Cost-Efficient Web Service Compositions

submitted by
Irma Sofia Espinosa Peraldi

supervised by
Prof. Dr. Ralf Möller
M.Sc. Sebastian Bossung

Hamburg University of Science and Technology
Software Systems Institute (STS)

Abstract

Web services are the latest standard technology in the area of software reusability for distributed systems. They are a new layer of abstraction on top of existing systems, helping to communicate them by the use of the internet network. Their aim is to share resources, more precisely, specific operations of an application, regardless of the platform or language in which they are implemented. Thus a web service can be seen as an application that can be accessed over the internet, providing service not only to human end users but also to other systems. The advantages that web services offer can be increased when they are used within compositions. Composed web services define a new web service interface above a set of existing ones, allowing to create complex processes by using prebuilt applications at different internet network locations as if they where part of a single application. Therefore, the typical way to reuse code (functions, classes, etc.) in a programming language by the use of resources found in local repositories is now extended to a wider scope the internet. As a well-adopted technology, the number of available web services that offer the same functionality is increasing, making management decisions over the selection of the most convenient one become more important. It is in this area of management that this thesis focuses. Having as a starting point a given web service composition, and having the possibility to choose from a list of web services of the same type for each web service partner invoked in the composition, the problem addressed in this thesis is to analyze the different aspects involved in the process of creating an optimal composition, expressed in terms of a configuration problem, in which candidate web services become the primitive objects to be combined such that they satisfy a given criteria.

Declaration

I declare that:
this work has been prepared by myself,
all literal or content based quotations are clearly pointed out,
and no other sources or aids than the declared ones have been used.

Hamburg, 14th July 2005
Irma Sofia Espinosa Peraldi

Acknowledgments

This report describes the thesis project that I developed from January to July 2005 as part of my studies at the Master program *Information and Media Technologies* in the STS (Software Technology Systems) department of the Technical University Hamburg-Harburg.

I would like to thank my parents for giving me always their support, it is because of them and the education they have given me, that i am able to come to Germany and live an experience of a life time.

To CONACYT for granting me a scholarship during my first 2 years of studies as part of a program with DAAD to promote the scientific research of Mexicans in Germany.

I am grateful to my supervisor Prof. Dr. Ralf Möller, for the hours of attention specially at the beginning of the project which were very important to guide me during my research and also for trusting me an interesting and challenging topic of research.

To Sebastian Bossung for the great discussion sessions and careful review of my work.

To Michael Wessel for the professional and fast answer of my questions during the multiple Racer sessions.

To Savvas Katemliadis for the comprehension and spiritual support in difficult times.

This project could not have been as fruitful as it was, without the support, interest and inputs from individuals and their willingness to share their experience and knowledge, and the time given to discuss related topics.

Contents

List of Figures

Preface

Document Distribution

Chapter 1: Introduction

This chapter aims to introduce the background of the project. First it is introduced how web service technologies facilitate software reusability, then an overview about the core technologies that realize web services and their composition is described. Finally an introduction to the problem addressed in this thesis is given.

Chapter 2: Object to Optimize

This chapter gives a more detailed description about what web service compositions are and how they are considered in this project as means for the definition of a querying process. Finally a specific web service composition is defined, which serves as an example for the design of the optimization strategy.

Chapter 3: Criteria Analysis

Having a web service composition as an object to be optimized, this chapter aims to search for suitable criteria that can help to measure the optimality of the composition, once the strategy developed in this project is applied.

Chapter 4: Algorithms for Optimization

This project considers the optimization problem as a configuration problem, where web services are being combined and compared according to some criteria to find an optimal solution. Thus, this chapter presents a number of optimization algorithms considered suitable to solve the problem addressed in this project. Finally, the adopted approach is presented.

Chapter 5: Materials and Methods

It aims to give a detailed description about the design of the *Locator's* implementation. Thus, how the considered criteria obtains its values, some pseudocode of the optimization algorithm, communication style with the web service partners and clients, etc.

Chapter 6: Results

This chapter presents some test results to show how the implementation behaved according to specific settings. Moreover advantages and disadvantages of the design are discussed.

Chapter 7: Future Work, Shortcomings and Contributions

Finally shortcomings, significance of the findings and future work are pointed out.

Chapter 1

Introduction

This chapter aims to give an overview of the background related to this project. Web services and the core technologies that implement them are introduced and described how they are a new generation of technology for the reusability of software with the world wide web as scope. Finally the problem addressed in this project is pointed out.

1.1 Reusability of Software

Reusing something has as a natural consequence the possibility of saving resources and effort. Thus, there is a constant tendency for reusability in different areas from which software development is not an exception.

Techniques and tools for software development have evolved since the creation of computer systems towards reusability. This can be appreciated since the use of *functions* in *structural programming*, where ready made pieces of code are available from local libraries. Continuing to *object-oriented programming* where a whole concept is abstracted in so called *objects* which encapsulate attributes and functions describing their specific characteristics and behavior, such that not only a function but a whole concept can be modeled and reused. More over, with the development of distributed systems from the local area network to the world wide web, the scope for sharing and reusing software resources has also changed from local to remotely distributed locations. Still when considering the reusability in distributed environments, compatibility issues become a priority, since there is a variety of languages, operating systems, interaction protocols, etc. to deal with. An analogy to such compatibility issues can be observed for example in today's globalization, where humans are nowadays able to reach other countries easier than ever, but it is confronted to new languages (syntax) that he must understand, moreover customs and interpretations are also different (semantics), metric systems, power adapters should not be forgotten, to name only a few examples. Thus, the possibilities and facilities to interact with other environments are there, but to successfully

interact with them some additional knowledge is needed.
To overcome those heterogeneous environments in the area of software development, a new architecture has been introduced called *SOA*[1]. It has the aim of loosely couple different architectures by the use of *service* units. These units are considered as a black box, where only the interface is visible to indicate how the interaction with the service can proceed independently of how their internal processing is carried out.
Services in a SOA architecture must implement interfaces based on generic semantics such that it is understood by any client that wants to use it, thus facilitating the integration of software technologies.

1.2 Web Services

A SOA architecture that aims to integrate components distributed over the world wide web can be realized by web services. They represent the latest technology for reusability of software distributed in heterogeneous environments, that have facilitated the introduction of new business paradigms like the automated integration of IT systems between different trading partners for the collaboration in supply-chain processes, to name one example.
Thus, web services aim to overcome the compatibility issues between different architectures due to their XML[2] nature. XML is a descriptive language, which allows to define data independently from any system architecture, this is done through the creation of documents which are related to each other, namely the document instance which contains the data to transfer and the schema document which provide means for defining the structure, content and semantics of the XML instance documents[3], such that it is possible to interpret the meaning of the data contained in the instance document and it can be transformed according to the requirements of a specific system. XML is a syntax on top of which many specifications have been developed, in the area of web services.

1.2.1 Web Service Technologies

To be able to use web services it is necessary to first localize the service of interest, understand its interface and finally start the communication. For each of these activities, there is a web service specification that supports them, these are:

SOAP The Simple Object Access Protocol, which supports the communication with web services. It uses the HTTP[4] protocol used by the web to be able to handle XML based messaging between the parties involved. The styles

[1] Service Oriented Architecture
[2] Extensible Markup Language: http://www.w3.org/XML/
[3] As described in http://www.w3.org/XML/Schema
[4] HyperText Transport Protocol

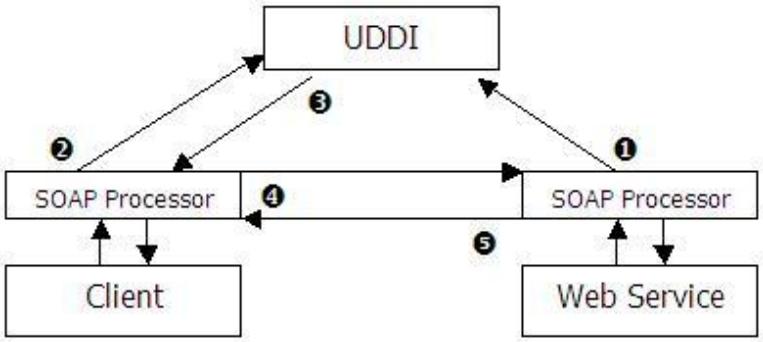

Figure 1.1: Core web service technologies in interaction

of messages that can be send with SOAP can be divided in *RPC*[5] which simulates a method call with parameters of input and output, and the *document style* which contains a plain XML document.
Both parties, the client and the service involved in the communication, need a SOAP processor, such that the protocol can be interpreted. The processor can map the received data to the underlying software that is providing the service. In this way every service independent of their architecture, transforms its outputs to a standard neutral syntax that can be sent through the internet and received by other systems that understand the same syntax, interpret the contents of the message and transform them according to their own needs.

WSDL The Web Service Description Language specification implements an XML file which represents the interface of the web service, such that a description of the location, its operations, their respective parameters and binding styles are published on the internet for the customers to access.

UDDI The Universal Description, Discovery and Integration Registry, is a directory that helps to localize the web services. The information contained in this repository can be categorized in *white pages*, where the name, address, web site, id number and other relevant information about a business can be found. Another category is *yellow pages*, where the information is categorized under different taxonomies, like type of business, location and products. And finally the *green pages* where technical information of the business service is described, for example the reference to the WSDL file can be found here.

In order to use a web service using the previously introduced technologies the process in figure 1.1 shows how they interact with each other.

1. A web service registers its WSDL file in the UDDI directory, through a SOAP request, thus, the UDDI registry provides a set of SOAP APIs to services and clients that wish to interact with it.

[5]Remote Procedure Call

2. Clients, can browse the UDDI registry using the SOAP APIs to discover the WSDL file of the desired web service and make a request.

3. The UDDI provides the WSDL information to the client. The file is examined to understand the requirements for interaction.

4. The client can post a SOAP request to the web service.

5. The service can send the requested information back to the client.

These web service technologies have been widely adopted and are considered as a standard for the interaction with applications over the internet. For a more detailed description refer to [7] and [3].

The number of applications that are being developed as web services is increasing, as a consequence, there is a tendency of constructing applications that try to use various web services together, this kind of applications are identified as *web service compositions*.

1.2.2 Web Service Compositions

A further level in the reusability of software is then observed when a new application is developed by composing a set of web services. Such applications are also referred to as orchestrations of web services, because the application is managing the flow of information between the composed services.

A web service composition has its own web service interface constructed on top of its web service partners, which are assembled together, to create a process. Therefore, advantage is being taken of prebuilt components from applications at different network locations as if they where part of a single software system. Specifications have been developed to standardize the way web services can be orchestrated, the most recent specification is called *BPEL4WS*[6] also referenced as BPEL. It merges two previous specifications, namely *WSFL* proposed by IBM and *XLANG* by Microsoft. This specification defines an XML base language, that allows to invoke web services, receive their response and reply to clients of the composition in a structural programming style[7].

When trying to compose web services together and as web services proliferate, thus finding more web services that offer equivalent functionality, a new situation emerges in which one wants to choose the one service that does not only offer the desired functionality but it is also convenient according to a certain criteria. For example, one can start to compare similar web services according to price, binding styles, response time, security protocols, etc.

[6] Business Process Execution Language for web services

[7] For a detailed documentation on BEPL refer to http://www-128.ibm.com/developerworks/library/specification/ws-bpel/

1.3 Addressed Problem

As explained in the previous section, a composition does not only obtain the functionality of the web services, but some additional characteristics are also expected to be fulfilled by the web service partners such that, they can be considered as part of the composition. It is in this area of decision making that this project takes place, where a given web service composition is to be optimized by choosing the best web service partners according to given criteria.

The kind of compositions considered in this project model a querying process, such that query requests are being send to the web service partners which offer similar query reasoning functionality. It is observed that the optimization strategy can be expressed as a configuration problem where combinatorial and optimization algorithms are suitable for its solution. A set of criteria is considered to test the optimization strategy, implemented by the *Locator*, which is able to compare and choose between the web service partners according to this set of criteria. The Locator is designed in such a way that it treats any criteria as black boxes with a value, so that there is flexibility for additional criteria any time it is necessary independently of their peculiarities. Thus, the Locator is developed to cope with multiple criteria expressed as constraints or objectives.

A strategy for optimization, as the one presented in this project, can profit from the current research in the area of quality of services, security and transactions for web services where more specifications that complement the core web service technologies are being developed to become standards. Future work can further focus on finding criteria suitable for the optimization of web service compositions towards the creation of a brokerage business model, such that web service compositions of a certain domain are being attracted by the knowledge that a middleware like the Locator possesses about candidate web service partners that fulfill their demands and expectations. While, on the other side web service partners can find a market place for the distribution of their services.

Chapter 2

Object to Optimize

This chapter aims to introduce the object to be optimized in this project. A first explanation of the nature of web service compositions and their behavior during instantiation is presented for the proper understanding of the scenarios that can occur during the life cycle of a composition; afterwards a specific composition is presented that will serve as an object for optimization. This last part, aims to clarify how compositions are considered in this project as means for designing a querying process to be target to a specific system.

2.1 Web Service Compositions

Web services provide a new layer of abstraction, a new interface through which it is possible to communicate via internet with existing software applications by the use of standardized technologies, extending the area of SOA[1] to the world wide web. As a consequence a growing tendency to reuse the functionality now available through this new interface, leads to the creation of web service composed applications.

While a composed application could be built upon any architecture, some standardization efforts have been taking place to ensure reliability regarding the correct specification of a composition. One of these efforts is BPEL4WS[2], which merges two older proposals, the WSFL[3] from IBM[7] and XLANG from Microsoft[7]. The BPEL4WS specification is adopted in this project for the description of the web service composition model to be optimized.

2.1.1 BPEL4WS

BPEL4WS is an XML based language for the standardized implementation of processes through the composition of web services.

[1] Service Oriented Architecture
[2] Business Process Execution Language for web services
[3] Web Service Flow Language

The composition can be exposed as any other web service by its own WSDL file, containing a list of port types describing the operations offered by the composition. This language provides a variety of XML tags that can be divided into three groups, the primitive activities, the structure activities and the consistency activities.

1. Primitive activities
   These are the tags that deal with the interaction of the composition with its web service partners. By web service partners is referred to any web service that is used by the composition as a provider or that makes use of the composition as a client. Since the main purpose of the language is to compose together a set of web service into an executable program, a BPEL program consequently makes use mainly of these activities. An example of these primitive activities is:

   - Invoke: To invoke an operation of some web service partner, which plays the role of provider to the process
   - Receive: Where the process waits for a message of someone external with a response, to start or continue its execution.
   - Reply: Generating a response to a partner of the process.

2. Structural activities
   Refer to those tags that help to combine the primitive activities, into a structure that defines their flow of execution. For example: in sequence, in parallel, alternative paths, loops, etc.

3. Consistency activities
   To support error handling and compensation actions, for example:

   - Throw and catch commands for the handling of errors.
   - Means to define scope units in which compensation activities will take place, useful also for fault handling.

The required components for a complete definition of a web service composition based in BPEL, as showed in figure 2.1, are the following:

1. BPEL file, which contains:

   - References to WSDL descriptions
     Represent the description files of the web service partners. They need to be identified in order to know exactly which operations to invoke and the messages to exchange.
   - Partner definitions
     Lists relations of the partners with their respective port type and the role they play with respect to the composition, as provider or as client.

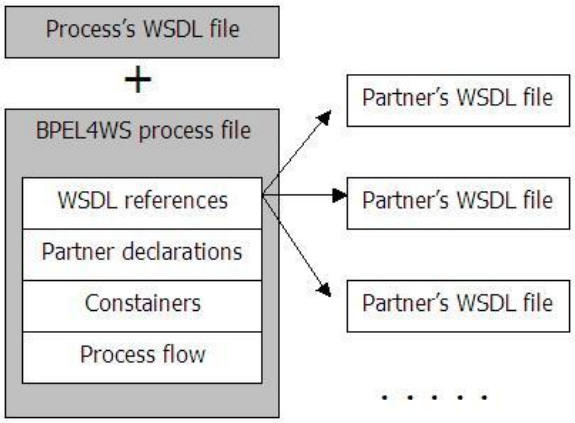

Figure 2.1: Required components for the definition of a BPEL composition

- List of containers
  They offer support to the primitive activities, such that requests and response messages for the respective partners are stored and available for the activities to manipulate them.
- Activity tags
  Finally the activities that describe the flow of the composed application.

2. WSDL file of the composition

3. WSDL file of web service partners

For a more detailed view of the BPEL standard, refer to [14].

Lifecycle of a composition

Once all these elements are defined, and the WSDL file of the composition is published, clients can start its execution when they send a message to the entry port type. The port type is related with a *receive* activity in the BPEL file which is in charge of creating an instance of the composition, thus multiple instances of a process can run simultaneously. Each of the instances is identified by a triple containing: port type, operation and partner. The triple is used as an ID of the instance so that it is possible for the composition to differentiate between messages of different clients and finally giving the corresponding answer to its respective client.

2.2 Web Service Compositions as a Querying Process

The type of composition being optimized in this project describes a querying process. This means, that queries are being send to the web service partners in

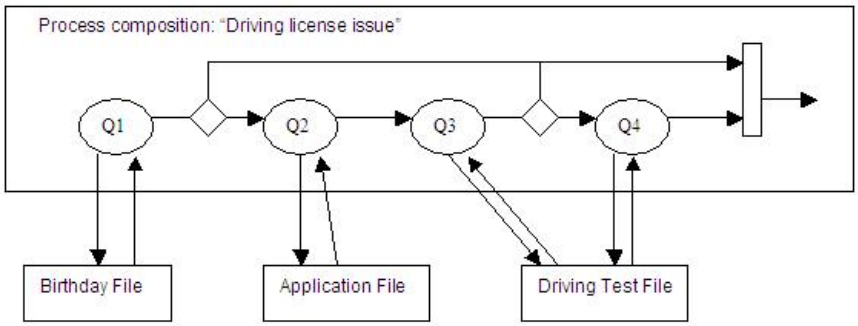

Figure 2.2: Web service composition example: A querying process

each invocation step. Thus, the set of queries that belong to a composition, are not randomly chosen, but they complement each other to obtain information according to the aim of the process's business logic. For example, lets consider a bureaucratic process, which aims at issuing a driving license, so that the next queries are being performed:

- Query 1: A first query accesses the *birthday registration file* to obtain the applicant's birthday date and prove his age.
- Query 2: If the applicant is old enough, a second query is directed to the *license application file* to check for the application number.
- Query 3: A third query accesses the *driving tests file* to obtain the test profile of the applicant.
- Query 4: If the applicant passed the examination a forth query also directed to the same *driving test file* will find the driving license number attached to the test profile.

From the past description and as shown figure 2.2, the partners of the composition are web services that provide querying reasoning and contain the needed information. The composition is in charge of organizing the order in which predefined queries are to be processed, thus it has a general knowledge about:

- The file that a certain query requires.
- An implicit knowledge about the order in which queries need to be processed.
- If a query subsumes another one, in this example, Q4 subsumes Q3, the subsumption concept will be explained in more detail in section 3.2.1.

As the "Driving license issue" process, many other querying processes can be developed for different domains and business logic. The process in figure 2.2 is just an example for illustration purposes, important to notice are the characteristics of the composition that are relevant to this project which are:

1. The partners: In every invocation step, partners of the same type are required, thus all the partners are offering the same functionality, but may contain different information. In this case the partners are *Description Logic Reasoners* that offer querying services over ontologies, a better description of their functionality is found in section 3.2.

2. The queries: There is a predefined set of queries, with a predefined place inside the process, according to the business logic of the process. The structure of the query does not change, only the information they search for, e.g. for the driving license the same process is used to give service to different applicants.

3. Global knowledge: Since the composition is defining a process of a specific domain, e.g. driving license issue, marriage registration, etc. The composition knows the domain of the information that a query needs to get its results from, and the relationships between the queries, mainly whether one query subsumes another or they are independent from each other by searching in different files, or whether they search in the same file.

To find an optimization strategy for a composition with such characteristics, a specific composition was defined to serve as basis for experiments.

2.2.1 Design of a Specific Composition

To define a specific composition the next elements need to be identified:

1. Knowledge Base for the domain of the composition:[4]

   - A Tbox that defines the ontology of the domain.
   - Aboxes that contain the instances of the concepts defined in the Tbox.

2. The set of queries describing the process of the composition.

3. The BPEL web service composition that will orchestrate the querying process.

4. The set of WSDL descriptions from the web service partners.

Chosen Knowledge Base

As previously explained, the composition describes a querying process, where the requests are send to web service partners which offer query reasoning functionality over ontologies. Ontologies are defined through Tboxes and instantiated through Aboxes, such that Tboxes an their respective Aboxes form the knowledge base of such web service partners. For more detailed description of the structure of a knowledge base refer to section 3.2.1.

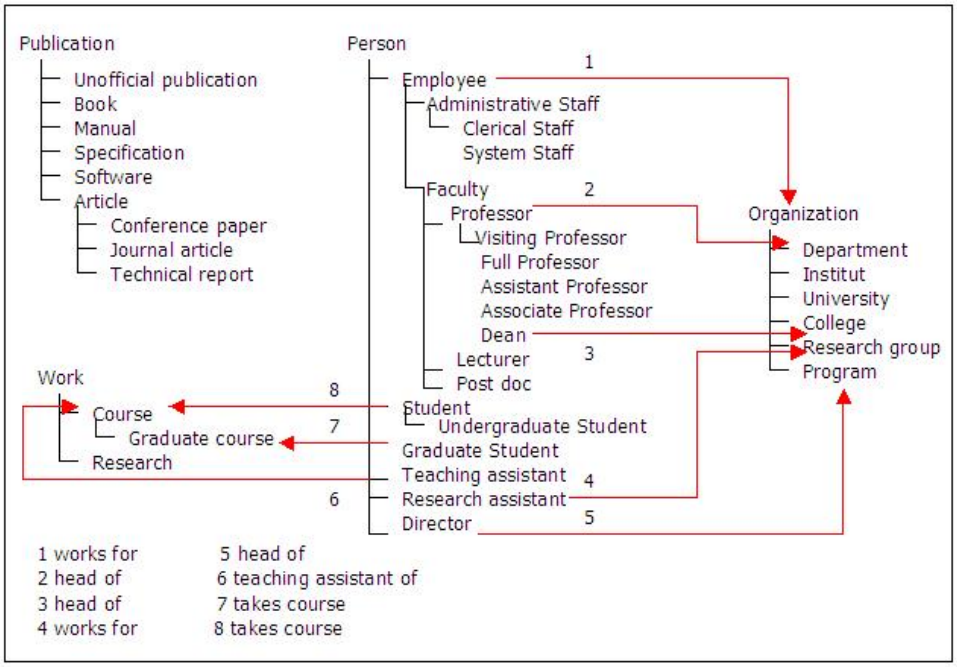

Figure 2.3: Univ-Bench Ontology

Since the creation of an ontology to represent the Tbox is not a trivial task, an existing ontology was adopted, this is the Univ-Bench ontology [16] developed by the Lehigh University. This ontology has been used for the evaluation of DAML+OIL[5] repositories built on RDF, which is one of various formats supported by the web service partners. An advantage of this ontology is not only that it defines a realistic domain but it is also understandable for most of the people, since it models, as figure 2.3 shows, the university domain and its respective organizations, people and work. Another advantage is that the language primitives used in the Univ-Bench ontology[6] consists mainly of primitive classes, some restrictions, object properties and few data type properties, which according to statistics [4] about the average usage of language primitives in ontologies, the primitives used in this ontology are categorized under the most frequently used. Thus, the Univ-Bench ontology can be used to represent a real workload.

To create the Aboxes, a tool called UBA[7] that supported the benchmark [16] was used. It produces random instances, where the university instance is the minimum unit of data being generated per file. Each file, represents a department of a university.

[4]For a description on Tboxes and Aboxes refer to section 3.2

[5]A language specification for the creation of ontologies, described at: http://www.daml.org/

[6]For a better overview of the ontology you can refer to http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl

[7]Univ-Bench Artificial data generator

Chosen Set of Queries

The set of queries that describe the process flow were created such that are suitable for the chosen knowledge base and also represent common used queries, thus the process is as follows:

Name: "Professor for Exchange Program".

Purpose: With the aim of cooperation and improvement in education, different universities around the country participate in an exchange program. In this program, a professor is selected every year and sent to another university to work for one year as a visiting professor. For this process, a university chooses from two given departments, the one with the higher amount of full time professors with respect to the total number of faculty members in that department, such that the absence of a professor is least inconvenient. Afterwards a list of professors and the courses they teach is queried, to obtain the professor that teaches the less number of courses. Finally the contact information of the selected professor is given back.

In the next set of queries the next URL:

http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl

where the ontology is located, was abbreviated by *TboxURL* for illustration purposes.

Q_1: *(retrieve (?x)(?x |TboxURL#Faculty|))*

This is a unary concept query atom, that helps to obtain all the instances of faculty members in the department, to later obtain the relation between total faculty members and number of full time professors in the department.

Q_2: *(retrieve (?x) (?x |TboxURL#FullProfessor|))*

This is a unary concept query atom, to get all instances of full time professors in the department.

Since this two first queries look into different departments, thus into different Aboxes, they can be executed in parallel. Moreover, this query subsumes query Q_1.

Q_3: *(retrieve (?x ?y) (and(?x |TboxURL#FullProfessor|)*
(?x ?y |TboxURL#teacherOf|)))

This is a binary role query atom, which returns a list of full time professors with the courses they teach, it is necessary to obtain the professor with the lowest number of courses under their responsibility. Only after Q_1 and Q_2 are answered and the information about the percentage of full time professors w.r.t. the total number of faculty members of each department is known, a decision to choose between the two departments can take place, so that Q_3 is directed to the chosen department (Abox file).

Q_4: *(retrieve ((annotation (|TboxURL#emailAddress|*
|http://www.Department0.University1.edu/FullProfessor0|)))

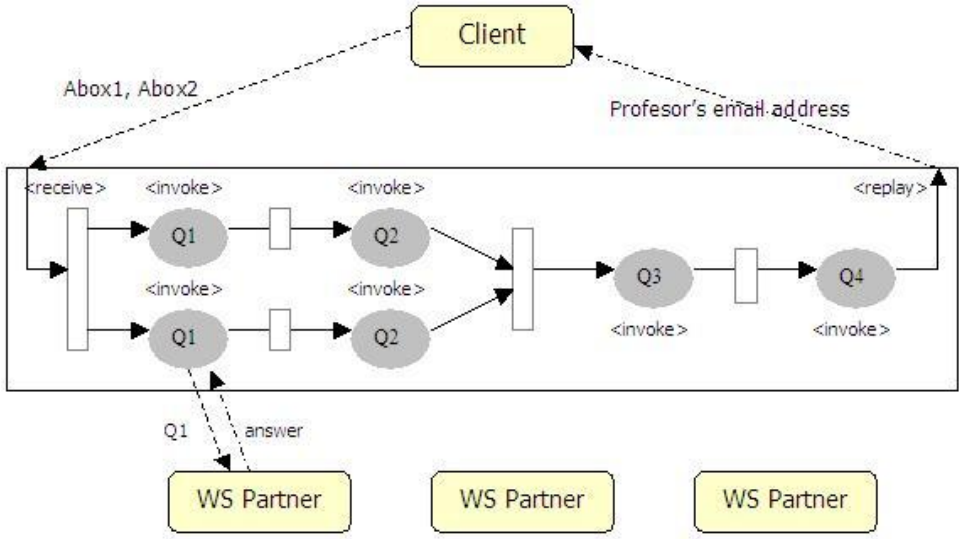

Figure 2.4: A specific composition: *Professor for exchange program*

(|http://www.Department0.University1.edu/FullProfessor0| top))
Finally the email of the professor who teaches the lowest number of courses is obtained.

As shown in figure 2.4 before a query is sent to a web service partner, some computation needs to be done in order to obtain for example, the total number of entities in the answer, to calculate the percentage of full time professors, etc.

The composition's entry point, consists of one port which requires two parameters with the names of the Aboxes to query, as a reply the composition gives back to the client an email address of the chosen professor. Even though the queries for the process are given as a kind of template for the composition, the computation tasks found in between the invoke statements, help to complement the query with the needed information, for example, for Q_4 the name of the full time professor varies for every instance of the composition, so according to the professor's name, the query is changed, also the Aboxes for every instance of a composition vary, so the corresponding changes to the query should be done.

BPEL Web Service Composition

As explained before, the BPEL4WS standard was adopted to define the querying process. To ensure the correctness of the BPEL file, the BPEL Designer from Oracle[8] was used, since it produces BPEL compatible files and facilitates their creation.

[8]http://www.oracle.com/technology/products/ias/bpel/index.html

WSDL Description of Web Service Partners

As explained in section 2.2 the web service partners offer the same functionality, they are services of the same type, thus they have similar WSDL files. The web service partners contain only one port type with one operation, the operation gets as input one parameter of type string that should contain the query and as output a string parameter containing the response to the query is given back. The service is bound to the SOAP messaging protocol, through an RPC/encoded style.

2.3 Conclusions

With the increasing adoption of web service standards, more web services offering the same functionality are available, thus opening a variety of possibilities to choose from, as a consequence the creation of composed applications based in web services become more common. Thus, leading IT companies have gathered to develop a standard for the definition of web service compositions called BPEL4WS.

In this project an optimization strategy for a web service composition is to be found. Still, optimization possibilities can vary from composition to composition, according to the kind of web service partners they use and the architecture on top of which the composition is built. Therefore particularities of the kind of composition to be optimized were identified, such that according to these characteristics a specific composition was defined to serve as an example basis for the optimization strategy.

The kind of composition to be optimize in this project, is one, which uses in every invocation step a query reasoning service over RDF documents, thus the composed application describes a querying process. The BPEL standard was adopted for the description of the composition.

Chapter 3

Criteria Analysis

> The optimization of an object or process means, its improvement towards a certain criteria. This chapter presents a variety of criteria suitable for the optimization of the previously introduced web service composition. It aims to show that most of the time in an optimization project, one has to decide on multiple ways to optimize something to be able to determine objective and constraint functions that will help to probe the cost efficiency of the solution to the given problem.

The research of parameters for the optimization of a problem is a task that is confronted to a wide area of possibilities. For this reason, a good approach would be one that first takes a high level overview of the possible optimization areas, then decided on the area considered as most interesting or convenient to the problem and finally focus the research on the selected area, such that the first optimization steps can take place. Considering that this project tries to optimize a web service composition based on the characteristics of the web service partners, and that all the required partners, as described in section 2, provide the same functionality, the optimization criteria was divided into two major groups, namely service-dependent and service-independent criteria. The first focuses in characteristics related to the functionality offered by the web service partners, while the second refers to characteristics found in the technologies that realize web services in general, regardless of their functionality.

3.1 Service-Independent Criteria

With the increasing adoption of WSDL, SOAP and UDDI as core standard technologies for the implementation of web services, there is a continuous research focused on the creation of additional complementary technologies that cope with issues like security, transactions, quality of service management, etc. that the core technologies have not covered. It is in the different technologies and standards, which realize web services, that it is possible to find optimiza-

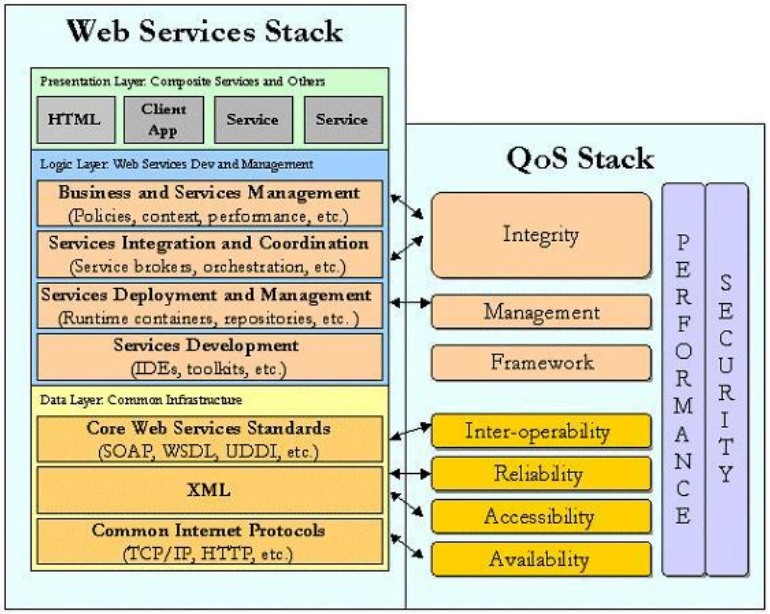

Figure 3.1: WS QoS Stack [13]

tion criteria for web service compositions.

Research has been done in the area of Quality of Service[1] for web services, following different paths, which I classify in three groups:

1. Classification of QoS concepts
2. Descriptive languages for the specification of QoS
3. Frameworks for QoS aware web service management

For each of them a selection of related work in the area is introduced that was considered as relevant for this project.

3.1.1 Classification of QoS Concepts

While many articles talk about ways to specify QoS for web services and propose frameworks for their management, a few have given the time to try to classify QoS aspects, for these reason I found the classification proposed by [13] relevant for this project. The so called QoS stack, see figure 3.1, provides a relation between the technologies which realize web services, divided in data layer, logic layer and presentation layer and their QoS concepts, divided in:

- Performance: Which deals with throughput, latency, execution and transaction time as issues concerning with the application logic, network and messaging and transport protocols.

[1] hereafter referred as QoS

- Reliability: Focus on the transport protocols.
- Integrity: Focus on transactions and security issues.
- Accessibility: With focus on scalability and infrastructure of the system.
- Availability: With focus on fault-tolerant systems.
- Interoperability: With focus on the implementation of web service standards.
- Security: With focus in authorization, encryption and access-control.

3.1.2 Descriptive Languages for the Specification of QoS

There is a proposal of IBM, called *Web Service Level Agreements* which extends the WSDL file to describe quality of service level agreements[2], between the involved parties. A SLA represents a contract containing three main groups of information:

1. Parties: Involves not only the client and provider, but optionally also a third party, for the support in service evaluation and service management, to take actions when a guarantee has been violated.
2. Service description: This is the core part of the contract where the SLA parameters for every operation are specified with higher and lower metrics, to help the parties determine if they meet their expectations, for example, binding possibilities, response time, availability.
3. Obligations: The obligations describe guarantees and constraints upon the agreements, for example, validity period of a SLA parameter.

A general scenario starts when a service provider publishes its WSDL file together with the SLA for multiple operations or for the service as a whole. The interested client starts a negotiation procedure supported by an authoring tool, which retrieves the metrics offered by the provider, allowing the client to choose and combine them into its own SLA and finally send them for approval, such that at the end only one SLA document between the two parties is established. Thus, the language provides with a variety of XML commands for the specification of predefined element that compose a SLA, plus the possibility to extend the language to define other parameters. For a complete reference on the WSLA language you can refer to the XML-Schema found in [1].

There is a similar proposal from HP, called *Web Service Management Language* [2], also based in SLA containing the information previously described, plus a Purpose section, describing the reasons why the SLA was created, and the Exclusions section, which describes what is not covered in the SLA, another difference is that 3rd parties can not be specified.

[2]hereafter referred as SLA

These two proposals, are oriented towards inter-enterprise scenarios, trying to standardize an unambiguous set of XML commands that can express offers and demands for the creations of contracts, and legally formalize agreements between parties, thus formalizing a way to express the criteria upon which a web service can be measured. Still the infrastructure to obtain the measurements and manage decisions has to be build.

Another proposal with a slightly different view comes from Carleton University with a language called *Web Service Offerings Language* [15]. It is based in the so called Service Offerings, which are a kind of SLA, the main idea is that different kinds of *service offerings* can be specified for the same service as a whole, such that more flexibility is offered for the customer to choose between the available offerings while reducing overhead in negotiation processes to form a whole new and unique SLA between parties, as the previously described languages do. A service offering is composed of:

- Constraints: To specify QoS constraint, functional constraints and access rights.
- Statements: To specify price/penalty statements, management responsibility statements and in general any construct which is not considered a constraint is classified under this group.

The language also enables the reusability of offerings, for the specification of new offerings in a inheritance like way. Constraints can be grouped into units of constraints that can further become a template, there is also the possibility to extend the language for the definition of new constraints.

3.1.3 Frameworks for QoS Aware Web Service Management

The *WS-QoS* Framework developed at the Free University of Berlin, allows for the definition, publication and matching of requirements and offers for web services [11], to achieve this the next tools are being offered:

1. QoS XML-Schema: A language to specify predefined QoS requirements of server performance, network performance, security, transactions, pricing, for both client and service provider. The language can be extended through an attribute called ontology, which references an ontology file, where the new parameters are defined.
2. Service broker: To help in the service lookup process, so that clients send the request to the broker together with their requirements, according to these requests the broker looks into its UDDI registries, finds the appropriate service and starts the communication with the services to ask for their offers, the broker compares the offers and decides for the best choice, finally gives the web service reference back to the client, so that it can start its communication with the web service.
3. QoS Proxy: Which aims to support the QoS for the network layer, by mapping the transport QoS parameters specified by the client to the

underlying technologies supporting the QoS transport, thus relays on other technologies like ATM, UMTS.

4. GUI: For the monitoring of server and network performance in real-time.

Thus, the described research efforts try to complement the existing web service standards for the specification of a variety of constraints for a better description of web services and try to find a standard language for their specification. They represent good examples for different paths that research on web service management can take. Still, I consider that without a deeper study in the different areas of QoS parameters, (as the areas presented in figure 3.1) that is sponsored by a standardization group in charge of the definition of ontologies for such diversity of parameters, no standard unambiguous language can be developed that could truly support the further development of management tools for web services.

3.2 Service-Dependent Criteria

To identify service dependent criteria, the functionality offered by the web service partners needs to be understood. As previously explained, the web service composition considered in this project makes use of the services offered by the inference engine called RACER.

3.2.1 Web Service Functionality

RACER: Renamed Abox and Concept Expression Reasoner

The information found in the current world wide web, is described with the use of the HTML language, such that it is possible to combine and organize multimedia objects and texts on top a structure that mainly describes the visual layout of the elements in a web page. Thus, the information is syntactically described while the semantics are left to the interpretation of human end users. Therefore HTML descriptions are insufficient for a computer to reason about the real meaning of the rendered information, for example, to distinguished that "€50" is not only a string but a price of something.

If a computer can understand the real meaning of the information found in the web, then it is possible to automate the gathering of information in a more intelligent way. For this reason, some descriptive technologies have been developed to extend the current web and give well-defined meaning to the information, such that is also understandable for computers. This new extended web is called *Semantic Web.* RDF[3] and OWL[4] are such descriptive technologies and they complement each other to provide descriptive vocabulary. Still, the concepts described with these vocabularies, need a structure that determines their relationship with other concepts with respect to a domain, such a structure is called *ontology.*

[3]Resource Description Framework: http://www.w3.org/RDF/
[4]Web Ontology Language: http://www.w3.org/TR/owl-features

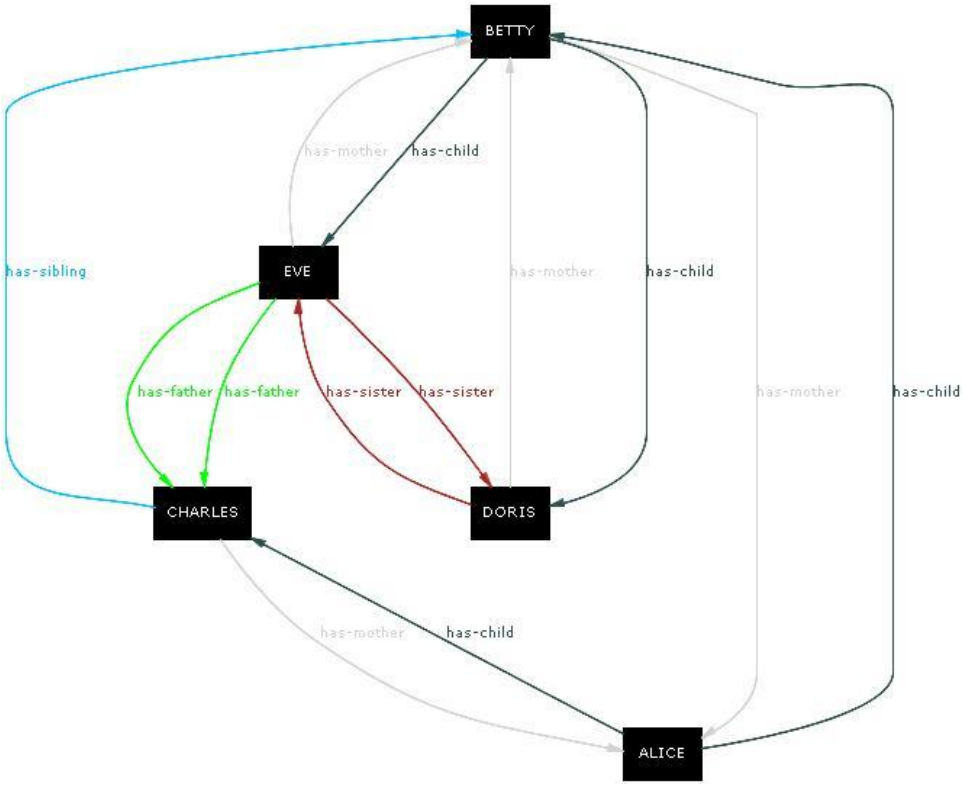

Figure 3.2: Abox graph.

The RACER system is an inference engine over documents which contain descriptive information written in RDF, DAML, Racer and other descriptive technologies. It can reason over the ontologies found in its knowledge base.

Knowledge Base

As description logic reasoner the knowledge base of a Racer server is composed of Tboxes and Aboxes.

A *Tbox* represents an ontology where a group of concepts and roles (which represent the relationships between the concepts) for a specific domain are modeled. In a Racer server, the concepts of a Tbox can have several axioms, allowing to built knowledge upon already existing knowledge. An example of an ontology well know around the Racer documentation [9] is the family ontology Tbox, where for example, the relationships between the concepts sister and mother, man and woman, etc. are defined.

Instances of concepts with respect to a Tbox can be created in the so called *Aboxes*, see figure 3.2, in these Aboxes each instance is mapped to a concept, this means that different names refer to different individuals, such a characteristic is called UNA (Unique Name Assumption).

The distribution of the knowledge base is such that for every Abox file a related Tbox file must be found, an Abox can refer only to one Tbox, while

a Tbox can have many related Aboxes. In case that the Abox and its corresponding Tbox are separated in different files, a reference must be declared, another possibility is that the Abox file can directly contain the Tbox description.

Inference Services

Being an inference engine, a RACER server, thus obtains a query as input and sends back the corresponding result. This is reflected in its web service interface where only one operation is offered, requiring one string as parameter containing the query command and giving back a string containing the corresponding result. There are different inference services offered for both Tboxes and Aboxes, they can be classify as follows:

- Satisfiability: To check consistency/coherence of a concept in a Tbox, while for an Abox, it is checked if the instance assertions are satisfiable with respect to other individuals.
- Subsumption: The hierarchy level of a concept in the taxonomy of a Tbox can be obtained, while for an Abox, it is possible to know if a concept subsumes an instance.
- Realization: The concepts in the Tbox are classified according to the hierarchy in the taxonomy from the most general to the most specific, the same is applied for the individuals of an Abox.
- Concepts and roles: For an Abox, instances of a inquired concept or role are obtained.

The query services for Aboxes are relevant for this project.

Querying Process

When a query is received for its execution, the server will try to obtain the information from the Tbox/Abox considered as current, therefore, the next server states can be found when a query arrives:

1. The server considers the default knowledge base as current:
   The default knowledge base[5] is automatically loaded when the server is started and will be considered as current as long as no other KB is loaded. The default KB consists of a pair of an empty Tbox and Abox. When such a status is found, the answer to any query will be a "NIL" string. NIL is an answer given back when the server recognizes the absence of the requested information, in this case there is no information in the default KB, but it can also be the case that the KB do contains information, still the concept or role is not defined, it is absent, so the server interprets absence of information as something that can't be proved, this is because of the adopted OWA[6] concept, which says that something that cannot

[5]Knowledge base will be hereafter referred as "KB"
[6]Open-World-Assumption

be proven to be true is not believed to be false. This is why the name of the related Tbox/Abox should be specified together with the query to warranty a consistent answer.

2. The current KB is not the desired one:
   When the current Abox is another than the desired one, but still is based on the same Tbox, after a concept or role-like enquiry, the answer will be misleading.

3. The desired KB is considered as current:
   According to the query, a result to the desired KB can return either the enquired tuples or the "NIL" string.

It is recommended to specify the name of the required Abox together with the query command, such that misleading answers are avoided and the previously described server states do not interfere.

The very first time that an Abox is queried, index structures for the Abox are build, which is a procedures that can take some time, since the related Tbox needs to be first classify and then the realization process of an Abox can take place. The Abox realization has the purpose of computing the concept names for all the individuals, this is an optimization step which helps future queries to be answered much faster. The information in table A.1, see appendix A.1, shows an example relation between indexing time and the size of an Abox file. But once the index structures are build, the following queries to the same Abox will use the same index structures, thus the response time is clearly faster.

If one has the choice between one server that has the needed Abox with the index structures already computed, against a second one who has no index structures for the same Abox, it is clear that the first option is preferred. Thus this is one criteria that can be consider for the optimization of our composition, hereafter referred as Abox indexation.

The Query Repository

A Racer server can be configured according to different processing modes, when the query repository also called Qbox is activated, then besides executing the query, the server will also classify the query in a subsumption-like hierarchy, such that for the set of queries that have been sent to an Abox, the ancestors of the query, which represent its most general subsumees and successors, which represent its more specific subsumers are computed.
This classification process is based on the *object vector* of the query, thus a query is composed of head and body, for example for the next query:

(retrieve (?x) (and (?x woman) (?y person)))
The head is: (?x)
The body is: (and (?x woman) (?y person))

Each query has an associated object vector of variables and individuals, which are obtained from the body of the query. For example:

For the query body: (and (?y woman) (?a man))
The object vector of variables is: (?a, ?y)
The object vector of individuals is: (man, woman)

The vector specifies how the answer tuples are internally formatted in a lexicographic order, which is later reordered according to the given head of the query, to present the answer. Thus the internal vectors are the ones used for the entailment comparison between two queries. For example, according to the next queries and vectors:

Query1: (retrieve (?x) (and (?x woman) (?y person)))
Vectors: (x,y) (woman,person)
Query2: (retrieve (?a) (and (?b woman) (?a person)))
Vectors: (a,b) (person,woman)
Query3: (retrieve (?a) (and (?b person) (?a woman)))
Vectors: (a,b) (woman,person)

Query3 and *Query1* entail each other because their internal vector is the same (woman,person). While *Query2* has a vector different to the others so it can not be entailed by any of the other queries. If the query bodies where different, then entailment applies between the most general and the most specific one. For example, according to the next queries and vectors:

Query1: (retrieve (?x) (?x mother)) Vectors: (x) (mother)
Query2: (retrieve (?a) (?a woman)) Vectors: (a) (woman)

Query1 entails *Query2*, but not the other way around.

Moreover, the Qbox feature maintains in memory the results of the executed queries over a specific Abox, this means that, once the query repository is enabled in the processing mode of the server, different query repositories for every accessed Abox will be created.

The cached information found in the Qbox can help for optimization purposes in a way that futures queries can totally or partially profit from the cached information, thus reducing computational effort of recomputing the total amount of information.

If we see the Qbox after a session of queries, the figure 3.3 shows an example of a Qbox for the Abox "University1". One can see that *Query-4* is the most specific subsumer of *Query-16*, while *Query-26* has an equivalent query (*Query-31*) but no subsumers or subsumees. As showed in figure 3.4, the answer of the forth inquiry (*Query-16*) to the University1 Abox, was partially computed, taking advantage of the information stored in *Query-11* and computing only the left information, corresponding to the highlighted area. Thus, a server that contains cached information that is useful for the present query can be considered as a good candidate. Since this information is obtained from the Qbox hierarchy, query subsumption is another criteria useful for optimization.

```
;;; QBOX FOR ABOX file://University1_0.owl
;;;
;;; 0:MASTER-TOP-QUERY
;;;  |---5:QUERY-26 = (QUERY-31)
;;;  |   \___0:MASTER-BOTTOM-QUERY
;;;  |---2:QUERY-8
;;;  |   \___0:MASTER-BOTTOM-QUERY
;;;  \___4:QUERY-16
;;;      \___3:QUERY-11
;;;          \___1:QUERY-4
;;;              \___0:MASTER-BOTTOM-QUERY
```

Figure 3.3: Qbox for Abox University1

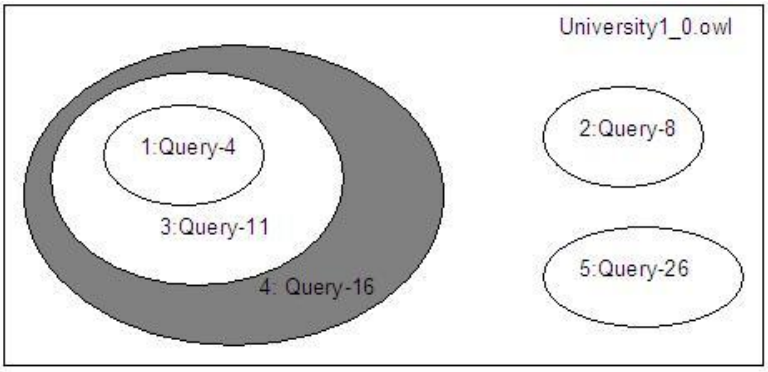

Figure 3.4: Distribution of cached information.

3.3 Conclusions

Considering the characteristics of the web service compositions to optimize in this project, it was decided that the first optimization steps could better profit when performance parameters from the application logic of the web service partners are considered, due to the fact that all of them offer the same functionality.

The first optimization steps consider Abox indexation and Query subsumption as criteria parameters to cope with the performance issues of the partner's functionality.

Chapter 4

Optimization Design

This chapter presents how the optimization of the web service composition can be considered as a configuration problem and the different kinds of optimization algorithms that were applied in trying to find the best choice suitable for our problem. Finally the adopted approach is presented.

4.1 Algorithms for Optimization

From the description given in section 2 about the kind of web service composition to be optimize in this project, it becomes clear that the aim of the optimization is to find for every invocation step in the composition a convenient web service among candidates such that it is possible to find a combination of them that satisfies certain criteria. This kind of problem is known as configuration design, which is described by [5] as following: "An artifact is said to be configured if it is made by combining objects that are chosen from a given finite set of generic components. The final configuration satisfies a given criteria, expressed either as constraints or objectives." The next problem description, illustrates this better:

> Suppose n WS partners $WS_1 \ldots WS_n$[1] required in a BPEL composition. For each and every one of those partners there are 0 until ∞ other WS offering the same functionality and containing the required information. For each WS there is a cost $c(WS_i)$ and a time unit representing the execution time $t(WS_i)$. Which WS fits best each partner, such that the combination of all partners minimizes the total time without exceeding a given budget? Consider the next problem instance:
>
> No. of WS partners in the composition n = 4
>
> For WS_1, X_1= 0 WS candidates are found
>
> For WS_2, X_2= 4 WS candidates are found

[1] They can be considered as a class of web service

For WS_3, X_3= 2 WS candidates are found

For WS_4, X_4= 6 WS candidates are found

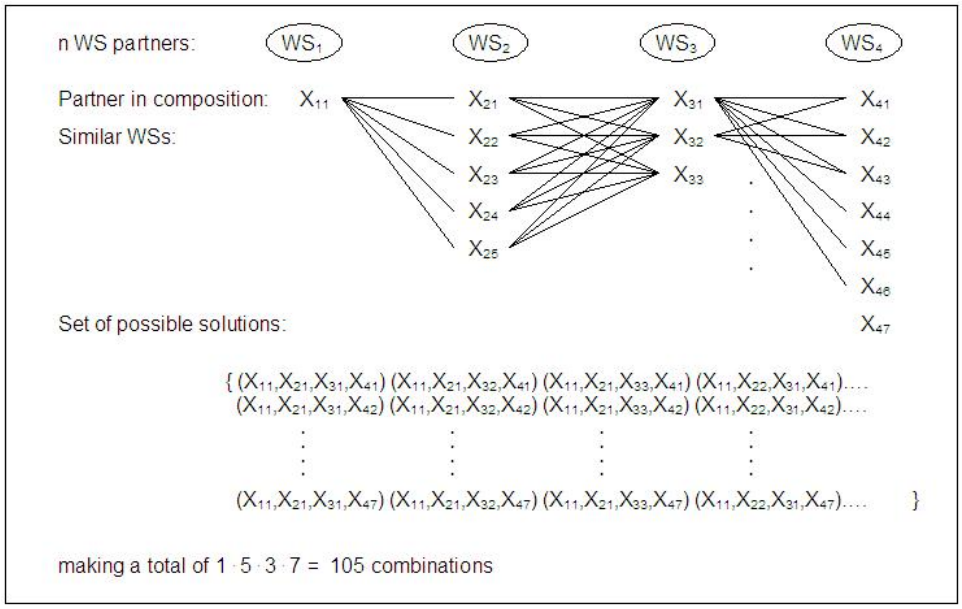

Figure 4.1: A complete enumeration approach.

This implies that the total of possible combinations can be expressed as:

$$\prod_{i=1}^{n} X_i + 1$$

As seen in figure 4.1, the number of possible combinations can dramatically increase according to the number of partners in the composition and the number of similar WS found for each of these partners, making the task of examining all possible solutions in order to find the best combination that fulfils the requirements, unpractical and most important, time consuming. For these reason it is reasonable not to solve the problem by a complete enumeration approach, thus it is necessary to find a solution that reduces search efforts.

There has been extensive research in the area of algorithmics, to find clever methods that reduce the size of the search space. Since this is a broad field, a solution starting from fundamental algorithm design techniques was intended to be found. For a better understanding of algorithmics you can refer to [10].

4.1.1 Fundamental Design Techniques

A Combination of Backtracking and Greedy Methods

To find an optimization strategy for this combinatorial problem, it was decided to look into some general techniques that are considered as fundamental because they provide efficient algorithms for different kinds of problems, two

of these techniques are the Backtracking and the Greedy methods, that can be used for solving optimization problems.

A *Backtracking* method carries out a systematic search on top of a structure, the most commonly used structure is a tree which is considered as an implicit directed graph with no cycles. With implicit graph is meant, that every node in the tree is being constructed during the search progress. A search cycle starts from the root of the tree towards a leaf, a leaf represents a feasible solution and the end of the cycle. If during the cycle the solution is found unsatisfiable, the edges used in the algorithm are discarded and a new search cycle begins.

A *Greedy* method refers to one that for each search step the most convenient parameter is considered according to the specifications of a desired solution. This means the most convenient local parameter is chosen regardless of the situation that can come in the future.

Thus, combining this two methods an optimization algorithm was found, which can be better illustrated with an example. Consider the afore given problem instance, where a composition requires four web service partners. As depicted in figure 4.2[2] each candidate web service is given a pair of criteria, time and cost with their respective values. The specifications of the desired solution is to minimize the time, while a budget less or equal than 14 is not exceeded. For every step in the search iteration, the edge with the lowest time was considered as the most convenient, and the cost of every edge was added. The first iteration fails, because the budget was exceeded with a cost value of 20. The edges participating in this iteration are discarded[3]. The next cycle starts with a new set of edges and the feasible solution is found.

Even though, the answer was found in the second step, this is still an exhaustive search process which in the worst case, it would have to evaluate all the non-eliminated nodes for each step. This could happen for example, when the given budget constraint is too low to discover that no combination is possible to fulfill it.

The given example in figure 4.2 represents only one instance of a composition, but in this project, it is intended to find a suitable algorithm for compositions with different input sizes, so it should be possible to cope with problems of larger input sizes and still maintain a reasonable computational cost.

4.1.2 A Heuristic Approach

An alternative solution are the heuristic techniques which are part of the combinatorial optimization field and are suitable to approach the so called hard problems. Hard problems represent any kind of problem with an exponential complexity that could take much time to be solved by deterministic exhaustive

[2]All the edges in the tree are expanded for illustration purposes, but the tree is still implicitly considered.

[3]For additional efficiency of the algorithm, edges considered poorer than the discarded ones could also be discarded.

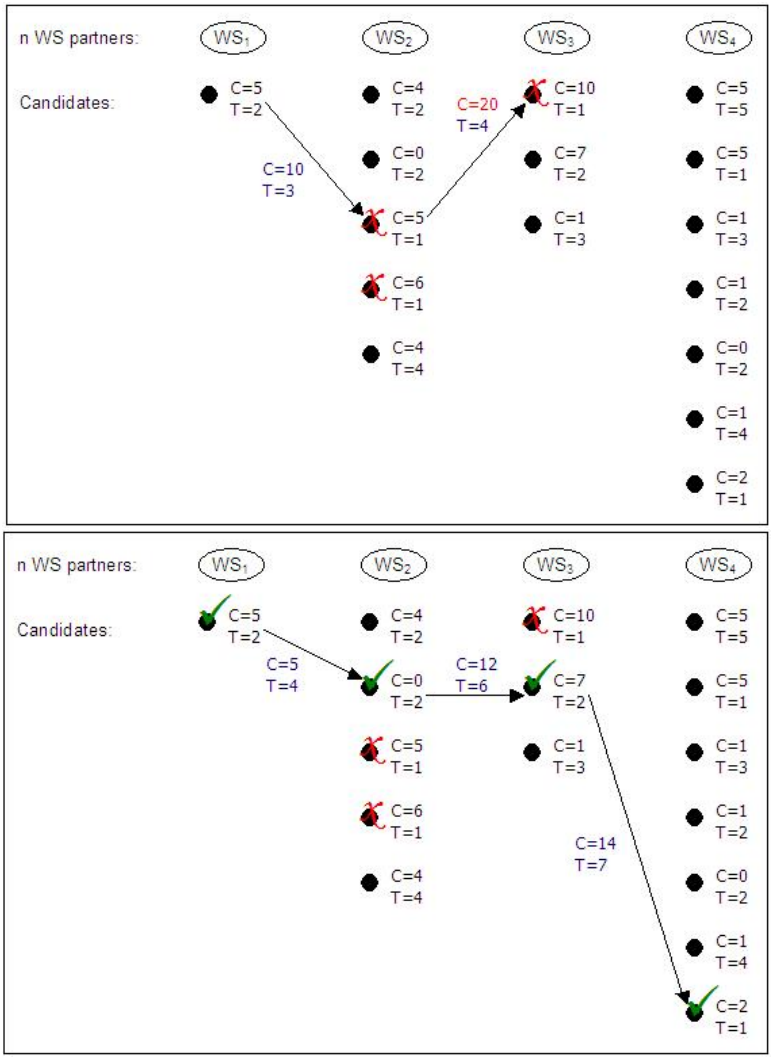

Figure 4.2: Greedy-Backtracking method

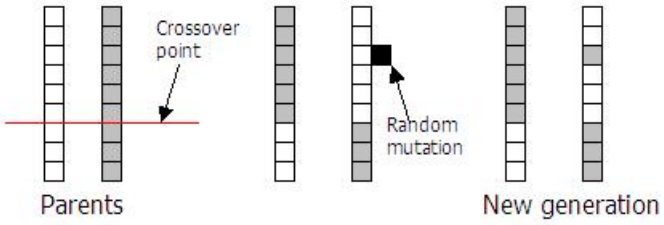

Figure 4.3: Genetic Algorithm: Crossover and Mutation

search programs, when considering a realistic input size. The main characteristic of heuristics algorithms in general, is that they find solutions among the universe of all possible solutions, without any guarantee that the best will be found. But they can find solutions close to the best one, in this sense they are approximating algorithms.

Several heuristic techniques are inspired by nature, biology and physics, for example the *Ant Colony Optimization* or *ACO* [6] inspired by the observation of how ants find the shortest path to their food source and why all of them follow the same path which appear to require much coordination. It was discovered that ants leave pheromone trails on their way, such that the following ants will use the same path by following the pheromones. If by chance the larger path was first chosen and not large enough to give time to the pheromone to evaporate, the following ants will come the same trail laying out more pheromone, thus maintaining a mediocre path. Still this is not always the case and has helped computer scientists in finding new optimization solutions, by using weight or "artificial pheromones" for each possible path with an artificial degree of evaporation, such that bad paths (paths of larger size) will evaporate first and good paths (paths of smaller size) will maintain a higher degree of pheromones according to their size.

Genetic Algorithm

For this project the Genetic Algorithm was also applied, aiming to find a suitable algorithm for input sizes of bigger dimensions.

The *Genetic Algorithm* simulates the natural optimization process found in evolution where the sequential recombination of chromosomes (DNA sequences) produce new generations from which the bests are chosen to be recombined again, thus producing in each iteration a better generation.

To simulate this process the population or the artificial chromosomes are represented with vectors or strings. Two chromosomes representing the parents are needed to create two new individuals, for these process the *crossover* operation is used, see figure 4.3, in which parts of the chromosomes are exchanged, thus producing a new generation with characteristics of both parents[4]. Then, the *mutation* takes place, in which random genes of the new chromosomes are mutated. The so called *fitness value* determines if a chro-

[4]How parts of chromosomes are mixed together is fixed by the designer of the algorithm.

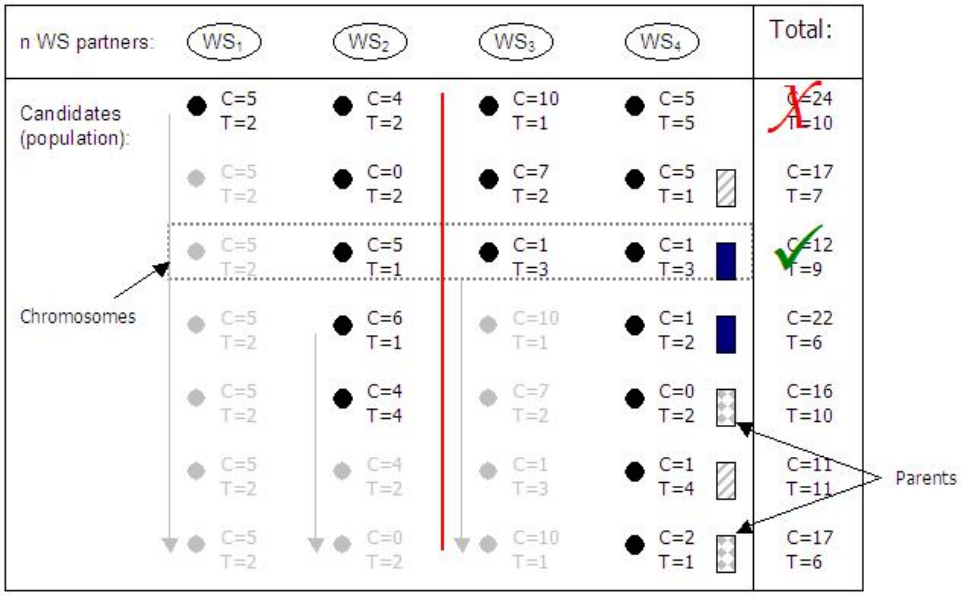

Figure 4.4: Population of Genetic Algorithm for a problem instance

mosome is kept or discarded for further reproduction, thus generations are compared against the fitness value and depending of their value, they are discarded or chosen to be parents. For a more detailed documentation of this algorithm you can refer to [10].

To explain how this algorithm suits our optimization problem, lets look at an example, for which we take the same problem instance given in section 4.1.1. First we need to find the population, as observed in figure 4.4 each web service partner has a different number of web service candidates. To produce chromosomes of the same size for a proper crossover, the candidates for the same partner are duplicated in the rest of the column. A cycle would then progress as follows:

1. Fitness value check: With a fitness value of cost less or equal to 14 and a minimum time, the best chromosome according to its fitness value, is kept for the next generation, while the chromosome with the worst fitness value is not considered for crossover.

2. Crossover: The crossover could only then take place in such a way, that only the candidates for the same web service are mixed with each other, from the figure 4.4 the crossover is taking place when the chromosomes are considered horizontally. The crossover point is taken randomly.

3. Mutation: Each new chromosome, see figure 4.5, is exposed to a random but low number of mutations, where the value of the mutation can only take a value from a random candidate from the same group of candidates (column) of their respective web service partner. The lowest number of

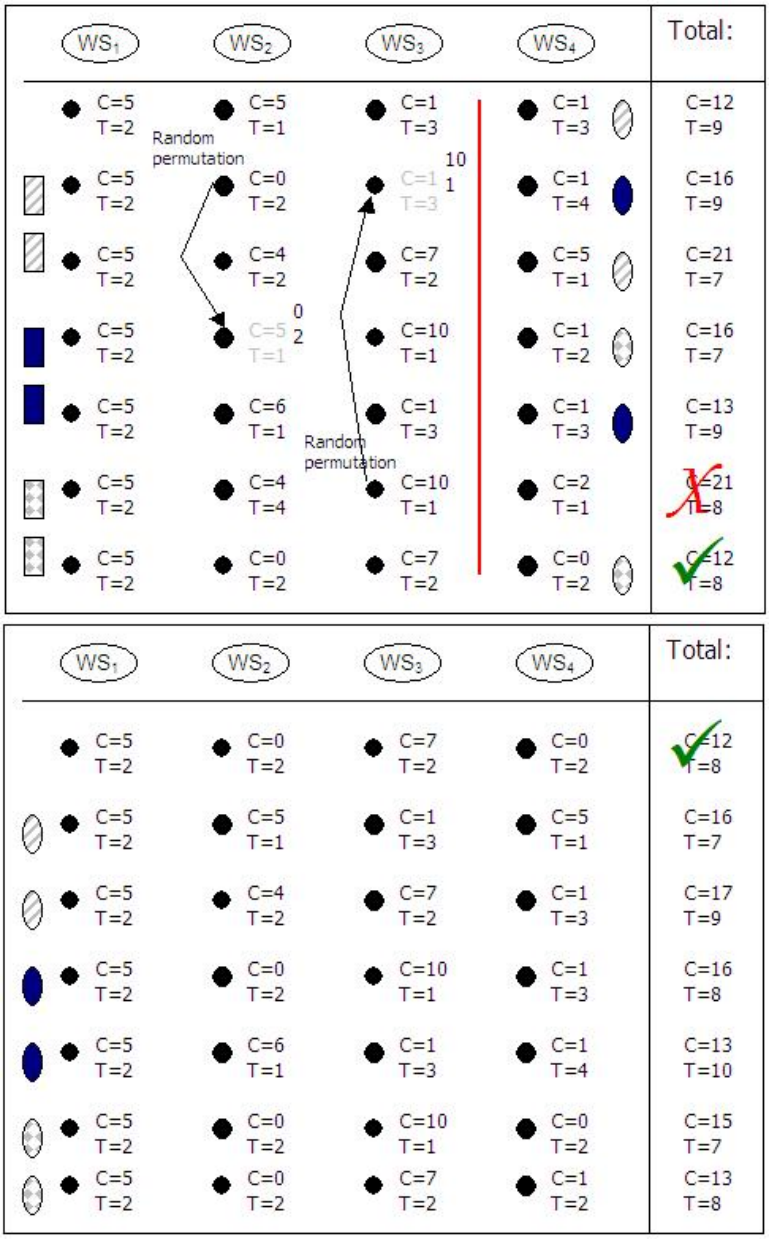

Figure 4.5: Crossover and mutation for a problem instance

permutations the better, to avoid falling into a pure random search, the experts[5] recommend 0.5% to 1% of permutation.

For choosing the parents, the so-called steady-state selection was applied, such that elitism is used, to avoid loosing the best present chromosome for the next generation. Together with a rank-selection, according to their fitness value. More than a couple of parents was chosen other wise by choosing only the best two parents for their crossover in each cycle, prunes out chromosomes that could lead for better solutions, the number of parents for crossover can be chosen by percentage in concordance with the size of the population, a percentage between 60% and 80% has been recommended. This cycle can repeat until a number of populations are reached.

4.1.3 Adopted Approach

The result provided by a robust optimization algorithm is a solution that satisfies a set of criteria. Criteria can be divided in constraints and objectives, where constraints are expressed as equalities or inequalities and objectives are expressed as the maximum or the minimum possible value. As a consequence, when too many constraints are applied, it is possible that no solution can be found that satisfies all of them, while objectives are more flexible.

The previously presented algorithms solved the given problem instances in the examples, while considering only a pair of criteria. Still, this project intents to cope with criteria to be added in the future, thus it should be possible to handle multiple objectives and multiple constraints.

The solutions presented before in section 4.1.1 and 4.1.2 can easily cope with a single objective, but when considering multiple objectives, with no total order, then it is better to consider an approach such as the one proposed by [5], namely the $PO - A^*$ algorithm. With no total order is meant, that the objectives are not mutually comparable and there is no factor to convert all the criteria into a single metric. The $PO - A^*$ algorithm gets its name from the combination of the next two algorithms.

1. The A^*[12] is a single-objective search algorithm that guides the exploration on top of a tree and uses the next evaluation function:

$$f^*(n) = g^*(n) + h^*(n)$$

This evaluation function determines the goodness of a node in the tree, where $g^*(n)$ (see figure 4.6), refers to the cost from a given node s to a given node n, and h^*(n) refers to the optimal path from node n to the *goal*. Since this is a best-first algorithm, the value of g^*(n) will be optimal, and an optimal solution can be guaranteed, only if for the value h^*(n) optimistic nodes are being considered. Since this a single-objective function, there is a unique solution for the optimization expressed with

[5] http://cs.felk.cvut.cz/ xobitko/ga/

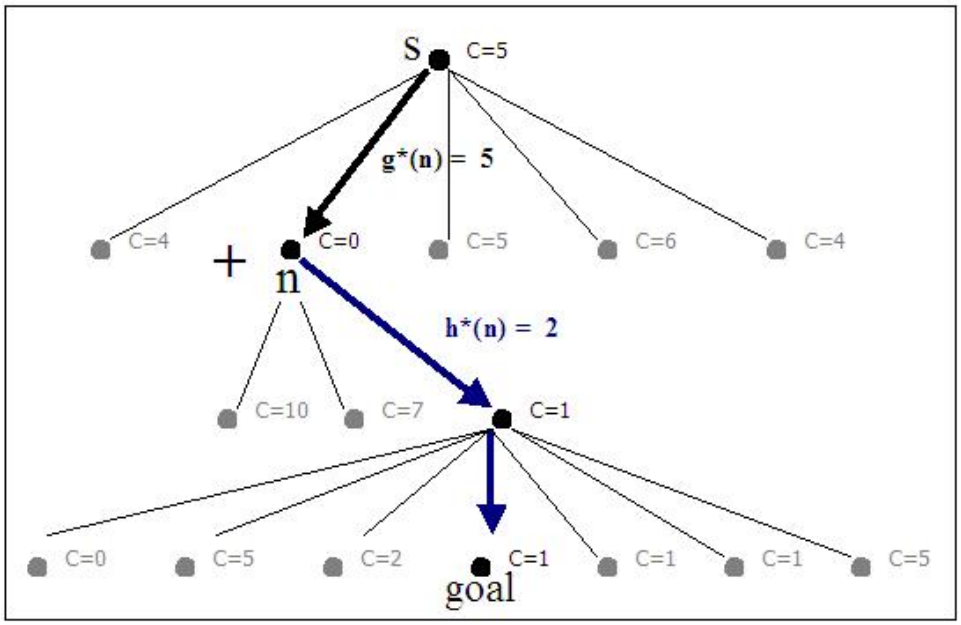

Figure 4.6: Evaluation function of A^* Algorithm

a scalar value. In figure 4.6 the unique optimal solution is $f^*(\mathrm{n}) = 5 + 2 = 7$, the value of $f^*(\mathrm{n})$ is then the total cost of an optimal path in the tree from node s to the *goal*, constraint to pass through node n[12].

2. The *Pareto-Optimal* is a multi-objective optimization algorithm, where the values of the objectives are expressed by dimensions in a graph, see figure 4.7 each vector is placed according to its value inside the dimensions. As a rule of thumb, a multi-objective problem has more than one solution, the vector *5* and *2* are pareto optimal because they dominate all the other vectors, this is the so called *pareto-optimal set* or *non-dominated set*.

   A vector in a multi-objective problem is characterized by the fact that it can't be increased in one of its objective functions without decreasing on some of the other objective functions, this behavior can be appreciated in figure 4.7, where vector *5* increases its objective towards A, while decreasing its objective function B. There is always a trade-off to choose from one solution or the other, so the decision can be made when priorities to each function are given.

   The *PO-A** algorithm is therefore the result of combining these two algorithms, so that it is possible to search into the tree and decide on the goodness of a node for multiple objectives.

   It can be observed that a pareto optimal strategy can be applied to other optimization algorithms when it comes to compare a set of decision variables with different objectives, for example, the pareto optimality can

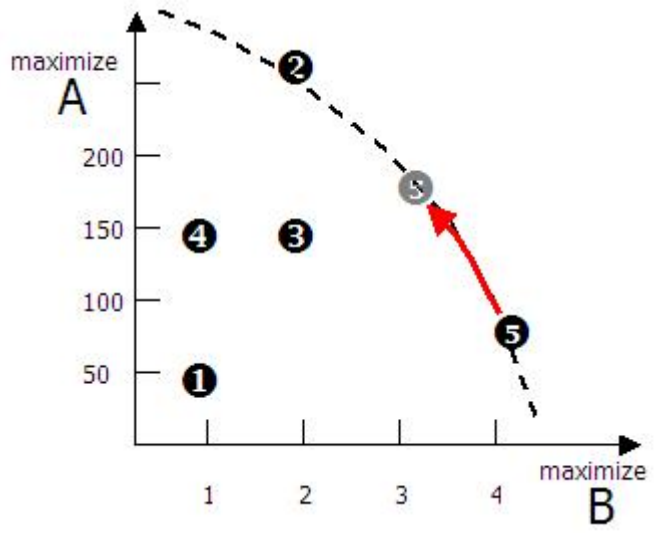

Figure 4.7: Pareto Optimal curve for two evaluation functions

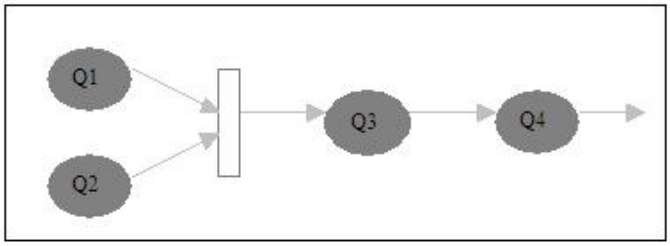

Figure 4.8: Composition of queries

determine the goodness of a chromosome in the genetic algorithm, such that non dominated chromosomes are considered for crossover.

The pareto optimality strategy was adopted to solve the multi-objective problem.

4.2 Candidate Solutions

This section describes two of the most relevant approaches considered to solve the configuration problem for a web service composition, using the criteria described in section 3.

4.2.1 Global Optimization

A layout language represents a set of relationships between the components of a design, in this project, the design is represented by the web service composition and its components are the queries to be sent as parameters when invoking a web service partner. To easily understand this section, a scenario is given.

Composition ID	Query ID	Abox	Predecessor	Successor	Subsumed by	Related query
1	Q1	Abox1	-	Q3	-	Q2
1	Q2	Abox1	-	Q3	-	Q1
1	Q3	Abox3	Q1,Q3	Q4	-	Q4
1	Q4	Abox3	Q3	-	Q3	Q3

Table 4.1: Layout language for a composition of queries (Scenario 1)

Web Service	Aboxes	Indexed Aboxes
S1	Abox 1	
S2	Abox 1, Abox 2	Abox 2
S3	Abox 3	
S4	Abox 3, Abox 2	Abox 3
S5	Abox 2	Abox 2

Table 4.2: States of web service partners (Scenario 1)

Scenario

Consider the composition in figure 4.8, four queries are involved, where Q_4 is subsumed by Q_3.

If the table 4.1 is used as the layout language[6] for the composition in figure 4.8, the next relationships are described:

- Composition ID: Represents the ID of the composition to which all instances of that composition belong to.
- Query ID: An ID that identifies the query inside of the composition.
- Abox: The enquired Abox
- Order of execution: The successor and predecessor of a query, helps to identify the order in which the queries are executed.
- Subsumption: Is the ID of a subsumee(s) of the query. This information should be known before instantiation.
- Related query: Queries in this rubric represent the queries that use the same Abox.

Now consider the status information given in table 4.2, where the candidate web service partners, the Aboxes they contain and the indexed Aboxes are listed. The ideal solution would assign the queries as follows:

- *Q1 - S1: Q1* is assigned to *S1* because the needed Abox is found there.
- *Q2 - S1: Q2* is assigned to *S1* because the previous query used the same Abox so it is indexed.
- *Q3 - S4: Q3* is assigned to *S4* because the needed Abox is indexed.

[6]The content of the layout language should be given by the designer of the composition.

- *Q4 - S4: Q4* is assigned to *S4* because its subsumee (*Q3*) was assigned to *S4*.

Comments

Making assignment decisions according to the relationship of a query with other past queries in the composition as the last solution showed, dramatically restricts the number of candidates, thus hindering the possibility of finding better solutions.

The ideal situation for a global optimization algorithm, is to consider the values of each criteria for each web service candidate, such that it is possible to play with the combination of values, as shown in sections 4.1.1 and 4.1.2.

The advantage of the algorithms in section 4.1 is that global optimization can be guaranteed.

The disadvantage is, that such an approach is suitable when only one instance of a composition is running at a time, thus only for one client composition. This is because the values of the chosen criteria, namely Abox indexation and level of query subsumption are dynamic, meaning that their value is constantly changing, so if other clients are accessing the web service candidates at the same time, the status of the servers change. Therefore, to obtain an accurate solution from a global optimization algorithm for such dynamic values, it is necessary that no other clients access the web service candidates, but this is not a realistic scenario, although it can be argued that the optimization solution is optimal according to the status of the web service candidates found in the past at designation time, but by the time a composition has finished executing its last query, the status of the servers will be different.

This project focuses on a solution that could serve multiple clients, as explained in the next section.

4.2.2 Local Optimization

As explained in section 4.2.1, having a global knowledge of the composition is useful when the constraints have a constant value or at least a value that will remain unchanged during the lifecycle of the composition, such that, with the use of a heuristic, it is possible to find in advance an optimal combination of resources (web service partners) to be assigned to each invocation step in the composition. But, when the value of the criteria is constantly changing, it is better to look for the proper web service only until run time, this has two important reasons, which are closely related to the criteria that this project considers as first steps for optimization, namely Abox indexation and query subsumption. Both criteria are affected by the changes in the state of the server, caused by the execution of queries for other clients. Figure 4.9 shows these two reasons:

1. By the time the composition invokes a partner, the Racer server has most probably changed its state, because other client's queries can arrive at any time in between changing the state of the server. Thus, the changes

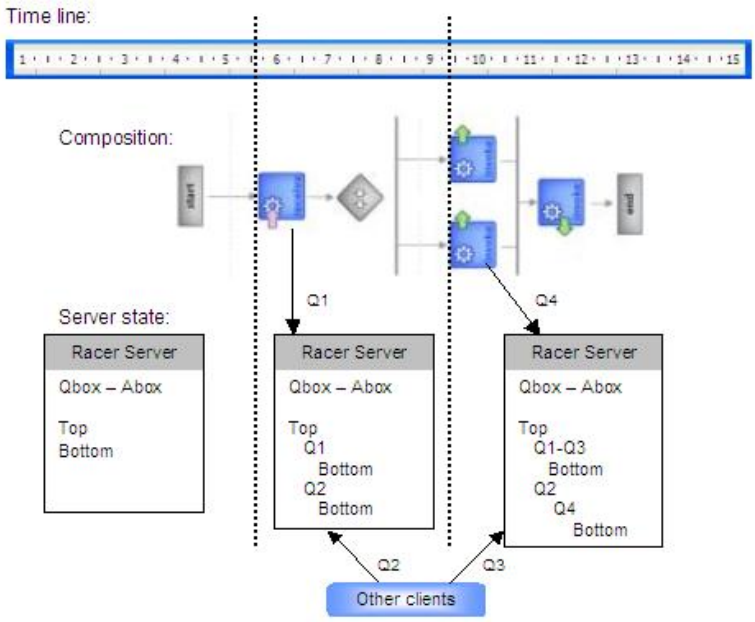

Figure 4.9: Changes of the server's state through time

are reflected in the state of the Qboxes that cache results of the executed queries, increasing the possibility to find a subsumee for the present query. Furthermore, a query that has been executed on an Abox for the first time, causes the Abox to be indexed. In this way, the more queries the server executes the more possibilities there are to find a convenient server state with respect to query subsumption and Abox indexing as criteria.

2. The name of the inquired Abox is known only until some decisions in the business logic of the composition have been made during its runtime, thus the decision to find a server that contains the required Abox can only be made at run time.

These two reasons point out for a solution that looks for a local optimization when considering criteria, which values are constantly changing. Local optimization means that only possible web service resources are taken into consideration for the present query and no decisions are taken on behalf of future queries that are to come, since there is not a way (at least for now) to predict the future state of the servers, moreover, the knowledge about the characteristics of the composition discussed in section 4.2.1 are for the moment not useful for decision making.

Comments

This approach can serve to multiple clients, avoiding the bottleneck that can be caused by blocking the access to web service candidates used by a client

composition. Another advantage is that, before the end of its life cycle, the composition can profit from the changes in the state of the servers that are being produced by the inquiries of other clients.

The local optimization applies pareto optimality to compare the goodness of the web service candidates.

4.3 Conclusions

The pareto optimal algorithm was adopted to be able to cope with multiple objective functions.

It could be observed that other optimization algorithms that deal with only one objective function can be helpful to solve multi-objective problems when they are combined with the Pareto Optimal algorithm.

After analyzing the advantages and disadvantages of a global optimization approach against a local optimization, it was decided to go for a local optimization which copes with multiple clients, giving the possibility of improving the values for the criteria of Abox indexing and query subsumption produced by the inquiries of multiple clients before a composition reaches the end of its life cycle.

Chapter 5

Materials and Methods

Until this moment, only the strategy applied by the optimization algorithm has been discussed, but how the strategy will be technically applied is to be introduced in this chapter. Thus, the next aspects are presented: type of communication between the client and the web service partners, the different criteria categories that can be handled by the Locator, so as the processes to obtain their respective values and finally the way in which the pareto optimality for comparison of multiple objectives was implemented is described.

5.1 Locator's Design

5.1.1 Components

To implement the optimization strategy, as figure 5.1 shows, a web service middleware, hereafter referenced as Locator, will be active between the composition instances and the web service entities, to generate the proper decisions. These decisions are based according to the following information:

1. Query to execute
2. Abox: Represents the name of the Abox to be inquired.
3. Candidate web services: Represents a list of web service URIs for each server that the locator will consider as candidate.
4. Knowledge-Base: Represents a list of names of Tboxes and Aboxes that each server contains, since the knowledge base of every server is not identical. It is assumed that Tboxes and Aboxes with the same name, will contain the same information in each server.
5. Qboxes: Represents the Qboxes in every candidate server, reflecting the querying activity that an Abox has had until the present, indicating which information has been cached. Thus, it is from this information

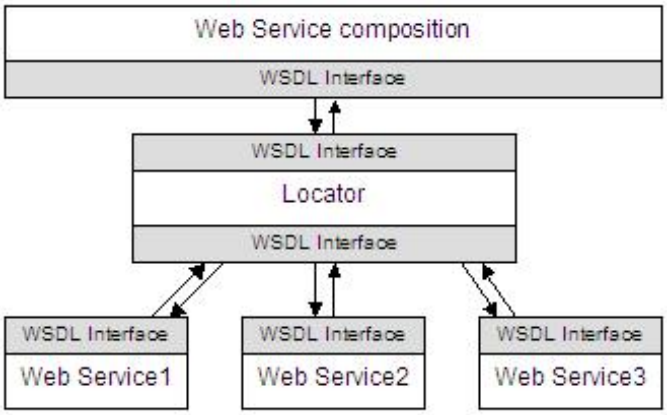

Figure 5.1: The Locator middleware

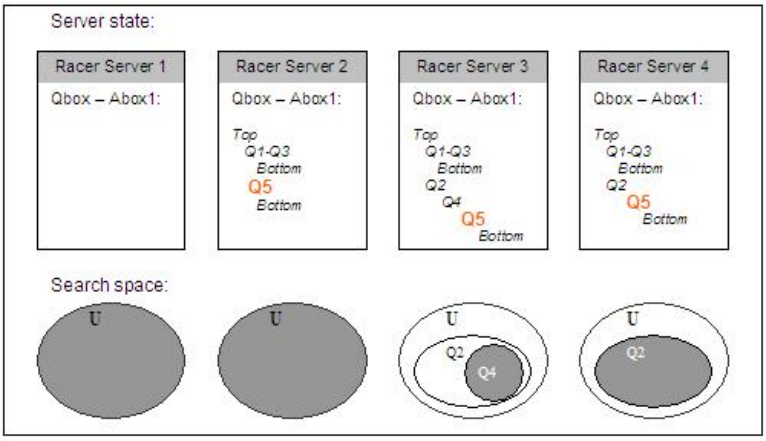

Figure 5.2: Qbox states per server

that the criteria for query subsumption and Abox indexing can obtain their respective values.

As explained in section 3.2 Abox indexing and query subsumption are the criteria towards which the composition is to be optimized, which can be expressed as constraints or objectives. While constraints are expressed as equalities or inequalities, objectives aim to maximize or minimize its value.

The Qbox

Knowing the state of the servers with respect to its Qboxes helps to know if there is any cached information that the present query subsumes. Thus, a server that has cached the needed information is considered more convenient than another with no subsumee.

To profit from the Qbox processing mode, the locator needs to know the state of every server with respect to its Qbox for each Abox. As the scenario in figure 5.2 shows, different servers have a different state with respect to

the Qbox of the same Abox ($Abox_1$), Q_5 represents the present query to be executed and the different places where it fits in the Qboxes.

- In server *1*, there is still no Qbox for $Abox_1$, this means, that no query has been executed for $Abox_1$, thus the $Abox_1$ in server *1* is not yet indexed.
- In server *2*, the parent of Q_5 in the Qbox hierarchy is *Top*, this means that there is no other query that this one subsumes, but only the entire universe.
- In server *3*, there are two ancestors (besides *Top*) to Q_5, while in server *4* only one ancestor is found, server *3* is then the most convenient server, since Q_5 does not only have a subsumee but its hierarchy in the Qbox is deeper than in server *4*, this means that the search space is smaller.

This example shows that considering query subsumption as criteria is expressed as an objective, such that the subsumption is maximized, according to the number of the query's ancestors. Moreover, knowing the existence of a Qbox in a server reflects that the Abox is indexed, thus Abox indexation is considered as a constraint where its values either resolve to true or false.

Obtaining Criteria Values

It is then clear that to obtain information about a possible subsumee of the present query, the Qbox information of every Abox in every server must be known to the Locator. Still, to obtain the Qbox information, the following requirements must be fulfilled to maintain the Locator's efficiency:

- The Qbox information should be found locally: Since the Qbox information is remotely located and the Locator can receive a high amount of queries to be executed, it is not desirable to ask the Qbox state to every server every time a query arrives, therefore this information should be found locally.
- Avoid duplicating Qbox's cached content: The cached information found in every server, should not be duplicated for the local Qbox knowledge, otherwise the Locator itself would be able to answer the queries.

To cope with these requirements, the proposed solution in figure 5.3 makes use of a Racer server on the side of the Locator that performs Qbox reasoning and helps to maintain a local copy of the status of each remote Qbox. First, the Locator inquires the local Racer server through the JRacer to classify the query against each Qbox. According to the classification, the proper server can be obtained. In this way the state of the Qbox in the destination server is locally reflected before it is actually changed in the remote server.

The local Racer server contains a copy of the Tboxes and empty Aboxes of every server, which are necessary to perform the Qbox reasoning. To accurately reflect the status of the remote servers, it is required that:

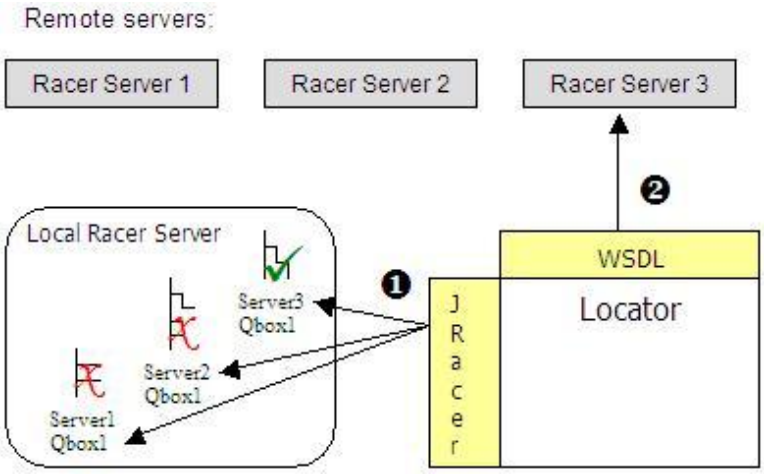

Figure 5.3: Local Racer server for reasoning.

1. Every query should be directed to the locator: For the locator to have a sufficient knowledge about the state of the servers, it is necessary that every client that intents to inquire the Racer servers, sends the query to the Locator, so that there is knowledge of every query that has reached a server, otherwise allowing the clients to directly access the servers will cause unnoticeable changes in their state. For this purpose and to make it transparent to the client, the locator will have the same interface as the Racer's web service, namely it will offer only one operation that receives a query string and gives back a result string.

2. The queries should specify the name of the Abox: Since the state of the servers can also change with respect to the current Tbox or Abox, then it is necessary that the client specifies the Abox to enquire, otherwise the query will be executed on top of the current Abox, thus giving a false answer if the Abox is not the correct one. The NRQL language allows specifying the Abox name, right after the query command, for example:

   (retrieve (?x) (?x top) :abox file://University1-0.owl)

   Specifying the Abox name right after the query, makes it unnecessary to send previous commands to load the required Abox before the query can be executed, which is a task that either the client or the Locator would have to take care of in order to assure the correct answer.

Pools

To help in the management of query requests and query answers, different query-pools are used such that there is a common place to obtain requests waiting to be assigned to a server, assigned queries waiting to be send to the web service partner and query answers waiting to be given back to the clients.

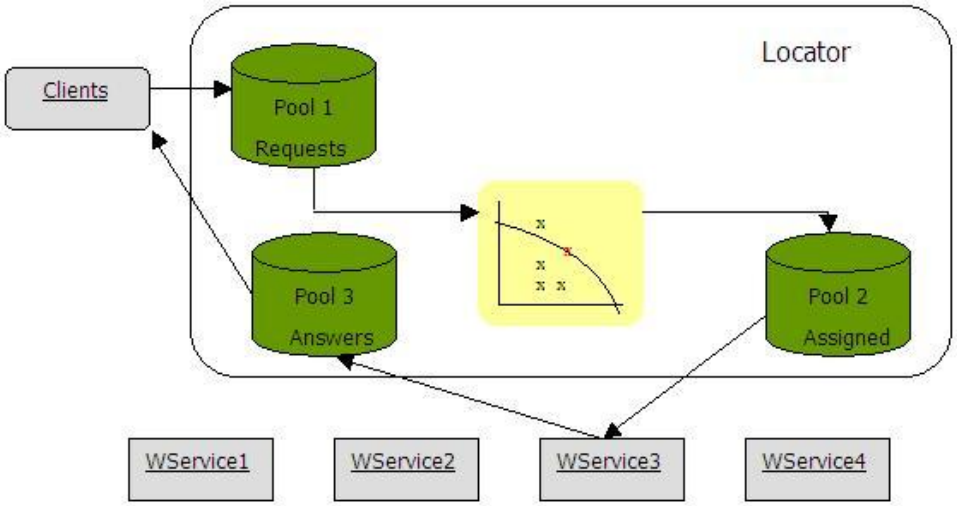

Figure 5.4: Locator's Pools

Thus to cope with multiple clients, the Locator uses three pools, see figure 5.4.

1. *Pool1* of requests: While the Locator is busy assigning a service to other requests, clients can store their requests in this pool, from which the Locator will pick them up to start the assigning process in a First-in First-out model.

2. *Pool2* of assigned queries: Afterwards, the queries assigned to a web service partner are stored in this second pool, which stores the queries that cannot yet be executed, because the web service partner is busy. Once their respective web service is available, they can be send for execution. Since the queries in *Pool1* were assigned in order of arrival, the optimization decisions were made also according to this order, in consequence queries in this pool are also picked up in a Fist-in First-out model to be sent to the corresponding service.

3. *Pool3* of answers: When the web service partner finished executing the query, the responses are kept in this pool until they are delivered back to their respective client.

Due to this design, it is possible to know the waiting queue of a web service partner, since this information is found in the pool *Pool2* of assigned queries, which contains requests that are only waiting to be executed. To take advantage of this facility, the waiting queue can be considered as another objective function to be included in the optimization, where the minimum is desired.

Comments

Thus, as previously explained, for the efficiency of the Locator's strategy the following components are used:

- One local Racer server: To avoid the inquiry to every remote server about their Qbox state and profit from its query classification reasoning without having to execute the present query in the remote servers.
- Knowledge Base: The query classification performed by the local Racer server, is done on top of Tboxes with empty Aboxes, to avoid duplication of remote information.
- Three pools: They are helpful to cope with multiple clients. Since the main task of the Locator is to find a suitable service to the present query, these pools help to maintain requests and answers while the Locator is busy finding the suitable service for a query.

5.1.2 Multiple Threads

To cope with multiple clients, the Locator is designed as a multithreading application. As shown in figure 5.5, it is divided in a front end and a back end. The front end is instantiated when a SOAP request reaches it, thus, the only method found in this class is published as web service, duplicating the interface of the web service partners. The main tasks of the Locator's front end are:

1. Interception of the client's request.
2. Delivery of the request to the Locator's back end, for its assignation to a convenient service.
3. Delivery of the answer back to the client.
4. Maintain a synchronous communication link with the client.

The back end of the Locator is a singleton class which aims to find the convenient service for the requests given by the front end, thus, it is activated every time that a front end hands a request to it. This class uses the following three threads:

1. Assign thread:
   Its aim is to assign all queries found in *Pool1* of requests, to a convenient server. After a service is assigned the query is transferred to *Pool2* of assigned queries. Thus, this thread is active as long as there are requests waiting to be assigned in *Pool1*.
2. Send thread:
   This thread is executed right after the first assigned query is found in *Pool2*, its aim is to supervise that there are no idle service partners that

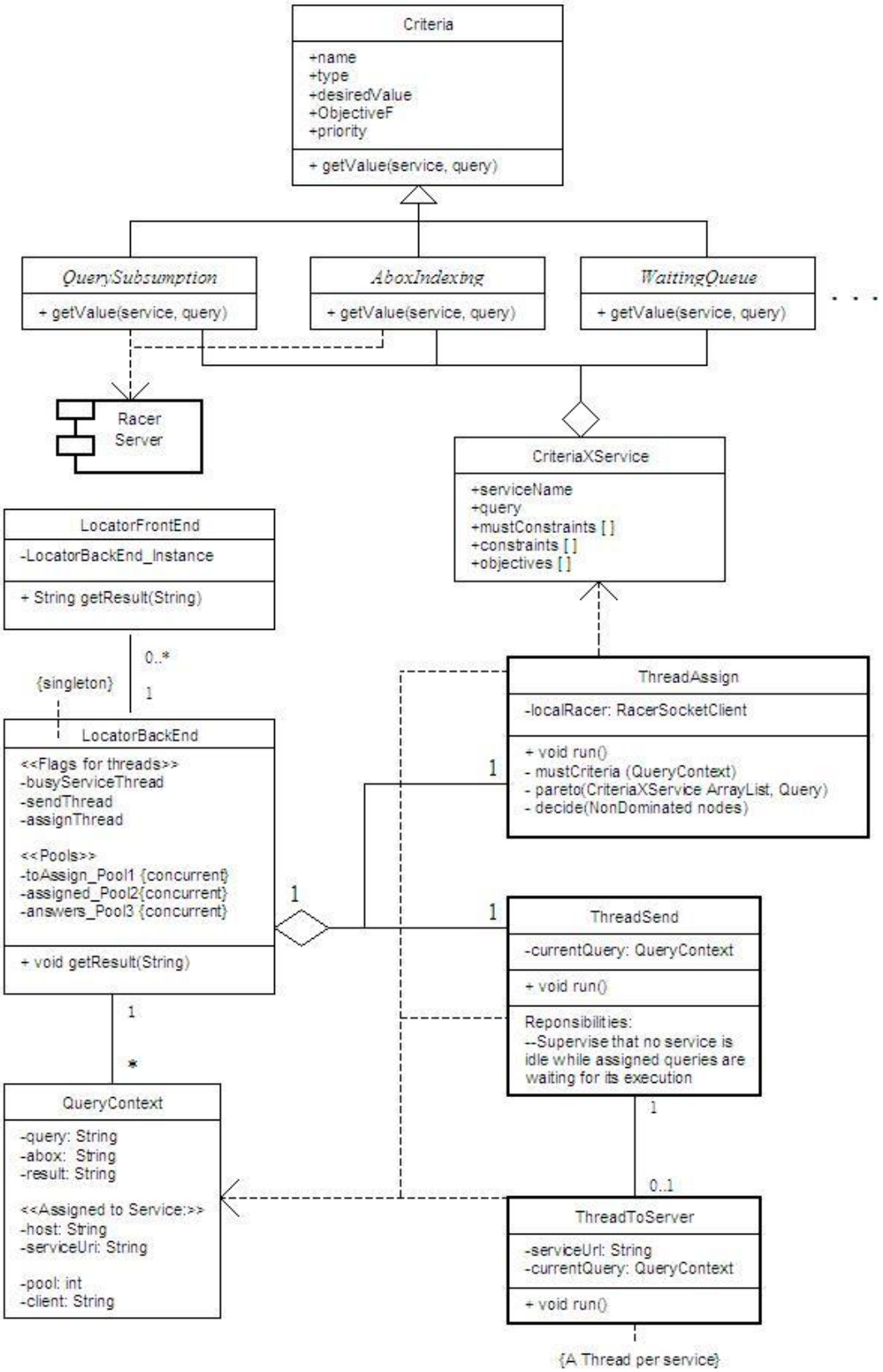

Figure 5.5: Simple class diagram

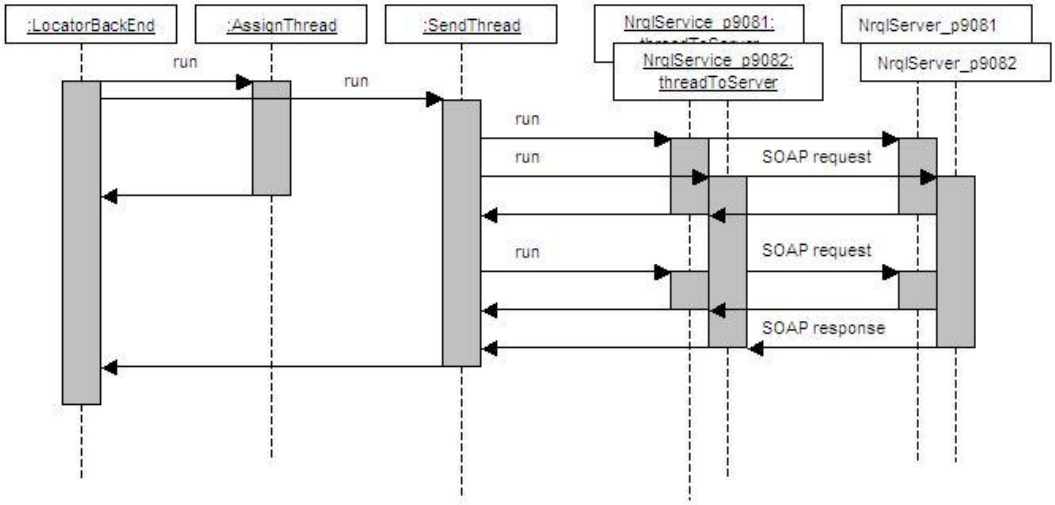

Figure 5.6: Sequence diagram

can solve the requests that are waiting to be executed. Thus, this thread has knowledge about the available service partners and their activity status, being busy or idle. Once a service is idle and there is a request that can be solve by it, this thread runs a third thread, called *Thread to Server*, which is following explained. Thus, this thread is active as long as there are waiting requests in *Pool2* and until there are no more active *Threads to server*.

3. Thread to server:
   This thread sends a SOAP request to a specific web service partner containing the query as parameter and maintaining a synchronous communication link. Once the answer is obtained, it is stored in *Pool3* and the thread is finished. There is one of this thread per service partner, in this way the Locator can control the activity of the web service partners by assigning a request at a time to each partner and maintaining the waiting requests in *Pool2*, such that it is possible to know how many request are waiting to be solved by a specific partner. Thus, the *Send thread* administers their instantiation and hinders the instantiation of more than one thread per service. Only if the query request was successfully answered, the registry of the request from *Pool2* will be deleted, otherwise, this thread will be created each time to try to send the request until the answer is successfully obtained.

Thus, as figure 5.6 shows, the back end can be considered as the main thread from which the previously described threads are launched and managed, so that no thread is idle while there are requests to be assigned and requests to send. The back end finishes its activity, once the 3 repositories are

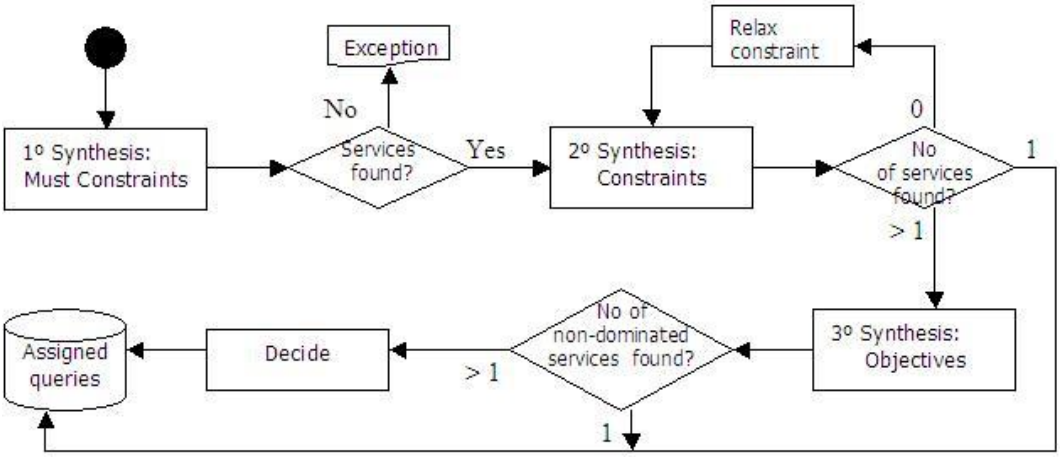

Figure 5.7: Assignment flow chart

left empty and the other threads have finished.

Every request, contains the query to execute as only parameter, once the request arrives at the Locator, the query is encapsulated in an object called *Query Context*, which aims to identify the query inside the Locator, such that the following information is known:

- Client: Represents an ID of the client that requested the execution of the query, to identify which of the answers found in *Pool3* correspond to a certain client.
- Pool: An ID of the repository in which the query is stored at the moment, thus, the three repositories store *Query context* objects.
- Host and Service URI: Represent the service that answered or will answer the query.
- Abox and Result of the query.

5.1.3 Service Assignment Logic

The previous sections have focused mainly in the communication of the Locator between the clients and the web service partners, but the strategy to find the convenient web service is to be explained in this section.

The strategy to find an appropriate service for a request is implemented in the *Assign Thread*. As the flow chart in figure 5.7 shows, it consists of three synthesis steps each of them corresponding to categories of criteria, thus the criteria are divided as follows:

- Must Constraints: Represent the set of minimum constraints which a web service partner must satisfy in order to give a response, for example that the service contains the Abox to inquire, or access rights are

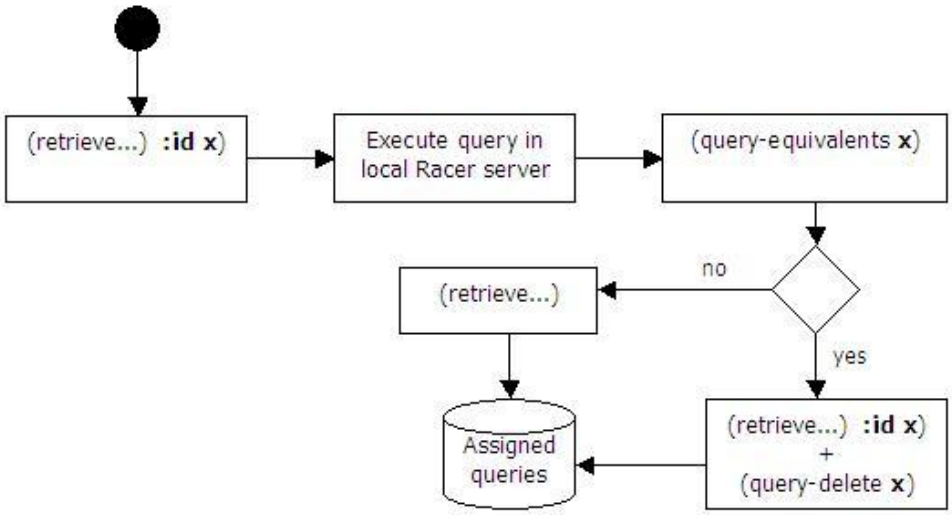

Figure 5.8: Memory management flow chart

satisfied, etc. If none of these constraints are satisfied, it is not possible to execute the query, thus this kind of constraints help to find partners that satisfy the minimum requirements to obtain an answer. The set of partners obtained after this synthesis are passed to the second synthesis stage.

- Constraints: Represent all the other constraints that are not part of the first category. This means, that their absence does not hinder the possibility of executing the request. Still it is not recommended to include too many constraints of this category, otherwise the web service partners might not satisfy all of them, thus decreasing the possibility of finding any candidates after this synthesis stage. If this is the case, the Locator will start to iterate in a cycle, which relaxes constraint by constraint until a candidate partner is found. In the worst case, the whole set of criteria is relaxed. The relaxation process is explained in section 5.1.4

- Objectives: Represent the criteria who's value needs to be maximized or minimized according to its objective function. In this synthesis stage, the values of the objectives are compared to obtain a set of non dominated candidates according to a Pareto algorithm, which is explained in section 5.1.5. If a set of candidates is obtained after this synthesis stage, a decision process chooses one of the candidates and assigns it to the request by writing its URI information to the *Query Context* object of the request. Finally the *Query Context* object is stored in *Pool2* of assigned queries, ready to be sent to the corresponding web service partner

Once a service has been assigned, a memory management procedure starts to check if the service has cached already an equivalent query for the same Abox, if this is true, a delete command is sent right after the request, with

the aim of deleting the cached information of the present request in the corresponding Qbox to avoid duplicated cached information. This is realized as figure 5.8 shows, by giving a name to the assigned query, afterward the query is executed in the corresponding Abox in the local Racer server. Thus, the local Racer server contains many empty Aboxes named after *Service URL + Abox name*. Afterwards, it is asked if the query with the given name has an equivalent query, if this is true, the query maintains its name and a delete command with the same name are stored in *Pool2* of assigned queries. While if there are no equivalents, then the query without any name is stored in *Pool2*.

The same procedure is done if it is a top like query, which duplicates the whole content of the Abox in the Qbox. To get to know if a query is a top like query, the first query atom of the request is obtained and kept in variable ***X=?z***, afterwards a string of the form *(retrieve(**X**)(**X**top))* is compared with the query request (with no spaces), if they are equivalent, then it is a top like query.

To manage the different categories of criteria, as shown in figure 5.5, every criteria that the Locator considers, should be implemented in its own class which must inherit from the *Criteria class* and should override the *getValue* method. Thus, it is considered that every criteria should have the next characteristics:

- A name.
- A type indicating the category of criteria that it belongs to.
- A desired value, if it is a constraint.
- An objective function, if it is an objective and to indicate the way a constraint should be relaxed.
- Priority, to support the order of the relaxation process.
- A specific method to obtain its value.

The class *CriteriaXService* is there to relate the different criteria and their value according to a specific service and request. Thus, the *Criteria* and *CriteriaXService* classes have the purpose of handling every criteria in a similar way and leaving only the implementation of the *getValue* method to the specific requirements of each criteria. For example, the objects of criteria *QuerySubsumption* and *AboxIndexing* make use of the JRacer API to communicate with a Racer server and obtain their respective values, while the *WaitingQueue* reads the waiting queries for a service found in *Pool2*, etc.

As the following pseudocode shows, the *Criteria* and *CriteriaXService* classes help to manage the different criteria in every synthesis stage in a dynamic way, independent of their peculiarities.

In first synthesis stage:
for every service **begin**
    for every *Must Constraint* in *CriteriaXService* **begin**
        value = *mustConstraint.getValue( );*
        if value != 0
            add to number of satisfied constraints
    end
    if all constraints where satisfied
        add service as candidate service for next stage
end

In second synthesis stage:
while no candidates for next stage are found **begin**
  for every candidate service **begin**
      for every *Constraint* in *CriteriaXService*
          value = *Constraint.getValue( );*
      if value != 0
          add to number of satisfied constraints
  end
  if all constraints where satisfied
    add service as candidate service for next stage
  if no candidate satisfied all constraints
    relax one constraint
end

In third synthesis stage:
for every candidate service **begin**
  for every *Objective* in *CriteriaXService*
    value = *Objectives.getValue( );*
  add *CriteriaXService* object to set of Pareto nodes
end

Still there is a restriction that every criteria must satisfy, namely the *getValue* method must return an integer value, where **0** means "not satisfied". Since it is not possible to expect that the value of every criteria is numeric, then it is left to the implementation of the criteria class to decide on a metric that will represent the kind of values it can get, for example:

> The criteria *Indexing* has a boolean value, where false is considered as not indexed, then the metric rule of that criteria can be:
> false = 0
> true = 1 (or another integer)

Thus, numeric values are required so that they can be compared during the third synthesis stage of objectives.

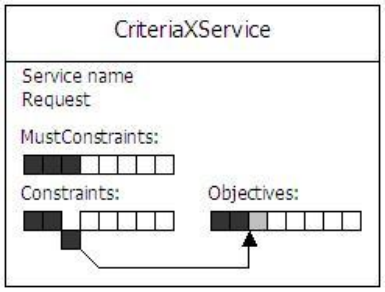

Figure 5.9: Constraint relaxation

In this way, new criteria can be added to the Locator by creating its own class and including it in the *CriteriaXService* class as a *mustConstraint*, *Constraint* or *Objective*.

5.1.4 Constraint Relaxation

As described in the previous section, constraints are relaxed during the second synthesis stage. The relaxation process takes place when at the end of the synthesis stage, there were no candidates found that satisfied all the constraints. From the set of constraints the one with the least priority will be chosen for relaxation, thus the order of relaxation in each iteration goes from the least important to the most important according to the priority attribute found in the criteria object. The process consists in converting the constraint into an objective. Thus as figure 5.9 shows, in the *CriteriaXService* object used for the classification of criteria in different categories, the desired criteria is moved from the category constraints to the category objectives. In this way, during the third synthesis stage the *CriteriaXService* object of every service will consider also as objectives, all the constraints relaxed in the previous synthesis stage. This procedure aims to find candidates to be considered in the third stage and still compare the service against the criteria now relaxed to an objective, thus the search still optimizes against the relaxed constraint.

For this reason every criteria should inherit from *Criteria* class and specify an objective function independent of the category that they initially belong to.

Even though candidate services can be found after the second stage without the need of relaxing criteria, the services that were discarded in this stage could be considered as non-dominated in the third stage where other criteria come into consideration. Moreover some criteria could block other services to ever be considered in the third stage, for example, consider the following case:

Constraints : Indexing
Objectives : Subsumption Level, Waiting queue
State of the servers: No Aboxes are indexed
Enquired abox : Abox1

1. For the first request the Locator relaxes criteria *Indexing*,

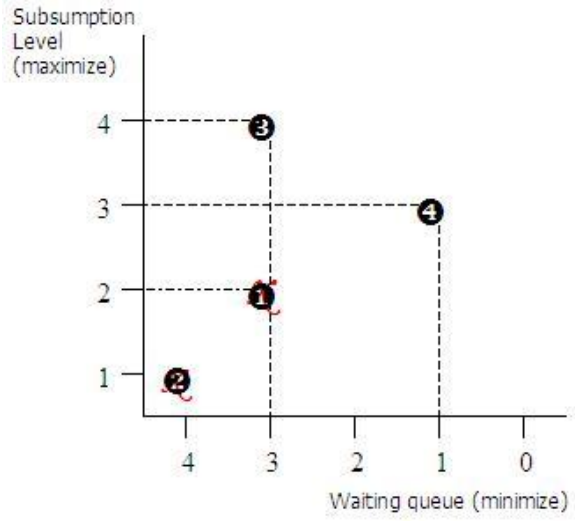

Figure 5.10: Obtaining non-dominated nodes

since non of the services have the abox *Abox1* indexed, consequently all services are considered in the third stage and one service *Service1* is chosen.

2. The second request inquires the same abox *Abox1*, consequently only *Service1* is considered after the second stage and as long as no other Abox is enquired, the same service *Service1* will be chosen, thus the waiting queue for *Service1* can easily increase, while the other services are left idle.

5.1.5 Pareto

For the comparison of multiple objectives, the third synthesis stage finds a set of non-dominated services based on the Pareto curve as explained in section 4.1.3. To understand how the non-dominated services are obtained an example for 2 dimensions is presented in figure 5.10. Four services are considered where their value for criteria *Waiting queue* is to be minimized and criteria *Subsumption level* is to be maximized then:

Service 1 dominates *Service 2*
Service 2 dominates none
Service 3 dominates *Service 1* and *Service 2*
Service 4 dominates *Service 1* and *Service 2*
Service 3 and *Service 4* are non-dominated

Thus, the non-dominated nodes are obtained as the following pseudocode shows:

for every service to compare i **begin**
$current = service_i$

```
        for every service to compare i+1 begin
            compare = service_{i+1}
            for every objective begin
              if objective function is minimize
                 if current.objective.value <= compare.objective.value
                    current dominates
              if objective function is maximize
                 if current.objective.value >= compare.objective.value
                    current dominates
            end
        end
        if current dominates compare in every objective
            add compare to set of dominated nodes
            delete compare from set of services to compare
    end
```

After the third stage, if more than one non-dominated nodes is found, a decision process should follow to pick one of the set of non-dominated nodes. In this case the decision process consist in randomly choosing one of the set and finally assigning the request to the selected service (non-dominated node), such that the request can be stored in *Pool2* of assigned requests.

5.1.6 Summary

- The Locator was implemented as a middleware between the client compositions and the web service partners. It is a multithreading application divided in front-end and a singleton back-end, where the front-end is exposed as a web service having the same interface as the web service partners and helps to maintain a synchronous communication link with the clients (compositions).

- The back-end is a singleton which manages the activity of three threads, namely the *Assign thread*, where the main strategy of the Locator for assigning a service to a request is found, and two other threads that are in charge of the communication with the web service partners, namely the *Send thread* which manages and launches the *Thread to server* threads which maintain a synchronous communication link with the web service partners.

- The assignment strategy is divided in three synthesis steps, each one corresponding to a category of criteria. Thus, the first stage finds services which satisfy the *Must Constraints* representing the set of minimum constraints which a web service partner must satisfy in order to give a response. The second stage verifies the fulfillment of a set of *Constraint* and finally the third stage is for the comparison of multiple objectives.

- For the comparison of multiple objectives with no total order the Pareto optimality approach is used.

- Three criteria were considered for optimization, namely the query *Subsumption level*, the *Abox indexing* which both of them make use of a locally found Racer server to obtain its respective values, and the *Waiting queue* criteria which obtains its value from the queries waiting to be executed in *Pool2* of assigned queries.

- Every criteria used by the locator should be implemented in its own class, with the next conditions:

  - It should inherit from *Criteria* class.
  - It should override the *getValue* method.
  - It should implement a metric rule such that the values of the criteria are returned as integers in the *getValue* method, where the value **0** is considered as not satisfied.
  - Every new criteria should assign a value to all the attributes inherited from the *Criteria* class, regardless of the category they belong to.

- Every criteria can be considered under any of the categories *Must Constraints, Constraints* or *Objectives*, this is defined in the *CriteriaXService* class, where the criteria object is assigned to a vector of a specific category.

- It is recommended to classify the least number of criteria under the category *Constraints* to avoid pruning off service candidates that could turn out to be non-dominated in the third synthesis stage where other criteria objectives come into consideration.

Chapter 6

Results

6.1 Locator's Behavior

In this section the behavior of the Locator is being presented through experimental results. Relevant examples are presented together with the status of the web service partners and their respective Qboxes, before and after, such that it is possible to observe the decisions made by the Locator. In the examples, four services were used containing different Aboxes as the following relation shows:

- Service name: NrqlService_p9081
  Contained Aboxes: University1_0.owl, University1_1.owl, University1_2.owl
- Service name: NrqlService_p9082
  Contained Aboxes : University1_1.owl, University1_2.owl
- Service name: NrqlService_p9083
  Constained Aboxes : University1_0.owl
- Service name: NrqlService_p9084
  Contained Aboxes : University1_0.owl, University1_2.owl

Example 1

In this example, the behavior or the *Assign thread* is presented, thus it is shown how the Locator finds a service to be assigned to execute the request. The *Assign thread* uses four criteria objects:

1. *Abox*, which helps to find the service that contains the required Abox.
2. *Waiting queue*, which obtains the number of requests waiting to be executed by a service.
3. *Subumption level*, which obtains the hierarchy of the query in the respective Qbox.
4. *Indexing*, which obtains if the required Abox has been indexed or not.

Criteria three and four make use of a local Racer server to obtain their respective values.

The *Assign thread* represents the main logic of the Locator, while the *Send thread* and *Thread to Service* threads, as explained in section 5.1.2, manage the connection with the web service partners, thus no interaction with the web service partners is being observed in this example, but only the internal reasoning of the Locator.

The context in which this example ran is as follows:

- Composition:
  The set of queries of the composition presented in section 2.2.1 is used, where Aboxes *University1_0.owl* and *Univeristy1_1.owl* are inquired.

- Criteria:
  The existence of the Abox in a service is considered as a *must constraint*. Indexing is considered as a *constraint* and subsumption level together with waiting queue are considered *objectives*.

- server states:
  Non of the Aboxes in any server is indexed, meaning that no query has been send before. Therefore, no Qbox for any of the Aboxes exist.

- Comments:
  The activity of the *Send thread* and *Thread to server* threads for the interaction with the web service partners was paused, so that the *waiting queue* criteria can be better observed. The set of queries were previously stored in the *Pool1 of requests*.

Results

As the run print out in appendix A.2.1 shows, for every request the three synthesis stages explained in section 5.1.3 are being carried out.
Six requests were assigned to a service for its execution:

1. First request: As the first query request arrived, only three services were found that contain the Abox *University1_0.owl*. In the second stage, because non of the Aboxes is indexed, no service satisfied criteria Indexing, therefore the constraint was relaxed and considered as an objective.
   Since it is expected that more criteria is considered as constraint, a second iteration in this stage was done to find services that satisfies the rest of the constraints. In this example, only the criteria *Indexing* was considered as constraint, therefore all the candidates are considered for the next stage, because there are no more constraints to satisfy.
   In the third stage, all candidates had the same values for every objective, thus the first service *NrqlService_p9081* was considered as non-dominated and assigned for this request. As explained in section 5.1.5, a service is

considered non-dominated when it is greater or equal than another service in all dimensions.
The waiting queue for service *NrqlService_p9081* increases to one.

2. Second request: The second query request requires another Abox *University1_1.owl*, thus a similar situation as in the first query is presented, with the difference that *NrqlService_p9081* has one request waiting for its execution, thus *NrqlService_p9082* dominates *NrqlService_p9081* in the waiting queue dimension, which is being minimized. Finally *NrqlService_p9082* increased its waiting queue to one.

3. Third request: During the second stage only service *NrqlService_p9081* satisfied the indexing criteria, thus is the only considered in the third stage, consequently the only non-dominated. The waiting queue for this service increases to two. Observe that the value for subsumption level is two, thus this query subsumes the first query in the composition, such that their query ancestors are *faculty* inquiry and the *top* of the Qbox.

4. Forth request: For this request, a similar situation than for the third query occurred. The *NrqlService_p9082* increases its waiting queue to two.

5. For the next 2 queries only *NrqlService_p9082* satisfied criteria indexing, thus the waiting queue increases finally to four.

It can be observed that considering criteria *Indexing* as a constraint, causes *NrqlService_p9081* and *NrqlService_p9082* to be chosen every time, discarding the possibility of comparing them against other services in the third stage. Thus, if another query that inquires Abox *University1_1.owl* arrives, the waiting queue of service *NrqlService_p9082* is much larger than the one of other services which would emerge as non-dominated during the third stage, since their waiting queue is shorter. The constraint is hindering the possibility of finding better candidates, therefore is not recommended to consider too many criteria as constraints but only the very necessary ones. Even though, this could be considered as a weakness of the Locator, this design aims to fulfill the clients request of considering certain criteria as constraint, by nature a constraint is strict, thus it is either fulfilled or not, while an objective is more flexible.

In fact, it is recommended that only must constraints and objectives are considered as criteria, thus, the third stage prunes out the dominated services and obtains the set of best considered services to choose from. This can be better observed in the next example where no constraints are considered.

Example 2

This example shows the decisions made for a second composition that was executed after the composition shown in the previous example. Thus the context is as follows:

- Composition:
  The same set of queries of the composition in example 1 were used, where the Aboxes *University1_1.owl* and *Univeristy1_2.owl* are inquired.

- Criteria:
  The existence of the Abox in a service is considered as a *must constraint*. Indexing, subsumption level together with waiting queue are considered *objectives*.

- Server states:
  After the execution of the composition in the first example, the servers were left with the following states[1]:

  - NrqlService_p9081:
    Indexed Aboxes: University1_0.owl
    Qbox for Abox University1_0.owl cached: *Faculty* and *FullProfessor* inquiries.

  - NrqlService_p9082:
    Indexed Aboxes: University1_1.owl
    Qbox for Abox University1_1.owl cached: *Faculty*, *FullProfessor*, *TeacherOf* and *Email* inquiries.

- Comments:
  The set of queries were previously stored in the *Pool1 of requests*. There are no queries waiting to be executed for any of the servers.

Results

As the run print out in appendix A.2.2 shows, for every request only the first and third synthesis stages were carried out, since there are no constraints to consider. Six requests were assigned to a service for its execution as follows:

1. First request: For the first query request three services were found containing Abox *University1_0.owl*. There were no constraints to consider. In the third stage, service *NrqlService_p9081* dominates the other twp services in all the dimensions. Thus the request is assigned to *NrqlService_p9081*.
   From the execution of the previous composition, an equivalent query was cached, thus a delete command for the present query is to be sent right after it. Therefore *NrqlService_p9081* increases its waiting queue to two.

2. Second request: For the second query request, the service *NrqlService_p9082* dominates the other two services, thus, *NrqlService_p9081* is dominated in the waiting queue dimension, while *NrqlService_p9084* is equivalent. Thus, *NrqlService_p9082* is chosen as non dominated and its waiting queue increases to one.

[1] Only the relevant states are presented

3. Third request: For the third query request, service *NrqlService_p9083* dominates *NrqlService_p9084*, while *NrqlService_p9081* and *NrqlService_p9083* are non-dominated. Even though *NrqlService_p9081* is indexed and has two ancestors for the present query, the waiting queue has a value dominated by *NrqlService_p9083*, thus both are considered as non-dominated. Finally *NrqlService_p9081* was randomly chosen and since it has already cached an equivalent query in the same Qbox, a delete command is sent, thus the waiting queue increases to four.

4. Fourth request: *NrqlService_p9082* dominates *NrqlService_p9081*, but because of the waiting queue it does not dominate *NrqlService_p9084*, thus both are non-dominated. By random *NrqlService_p9082* is assigned, increasing its waiting queue to two.

5. Fifth request: *NrqlService_p9082* dominates in all the dimensions to *NrqlService_p9081*, thus it is chosen. Since there is an equivalent query already cached in the Qbox a delete command is sent. The waiting queue increases to four.

6. Sixth request: a similar situation as for the previous query is presented.

It can be observed that considering only must constraints and objectives, more optimal choices among the total set of services are being obtained avoiding a situation where only certain services are chosen every time during the constraint synthesis stage as observed in the last example.

6.2 Conclusions

Finding the appropriate service to solve a certain request was resolved by choosing the service that satisfied a set of criteria that is based on the functionality of the web service partners. The considered criteria is:

1. Abox indexing: As explained in section 3.2.1 index structures are build for an Abox the first time it is inquired, which is a process that takes some time, as shown in appendix A.1. Therefore finding a service that contains an indexed Abox is advantageous.

2. Query sumbsumption: As explained in section 3.2.1 the Qbox, maintains the cached information of inquiries for a specific Abox, classifying the queries in a relation from most general to more specific, from which it is possible to know if the needed information has been cached before. If this is the case, the answer for the query will be totally or only partially computed, thus reducing computational effort.

3. Waiting queue: Due to the design of the Locator, in which the activity of a service partner is being observed and managed by the *Send thread*, such that only one *Thread to server* is active at a time. The *Pool2 of waiting queries* reflects the waiting queue for a specific service, thus

helping to know how busy a service is, and helping for load balancing purposes.

4. Abox: Ensures that only services that contain the required Abox are being considered.

The criteria were organized under three categories, which order also determined synthesis stages:

1. Must constraints: representing the minimum criteria expressed as equalities or inequalities that a service must satisfy in order to execute the query. Thus, these criteria are considered in the first synthesis stage.

2. Constraints: represent criteria expressed as equalities or inequalities that are considered as very important to be satisfied, knowing that its satisfiability might pruned out other services that could be advantageous in other criteria. This type of criteria is considered in the second synthesis stage.

3. Objectives: represent criteria expressed as minimization or maximization, thus being more flexible in its satisfiability. They are considered in the last synthesis stage.

Since there is no human interaction that intervenes in the decisions of the Locator, this classification of criteria, helps to know which constraints can be relaxed to objectives, when there are no services found that satisfies all the constraints. Thus, it is left to the designer to decide on the classification under which a criteria should be considered, such that they are proved in a three stage order, from the most strict to more flexible.

Considering the criteria as black boxes with a value, such that they can be dynamically classified under any of the criteria categories, probed to be very flexible and suitable for the different synthesis stages. Moreover, considering the content of each criteria classification as a set of black boxes, facilitates procedures like constraint relaxation, where a black box is removed from the constraint classification and added to the objective classification.

Obtaining the values of functionality dependent criteria[2] by the use of a local Racer server, prevents the invocation of the web services partners only for questioning their status. Moreover, the procedure to obtain the value for a criteria like subsumption level, as explained in section 5.1.1, requires sending some extra commands to obtain the hierarchy of the query in the remote Qbox.
For example, commands like (*prepare-abox-query)*[3], such that the query is classified without being executed, then a *(query-ancestors :query-id)* command should be sent to obtain its hierarchy, etc. Moreover to identify the ID of

[2]In this project also referred as service-dependent criteria
[3]For a detailed description refer to [8]

the query in the remote service a extra ID must be given by the Locator to be able to refer to the query in the remote service such that commands like *(query-ancestors :query-id)* are correctly executed. Thus having to send multiple commands to a remove service, means much processing and delay, which should be avoided.

Having a local Racer server, reflecting the state of the different web service partners, helped to manage the state of the remote servers, such that, if an equivalent query was already cached in the Qbox, the cached information of the executed query is deleted, to avoid the storage of duplicated information. Also Top like queries, are being deleted to avoid duplicating the total Abox content in the cached information of the Qbox.

Chapter 7

Future Work, Shortcomings and Contributions

7.1 Future Work

7.1.1 Criteria Management

The Locator carries out the different synthesis stages according to the configuration of each criteria object contained in the *CriteriaXService* class. As figure 7.1 shows, the criteria objects with some predefined attribute values are classified under a criteria category. Thus, the attributes like priority, desired value and objective function of the criteria objects are being fixed and the same configuration is applied to optimize every composition.
To offer more flexibility, the Locator can be extended such that the attributes of the criteria objects and their classification in the *CriteriaXService* object, can be dynamically handled for each composition. For this purpose an appropriate way for the entry of parameters should be designed, such that they can be transferred to constructors methods of each criteria and *CriteriaXService* objects.

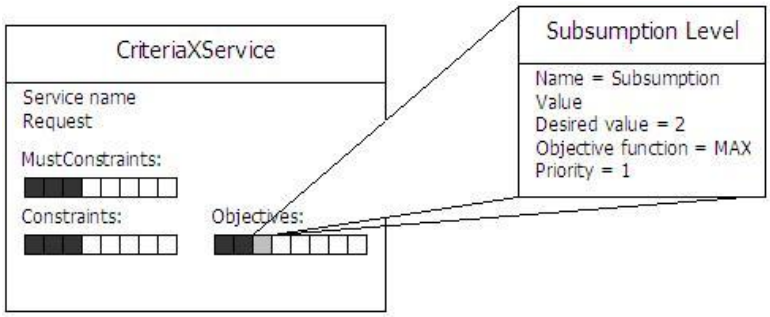

Figure 7.1: Criteria management.

7.1.2 Further Introduction of Criteria Classes

As explained in section 3.1, criteria for optimization can also be searched outside the functionality offered by the web service partners. Thus, research to find suitable criteria for general web services focuses on quality of service concepts like performance, reliability, integrity, accessibility, etc. Therefore, a variety of possibilities are there to extend the set of criteria for optimization. Considering the extension of the Locator for the dynamical management of criteria, appropriate criteria in the area of service level agreements can extend the present set to handle different classes of clients, such that each client profile corresponds to a different configuration of criteria.

7.1.3 Global Optimization

Once more criteria classes are considered the necessity for a global optimization also increases, such that criteria like total cost can be guaranteed. For this purpose, the algorithms for combinatorial optimization explained in section 4.1 are helpful.

7.1.4 Usage of the UDDI Repository

As observed in section 3.1.2, resent research in the specification of quality of service issues with respect to web services has been focused in the development of language specifications. For example, the WSOL [15] and WSLA [1] specifications that complement the WSDL service description for the specification of service level agreements, such that quality descriptions can also be obtained from the UDDI registry in a similar way as for the WSDL descriptions.
Thus, the Locator can be extended to use the UDDI technology, such that other web services which want to be considered as web service partners for the optimization strategy, can register to the UDDI. In this way, the Locator can use the UDDI registry to obtain the set web service candidates plus some extra quality descriptions, from which criteria values can be obtained, leaving the computation of those values to the web service partners.

7.2 Shortcomings

7.2.1 Locator Interface

Until now, the present solution works properly when considering web services that offer the same functionality with a similar interface. As explained in section 5.1 the Locator tries to be transparent to the client by offering a similar interface as the web service partners. Therefore if new web service partners are to be considered which implement a different interface, the transparency is not fulfilled anymore. Moreover, when considering extending the Locator to handle criteria management, additional information is required to be transferred while invoking the Locator. For these reasons, if other service interfaces and criteria

management is considered, another approach can be proposed to cope with these problems as follows:

- To change the Locator interface from the present *rpc* binding style to a *document* binding style, such that an XML document containing criteria preferences can be transferred. Even though this implies some extra parsing computation.
- Another feasible solution to avoid changing the Locator's interface is such where client preferences are registered in a repository, such as the UDDI, thus when ever a client is recognized the preferences are obtained and the optimization process can take place. Thus, the client would have the responsibility to update its own preferences. This is a similar approach as the Service Level Agreements described in section 3.1.2. This approach implies also the same extra parsing computation as the previous proposal.

7.2.2 Local Optimization

As described in section 4.2.2 a local optimization strategy was applied to profit from changes in the state of the servers caused by the inquiry of other client compositions that are being executed in parallel. It was considered that for a global optimization, locking the required web service partners, with the aim of maintaining a consistent state of the servers until the last invocation of the composition was processed, would lead to undesired bottlenecks. Though the global optimization can be applied without locking the use of the web service partners, arguing that decisions were made upon the service's states during the combinatorial optimization, without the warranty that the same states will be maintained. This is beneficial to criteria like subsumption level and Abox indexing which values can only improve (if considering that servers do not discard anything from their cached information), while for the waiting queue is not.

7.3 Contributions

7.3.1 Optimization Criteria

A solution for the optimization of web service compositions with the characteristics described in section 2, was developed based on the selection of web service partners according to criteria related to their functionality. The considered criteria deal with query subsumption, Abox indexing, existence of required Abox and waiting queue of a service. For a detailed description of this criteria refer to section 3.2.

7.3.2 Criteria as Black Boxes

Handling the criteria in a uniform way, independently of any peculiarities by considering them as black boxes with a value, provides a very flexible way

of dealing with multiple criteria. Thus, different optimization algorithms can profit from this flexibility, while leaving the specifics to the different criteria classes.

7.3.3 Memory Management

A way to avoid the duplication of cached information, while the Qbox feature is activated in the web service partners was implemented, such that delete commands are being send to the web service partners for the removal of equivalent queries and top like queries.

7.3.4 New Business or Service Paradigm

Providing an optimization strategy for this specific type of web service compositions, offers the possibility of creating a business or service paradigm in which clients (web service compositions) are being attracted to profit from the Locator services, while web services capable of fulfilling the clients request can find a market place for their service offers.

Appendix A

Tests

A.1 Indexing Time for Aboxes of Different Sizes

To obtain the time that it takes an Abox to be indexed, a Java program using the JRacer API, executed a query over Aboxes of different sizes. The tests where performed under the following conditions:

Used Knowledge Base

The Univ-Bench ontology [16] that has been used for the evaluation of DAML+OIL repositories, and a set of Aboxes produced by the UBA[1] generator where used. To change the size of the produced Aboxes, different random instances were added or deleted.

Thus the unique Tbox is separated from the Abox files, to avoid recomputation of the Tbox for the related Abox every time a query is sent.

Results

Abox size	Indexing time
6000 Kb	51 seconds
5000 Kb	41 seconds
4000 Kb	26 seconds
3000 Kb	22 seconds
2000 Kb	11 seconds
1000 Kb	1 second

Table A.1: Indexing time for an Abox

Comments

Once the indexing structures are build, the next query is answered in less than a second.

[1] Univ-bench artificial generator

Even though, the processing time for the indexation of an Abox can vary according to the characteristics of the computational resources, the tests give an example of difference between response time before and after indexation.

A.2 Assign thread's run results

A.2.1 Example 1

```
Start Assign Thread
-------------------

Assign- Assigning query: (retrieve (?x) (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Faculty|))
        Abox: /KB/University1_0.owl

     First synthesis: Must Constraints

      Service NrqlService_p9081 satisfies criteria: Abox
      Service NrqlService_p9083 satisfies criteria: Abox
      Service NrqlService_p9084 satisfies criteria: Abox

     Second synthesis: Constraints

      Service NrqlService_p9081:
      -Index checking
      abox is not indexed

      Service NrqlService_p9083:
       -Index checking
       abox is not indexed

       Service NrqlService_p9084:
       -Index checking
       abox is not indexed

       Non of the candidates satisfy all the constraints, proceeding to relaxation:
       Criteria Indexing was relaxed
       Next iteration for this synthesis stage:

       Service NrqlService_p9081:
       NrqlService_p9081 satisfies all constraints. Added as candidate for next stage

       Service NrqlService_p9083:
       NrqlService_p9083 satisfies all constraints. Added as candidate for next stage

       Service NrqlService_p9084:
       NrqlService_p9084 satisfies all constraints. Added as candidate for next stage

     Third synthesis: Objectives

       Obtaining criteria values:

       For service: NrqlService_p9081
       -Checking subsumption level:
       There is no subsumee
       -Waiting queue:
       Total number of queries waiting: 0
       -Index checking
       abox is not indexed

       For service: NrqlService_p9083
       -Checking subsumption level:
       There is no subsumee
       -Waiting queue:
```

Total number of queries waiting: 0
-Index checking
abox is not indexed

For service: NrqlService_p9084
-Checking subsumption level:
There is no subsumee
-Waiting queue:
Total number of queries waiting: 0
-Index checking
abox is not indexed

Obtaining non-dominated nodes:

NrqlService_p9081 dominates:
node: NrqlService_p9083
node: NrqlService_p9084
NrqlService_p9083 dominates:
NrqlService_p9084 dominates:
Service NrqlService_p9081 is non-dominated

Assigned to service NrqlService_p9081
Memory management:
-Looking for query-equivalents
there are no equivalents

-Checking if this is a top like query
is not a top query

Assign- Assigning query: (retrieve (?x) (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Faculty|))
Abox: /KB/University1_1.owl

First synthesis: Must Constraints

Service NrqlService_p9081 satisfies criteria: Abox
Service NrqlService_p9082 satisfies criteria: Abox

Second synthesis: Constraints

Service NrqlService_p9081:
-Index checking
abox is not indexed

Service NrqlService_p9082:
-Index checking
abox is not indexed

Non of the candidates satisfy all the constraints, proceeding to relaxation:
Criteria Indexing was relaxed
Next iteration for this synthesis stage:

Service NrqlService_p9081:
NrqlService_p9081 satisfies all constraints. Added as candidate for next stage

Service NrqlService_p9082:
NrqlService_p9082 satisfies all constraints. Added as candidate for next stage

Third synthesis: Objectives

Obtaining criteria values:

For service: NrqlService_p9081
-Checking subsumption level:
There is no subsumee

```
        -Waiting queue:
        Total number of queries waiting: 1
        -Index checking
        abox is not indexed

        For service: NrqlService_p9082
        -Checking subsumption level:
        There is no subsumee
        -Waiting queue:
        Total number of queries waiting: 0
        -Index checking
        abox is not indexed

        Obtaining non-dominated nodes:

        NrqlService_p9081 dominates:
        NrqlService_p9082 dominates:
          node: NrqlService_p9081
        Service NrqlService_p9082 is non-dominated

      Assigned to service NrqlService_p9082
Memory management:
        -Looking for query-equivalents
        there are no equivalents

        -Checking if this is a top like query
        is not a top query
```

```
Assign- Assigning query: (retrieve (?x) (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#FullProfessor|))
          Abox: /KB/University1_0.owl

      First synthesis: Must Constraints

        Service NrqlService_p9081 satisfies criteria: Abox
        Service NrqlService_p9083 satisfies criteria: Abox
        Service NrqlService_p9084 satisfies criteria: Abox

      Second synthesis: Constraints

        Service NrqlService_p9081:
        -Index checking
        abox is indexed
        Service NrqlService_p9081 satisfies criteria Indexing
        NrqlService_p9081 satisfies all constraints. Added as candidate for next stage

        Service NrqlService_p9083:
        -Index checking
        abox is not indexed

        Service NrqlService_p9084:
        -Index checking
        abox is not indexed

      Third synthesis: Objectives

        Obtaining criteria values:

        For service: NrqlService_p9081
        -Checking subsumption level:
        Total number of subsumees(ancestors): 2
        -Waiting queue:
        Total number of queries waiting: 1

        Obtaining non-dominated nodes:
```

```
        NrqlService_p9081 dominates:
        Service NrqlService_p9081 is non-dominated

      Assigned to service NrqlService_p9081
Memory management:
        -Looking for query-equivalents
        there are no equivalents

        -Checking if this is a top like query
        is not a top query
```

```
Assign- Assigning query: (retrieve (?x) (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#FullProfessor|))
        Abox: /KB/University1_1.owl

      First synthesis: Must Constraints

        Service NrqlService_p9081 satisfies criteria: Abox
        Service NrqlService_p9082 satisfies criteria: Abox

      Second synthesis: Constraints

        Service NrqlService_p9081:
        -Index checking
        abox is not indexed

        Service NrqlService_p9082:
        -Index checking
        abox is indexed
        Service NrqlService_p9082 satisfies criteria Indexing
        NrqlService_p9082 satisfies all constraints. Added as candidate for next stage

      Third synthesis: Objectives

        Obtaining criteria values:

        For service: NrqlService_p9082
        -Checking subsumption level:
        Total number of subsumees(ancestors): 2
        -Waiting queue:
        Total number of queries waiting: 1

        Obtaining non-dominated nodes:

        NrqlService_p9082 dominates:
        Service NrqlService_p9082 is non-dominated

      Assigned to service NrqlService_p9082
Memory management:
        -Looking for query-equivalents
        there are no equivalents

        -Checking if this is a top like query
        is not a top query
```

```
Assign- Assigning query: (retrieve (?x ?y) (and (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-
bench.owl#FullProfessor|) (?x ?y |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#teacherOf|)))
        Abox: /KB/University1_1.owl

      First synthesis: Must Constraints

        Service NrqlService_p9081 satisfies criteria: Abox
        Service NrqlService_p9082 satisfies criteria: Abox

      Second synthesis: Constraints
```

```
      Service NrqlService_p9081:
      -Index checking
      abox is not indexed

      Service NrqlService_p9082:
      -Index checking
      abox is indexed
      Service NrqlService_p9082 satisfies criteria Indexing
      NrqlService_p9082 satisfies all constraints. Added as candidate for next stage

    Third synthesis: Objectives

      Obtaining criteria values:

      For service: NrqlService_p9082
      -Checking subsumption level:
      Total number of subsumees(ancestors): 1
      -Waiting queue:
      Total number of queries waiting: 2

      Obtaining non-dominated nodes:

      NrqlService_p9082 dominates:
      Service NrqlService_p9082 is non-dominated

    Assigned to service NrqlService_p9082
Memory management:
      -Looking for query-equivalents
      there are no equivalents

      -Checking if this is a top like query
      is not a top query
```

```
Assign- Assigning query: (retrieve ((annotation (|http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-
bench.owl#emailAddress| |http://www.Department0.University1.edu/AssistantProfessor0|)))
(|http://www.Department0.University1.edu/AssistantProfessor0| top))
  Abox: /KB/University1_1.owl

    First synthesis: Must Constraints

      Service NrqlService_p9081 satisfies criteria: Abox
      Service NrqlService_p9082 satisfies criteria: Abox

    Second synthesis: Constraints

      Service NrqlService_p9081:
      -Index checking
      abox is not indexed

      Service NrqlService_p9082:
      -Index checking
      abox is indexed
      Service NrqlService_p9082 satisfies criteria Indexing
      NrqlService_p9082 satisfies all constraints. Added as candidate for next stage

    Third synthesis: Objectives

      Obtaining criteria values:

      For service: NrqlService_p9082
      -Checking subsumption level:
      Total number of subsumees(ancestors): 1
      -Waiting queue:
      Total number of queries waiting: 3
```

Obtaining non-dominated nodes:

NrqlService_p9082 dominates:
Service NrqlService_p9082 is non-dominated

Assigned to service NrqlService_p9082
Memory management:
-Looking for query-equivalents
there are no equivalents

-Checking if this is a top like query
is not a top query

Finish Thread to Assign
Finished

A.2.2 Example 2

Start Assign Thread

Assign- Assigning query: (retrieve (?x) (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Faculty|))
Abox: /KB/University1_0.owl

First synthesis: Must Constraints

Service NrqlService_p9081 satisfies criteria: Abox
Service NrqlService_p9083 satisfies criteria: Abox
Service NrqlService_p9084 satisfies criteria: Abox

Second synthesis: Constraints –> no constraints where found

Third synthesis: Objectives

Obtaining criteria values:

For service: NrqlService_p9081
-Index checking
abox is indexed
-Checking subsumption level:
Total number of subsumees(ancestors): 1
-Waiting queue:
Total number of queries waiting: 0

For service: NrqlService_p9083
-Index checking
abox is not indexed
-Checking subsumption level:
There is no subsumee
-Waiting queue:
Total number of queries waiting: 0

For service: NrqlService_p9084
-Index checking
abox is not indexed
-Checking subsumption level:
There is no subsumee
-Waiting queue:
Total number of queries waiting: 0

Obtaining non-dominated nodes:

NrqlService_p9081 dominates:
node: NrqlService_p9083
node: NrqlService_p9084

```
        NrqlService_p9083 dominates:
        NrqlService_p9084 dominates:
        Service NrqlService_p9081 is non-dominated

       Assigned to service NrqlService_p9081
Memory management:
        -Looking for query-equivalents
        there are equivalents
        A delete command will be sent for this query
```

```
Assign- Assigning query: (retrieve (?x) (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#Faculty|))
        Abox: /KB/University1_2.owl

       First synthesis: Must Constraints

        Service NrqlService_p9081 satisfies criteria: Abox
        Service NrqlService_p9082 satisfies criteria: Abox
        Service NrqlService_p9084 satisfies criteria: Abox

       Second synthesis: Constraints –> no constraints where found

       Third synthesis: Objectives

        Obtaining criteria values:

        For service: NrqlService_p9081
        -Index checking
        abox is not indexed
        -Checking subsumption level:
        There is no subsumee
        -Waiting queue:
        Total number of queries waiting: 2

        For service: NrqlService_p9082
        -Index checking
        abox is not indexed
        -Checking subsumption level:
        There is no subsumee
        -Waiting queue:
        Total number of queries waiting: 0

        For service: NrqlService_p9084
        -Index checking
        abox is not indexed
        -Checking subsumption level:
        There is no subsumee
        -Waiting queue:
        Total number of queries waiting: 0

        Obtaining non-dominated nodes:

        NrqlService_p9081 dominates:
        NrqlService_p9082 dominates:
          node: NrqlService_p9081
          node: NrqlService_p9084
        NrqlService_p9084 dominates:
        Service NrqlService_p9082 is non-dominated

       Assigned to service NrqlService_p9082
Memory management:
        -Looking for query-equivalents
        there are no equivalents

        -Checking if this is a top like query
        is not a top query
```

Assign- Assigning query: (retrieve (?x) (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#FullProfessor|))
Abox: /KB/University1_0.owl

First synthesis: Must Constraints

Service NrqlService_p9081 satisfies criteria: Abox
Service NrqlService_p9083 satisfies criteria: Abox
Service NrqlService_p9084 satisfies criteria: Abox

Second synthesis: Constraints –> no constraints where found

Third synthesis: Objectives

Obtaining criteria values:

For service: NrqlService_p9081
-Index checking
abox is indexed
-Checking subsumption level:
Total number of subsumees(ancestors): 2
-Waiting queue:
Total number of queries waiting: 2

For service: NrqlService_p9083
-Index checking
abox is not indexed
-Checking subsumption level:
There is no subsumee
-Waiting queue:
Total number of queries waiting: 0

For service: NrqlService_p9084
-Index checking
abox is not indexed
-Checking subsumption level:
There is no subsumee
-Waiting queue:
Total number of queries waiting: 0

Obtaining non-dominated nodes:

NrqlService_p9081 dominates:
NrqlService_p9083 dominates:
node: NrqlService_p9084
NrqlService_p9084 dominates:
Service NrqlService_p9081 is non-dominated
Service NrqlService_p9083 is non-dominated

Assigned to service NrqlService_p9081
Memory management:
-Looking for query-equivalents
there are equivalents
A delete command will be sent for this query

Assign- Assigning query: (retrieve (?x) (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#FullProfessor|))
Abox: /KB/University1_2.owl

First synthesis: Must Constraints

Service NrqlService_p9081 satisfies criteria: Abox
Service NrqlService_p9082 satisfies criteria: Abox
Service NrqlService_p9084 satisfies criteria: Abox

```
    Second synthesis: Constraints –> no constraints where found

    Third synthesis: Objectives

      Obtaining criteria values:

      For service: NrqlService_p9081
      -Index checking
      abox is not indexed
      -Checking subsumption level:
      There is no subsumee
      -Waiting queue:
      Total number of queries waiting: 4

      For service: NrqlService_p9082
      -Index checking
      abox is indexed
      -Checking subsumption level:
      Total number of subsumees(ancestors): 2
      -Waiting queue:
      Total number of queries waiting: 1

      For service: NrqlService_p9084
      -Index checking
      abox is not indexed
      -Checking subsumption level:
      There is no subsumee
      -Waiting queue:
      Total number of queries waiting: 0

      Obtaining non-dominated nodes:

      NrqlService_p9081 dominates:
      NrqlService_p9082 dominates:
        node: NrqlService_p9081
      NrqlService_p9084 dominates:
      Service NrqlService_p9082 is non-dominated
      Service NrqlService_p9084 is non-dominated

    Assigned to service NrqlService_p9082
Memory management:
      -Looking for query-equivalents
      there are no equivalents

      -Checking if this is a top like query
      is not a top query
```

```
Assign- Assigning query:
(retrieve (?x ?y) (and (?x |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#FullProfessor|)
(?x ?y |http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#teacherOf|)))
  Abox: /KB/University1_1.owl

    First synthesis: Must Constraints

      Service NrqlService_p9081 satisfies criteria: Abox
      Service NrqlService_p9082 satisfies criteria: Abox

    Second synthesis: Constraints –> no constraints where found

    Third synthesis: Objectives

      Obtaining criteria values:

      For service: NrqlService_p9081
      -Index checking
```

```
abox is not indexed
-Checking subsumption level:
There is no subsumee
-Waiting queue:
Total number of queries waiting: 4

For service: NrqlService_p9082
-Index checking
abox is indexed
-Checking subsumption level:
Total number of subsumees(ancestors): 1
-Waiting queue:
Total number of queries waiting: 2

Obtaining non-dominated nodes:

NrqlService_p9081 dominates:
NrqlService_p9082 dominates:
  node: NrqlService_p9081
Service NrqlService_p9082 is non-dominated

Assigned to service NrqlService_p9082
Memory management:
-Looking for query-equivalents
there are equivalents
A delete command will be sent for this query
```

```
Assign- Assigning query: (retrieve ((annotation (|http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-
bench.owl#emailAddress| |http://www.Department0.University1.edu/AssistantProfessor0|)))
(|http://www.Department0.University1.edu/AssistantProfessor0| top))
 Abox: /KB/University1_1.owl

First synthesis: Must Constraints

  Service NrqlService_p9081 satisfies criteria: Abox
  Service NrqlService_p9082 satisfies criteria: Abox

Second synthesis: Constraints –> no constraints where found

Third synthesis: Objectives

  Obtaining criteria values:

  For service: NrqlService_p9081
  -Index checking
  abox is not indexed
  -Checking subsumption level:
  There is no subsumee
  -Waiting queue:
  Total number of queries waiting: 4

  For service: NrqlService_p9082
  -Index checking
  abox is indexed
  -Checking subsumption level:
  Total number of subsumees(ancestors): 1
  -Waiting queue:
  Total number of queries waiting: 4

  Obtaining non-dominated nodes:

  NrqlService_p9081 dominates:
  NrqlService_p9082 dominates:
    node: NrqlService_p9081
```

```
        Service NrqlService_p9082 is non-dominated

      Assigned to service NrqlService_p9082
Memory management:
       -Looking for query-equivalents
       there are equivalents
       A delete command will be sent for this query

      Finish Thread to Assign
 Finished
```

Bibliography

[1] A. Dan, A. R. Franck, A. Keller, R. King, H. Ludwig, *Web Service Level Agreement Language Specification - IBM*, 2002. http://dwdemos.alphaworks.ibm.com/wstk/common/wstkdoc/services/utilities/wslaauthoring/WebServiceLevelAgreementLanguage.html.

[2] Akhil Sahai, Anna Durante and Vijay Machiraju, *Towards Automated SLA Management for Web Services.*, 2002.

[3] Alonso Gustavo, Casati Fabio, Kuno Harumi and Machiraju Vijay, *Web Services: Concpets, Architechtures and Applications.*, 2004.

[4] Christoph Tempich and Raphael Volz, *Towards a benchmark for Semantic Web reasoners - An analysis of the DAML ontology library.*, 2003.

[5] D. Navinchandra, *Exploration and innovation in Design. Towards a Computationl Model.*, 1991.

[6] E. Bonabeau, M. Dorigo and G. Theaulaz, *Inspiration for optimization from social insect behaviour.*, 2000.

[7] Eric Newcommer, *Understanding Web Services: XML, WSDL, SOAP and UDDI.*, 2002.

[8] Haarslev Volker, Moeller Ralf, Wessel Michael, *The New Racer Query Language-nRQL*, 2005.

[9] Haarslev Volker, Möller Ralf, Wessel Michael, *RACER User's Guide and Reference Manual Version 1.8.*, 2005April.

[10] Juraj Hromkovic, *Algorithmics for Hard Problems: Introduction to Combinatorial Optimization, Randomization, Approximation and Heuristics.*, 2002.

[11] M. Tian, A. Gramm, T. Naumowicz, H. Ritter and J. Schiller, *A Concept for QoS Integration in Web Services.*, 2003.

[12] Nils J. Nilsson, *Principles of Artificial Intelligence.*, 1982.

[13] Rajesh Sumra and Arulazi D., *Quality of Service for Web Services-Demystification, Limitations, and Best Practices.*, 2003March. http://www.developer.com/services/article.php/2027911.

[14] Sanjiva Weerawarana and Francisco Curbera, *Business Process with BPEL4WS.*, 2002. http://www-128.ibm.com/developerworks/library/ws-bpel/.

[15] Vladimir Tosic, Bernard Pagurek and Kruti Patel, *WSOL - A Language for the Formal Specification of Various Constraints and Classes of Service for Web Services.*, 2002November. Technical Report OCIECE-02-06.

[16] Y. Guo, J. Heflin and Z. Pan, *Benchmarking DAML+OIL Repositories.*, 2003.