and scores 0 points.

3. The RuGQoS solution

I was part of a team of the RuG that participated in WSC '09. Last year, another team from the RuG had also participated and left a strong legacy by finishing first in the architectural challenge and second in the computational challenge with their RuGCo system. Luckily for us, the similarities between last year's challenge and this year's challenge made it possible to use the award winning architecture from last year again. The big difference between the WSC '08 and the WSC '09 is the addition of non-functional attributes to the services. The objective has changed accordingly: last year the solution that used the fewest services scored highest. This year the solutions with the lowest response time and highest throughput score highest.

Since we were given a solid architecture to work from, we have decided to set ourselves an ambitious goal. We will guarantee that the solution returned by our software was the best solution possible. Since the problem of web service composition is generally NP-hard, our algorithm needs to implement a smart exhaustive search.

One of the first problems have to faced was adapting previous years RuGCo system to accept the new QoS attributes. To solve this, we have created a small WSLA parser; the technical details are discussed in the next chapter. Next we have to look for an algorithm that will guarantee the best answer. We have considered many algorithms, but always came across several main problems.

Most search algorithms require some knowledge of the search space. They need a measure to see if the step they take will take them closer to their goal or further away. The clearest example would be the A-star algorithm, which has this measure as part of its heuristic. In the search for a composition with a given set of services, determining this measure is not possible. While working on a path in the composition, there is no way to know if another path is not better unless one would perform an exhaustive search.

Mapping the entire search space is an exponential problem, both in number of services and in number of concepts (12). Without any relation between the in- and output of service, there is no way to diminish the search space during composition construction.

Searching for a path without the given input instances is not possible. It is not known which instances are given in the request and each of the given instances can break any pre-created path by allowing a service to be invoked earlier in the composition.

Given these issues, we have settled for a simple greedy search algorithm to be able to guarantee finding the optimal solution. The algorithm follows a simple execution loop and can make that guarantee. The downside is a longer execution time and larger memory usage, but in light of our goal and the scoring system of the WSC '09, awarding the best solutions, these drawbacks do not outweigh the advantages.

In the initialization phase, the services with their attributes are read into the repository and the ontology tree is constructed. After that, the system will await the associated query. When this query is received, the loop is initialized. It will continue until either a solution is found, or the search options are exhausted and we can conclude that no solution exists. This loop is executed twice, once while searching for the composition with the lowest response time and once while looking for the highest throughput.

HIGH LEVEL OVERVIEW

The core of the algorithm is a priority queue for (incomplete) compositions with an associated comparator. The comparator governs the ordering of the items in the priority queue and differs when searching for response time or throughput. The input and output nodes discussed in the previous chapter are simulated using a list of satisfied and unsatisfied concepts for the entire composition.

The algorithm searches for a solution in a backwards fashion, adding services in reverse order until no unsatisfied input concepts remain. This limits the search space for each step (12). To start, an empty composition is created. The instances given by the request are added to the known concepts list, and the requested instances are added to the required concepts list of the composition. The composition is now assigned a response time of plus infinity and throughput of minus infinity, and inserted as the only element in the priority queue.

The loop is now started, popping the first element of the queue, if no element remains, no solution exists. The popped composition is checked to see if it is a solution, if not, all its successor compositions are created and inserted back into the queue.

On the next page a high level overview of the RuGQoS greedy search algorithm is given. The input consists of the indexed repository created in the initialization, the provided and required concepts and the desired comparator. The function unsat(c) which is referred to in line 7 and 10 is a function that will retrieve the set of unsatisfied concepts from composition c. The function providers(u) queries the repository and returns the set of services which have u, or one of the sub classes of u as one of their output parameters. The function add(c, s) returns a new composition created by adding service s to composition c. This involves,

- adding the service to the DAG,
- adding edges between the service and services that depend on it,
- updating the available and required concepts and
- updating the response time and throughput of the composition using the rules specified in chapter two.

```
input: indexed repository $repos$, proved instances $prov$, requested concepts $req$,
        composition comparator $comp$
output: optimal composition based on $comp$ or failure if no solution is possible
Initialize priority queue $q$ using $comp$
create initial composition and insert into $q$
while $q$ not empty
    $c \longleftarrow q$
    if unsat$(c) =$ then
        return $c$
    else
        for all concepts $u \in$ unsat$(c)$ do
            for all services $s \in$ providers$(u)$ do
                $q \longleftarrow$ add$(c, s)$
            end for
        end for
    end if
end while
return failure
```

Algorithm 1: RuGQoS high level algorithm overview.

We conclude our high level overview with an example run of the algorithm to create a clear picture of the overall workings before discussing the architecture of the RuGQoS system in detail. In this example, we limit the number of services and concepts to four since the graphical representation would otherwise grow too large. The concept space $T = (a, b, c, d)$, and there is no super- and subclass relationship between these concepts. The provided instance set is $\{a\}$, the required set is $\{d\}$. We only provide a response time for the services as response time and throughput are unrelated, and the example only runs the algorithm for response time.

Service	Input concepts	Output concepts	Response time
1	$\{a\}$	$\{b, c\}$	1
2	$\{b\}$	$\{c, d\}$	2
3	$\{a, c\}$	$\{a, c\}$	3
4	$\{a\}$	$\{d\}$	4

Table 1: Example services.

The first search will use a comparator that will order based on response time of the composition. We begin with the initial composition element in our priority queue and then start our loop.

In the figure on the next page, Sa stands for satisfied concepts, Uns for unsatisfied concepts. Re for response time and T□ for the throughput of that composition. The priority queue (PQ) q can be seen at the top of every iteration, the numbers in it denote the response times of the compositions at that position. Instances generated by services that are not used are not shown in this example. Also, the second run for throughput is omitted, as the same mechanic is used and the answer would be found in iteration two, using only service 4.

We see in the first iteration that we find a successor composition with no unsatisfied concepts remaining. This is a solution to our query, but as we can see, it is not optimal: a solution with a response time of three exists. We thus only conclude we have found the optimal solution if the just popped composition is a solution.

Now that we have a good view of the overall workings of the algorithm, we can dig deeper into the way this solution is implemented in actual code.

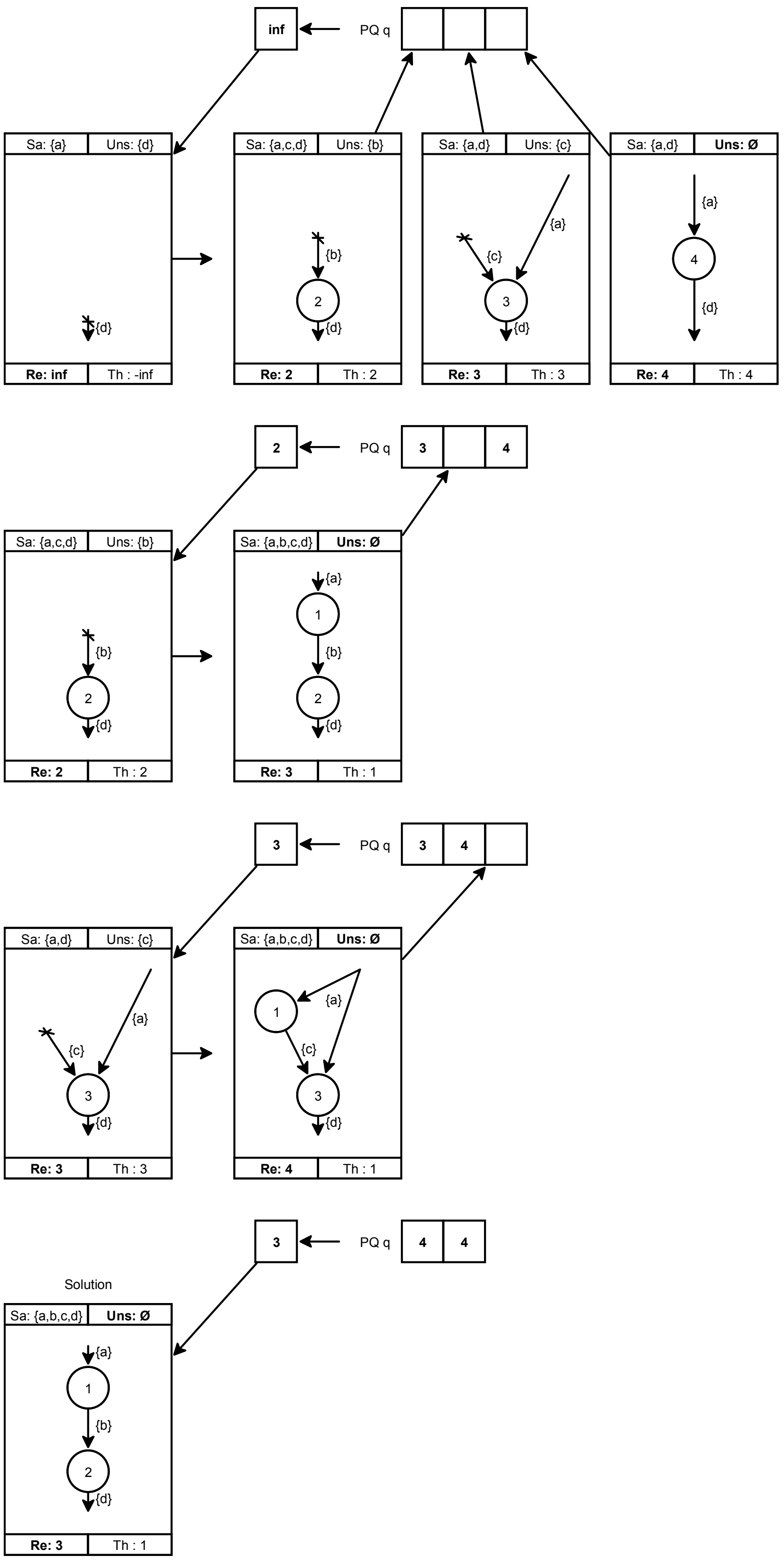


Figure 3: Example run of RuGQoS algorithm.

THE RUGQOS IMPLEMENTATION

As mentioned before, our system is built on the foundations of the architecture from the team from last year. This architecture was build around the Graphical User Interface (GUI) provided by the organizers, and to understand the architecture properly, we must first explain this graphical interface.

WEB SERVICE CHALLENGE GRAPHICAL USER INTERFACE

The WSC organizers have created a small GUI for the participants clients to interface with. To stimulate and facilitate the use of different programming languages, this GUI invokes the participants software using a web service interface. The GUI has three methods the participant's software must implement.

- initialize(wsdlServiceDescriptionsURL, owlTaxonomyURL, wslaSLAgreementsURL)
- startQuery(wsdlQuery, callbackURL)
- stopComposition

The first method, *initialize*, carries the information containing the location of the service descriptions, the ontology and the Service Level Agreements (SLAs). When this method is called, the participant's client can begin preprocessing the data and indicate on the command line when it is ready for the actual query.

The query itself is delivered using the startQuery method. Here the location of the query file is passed and the callback URL where the client will have to send the answer once it has found it.

The last method, stopComposition, is used when the GUI wishes to abort the search of the client.

HIGH LEVEL OVERVIEW

The architecture of the RuGQoS system can be subdivided into five different main components as can be seen in the figure below. Each has its own distinct functionality and they are loosely coupled.

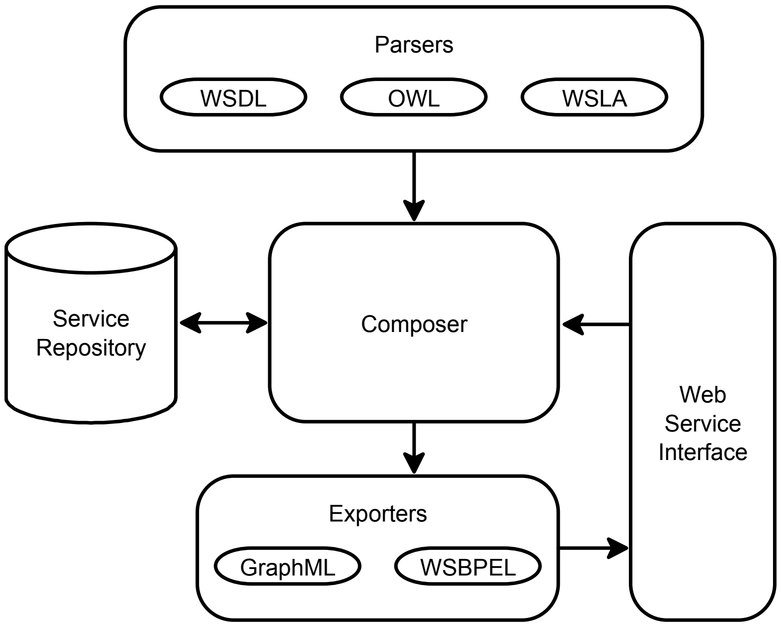


Figure 4: RuGQoS high level overview.

We will go into detail later in this chapter, but the main functionalities of these components are listed by component.

- Parsers: these provide all the required parsing for the system. There are three parsers in this package, a WSDL parser, an OWL parser and a WSLA parser.
- Service Repository: this is where all of the services with their attributes are stored so they can be searched quickly.
- Composer: the main class of the system. This executes the search for a composition.
- Exporters: facilitate the exporting of solutions to different formats. Currently the system can export its solutions into GraphML and WSBPEL format.
- Web Service Interface: this class allows interfacing with any external service that implements its interface; in this case the interface is tailored to fit the GUI of the WSC '09.

PARSERS

The parsers are what allow the RuGQoS system flexibility in the types of documents it can handle. The current parsers are all XML-based parsers, as all supported languages are XML-based in origin. The parsers all use the SAX parser from the SAX project[1], which provides a clean and simple interface to parse any XML based file.

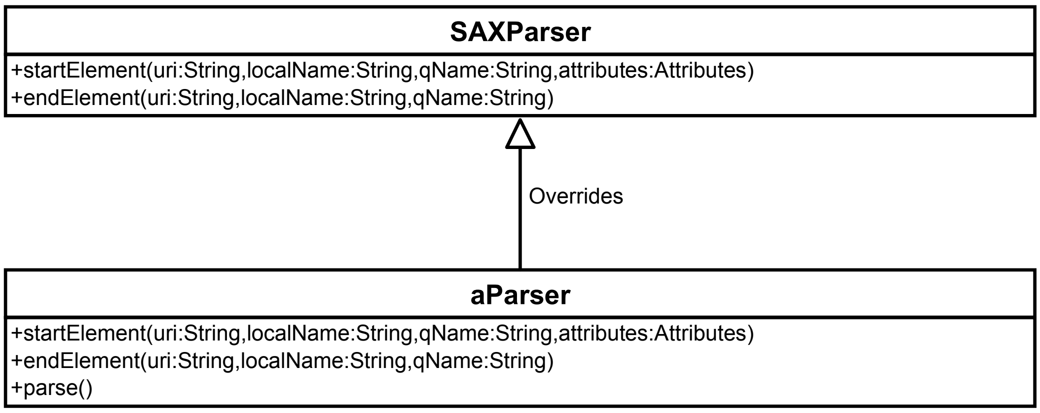

SAXParser
+startElement(uri:String,localName:String,qName:String,attributes:Attributes)
+endElement(uri:String,localName:String,qName:String)
Overrides
aParser
+startElement(uri:String,localName:String,qName:String,attributes:Attributes)
+endElement(uri:String,localName:String,qName:String)
+parse()

Figure 5: Standard SAX implementation.

To parse an XML file with the SAX package, two methods must be overridden. The startElement and endElement methods, which are respectively called when the parser parses a start token and end token. In these methods, the required actions can be described for each individual parser. For further information on the use of SAX, please see their website listed below.

For the ease of viewing, the overridden methods startElement and endElement have been omitted in the following figures, but they are still there in the implementation.

[1] The SAX project can be found at http://www.saxproject.org, verified august 16th 2009.

WSLA Parser

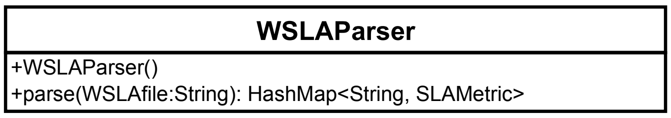


Figure 6: The WSLA parser.

The WSLA parser is the simplest of the parsers. The WSC '09 challenge and test set generator has the convention to name all items corresponding to one service, in such a way that the name of that service is present in the name of the metric being described. For instance:

```
<SLAParameter>AverageResponseTimeService12345</SLAParameter>
<Value>200</Value>
```

This convention allows for faster parsing of the WSLA file, by skipping the normal connection between the WSLA description of a service and the WSDL description of a service. Normally, the connection between these descriptions is made using the different binding names in both documents. See the official description on WSLA for more information on this. In the RuGQoS, as it was tailor-made for the challenge and parsing in not part of the spirit of the challenge, we used the knowledge of the structure of the dataset to look up the name of the services in the name of the metric.

The WSLA parser uses a helper class SLAMetric to represent the two metrics used in the WSC '09, response time and throughput. The parser returns a HashMap<String, SLAMetric> directly linking the names of services to their metrics. The name has to be used here as the WSDL has not been parsed, and instances of the services have not been created yet.

Ontology Parser

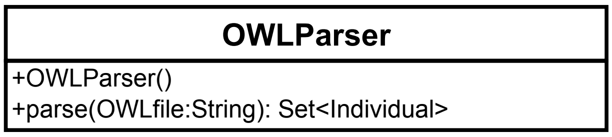


Figure 7: The OWL parser.

The OWL parser, or ontology parser, is responsible for parsing the file with the relations of the concepts. The parser returns a set of individual concepts and in these concepts the child parent relations are already stored. The parser is namespace sensitive and the set of individuals that it produces is namespace sensitive as well.

The OWL parser uses two helper classes, Individual and Concept to represent the different concepts and their instances as they are parsed. The hierarchy of the concepts is implemented using a nested tree set of nested tree set elements[2]. This allows fast checks between specialization and/or generalization of different concepts. The Individuals are directly linked to the concepts they instantiate.

The parser returns a Set<Individual> (a HashSet<Individual> to be more specific) of the instances found in the ontology file, ready to be referenced by the WSDL parser.

[2] See: http://dev.mysql.com/tech-resources/articles/hierarchical-data.html, verified august 15th 2009.

WSDL Parser

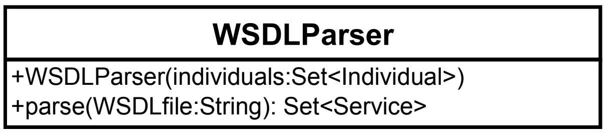

WSDLParser
+WSDLParser(individuals:Set<Individual>)
+parse(WSDLfile:String): Set<Service>

Figure 8: The WSDL parser.

The WSDL parses the service descriptions and uses the data gathered by the ontology parser to create a Set<Service>, containing all services with their in- and output directly linked to the instances of the ontology. This is the most complicated parser, as it is required to keep track of the XML element stack. A service definition can only be fully extracted by using the stack as several attributes of a service are listed on different levels. The Composer later populates the Service Repository using the output set that was created, along with the data from the WSLA parser.

Service Repository

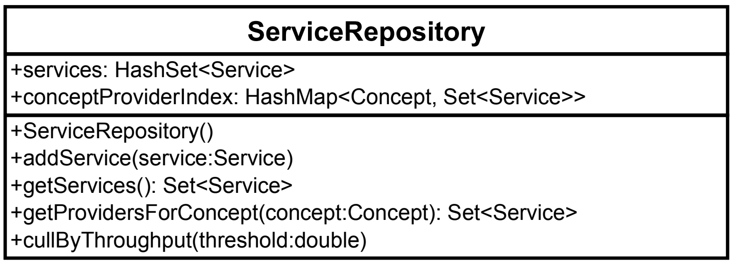
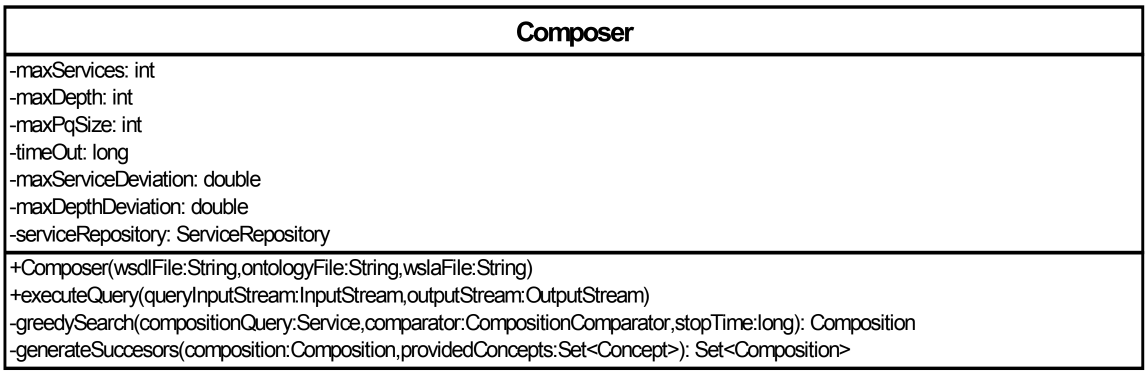

ServiceRepository
+services: HashSet<Service>
+conceptProviderIndex: HashMap<Concept, Set<Service>>
+ServiceRepository()
+addService(service:Service)
+getServices(): Set<Service>
+getProvidersForConcept(concept:Concept): Set<Service>
+cullByThroughput(threshold:double)

Figure 9: The Service Repository.

The service repository is where all services are stored. It is populated by the Composer using the data gathered by the WSDL parser. Each service is inserted into a reverse index, (13) mapping the generated output instances to their services. This reverse mapping is used in the main method of the repository, getProvidersForConcept(c), that will return the set of services that have the concept c as output concept. Since we perform a backwards search in our composition, optimizing this search to $O(1)$ improves performance greatly.

Further optimization is achieved by the cullByThroughput(t) method. This method is called by the Composer after the optimal solution has been found for response time to remove part of the services from the search space. The optimal throughput solution will have a throughput at least as high as the response time solution. With the throughput of a composition is defined as the minimal throughput of any service in the composition, we can remove all services with a lower throughput then the throughput from our response time solution from our search space. This function does exactly that, removing all services with a throughput lower than the given threshold, reducing the search space and speeding up the search for the optimal throughput.

COMPOSER


Composer

-maxServices: int
-maxDepth: int
-maxPqSize: int
-timeOut: long
-maxServiceDeviation: double
-maxDepthDeviation: double
-serviceRepository: ServiceRepository

+Composer(wsdlFile:String,ontologyFile:String,wslaFile:String)
+executeQuery(queryInputStream:InputStream,outputStream:OutputStream)
-greedySearch(compositionQuery:Service,comparator:CompositionComparator,stopTime:long): Composition
-generateSuccesors(composition:Composition,providedConcepts:Set<Concept>): Set<Composition>

Figure 10: The Composer.

The composer is the main class of the RuGQoS system and orchestrates the other classes. It contains all the configuration constants and only two public functions to simplify interfacing.

The configuration constants all govern the behavior of the search and limit its width and depth.

- maxServices, is the maximum number of services a candidate composition may have.
- maxDepth, is the maximum depth of a candidate composition. This is the length of the longest path in the composition.
- maxPqSize, the maximum depth of the priority queue used in the algorithm.
- timeOut, maximum amount of time in milliseconds a search may take.
- maxServiceDeviation, if an optimal solution is found, the maxServices constant is adjusted to the found solutions number of services, plus this deviation factor. Because of the way the challenge generator creates the challenges, the chance a maximum throughput solution having equal length is relatively high. In non-WSC operating circumstances this constant should not be used.
- maxDepthDeviation, equal to maxServiceDeviation, but then for maxDepth.

The Compose method is called to initialize the Composer. When called, this method will parse the three given files and create the Service Repository. This is all done in the initialization phase of the challenge and will not count towards the time of the participants.

```
1   input: WSDL File $wdsl$, OWL File $owl$, WSLA File $wsla$
2   output: indexed repository
3   initialize new ServiceRepository $repos$
4   $individuals \longleftarrow$ owlParser.parse($owl$)
5   $services \longleftarrow$ wsdlParser.parse($wsdl, individuals$)
6   $slas \longleftarrow$ wslaParser.parse($wsla$)
7   for all Services $s \in services$ do
8           addSLA($s, slas$)
9           $repos$.addService($s$)
0   end for
11  return $repos$
```

Algorithm 2: Service repository creation.

In the above algorithm, the method addSLA($s, slas$) on line 8 uses the name of the service s to access the index of the HashMap $slas$ and obtain the SLA's for the requested service. It then adds these metrics to the service. The addService(s) method from the repository adds a single service into the repository. It adds the service to the set of services and also maps all output concepts of this service to this service in the conceptProviderIndex of the repository.

The search algorithm started by invoking the executeQuery method of this class. The method uses the greedySearch method to find actual solutions. Since two solutions are required, executeQuery contains the logic of performing two searches and optimizing between the searches.

```
Input: Query Inputstream $query$
Output: WSBPEL with solutions if any
$stopTime \longleftarrow \text{currentTime()} + timeOut$
$responseTimeSolution \longleftarrow \text{greedySearch}(query, \text{ResponeTimeComparator}, stopTime)$
if $(responseTimeSolution \neq \emptyset)$ then
        Optimize search space and parameters using $responseTimeSolution$.
end if
$throughputSolution \longleftarrow \text{greedySearch}(query, \text{ThroughputComparator}, stopTime)$
return $BPELexporter.export(reponseTimeSolution, throughputSolution)$
```

Algorithm 3: executeSearch overview.

Above we can see the optimization that happens between the two searches. This optiomization consist of removing services with a lower throughput from search space using the cullByThroughput($cutOff$) method and updates the $maxServices$ and $maxDepth$ variables using the deviation constants.

A description of the greedySearch method has been already been given in the previous chapter and is omitted in this section.

Exporters

The exporters have the task of exporting a found solution into a format usable by any third party. In the case of the RuGQoS, two exporters are provided, one exports to WSBPEL for the competition and the other to GraphML to allow visual inspection of the found solutions.

WSBPEL exporter

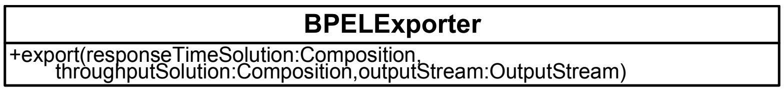
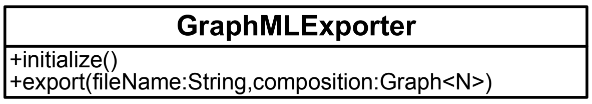

BPELExporter
+export(responseTimeSolution:Composition, throughputSolution:Composition,outputStream:OutputStream)

Figure 11: The WSBPELExporter.

The WSBPEL exporter can export a given composition into a subset of the WSBPEL process language. The subset is compliant with the WSC '09 specifications. Sequences and flows in the composition can be translated on a one-to-one basis. In some cases the resulting translation in WSBPEL might have a higher response time then the one in memory due to the inability to specify "depends on" relations, resulting in problems indicating parallel invocations. The case statement is used only once, to separate the two different result compositions for response time and throughput.

GraphML Exporter

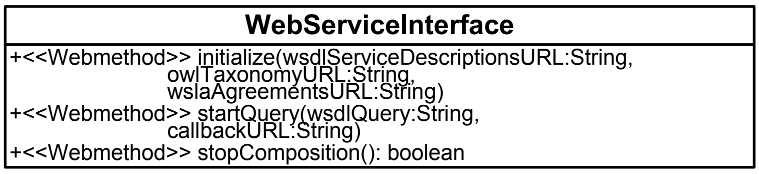


```
GraphMLExporter
+initialize()
+export(fileName:String,composition:Graph<N>)
```

Figure 12: The GraphMLExporter.

The GraphML exporter can export a given composition into a GraphML file. This is an open standard for graph representation. It maps the services to nodes in the graph and the "depends on" relationship as directed arrows. No layout information is added as this is beyond the scope of the project and most programs that can import graphs have methods to create a proper layout of their own.

Web Service Interface



```
WebServiceInterface
+<<Webmethod>> initialize(wsdlServiceDescriptionsURL:String,
               owlTaxonomyURL:String,
               wslaAgreementsURL:String)
+<<Webmethod>> startQuery(wsdlQuery:String,
               callbackURL:String)
+<<Webmethod>> stopComposition(): boolean
```

Figure 13: The WebServiceInterface.

The web service interface allows the RuGQoS system to be accessed like a web service. The methods required for the WSC '09 are all implemented and redirected from here.

Test Results

We have run a series of tests on our software to evaluate its performance. The tests consisted of test sets with 100.000 concepts and a solution depth of 30 created with the challenge generator provided by the WSC '09 organization. The generator guarantees a solution at the indicated depth, but more may be available due to random addition of services. We varied the amount of services in the tests from 2.000 to 20.000 with increments of 2.000 and performed every test 5 times on different sets.

In the first graph we have a look at the preprocessing times needed for each data set.

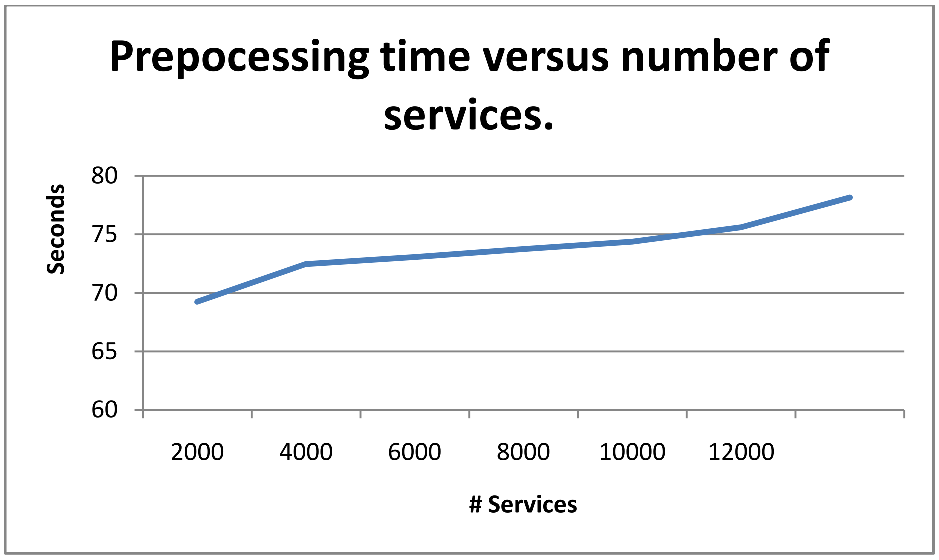


Figure 14: Preprocessing times.

We immediately see that we do not have data for 16.000, 18.000 and 20.000 services. This is because the system was not able to solve these data sets, we have therefore omitted their preprocessing times as well.

The preprocessing times grow in an almost linear fashion. This is as we expected as parsing the ontology takes up the most time and the services parsing is a linear process.

The processing times of the challenges that have been solved shows an almost linear correlation between the amount of time required and the size of the sets.

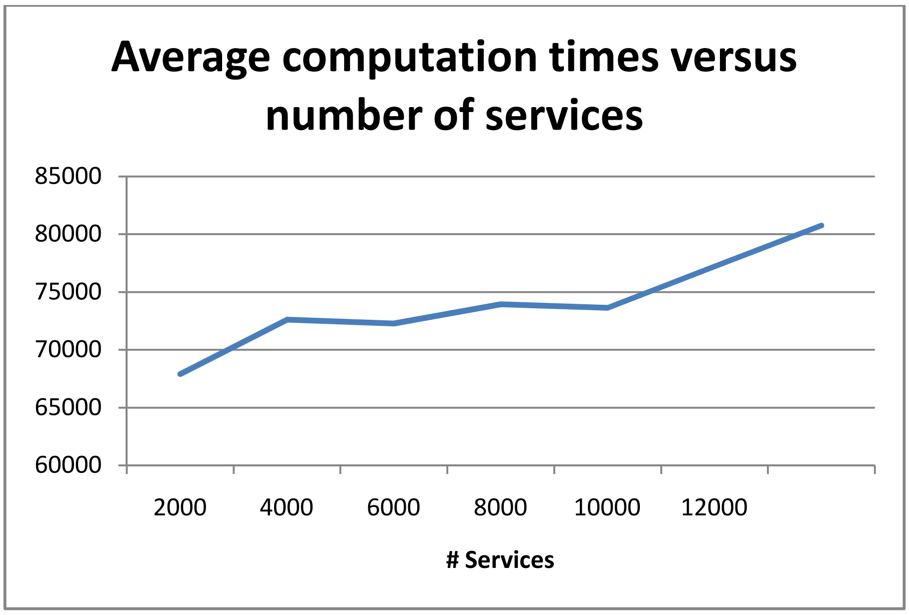


Figure 15: Average computation times for solved sets

Unfortunately, the promising results presented above are offset by the fact that this only an average over the sets the software did solve.

When we look at the number of test sets that have been solved by the algorithm, we see a spike at 8.000 services. This can be explained in the optimal path available through the services. In optimal conditions, the best choice for the next service will coincide with the prioritizing of the used Composition Comparator.

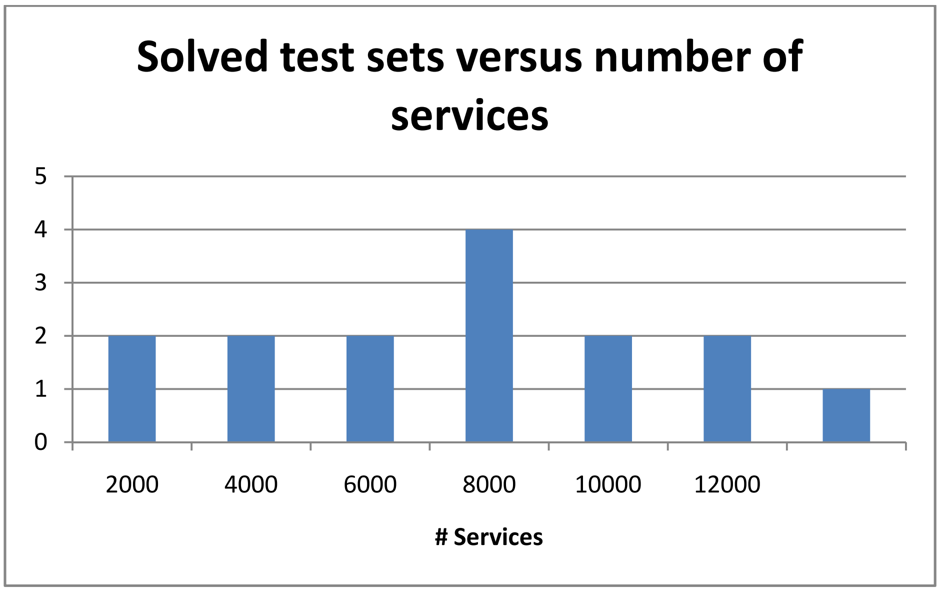
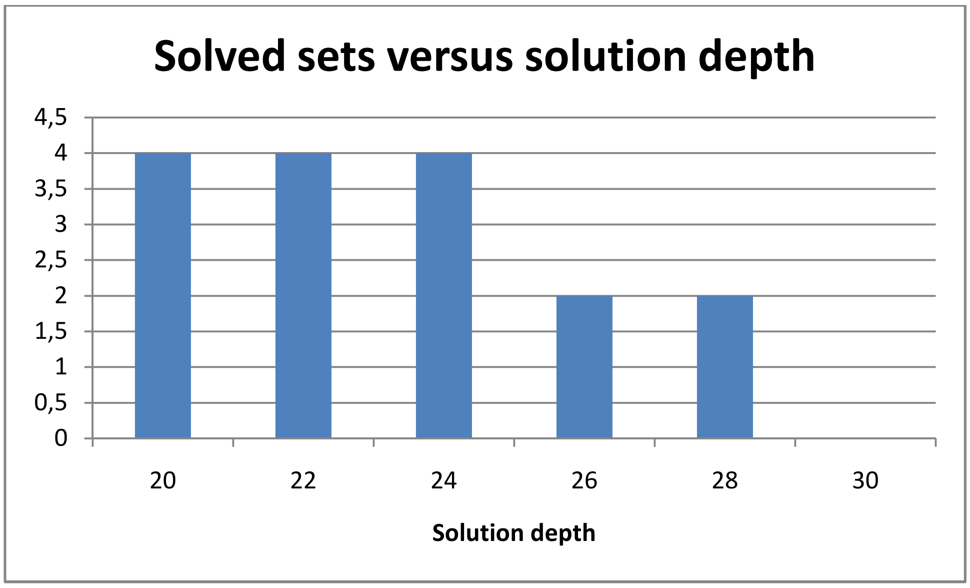


Figure 16: Number of solved test sets out of five offered.

The issue with the test sets is that they all have been generated with a depth of 30 services. Through more testing we have found that the algorithm has problems with compositions of more than 24 services. To show this clearly, we have constructed a new set of test sets, each with 8.000 services and 100.000 concepts. We now vary the depth of the solution from 20 to 30, again offering 5 test sets for each combination.



We see the performance of the algorithm drops dramatically when the depth is increased above 24 services and the algorithm is even unable to solve sets with a depth of 30 or more.

On a side note, the dependency on the of the algorithm of the structure of the optimal solution becomes very clear when plotting the average computation times of these last test sets.

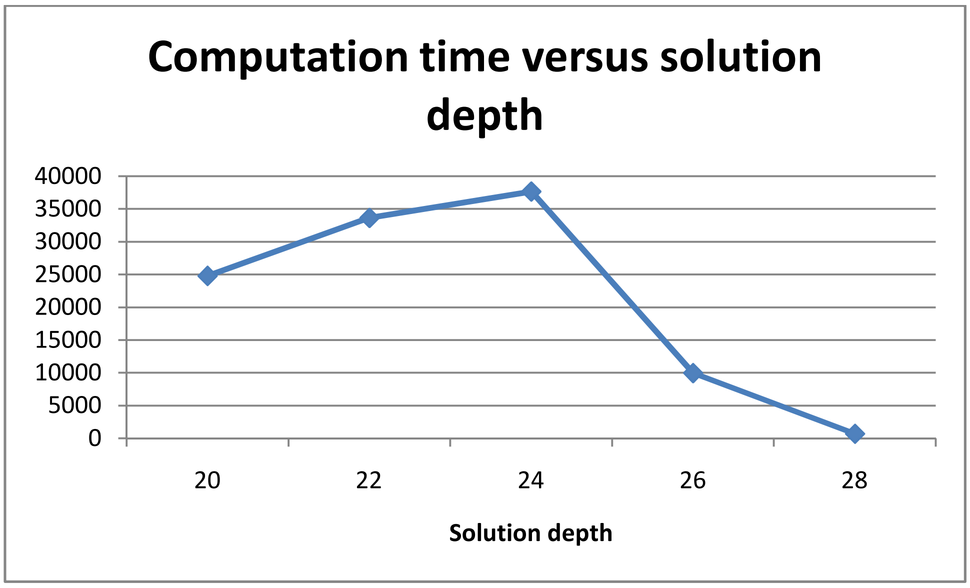


Figure 17: Computation times of solved sets with varying depth.

Here we see the algorithm improving its speed over 40 times when searching for a solution of depth 28 instead of a solution of depth 24, before being unable to solve any sets anymore at a depth of 30.

It is clear that the RuGQoS solution to the web service composition problem performs well if the required depth is small, but for deeper compositions another algorithm is needed. One of the architectures that does solve this problem, the layered approach, is discussed later in this thesis.

4. Discussion

In the adaption of the architecture of last year, we have encountered several problems, we have found one problem that is not solvable with our current approach. We had hoped that with current computing power and memory, our approach would not have to be limited in any way. If no limitations exist our algorithm will always yield the optimal answers. If a better answer would exist, we would have had to pass this earlier in our priority queue. This was however infeasible. The search space is too large and the system will simply run too long and run out of memory before the optimal answer is found. We have therefore had to limit our search with several parameters, described in the previous chapter. These parameters define the maximal amount of services and the maximal length of the longest chain a candidate composition can have. Unfortunately, these constraints come at a hefty price.

We can no longer guarantee returning the optimal solutions if they fall outside of our limitation parameters.

With the type of algorithm we use, this compromise is unavoidable. At the WSC '09, several algorithms have been suggested, some of which do not have this problem. We will discuss these later on.

The second big problem with the RuGQoS approach is the search for the optimal throughput composition. The problem here lies in the use of a priority queue with a comparator to order the compositions. While the loop of the algorithm runs new compositions are added and sorted based on their throughput. However, adding a service to a composition with a higher throughput then the composition has will not affect the throughput. Resulting in a priority queue filled with a lot of items with equal sorting value. The comparator will order these a bit more by unsatisfied concepts and amount of services, but this does not solve the problem. This "clumping" behavior renders the use of the ordering of the priority queue ineffective and the algorithm begins to behave like a brute force search. This problem is limited to the search for the optimal throughput though. While searching for the optimal response time, the priority queue is sorted based on the response time of the composition. With each service added, we add extra response time to at least one of the sequences of the composition. If this sequence has the longest response time, the composition will have a longer response time after addition of this new service. Should the sequence the service was added to not have the longest response time, the compositions response time will stay the same, however, the response time of the sequence will have increased. Given enough additions, this sequence will become dominant and increase the response time of the composition, changing the place it has in the priority queue.

This problem with search for the optimal throughput solution is not solvable with the approach of the RuGQoS system as the priority queue is at the heart of the solution. At the WSC '09 several other approaches were presented and some do not suffer from this problem. The three main solutions presented are discussed below.

Observations on the WSC '09

At the WSC '09 nine teams presented solutions for the web service composition problem. The solutions seen can be divided into four main categories.

Service addition searches

The first category encountered at the challenge is can best be described as the service addition searches. In this category searches are performed by adding a single service to a composition until a solution has been found. The RuGQoS solution falls in this category, but three other teams have created algorithms that also fall in this category. The two main types of searches used by all teams here are breath first searching and depth first searching.

The participants of Pennsylvania State University have used a learning-based depth first search algorithm for their composition system (6). The algorithm adjusts its search parameters every iteration and terminates if a solution is found that does not exceed given bounds. One of the problems noted is that algorithm requires the user to specify a maximal value for response time and a minimal value for throughput the system must deliver, but in the WSC, there is no indication what these bounds should be. In normal internet use, this might prove to be less of a problem though as these values can be extracted from normal use. The longer the algorithm can work, the better the search parameters will become as well.

The team representing the University of California (12) have implemented two algorithms, a uniform cost search for response time and a backwards greedy search for search for the throughput solution. The uniform cost search is implemented using an Iterative Deepening Depth First Search (IDDFS) algorithm. This type of algorithm combines the completeness of a breath first search with the memory efficiency of depth first searching. However, the algorithm requires a depth limit to be set and this removes the guarantee of finding the optimal solution. The greedy search used has problems with the search space size and is not very scalable.

General Motors and the Sungkyunkwan University (14) have joined forces and their team used an A* algorithm to search for a solution while adding services. A* algorithms work with two values on each step to decide the next; these values are the cost of the current path and the cost of the expected path to the goal. The main issue is that the latter cannot be predicted in the web services challenge. The heuristic chosen by the team is the amount of satisfied concepts. Though intuitively this is a good heuristic to get to a solution quickly, it gives no guarantees on finding a good solution.

AI PLANNING

The second category of solutions was chosen by only one team, the home team of Vienna University (15). Using a set of rules, they were able to translate the web service composition problem into a problem in the AI planning domain. The output of this translation is a PDDL file which they were able to feed into a variety of off-the-shelf AI planning problem solvers. In this case they used the "Blackbox planning tool"[3]. The team provided only results based on last year's test data and failed to represent at the challenge, I am therefore unable to conclude more on this matter.

NETWORK ANALYSIS

A very different approach was presented by the team from Pennsylvania State University (16), who proposed a network analysis type algorithm. The algorithm proposed builds matrix of the way services and concepts relate to one and another. This matrix is then used to build a graph of the network. In this graph, both services and concepts are represented as nodes, with edge representing the relations between them. This graph is then analyzed to extract the stronlgly

[3] The "Blackbox planning tool" can be found at the website of Henry Kautz at www.cs.rochester.edu/u/kautz/satplan/blackbox, verified august 15th 2009.

connected components and identify the bridges between them. In composition, these bridges are important as they mark paths that have a high probability of being followed. The algorithm is thus able to split the search into two parts, between the connected components and within the connected components. Sadly, no results of the system are known, but the approach seems useful in real life situations as it is capable of providing a lot of extra information on the web service network.

LAYERED APPROACH

The last category is the layered approach. It begins with a first layer with only the provided instance available. The next layer is build by adding all services that can now be invoked. The newly obtained concepts are added to the available concept set and the process is repeated. Only services that have not been placed in any previous layer are available for selection, thus every service can be added to only one layer. Each concept keeps track of which service provides it with the best QoS attributes and updates if a new service can provide a better QoS. The algorithm stops when no new services or concepts can be added and no QoS attribute is updated for any concept. The top three in the computation contest consists entirely of the three contestants in this group.

Two teams out of Beijing have implemented this solution. The team from the Tsinghua University (9) has implemented a system that is loosely based on the previous idea. It uses a three stage system to optimize the obtained layers. However, they use an approach which depends on the path length to the solution rather than the QoS attributes. They cannot guarantee their results to be optimal, but do can guarantee finding a solution if one exists. On top of that, they can do it fast, making the algorithm a place to start looking further.

The other team hailing from Beijing is from the Chinese Academy of Sciences (17). The algorithm they implemented builds the layered network as described above. It uses two threads for finding the optimal response time solution and the optimal throughput solution. In every layer that is added, the list of concepts is updated not only to contain the optimal values for response time and throughput, but also remembering which service has provided it. This approach allows a fast backtracking algorithm to build the final two solutions after the layered graph is completed. This is backed by good test results and a first place in the computation challenge of the WSC ’09.

The last team I would like to discuss hails from the Institute of Informatics and Software Engineering in Bratislava, Slovakia (18). Their algorithm combines three different processes and a high amount of preprocessing to achieve high speed in finding compositions. The algorithm preprocesses the data to find all possible paths available without input in the graph that can be constructed by the given services. It then uses this data to build the layered composition as described above, starting one thread from the given concepts forwards and another from the required concepts backwards. Using a relational database, both threads can simultaneously access the data for optimal speed. If a third processor or core is available, a third thread removes all services from the search space of the next layer that cannot be invoked to limit the search space even further. This approach is highly scalable as presented test results at the WSC ’09 have shown.

The results from the competition have been added in reference for reader below.

		1. Place		2. Place		3. Place		4. Place	
		Result	Points	Result	Points	Result	Points	Result	Points
Challenge Set 1	Lowest Response Time	500	+6	500	+6	780		880	
	Highest Throughput	15000	+6	15000	+6	15000	+6	15000	+6
	Composition Time (ms)	< 300	+6	< 300	+6	< 300	+6	531	
Challenge Set 2	Lowest Response Time	1690	+6	1690	+6	2100		2110	
	Highest Throughput	6000	+6	6000	+6	6000	+6	6000	+6
	Composition Time (ms)	< 300	+6	< 300	+6	< 300	+6	2219	
Challenge Set 3	Lowest Response Time	760	+6	760	+6	760	+6	950	
	Highest Throughput	4000	+6	4000	+6	4000	+6	4000	+6
	Composition Time (ms)	< 300	+6	< 300	+6	< 300	+6	21438	
Challenge Set 4	Lowest Response Time	1470	+6	1470	+6	2070			
	Highest Throughput	4000	+6	2000		4000	+6		
	Composition Time (ms)	< 300	+6	< 300	+6	< 300	+6		
Challenge Set 5	Lowest Response Time	4070	+6	4070	+6	4500			
	Highest Throughput	4000	+6	4000	+6	4000	+6		
	Composition Time (ms)	< 300	+6	938	+2	< 300	+6		
Sum		90 Points		80 Points		66 Points		18 Points	

Table 2: Web service challenge 2009 computational challenge results .

RECOMMENDATIONS FOR A FUTURE ARCHITECTURE

With the experiences we have from the WSC '09 we can make a few recommendations for future solutions for the web service composition problem.

The first and foremost recommendation I wish to make, is to use the layered approach for building compositions. This method restricts the order of searching to the number of services provided. Using a reverse index based on the in- and output concepts of services, finding the sets of services to be invoked on the next layer can be done efficiently. Keeping track of optimal paths while building the layers can be done without extra overhead and will immediately yield the optimal path after completion. It also enables one to guarantee the optimal results with respect to QoS attributes and can run in a single or multiple threads.

Almost every new PC that is sold these days has more than one core. This calls for architectures that can support working with more thread. The suggested layered approach can work with up to three threads, but it is conceivable more threads could be created to optimize processing even further. This scalability will play a big role in future web service composition solutions as the software to solve this problem will mostly be used as part of a server system handling more than one request at a time. Being able to optimize the number of cores and threads could yield great rewards in processing time.

Use a relational database. These software packages are optimized to handle large amounts of data in efficient ways and can rapidly group and return data. The database can often be configured to keep specific data in main memory and less important data on disk. It is also able to create persistent storage of data on the fly, allowing a recovery of the system after a failure without having to preprocess all data again.

Depending on the context, use extensive preprocessing or eliminate it. If one participates in the WSC, a lot of preprocessing is desired to reduce computation times. This preprocessing is possible as the supplied data sets are static and will not change. However, if we consider a real world situation, the services available will change as will the ontology of the concepts. The frequency at which this happens determines the usefulness of preprocessing, up to the point where preprocessing should be eliminated all together.

Optimize the storage of the ontology. The amount of concepts and instances in an ontology will often be much larger than the amount of services that use them. This makes efficient storage and traversal of the ontology tree important. In the WSC '09 only super- and subclass relations have been used between concepts, but OWL supports a lot more types of relations which will be present when looking at the ontology or larger systems. To store these relations efficiently we can again turn to the relational database, which is optimized to work with any kind of relation between data.

Recommendations for the WSC 2010

With the knowledge now from the WSC '09, I would also like to make some recommendations to the organizers of the WSC 2010.

A complaint expressed by many participants this year is the inadequacy of the output format. With only support for invocations, flows, sequences and case statements, many participants we not able to translate their solution into the optimal output format. In many cases, a direct translation would result in longer response times of the composition described in the WSBPEL output then in memory of the machine. Some participants have created a workaround and calculated the longest execution path of their composition and used this as the main sequence, however, this of course came at the cost of more computation time. The problem can be solved by adding a "depends on" relation in the output format, allowing the DAG of the solution to be translated into WSBPEL without loss. It also allows for the specification of parallel execution of services, which is a desired feature for the WSC 2010 according this year's challenge description.

In real world situations, data can be mined from a system that answers more than one query. This data can then be used to optimize future requests. If the organizers of the WSC 2010 wish to simulate this better, they could provide more than one query for each data set. Perhaps even correlation between queries is possible as would happen in real world situations. This approach would result in composition software that would be even closer to real world use.

Should the organizers wish to challenge the participants even further, one could consider changing the available services during the running of the software or between queries. The ontology of concepts might change accordingly. This would resemble the real world even more, where the stability of the network and services is not guaranteed.

5. Conclusion

The WSC ’09 draws researchers from all over the world together to solve the web service composition problem. The web service composition problem consists of finding a set of services that can be invoked in a specific order to fulfill a specific request. The in- and output instances of each service are related by the concepts they contain. These concepts have super- and subclass relations described in ontology. The challenge consists of two parts, an architectural challenge and a computational challenge. In the architectural challenge participants with high quality architecture of their solutions are rewarded and in the computational challenge, the computation time of the software and the correctness of the solution are measured. This year, I participated in a team from the Rijksuniversiteit Groningen and we achieved 2nd place in the architecture competition and 4th place in the computational challenge.

The top three of the computational challenge all used the same approach in finding solutions, giving us a clear sign of a direction for future work in this area. The approached used consists of building a layered network of services. The first layer contains only the provided concepts by the input instance(s). The next layer is build by adding all services that can be invoked with the available concepts; the set of available concepts is then updated to reflect the newly obtained concepts. Every concept keeps track of which service can provide it with the best QoS attributes and updates these when a new layer is added if needed. The algorithm terminates when no new services or concepts could be added and no concept has updated its QoS attributes.

This type of algorithm ensures the optimal answer and is fast as it is mainly dependent on the number of services, not the amount of concepts, which is generally a lot more. The algorithm is able to work from the provided instances forward as well as from the required instances backward to allow for two threads to work concurrently in finding the answer even faster.

The use of a relational database has also proven itself this WSC. A relational database allows several processes to concurrent access and being able to quickly group items based on relationships makes the use of a relational database a good choice as data storage. Most off-the-shelf packages can be configured to keep specific data in main memory and the ability to write the data onto persistent media creates reliability.

Even though the WSC is able to provide a good basis for research on web service composition, there are still significant differences between the real world and challenge environment. The challenge is semantic, so data representation is not part of the challenge, which is one of the main issues in the real world. In the challenge all concepts are artificially generated and have clearly defined relations. To take the challenge closer to the real world applications without losing its semantic property several things can be done.

The participant’s software might be queried more than once on a given data set, giving rise to the ability to optimize based on previous queries. Services might be added or removed while the software runs and the ontology might have to be altered to accommodate this. These kinds of changes would allow the contestants to create software that is to a real world useable solution.

References

1. *Optimal QoS-Aware Web Service Composition.* **Aiello, M, et al.** Groningen : IEEE Conference on Commerce and Enterprise Computing, pp 491-494, 2009. 978-0-7695-3755-9/09.

2. *Visualizing Compositions of Services from Large Repositories.* **Aiello, M., Benthem, N. van and Khoury, E. el.** Groningen : University of Groningen, 2008.

3. *Bringing Semantics to Web Services: The OWL-S Approach.* **Martin, D. et al.** San Diego, USA : Proc. of the First Intl. Workshop on Semantic Web Services and Web Process Composition, 2004. SWSPC-04.

4. *The Web Service Modeling Framework.* **Fensel, D. and Bussler, C.** s.l. : Electronic Commerce: Research and Applications 1(2). pp. 113-137, 2002.

5. *OWL Web Ontology Language Reference.* **Bechenhofer, S., et al.** Online availble at http://www.w3.org/TR/2004/REC-owl-ref-20040210/ : World Wide Web Consortium (W3C), 2004.

6. *QoS-Driven Service Composition Using Learning-based Depth First Search.* **Nam, Wonhong, Kil, Hyunyoung and Lee, Jungjae.** Pennsylvania, USA : IEEE Conference on Commerce and Enterprise Computing, pp. 507-510, 2009. 2009 IEEE Conference on Commerce and Enterprise Computing Proceedings. pp. 507 - 510. 978-0-7695-3755-9/09.

7. **Keller, A. and Ludwig, H.** The WSLA framework: Specifying and monitoring service level agreements for web services. *J.Network Syst. Manage. 11(1).* 2003.

8. *WSC-2009: A quality of Service Oriented Web Services Challenge.* **Kona, S., et al.** Vienna : IEEE Conference on Commerce and Enterprise Computing, p. 487-490, 2009. 978-0-7695-3755-9/09.

9. *A QoS-Driven Approach for Semantic Service Composition.* **Yixin, Yan, et al.** Beijing, China : IEEE Conference on Commerce and Enterprice Computing, p. 523-526, 2009. 978-0-7695-3755-9/09.

10. *Effective web service composition in diverse and large scale service networks.* **Oh, S.-C., Lee, D. and Kumara, S.R.T.** s.l. : IEEE Transaction on Service Computing. 1(1), pp. 15-32, 2008.

11. *master thesis RuG in preparation.* **Benthem, N. van.**

12. *Business Process Composition with QoS Optimization.* **Zhang, J., et al.** Irvine, USA : IEEE Conference on Commerce and Enterprise Computing, pp. 499-502, 2009. 978-0-7695-3755-9.

13. *Web service indexing for efficient retrieval and composition.* **Aiello, M., et al.** s.l. : IEEE EEE/CEC 2006/ pp/ 63-65, 2006.

14. *WSPR*: Web-Service Planner Augmented with A* Algorithm.* **Oh, S.-C., et al.** Gyeonggi-do, South Korea : IEEE Conference on Commerce and Enterprise Computing, pp. 515-518, 2009. 978-0-7695-3755-9/09.

15. *MOVE: a generic service composition framework for Service Oriented Architectures.* **Rainer, A. and Dorn, J.** Vienna, Austria : IEEE Conference on Commerce and Enterprise Computing. pp. 503-506, 2009. 978-0-7695-3755-9/09.

16. *Large-scale Network Decomposition and Mathematical Programming based Web Service Composition.* **Cui, L., et al.** Pennsylvania, USA : IEEE Conference on Commerce and Enterprise Computing, 2009. 978-0-7695-3755-9/09.

17. *Effective Pruning Algorithm for QoS-Aware Service Composition.* **Huang, Z., et al.** Beijing, China : IEEE Conference on Commerce and Enterprise Computing. pp. 519-522, 2009. 978-0-7695-3755-9/09.

18. *Semantic Web Service Composition Framework Based on Parallel Processing.* **Bartalos, P. and Bieliková, M.** Batsialava, Slovakia : IEEE Conference on Commerce and Enterprise Computing. pp. 495-498., 2009. 978-0-7695-3755-9/09.