Search in Web Service Bases

Dimitrios Pilios

Master Thesis

Ioannina, June 2014

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ & ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF IOANNINA

Οργάνωση και Αναζήτηση σε Βάσεις Υπηρεσιών Διαδικτύου

Η
ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

Υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης
του Τμήματος Πληροφορικής
Εξεταστική Επιτροπή

από τον

Δημήτριο Πήλιο

ως μέρος των Υποχρεώσεων

για τη λήψη

του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ ΣΤΟ ΛΟΓΙΣΜΙΚΟ

Ιούνιος 2014

INDEX OF CONTENTS

Page

INDEX OF TABLES

Page

INDEX OF FIGURES

Page

ΠΕΡΙΛΗΨΗ

Δημήτριος Πήλιος του Πέτρου και της Ιουλίας. MSc, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Ιούνιος, 2014. Οργάνωση και Αναζήτηση σε Βάσεις Υπηρεσιών Διαδικτύου. Επιβλέποντας: Απόστολος Ζάρρας.

Η ανάπτυξη υπηρεσιο-κεντρικού λογισμικού, παρά την αρχικά υποσχόμενη εξέλιξη της, δεν έχει καταφέρει να αναδειχθεί σε μια ευρέως χρησιμοποιούμενη τεχνολογία. Κύρια αιτία γι αυτό αποτελεί η περιορισμενη αποτελεσματικότητα και αποδοτικότητα των χρησιμοποιούμενων τεχνικών αναζήτησης: δεν προσφέρονται τρόποι αναζήτησης σχετικά με τη δομή των υπηρεσιών, για το οποίο ενδιαφέρονται περισότερο οι προγραμματιστές, ενώ και ο χρόνος αναζήτησης είναι υψηλός, γιατί απαιτείται έρευνα κάθε υπηρεσίας, οπότε ο χρόνος αναζήτησης είναι ανάλογος με τον αριθμό των υπηρεσιών.

Το σύστημα AoSBM (Abstraction-Oriented Service Base Management), ένα λογισμικό ανοικτού κώδικα, εισάγει μια τεχνική ομαδοποιήσης. Κάθε ομάδα χαρακτηρίζεται από μια σύνοψη, που λέγεται αφαίρεση υπηρεσιών. Μια αφαίρεση υπηρεσιών αντιπροσωπεύει μια ομάδα υπηρεσιών με παρόμοιες λειτουργίες (μεθόδους, παραμέτρους εισόδου/εξόδου, κτλ.). Οι ερωτήσεις αναζήτησης ερευνούν κάθε αφαίρεση υπηρεσιών, οπότε ο χρόνος αναζήτησης είναι ανάλογος με τον αριθμό των αφαιρέσεων υπηρεσιών, και όχι με τον αριθμό των υπηρεσιών.

Βασισμένοι στη έννοια της αφαίρεσης υπηρεσιών και στον αλγόριθμο εξαγωγής αφαιρέσεων υπηρεσιών που χρησιμοποιεί το AoSBM, διευκολύναμε την οργάνωση μεγάλων αδόμητων συλλογών υπηρεσιών καθώς και την αναζήτηση υπηρεσιών. Συγκεκριμένα, προτείνουμε ένα σύστημα αναζήτησης υπηρεσιών, το οποίο ονομάζουμε βάση υπηρεσιών. Τα κύρια συστατικά της βάσης υπηρεσιών είναι μια κατανεμημένη λειτουργία εξαγωγής αφαιρέσεων υπηρεσιών, η οποία κάνει εφικτή την εφαρμογή της ομαδοποιήσης σε μεγάλο αριθμό υπηρεσιών και μια φιλική προς τον προγραμματιστή μηχανή αναζήτησης υπηρεσιών, η οποία διενεργεί την αναζήτηση με βάση τις αφαιρέσεις υπηρεσιών. Επιπρόσθετα, αναπτύξαμε μια υπηρεσία η οποία δίνει πρόσβαση στη μηχανή αναζήτησης, επιτρέποντας έτσι τη χρήση της βάσης υπηρεσιών σε μια κατανεμημένη τοποθέτηση.

ABSTRACT

P, D. MSc, Department of Computer Science and Engineering, University of Ioannina, Greece. June, 2014. Organization and Search in Web Service Bases. Thesis Supervisor: Zarras, Apostolos.

Service-Oriented Computing (SOC), despite emerging as a very promising trend for application development, has failed to be widely used. The main reason for that is the limited efficiency and effectiveness of the current search technologies; structured queries, which mainly concern a developer, are not offered, while search time is high, since answering a query requires matching it against all the services, thus meaning that search time scales with the number of services.

Abstraction-Oriented Service Base Management (AoSBM) is an open source software that introduces a clustering technique; the summaries that characterize the clusters are called service abstractions. A service abstraction represents a group of services that have similar functional properties (operations, inputs, outputs, etc.). The lookup queries are matched against service abstractions, thus the query execution time scales with the number of services abstractions, instead of scaling with the number of service descriptions.

We build upon the notion of service abstractions and the abstractions mining algorithm offered by AoSBM to enable the organization of large unstructured collections of service descriptions and the execution of service lookup queries. More specifically, we propose a service discovery facility that we call service base. The main constituents of the service base are a distributed abstractions mining facility that enables the clustering of large collections of service descriptions, and a developer-friendly query engine facility that enables the execution of service lookup queries over abstractions. Moreover, we developed a Web service that provides access to the query engine and allows using the service base in a distributed setting.

CHAPTER 1. INTRODUCTION

1.1 Objectives

The faster and cheaper Internet becomes, the more intriguing Internet Computing gets. Service-Oriented Computing (SOC) is the most widely acknowledged paradigm for Internet Computing [22]. Using services as reusable software components makes application development faster, easier and cheaper. More than that, a great opportunity for distributed applications development is emerging, as one does not need to own vast computational facilities in order to execute heavy-weighted processes. In fact, such a process can be substituted by a distributed one which can be executed even in heterogeneous environments.

Although much attention has been focused on services over the last few years, both in terms of research and technology, the expectations that SOC would serve as a major medium for Business to Business (B2B) and Business to Consumer (B2C) interaction has not been fulfilled. As the authors in [7] denote, the business applications using third party web services as part of their functionality are very few. Rather than that, businesses prefer to develop their applications from scratch and, therefore, web services remain much more a convenient wrapping technology, than a basic construct for serious enterprise applications. To deal with this, there is a need for efficient and effective service discovery facilities.

The typical service discovery approaches involve two key actors; the service providers who register information about the services they provide in a service registry (centralized, or

distributed) and the service consumers who perform lookup queries over the contents of the registry. To enable efficient and effective querying the information that is stored in the service registries is typically organized according to a particular classification schema that is exploited by the providers towards the registration of the services that they provide.

Recently the availability of service crawlers that crawl the web and collect information about large collections of services introduced a shift in the conventional service discovery paradigm [1, 7, 25]. A large collection of services that is returned by the crawler is stored in the registry. By default the information is not structured in any way. The implication of this is that answering a query requires matching it against all the services, which in turn means that the query execution time scales with the number of services.

A possible solution to this problem is to treat service descriptions as documents and employ typical document indexing techniques for efficient information retrieval. Another possible solution is to employ typical clustering techniques that group similar services and construct a summary for each group (e.g. , tags). The main drawback of these solutions is that they only support keyword based querying, which may be adequate in the case of the retrieval of documents, but really unsuitable in the case of services. In the case of services we have to keep in mind that the typical query author is a developer. The main concern of the developers is to ask structured queries for services that offer certain operations, which have certain inputs, outputs, etc. To cover such requirements we need a clustering technique that not only groups similar services, but also summarizes each group in a form that can support the matching of structured queries.

Athanasopoulos et al. [6] proposed such a clustering technique; they call the summaries that characterize the clusters service abstractions. A service abstraction represents a group of services that have similar functional properties (operations, inputs, outputs, etc.). Moreover, a service abstraction is characterized by an abstract interface and a set of mappings between the abstract interface and the concrete interfaces of the represented services. The original purpose of this technique was to be executed over a small set of services, so as to find groups of similar services that can substitute each other. Nevertheless, the concept of service abstraction can also be applied to enable the execution of efficient service lookup queries. The main idea in this case is to group similar service descriptions with respect to service abstractions and match

lookup queries against service abstractions, instead of matching them against service descriptions. Doing so implies that the query execution time would scale with the number of service abstractions, instead of scaling with respect to the number of service descriptions.

Unfortunately, the main problem with this approach is that the algorithm that mines service abstractions is computationally and resource demanding. Hence, it does not scale for large collections of service descriptions. As stated in [13], a significant growth in the number of available service descriptions is anticipated. According to [1, 11, 14, 15], from 2003 till 2009, the amount of available service descriptions progressively increased from few hundreds to thousands. Moving on to the future, as documented in the EU FI Assembly vision document [12], from 2010 to 2015, the number of available service descriptions is expected to scale up to millions, while beyond 2015, this number is expected to grow up to billions, trillions and so on.

In this thesis, we build upon the notion of service abstractions and the abstractions mining algorithm proposed in [6] to facilitate the organization of large unstructured collections of service descriptions and the execution of service lookup queries. More specifically, we propose a service discovery facility that we call service base. The main constituents of the service base are the following;

- A novel, scalable, distributed abstraction mining facility that makes the clustering of large collections of service descriptions feasible. The proposed facility is part of a relational storage facility that stores service abstractions along with the collections of service descriptions that are represented by the service abstractions.

- A query engine facility that enables the execution of service lookup queries over abstractions. Moreover, we developed a Web service that provides access to the query engine and allows using the service base in a distributed setting.

1.2 Thesis Structure

We organize the rest of this thesis in six chapters; Chapter 2 probes further into the ideas beyond the existing Web services search systems, by analyzing the way they collect, organize, classify and retrieve Web services. Chapter 3 details the system proposed by [6], AoSBM, which served as the basis for our development. Chapters 4 and 5 present our contribution; in particular,

Chapter 4 presents our distributed version of the AoSBM's abstractions mining process, while Chapter 5 comprises our developed query engine. In Chapter 6, we present our experiments on the distributed abstractions mining process, measuring its execution time, as well as the execution time and the quality of the results for specific query workloads posed to the service base produced by our distributed abstractions mining process. Finally, Chapter 7 concludes this thesis, and sets a number of significant challenges concerning Web services organization and retrieval.

CHAPTER 2. RELATED WORK

The baseline approach adopted for the interaction between service providers and consumers is based on the idea of a public service registry, acting as the broker who brings together providers and users, as illustrated in Figure 2.1, while the data model mostly used is the one that has been proposed in the Universal Description Discovery and Integration (UDDI) specification. According to this, service providers who want to publish and advertise their services, must provide appropriate meta-data to the registry. These meta-data can be divided in three categories, (a) white pages, where businesses express their identity, (b) yellow pages, where they categorize their services and (c) green pages, where technical description for the services invocation is provided. For the technical description, UDDI proposes the use of tModels (technical models), while mappings to corresponding WSDL documents are also supported. Then, developers interested in finding useful web services as components for their applications, can search for them in two ways; either by browsing the registry, or via keyword (or value) - based search facilities. The most well-known attempt to provide such a public repository was the UDDI Business Registry (UBR) supported by IBM, Microsoft and SAP. The big drawback of such attempts, however, is the fact that they rely almost exclusively on human maintenance, i.e. on providers, without making any attempt of periodical checking for invalid or outdated services. Consequently, either the quality of their contents quickly degrades and becomes unusable, or, if maintenance is employed, the overhead of maintening is so large that far outweighs the benefits.

Search engines emerged as the newer trend for finding services, starting from the big search engine corporations, like Google, Amazon and Yahoo, who decided to publish their Web services through their own websites instead of using public registries. This enables developers to discover web services using a search engine model. Additionally, these sites generally exploit

their web page crawling technology by using it to capture WSDL documents, since nowadays there are plenty of WSDL documents publicly accessible. These crawlers also apply some kind of automatic maintenance, by periodically updating their contents and removing invalid services. Al-Masri et al. [1] show that services registered in public registries are decreasing in contrast with services crawled by search engine's crawlers. In addition to this, more than 53% of the UDDI business registry registered services are invalid, while 92% of Web services cached by Web service search engines are valid and active. Thus, it is more effective, and has become more common, to use search engines to discover Web services, in comparison with UDDI registries.

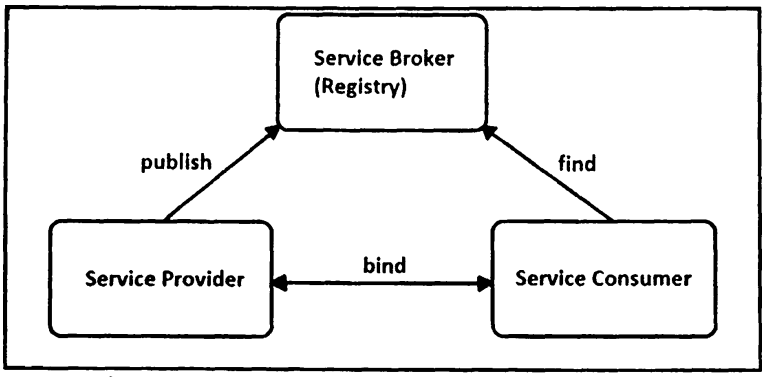


Figure 2.1 Standard service interaction model.

Atkinson et al. [7] created a crawler which crawls source code along with service desriptions, testing four different retrieval techniques. The two of them were signature matching and keyword searching, based on source code. The other two were name matching and abstraction matching, based on the service interface. The experiments showed that name and interface-driven forms of search provide significantly better precision than simple text-based approaches.

Indexing and managing Web services in the same manner as Web pages, though, results in very limited searching choices offered to users, specifically keyword matching on names, locations, businesses, and buildings defined in the Web service description file. This happens because typical search engines indexing processes do not take into consideration the semantic structure of a Web service description, which is the most essential conception regarding the user. For instance, a developer would like to search for a service that offers a function with 2 input parameters, both of type String, and 2 output parameters, one of type Double and one of type Integer. This kind of queries are not supported by search engines. In addition to this, the search

terms entered by the user must partially match those indexed by the search engine for a specific service, i.e. if the query term does not contain at least one exact word such as the service name, the service is not returned. The user must, therefore be aware of the concise keywords in order to retrieve the most relevant services that match the request, however, most developers are mainly concerned with functionality rather than exact naming. A user may not even retrieve services with synonyms on their descriptions, which often leads to low recall. For example, a query looking for "city" may fail to return a service containing "town" in the results.

A naïve approach to confront this problem is to perform a broad matching process which would return a large number of services, most of which may not be of interest to the user. This method would increase the recall but would also decrease the precision of the query results. Another approach that has been proposed is to annotate Web services descriptions with tags coming from a reference ontology. Using tools that exploit the semantic relationships of the ontology's terms, like synonyms, would take advantage of semantic information contained in services descriptions [18, 21, 23]. Nevertheless, such an approach would be an obstacle towards scalability.

A newer approach attempting to overcome these drawbacks is clustering Web services into semantically similar groups. This would dramatically reduce the search space while at the same time improving the matching process, leading to a much better trade-off between query execution time, precision and recall. The main idea is that each cluster has a representative (like summary tags), which is one of its contained services or an abstract one, and a query will search only representatives. If a representative matches the query terms, all the services of the cluster are returned as result, as all these similar services may be of user's interest. Regarding this approach, various methods have been proposed, which either employ well-known clustering algorithms [10, 20, 25] or classification techniques [16].

Authors in [10] presented a clustering approach that uses five key features extracted from WSDL documents in order to group Web services into clusters of functionally similar services. These features are service name, content, types, messages and ports. For the content extraction, they initially parse the whole WSDL document, remove tags and apply word stemming (for example, "connect", "connected", "connecting", "connection" all have the same stem "connect"). Afterwards, they remove function words using a Poisson distribution to model word

occurrence in the document, thus distinguishing function words from content words. Then, k-means algorithm with k = 2 is applied to the extracted words, in order to distinguish general computing words, like "data", "web", "port", etc. from the actual content words. The distance measure that is used between words is Normalized Google Distance (NGD) [8], which is also used to measure the distance between the names of the services. Quality Threshold (QT) clustering algorithm is used to cluster services, with a similarity function that counts for the five aforementioned features. The authors experimented on 400 services gathered from real-world service providers. They applied their approach as well as a similar one, presented in [17]. The latter approach differs only in that it counts for service context and service host name as well. They evaluated the two approaches by measuring the precision and recall of the clusters created. The comparison base was the clusters that the authors manually extracted. Experiments showed that their approach improves the quality of the retrieval, compared with the other approach.

The main drawback of these solutions is that they only support keyword based querying, which, as mentioned before, is not adequate for developers who want to ask structured queries. Taking a step further, in [6], the authors proposed a clustering method that could be used for more advanced queries. The authors of [6] were mainly concerned about the development of an adaptation middleware that provides an abstract level of service reuse, hiding the details of various alternative design options, i.e. the different services with similar functionality. The adaptation takes place, by substituting the concrete services that are hidden behind the composed service abstractions. For that, a systematic approach for extracting service abstractions out of the vast amount of services that are available all over the web, is analysed. The core of this approach is an agglomerative clustering algorithm that takes as input Web service descriptions gathered by crawling the Web, and constructs a hierarchy of service abstractions. The similarity function used for the clustering accounts for names of service, operations, messages and parameters, as well as the types of parameters. Finally, the woogle data set is used for the evaluation of this approach [9]. There were found relatively high percentages of useful abstractions, meaning those that actually represent semantically compatible services.

CHAPTER 3. BACKGROUND - BASELINE

3.1 Overview

The authors in [6] propose the idea of an Abstraction-Oriented Service Base Management (AoSBM), a stand-alone system designed to facilitate service discovery and adaptation, relying on the concept of service abstractions. The AoSBM is available as open-source software under GPL License[1].

Intuitively, a service abstraction represents a group of services that offer similar functional properties; it is characterized by an abstract interface, i.e, a service interface constructed to represent the interfaces of the group of services, and a mapping between the abstract interface and the interfaces of the represented services. The system manages a database in which both information concerning services and abstractions is stored and retrieved via the AoSBM's facilities. To assist in efficient and structured querying, a major need for programmers, the authors of [6] proposed a specific query language called Service Base Query Language (SBQL), a language tailored to the concept of abstractions. The SBQL query is matched against abstractions, instead of being matched against service descriptions of concrete services.

[1] http://www.choreos.eu/bin/view/Documentation/Abstraction_Oriented_Service_Base_Management

Therefore, the query execution time scales up with the number of abstractions, instead of scaling up with the number of available services. Figure 3.1 provides an overview of the main AoSBM facilities.

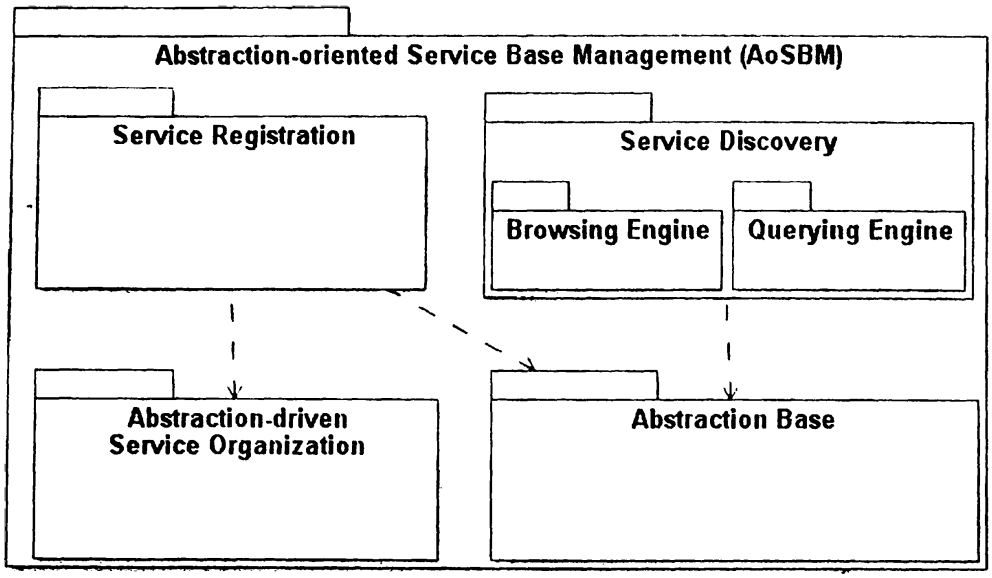


Figure 3.1 Overall architecture of the Abstraction-oriented Service Base Management[2].

- The `ServiceRegistration` facility is responsible for populating the AoSBM with information about services, gathered from collections of service descriptions. The collections of service desriptions are provided by the end-user of the AoSBM. The collections of service descriptions are given as input to the `ServiceRegistration` facility, then they are parsed and transformed to objects that comply with the service model, which we detail later.

- The `Abstraction-driven Service Organization` facility realizes the main algorithms that construct hierarchically structured abstractions for the service model that resulted from the registration of a collection of service descriptions to the `ServiceRegistration` facility. The abstractions comply with the abstractions model, which we detail later. In particular, the `Abstraction-driven Service Organization` facility comprises a hierarchical clustering algorithm that produces

[2] http://www.choreos.eu/bin/download/Share/Deliverables/CHOReOS_WP02_D2.3_CHOReOS-dyn-develop-process-meth-and-tools_V3.0.pdf

clusters of service descriptions, which provide similar functional properties. For each cluster, the hierarchical clustering algorithm constructs a corresponding functional abstraction.

- The `AbstractionBase` is the relational store (developed over MySQL) that is used to store information about service descriptions and service abstractions. The schema of the relational store, which mainly mirrors the service model and the abstractions model, is defined later.

The `ServiceDiscovery` facility provides the basic means for exploring the information that is stored in the `AbstractionBase`. In particular, the `QueryEngine` accepts as input SBQL queries, which are matched against the information that is stored in the `AbstractionBase`, and provides as output service and abstraction related information that satisfies the issued queries.

3.2 Main Memory Model

3.2.1 Service Conceptual Model

Figure 3.2 depicts the representation of the service model concepts that concern services, including service interfaces, service instances, messages and their structure. The center of the service model is the notion of `ServiceInterface` which comprises a collection of `ServiceOperation` objects. The `ServiceOperation` concept comprises information regarding input and output messages. The `Message` concept carries a set of types parameters/fields (coded as message types and components in the Web service model). The `ServiceInterface` concept is further associated with `ServiceInstance` objects that represent information concerning the actual service endpoints (URIs). The `ServiceCollection` concept groups service information that comes from a specific provenance.

3.2.2 Abstraction Conceptual Model

Figure 3.3 depicts the representation of abstractions, in terms of a class diagram. Abstractions form hierarchies that include `FunctionalAbstraction` objects. The

`FunctionalAbstraction` concept contains information concerning a set of structurally similar interfaces. More specifically, the `FunctionalAbstraction` concept is characterized by (a) the set of the interfaces that are represented by the functional abstraction, and, (b) an abstract interface, which stands out as a representative for the represented interfaces. The represented interfaces have been described in the service model of the previous subsection; the representative interface is similarly characterized by `ServiceOperation`, `Message` and `Parameter` objects. The interesting part is that we record the mappings among represented and representative interface (also keeping trace of their similarity in the form of a distance attribute); this happens for interfaces, operations and messages. Abstractions form tree-like hierarchies with abstractions at higher levels being composed as the "union" of a set of abstractions at the lower level.

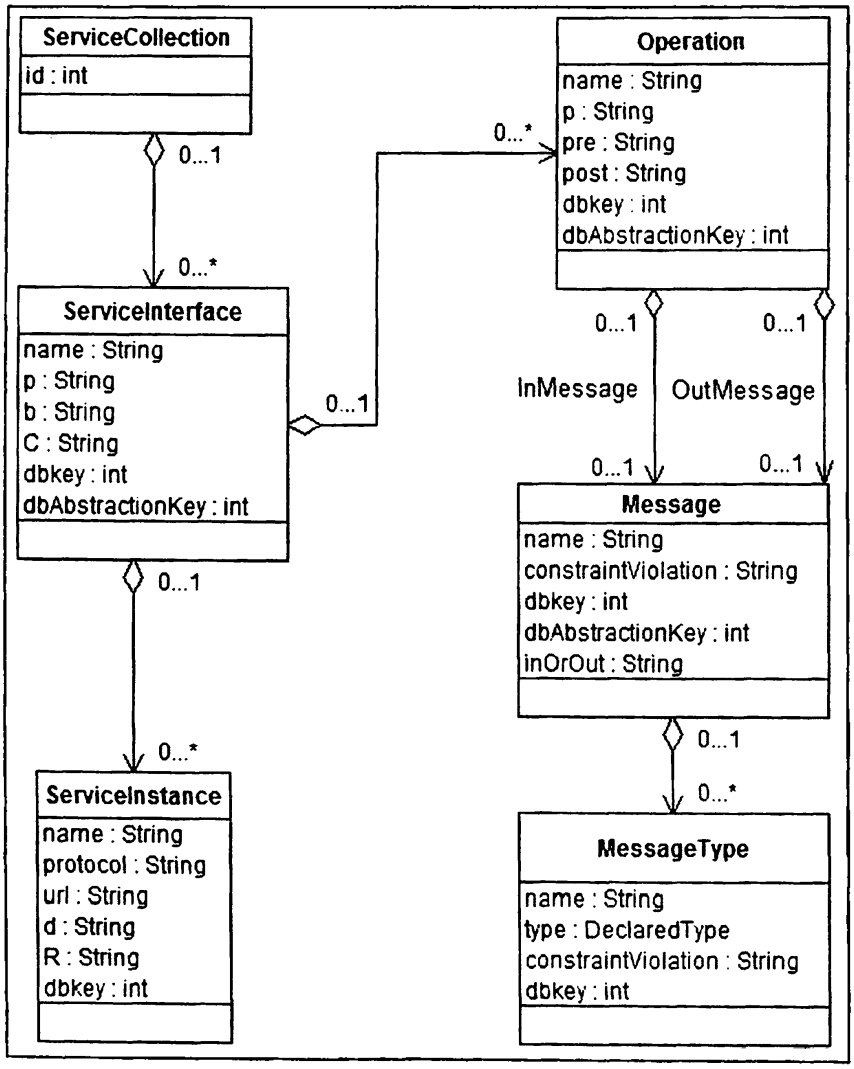


Figure 3.2 Service conceptual model.

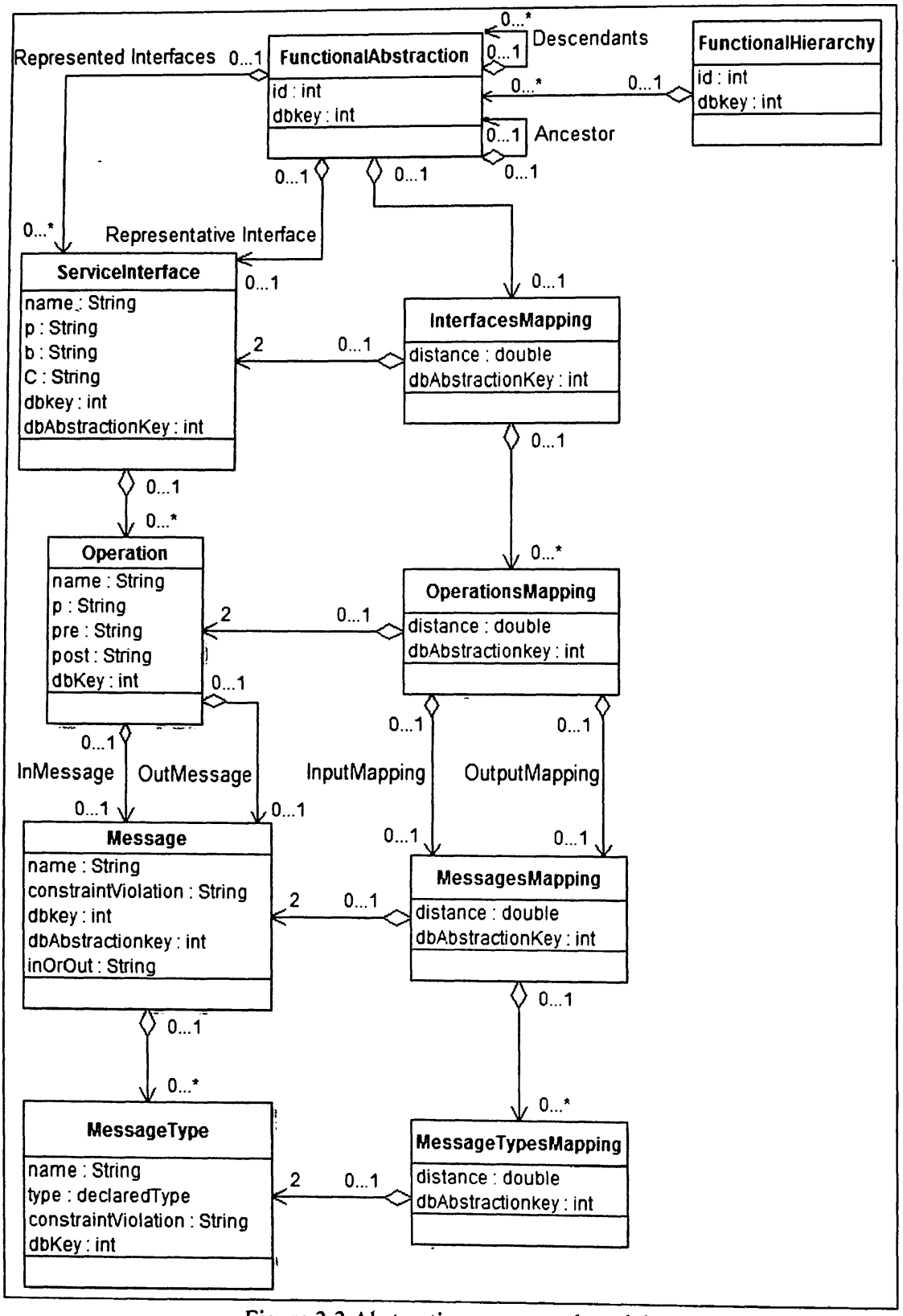


Figure 3.3 Abstraction conceptual model.

3.3 Database Model

We briefly describe the database model concerning the representation of services, service interfaces and their components, service instances and abstractions inside the `Service Base` that is used for persistent storage. The authors in [6] employed a relational DBMS, namely MySQL, for the storage and ultimate querying of the `Service Base`. They stressed that they employed typical and standard features of practically all mainstream relational DBMSs both in terms of representation and querying; therefore the usage of other DBMSs as persistent storage and querying engines is straightforward. Roughly, the database model comprises relations (a.k.a tables) that correspond directly to the concepts of the service representation model. In order to distinguish between the concepts of the service representation model and the relations of the database model, we employ a different naming convention for the relations; relations are named with lowercase letters.

3.3.1 Service Storage Model

Figure 3.4 depicts the part of the database model that concerns services and their internal structure. The notion of aggregation in the object oriented paradigm is modeled via foreign key relationships in the relational paradigm. Observe the upper middle part of the figure; there is a 1:M relationship between relations `serviceinterfaces` and `serviceoperations`, representing the fact that a `serviceinterface` object encompasses one or more `serviceoperation` objects. This is modeled via a foreign key from relation `serviceoperations` to the primary key of relation `serviceinterfaces`; specifically, attribute `OP_SI_ID` in `serviceoperations` is a foreign key (and thus, a subset of) attribute `SI_ID` in relation `serviceinterfaces`. The same pattern appears consistently throughout the database schema. In Figure 3.4, starting from right to left, service interfaces include operations that include messages that include parameters (relation message types). Service instances (bottom of the figure) are also linked to their respective service interfaces for a foreign key.

3.3.2 Abstraction Storage Model

Figure 3.5 depicts the part of the database model that concerns abstractions and how they are related to services. A functional abstraction includes several interfaces that it represents via relation `representedinterfaces`. A representative interface of a functional abstraction

is captured via relation `representativeinterfaces`. A representative interface's decomposition in its parts is captured via the line of relations `representativeoperations` and `representativemessages`. The mappings between the parts of the abstraction's represented interfaces and the respective parts of the components of the representative interface of the functional abstraction are depicted in the middle "line" of the Figure 3.5.

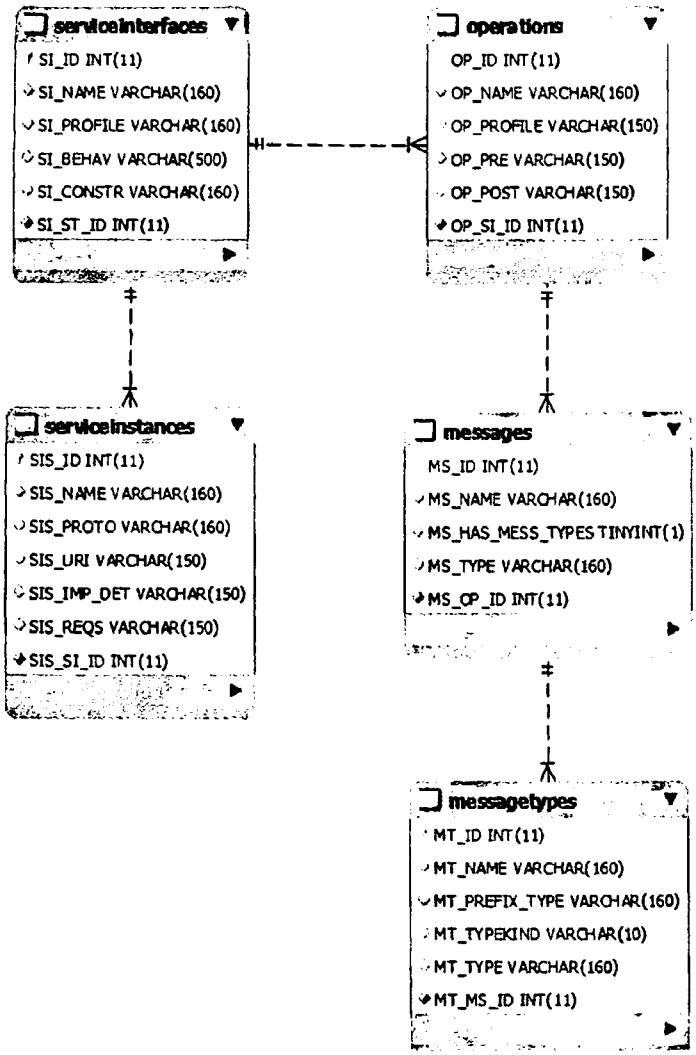


Figure 3.4 Service storage model.

Finally, in Figure 3.6 we give the part of the database schema that concerns the abstractions, their properties and inter-relationships. As one can see, service collections include hierarchies that, in turn, include abstractions. The abstractions form hierarchies and relation `hierEdges` captures the mother-child ancestor-descendant relationship.

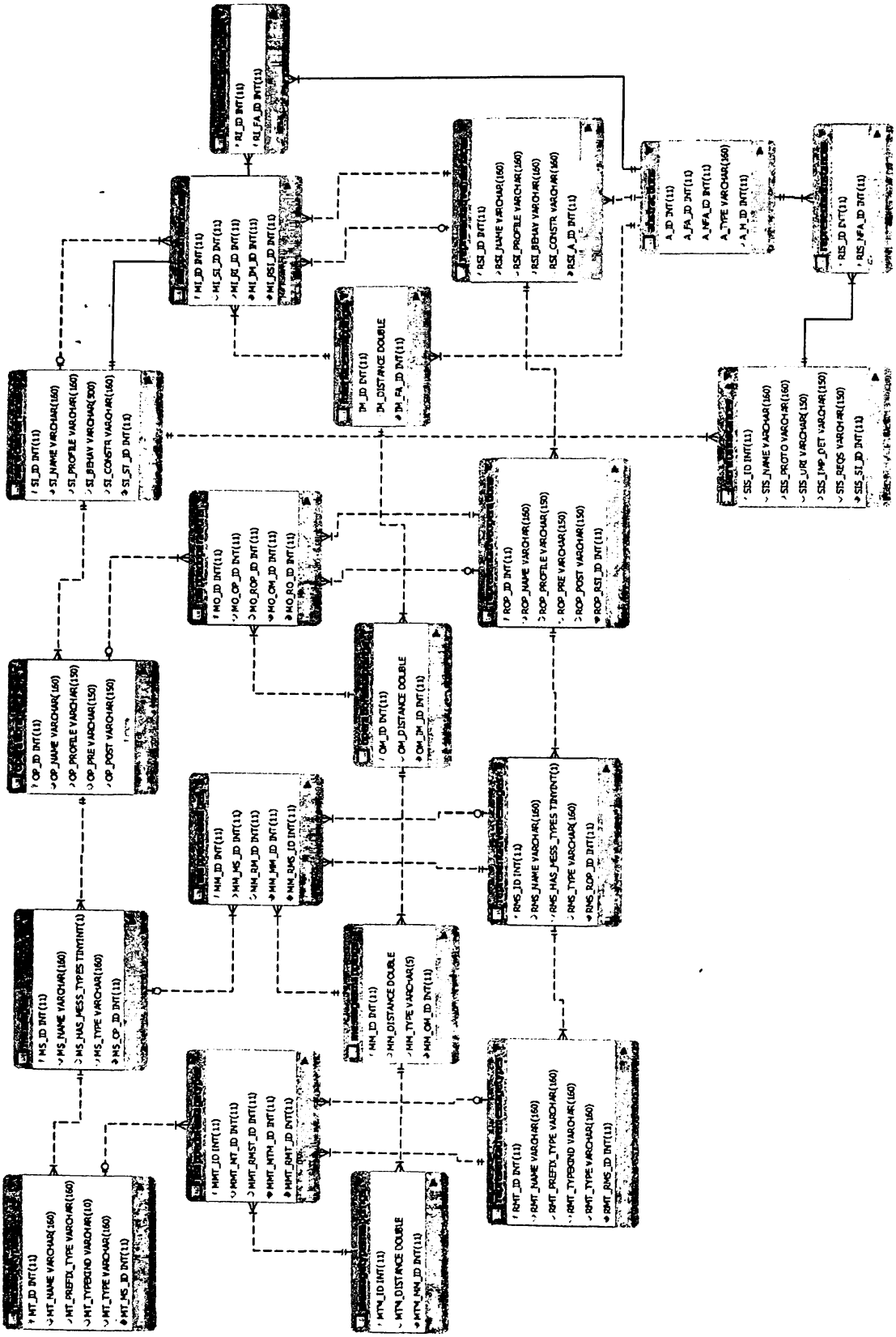


Figure 3.5 Abstraction storage model.

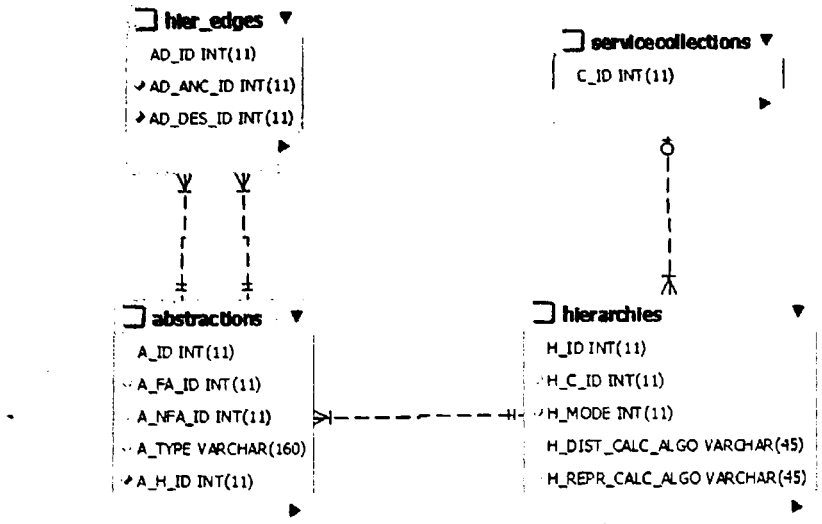


Figure 3.6 The basic database relations concerning abstractions and their hierarchies.

3.4 Service Base Query Language (SBQL)

Hereafter, we present the Service Base Query Language (SBQL). We begin with the main concepts of the language. Then, we give the syntax and the semantics of the SBQL language.

3.4.1 Basic Concepts - Generalized Trees for Querying Services

The querying of the `Service Base` requires the query author to think of the database as a generalized tree. As we will describe later, a generalized tree is a graph that resembles a tree a lot; however there are nodes that break the fundamental property that a tree's non-root node has exactly one father, and consequently, we use the -hopefully intuitive- term generalized trees.

To query the abstraction's part of the `Service Base` schema (i.e., to retrieve abstractions), we think of this part of the schema as a tree. The model that the query author has to keep in mind is depicted in Figure 3.7. We call this tree the *Generalized Tree of the Service Base at the Schema Level* and we textually detail the parts right away.

- A service collection contains several hierarchies of abstractions
- A hierarchy contains several abstractions.
- Functional abstractions have representative interfaces.
- These representative interfaces have operations.
- Each operation has input and output messages.
- Each message has a set of parameters.

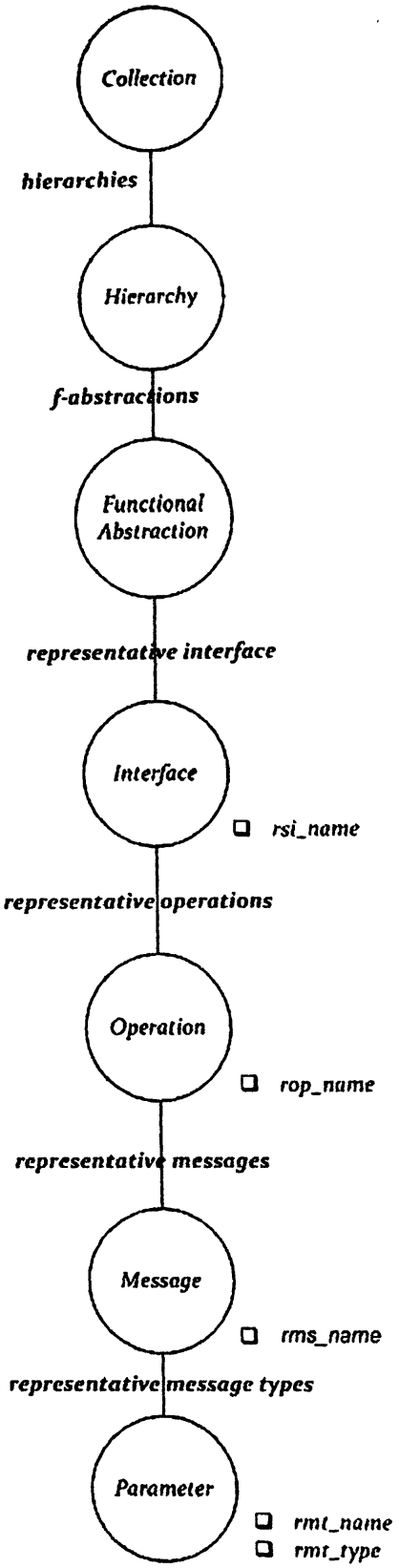


Figure 3.7 The tree that abstracts the structure of functional abstractions at the Schema Level.

```
let         $db = db('localhost/mySB')
for         $c in $db/servicecollections
for         $fa in $c/hierarchies/abstractions
for         $if in $fa/representativeinterfaces
for         $o1 in $if/representativeoperations
for         $o2 in $if/representativeoperations
for         $p1 in $o1/representativemessages/representativemessagetypes
for         $p2 in $o2/representativemessages/representativemessagetypes
for         $p3 in $o3/representativemessages/representativemessagetypes
where
            $if/rsi_name like '%SMSSend%' and
            $op1/rop_name like '%sendMe%'
            $op2/rop_name = 'exactSearch' and
            $p1/rmt_name = 'Sender' and
            $p1/rmt_type = 'String' and
            $p2/rmt_name = 'IP' and
            $p3/rmt_name = 'text'
return
            Abstractions.representativeInfo
```

Figure 3.8 A SBQL query example.

Except for the *Generalized Tree of the Service Base at the Schema Level*, the `Service Base` involves service instances too. Instances are represented via the *Generalized Tree of the Service Base at the Instance Level*. As the abstraction part of the `Service Base` obeys the schema of the *Generalized Tree of the Service Base at the Schema Level*, its contents can form the *Generalized Tree of the Service Base at the Instance Level*.

3.4.2 SBQL Syntax

Figure 3.8 gives an example of an SBQL query that queries the `Service Base` for functional abstractions that are characterized by the following characteristics:

The abstractions that belong to the result of the query:

- have a representative interface whose name includes 'SMSSEND'
- have two operations with the following characteristics
    - The first operation has

- a name which include the text 'sendMe'
- an input message with two parameters: (a) a parameter with type 'String' and name 'Sender' and (b) a parameter with name 'IP'

- The second operation has an output message with a parameter with name 'text'

The query returns to the user all the information concerning the functional abstractions that fulfill the aforementioned criteria.

The general syntax of a SBQL query is given in Figure 3.9.

```
let       databaseSpecifier
          variableDefinitionArea
[where    filterList]
return    returnExpression
```

Figure 3.9 The general syntax of a SBQL query.

As can be seen, the syntax of the language largely follows XQuery. The reserved words that distinguish the different parts of an SQBL query are presented with underlined format. We discuss each of these parts separately in the sequel, after we have formally defined their constituent elements. These are:

- **Variables.** A variable is an alphanumeric string that begins with a dollar sign '$'.
- **Path Expressions.** A path expression is of the form

  $variable_0/edge_1/.../edge_n$

  A path expression is well-defined if the sequence of edge names creates a linear path in the generalized tree of Figure 3.7.
- **Variable Definition.** A variable definition is of the form

  `for variable in pathExpression,`

  i.e., `for variable in` $variable_0/edge_1/.../edge_n$

  It is worth noting here that variables have a type, defined by the last edge of their path expression. In our abstract notation, $variable_0$ has type $edge_n$.
- **Filters.** A filter is an expression of the form

  `variable/field Θ value, Θ ∈ { =, LIKE, <, >, <=, >=, <>}`

- **Database Specifier.** A database specifier characterizes which database we will query. The syntax of a database specifier is

  ```
  databaseVariable = db(database)
  ```

  where `databaseVariable` is going to be used in subsequent variable definitions and `database` is a string with the name of the database that we query (as understood by the underlying DBMS).
- **Variable Definition Area.** A list of variable definitions. Variable definitions are separated by newlines in the variable definition area.
- **Filter List.** A list of filters. More than one filters are connected by `and` connectors in the filter list. The `where` clause is optional and consequently, the list may be empty; in this case we assume that a filter with semantics true is implicitly implied.
- **Return Type.** The return type dictates what the ultimate result will be in terms of main-memory representation and it is an expression of the form

  ```
  Abstractions.returnType
  ```

  where:

  ```
  returnType ∈ {RepresentativeInfo, fullObject}
  ```

3.4.3 SBQL Semantics

The semantics of the query language are largely based on the correct definitions of the individual parts of a SBQL query. We will employ the term *well defined* to refer to the individual parts whose declaration by the user makes sense.

- **Well Defined Variables.** Every variable definition in the variable definition area involves two variables, specifically, (a) the declared variable at the beginning of the definition and (b) an auxiliary variable at the beginning of the involved path expression.Then, every variable has

  - An abbreviated path, which is the one appearing in the variable definition area
  - A full path that is produced if we replace the auxiliary variable in the path expression with its own full path

  For this recursive definition to work, we need to define the full path for the auxiliary variable(s) that appear in the database specifier, which is the empty set.

A variable is *well defined* if its full path is a continuous path in the Generalized tree of the Service Base at the Schema level, starting at collections and ending at the type of the variable, as a simple line.

- **Well Defined Filters.** A filter is a triple of the form

```
variable/field Θ value, Θ ∈ { =, LIKE, <, >, <=, >=, <>}
```

Assuming the variable to be of type *T*, filter is well defined if the field appearing in the filter's expression belongs to type *T*.

In the sequel we assume that all variables and filters are well defined; if not, the query returns an error code and an empty result set.

- **Query semantics.** The semantics of a query, i.e., the list of returned objects that correspond to the application of the query expression over an arbitrary service base are defined via the sequence of the following four steps.

1) The query takes as input the Generalized tree of the Service Base at the Instance Level.
2) Each node is intended to be annotated with a variable as prescribed in the `let` clause of the query. For every abstraction appearing in the `let` clause of the query, all possible clones of its subtree with all the applicable combinations of variable assignments are produced.
3) Every such clone is passed via the set of filters prescribed in the `where` clause of the query. The semantics of the filter list are conjuctive; in other words, for a subtree to become part of the result, all the filters of the filter list must evaluate to true (see next). These subtrees are called *survivor* subtrees.
4) For every survivor subtree, its return graph of objects is computed, as prescribed from the `return` clause of the query.

- **Filter Semantics.** The semantics of a filter are as follows:

- The input to the filter is a subtree of an abstraction as previously defined.
- The filter annotates a node in the tree. The node which is annotated is the one resulting from the full path of the variable.
- The `variable/field` part of the filter's expression is replaced with the respective value of the node.
- If the resulting expression evaluates to true then the input path is added to the result of the filter, i.e., its output; else nothing is added to the output.

A filter list is the conjunction of filters and each of them is applied to the appropriate node. A subtree survives if all its filters evaluate to true. If the filter list of a query is empty, we assume that a single filter with semantics *true* is added; thus all subtrees evaluate to true without further checking.

- **Query Completion for the Return Type.** After all survivors have been computed as mentioned before, the part of their information that will constitute the final result depends on the result type of the query. Specifically, for every survivor the following is returned:

  - If the return type is `RepresentativeInfo`, the information returned is the one related to the `representativeInterface` attribute of the `FunctionalAbstraction` class. What is returned is actually the full `ServiceInterface` object that stores the information about the representative interface of the functional abstraction, and its components, belonging to classes `Operation`, `Message` and `MessageType`.
  - If the return type is `fullObject`, the full `FunctionalAbstraction` object is returned.

3.5 Mining Service Abstractions

3.5.1 Basic Concepts

Figure 3.10 defines the basic concepts regarding services, their components and service functional abstractions.These have already been depicted in the previous section, however a more formal view is utilized here, for the needs of the abstractions mining specification.

```
ServiceInterface = (n : String, O)
O = {op_i : Operation}
Operation = (n : String, In : Message, Out : Message)
Message = (n : String, Ps)
Ps = {p_i : MessageType}
MessageType = (n : String, type : XMLDataType)
XMLDataType = Builtin | Complex
Abstraction = (I : ServiceInterface, D, M)
D = {s_i : ServiceInterface}
M = {m_si : I.O → s_i.O}
```

Figure 3.10 Definitions of basic concepts.

A service interface is specified in terms of a name and a set of operations, *O*. Each operation is characterized by a name, an input message, *In*, and an output message, *Out*. In general, a message is hierarchically structured, consisting of a number of message types (also called "elements" or "parts"), characterized by their names and their XML data types. The data type of a particular message type could be either built-in or complex (i.e., a hierarchically structured element, consisting of further built-in or complex data types). In their mining process, the authors in [6] consider only the leaf elements of the message type's hierarchical structure. The reason for this choice is that the particular structure of the input and output data of an operation adds further complexity, while not providing much useful information to the mining process.

Ideally, a service abstraction should represent a set of available services that have in common a certain set of semantically compatible functionalities, realized by corresponding sets of operations, which most possibly would be syntactically different. Finding within a given set of services that were gathered by crawling the Web, services that provide common semantically compatible functionalities is very hard. However, is has been empirically observed that it is very frequently encountered to have semantically compatible services that provide syntactically similar interfaces.

$$D_I(s_i, s_j) = \frac{NED(s_i.n, s_j.n)}{2} + \frac{\sum_{\forall (op_i, op_j) \in M_{op_{ij}}} D_{op}(op_i, op_j)}{2 * |M_{op_{ij}}|} \quad (1)$$

$$D_{op}(op_i, op_j) = \frac{NED(op_i.n, op_j.n)}{2} + \frac{D_{io}(op_i, op_j)}{2} \quad (2)$$

$$D_{io}(op_i, op_j) = \frac{D_m(op_i.In, op_j.In)}{2} + \frac{D_m(op_i.Out, op_j.Out)}{2} \quad (3)$$

$$D_m(m_i, m_j) = \frac{\sum_{\forall (p_i, p_j) \in M_{m_{ij}}} D_p(p_i, p_j)}{|M_{m_{ij}}|} \quad (4)$$

$$D_p(p_i, p_j) = \frac{NED(p_i.n, p_j.n)}{2} + \frac{ND_T(p_i.type, p_j.type)}{2} \quad (5)$$

Figure 3.11 Distance formulas between service constituents.

Then, to assess the similarity between two service interfaces, the authors in [6] rely on a distance metric D_I, which is defined as follows (Figure 3.11). Given two interfaces s_i, s_j and a mapping $Mop_{ij} \subset s_i.O \times s_j.O$ between the most similar operations of the interfaces, the distance $D_I(s_i, s_j)$ is defined as the average of the Normalized Edit Distance (NED) between the names of the interfaces, and the average of the distances between the mapped operations. The distance $D_{op}(op_i, op_j)$ between two operations op_i, op_j is defined as the average of the normalized edit distance between the names of the operations and the average of the distances of their input and output messages. Given a mapping $Mm_{ij} \subset m_i \times m_j$ between the most similar parts of two messages m_i, m_j, the distance between the messages is defined as the average of the distances between the mapped parts. Finally, the distance between two message parts is defined as the average of the normalized edit distance between their names and the normalized distance between their build-in types $ND_T(type_i, type_j)$; if these types are in the same branch of the standard XML type hierarchy, then $ND_T(type_i, type_j)$ is the absolute difference of their depths, divided by the maximum height of the XML type hierarchy (see Figure 3.12), otherwise it is assumed that the types are incompatible and $ND_T(type_i, type_j) = \infty$.

Based on the previous concepts, a service abstraction is defined as a tuple that consists of: an abstract interface I and a set of represented service interfaces D (Figure 3.10). Each operation

of the abstract interface I is mapped, through a set of mappings M, to a set of operations, provided by the represented interfaces. Specifically, for each service interface s_i of D, M comprises a one-to-one function ms_i between the operations of I and the operations of s_i.

Finally, it should be noted that, in general, the interface $a.I$ of a service abstraction a may be included in the set of interfaces $a'.I$ of another service abstraction a'. In other words, it is possible to define a hierarchy that consists of higher level service abstractions, which represent lower level service abstractions.

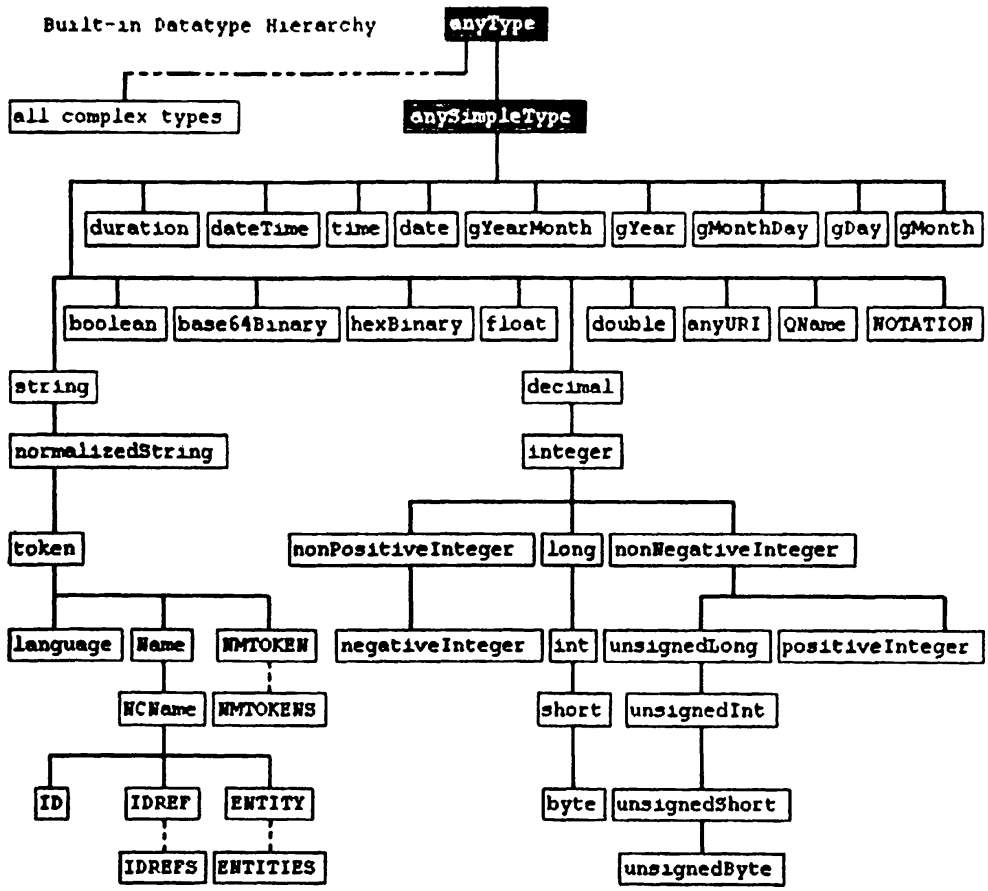


Figure 3.12 The standard XML type hierarchy [24].

3.5.2 Agglomerative Clustering

The ultimate goal of the mining process is to construct the interfaces of service abstractions, along with mappings between these interfaces and the interfaces of the represented services. Following, we discuss the core steps of the proposed algorithm, while the interested reader may

refer to the technical report [4] for further technical details. The mining algorithm accepts as input a set of interfaces $S = \{s_i : ServiceInterface\}$. The output of the algorithm is a set of hierarchically structured service abstractions $A = \{a_i : Abstraction\}$. To this end, the algorithm iteratively performs the following steps:

Step 1: For every pair of interfaces $s_i \in S$, $s_j \in S$ the algorithm finds the distance $D_I(s_i, s_j)$. To this end, the most similar pairs of operations $(op_i, op_j) \in s_i.O \times s_j.O$ (i.e., the mapping Mop_{ij} - Subsection 3.5.1) are found by solving the maximum weighted matching problem in a bipartite graph [19]. The nodes of the graph correspond to the operations of s_i and s_j, while the edges correspond to the distances between the operations. Finding the distances between two operations $op_i \in s_i.O$ and $op_j \in s_j.O$ involves finding the most similar pairs of elements for the input messages (respectively the output messages) of the operations (i.e., the mapping Mm_{ij} - Subsection 3.5.1). This problem is also solved by solving the maximum weighted matching problem in a bipartite graph that represents the input messages (respectively the output messages). Note that in this step it is possible to calculate a distance between two interfaces that equals to ∞. This case may come up in two circumstances:

- The first possibility occurs if the best possible matching between messages results in at least one pair of incompatible types. In such a case, it is considered that it is not possible to create an abstraction out of the two interfaces.
- The second possibility occurs if the distance calculated exceeds a threshold that authors in [6] have set.

Step 2: Based on the calculated distances the most similar pair of interfaces (s_i, s_j) is selected and an abstraction a is constructed as follows: By convention, the name of $a.I$ is the longest common substring of the names of s_i, s_j. For every pair of matched operations (op_i, op_j) found in the previous step, $a.I$ comprises a corresponding operation op_a, named by following the same convention. The input (respectively output) message of op_a, contains a message part p_a for every matched pair (p_i, p_j) of elements of the input (respectively output) messages of op_i, op_j. Concerning the type of the input (respectively, output) element p_a, we have $p_a.type = p_i.type$ if $p_i.type$ is higher (respectively lower) than $p_j.type$ in the standard XML type hierarchy; otherwise, $p_a.type = p_j.type$.

Step 3: The abstraction a is included in the result, i.e., $A = A \cup \{a\}$. Moreover, the services that are represented by a are removed from the input set, i.e., $S = S - a.D$. Finally, $a.I$ is included in S, i.e., $S = S \cup \{a.I\}$, so as to serve for the construction of higher level abstractions.

Step 4: The mining process repeats steps (1) to (3), until the input set comprises only one element, namely, the root abstraction of the resulting abstraction hierarchy A, which generalizes all the available service interfaces, or until no further abstractions can be recovered.

CHAPTER 4. DISTRIBUTED ABSTRACTIONS MINING

4.1 Overview

AoSBM posed some serious challenges towards, mainly, computational resources consumption and scalability. The process of functional abstractions mining that the authors in [6] proposed is not aimed at service discovery and cannot support large collections of services. Specifically, using a conventional personal computer, we found that the process is feasible only for a few hundreds of services, before the main memory is exhausted and the algorithm crashes. Even if we tackle this by providing much more memory in some way, the process is also very time

consuming, so that the number of services it could manipulate would not exceed a few thousands.

Our contribution is facilitating the organization of large unstructured collections of service descriptions, building upon the AoSBM's conceptional models and its abstractions mining algorithm. In particular, we have developed a distributed facility for abstractions mining, which can be executed in a set of computers (nodes) and exploits the proposed abstractions mining algorithm, extended with a pruning technique that retains only a part of the most useful abstractions, thus improving the effectiveness and feasibility of the overall abstractions mining process.

4.2 General Idea

Figure 4.1 illustrates a coarse-grained aspect of the architecture of our distributed abstractions mining facility. The rectangle at the bottom represents our extension to AoSBM, i.e., a standalone component which also uses remote software components to carry out its functionality. The elliptical objects represent the software components installed at the computational nodes that are to participate in the distributed process execution. These components form a tree structure. Intense consecutive arrows represent a call to the respective software component (the one at the arrow's target), while numbers on them correspond to the phase they are activated. The overall process comprises the following basic phases, which we briefly mention here and further detail in the following subsections.

Phase 1: The input collection of service descriptions is divided into a number of subcollections, equal to the number of leaf nodes. This phase is executed at the standalone component.

Phase 2: Subcollections resulting from Phase 1 are passed to a master node, which acts as the connector between the standalone component and the remote software components.

Phase 3: The master node distributes the subcollections to the leaf nodes, by calling the respective software component.

Phase 4: Each software component of a leaf node executes the following process (also depicted by the brief textual description beside the respective elliptical objects): Firstly, it mines

abstractions out of the subcollection of services, by employing the local algorithm proposed in [6], thus producing a hierarchy of functional abstractions. Then, by employing a pruning technique that we analyze later on, it retains a part of the abstractions of the initial hierarchy, thought to be the most useful ones. No hierarchical structure is formed between the retained abstractions. Next, the retained abstractions are passed to the parent node, which is an internal node, via a call to the respective software component.

Phase 5: Each software component of an internal node executes the following process (as depicted by the brief textual description beside the respective elliptical objects): Firstly, it gets the abstractions from the two child nodes and produces their union, i.e., a single collection of abstractions. Then, it mines abstractions out of this collection, by employing the algorithm proposed in [6], thus producing a hierarchy of functional abstractions. Then, by employing the pruning technique we mentioned before, it retains a part of the abstractions of the initial total hierarchy (forming no hierarchy). Finally, the retained abstractions are passed to parent node, via a call to the respective software component. This phase is executed repeatedly, starting from the leaves of the first category tree and ending to its root.

Phase 6: The initiator standalone component calls root node to get the final result.

As can be deduced from Figure 4.1, the software components that must be installed to the depicted tree's computational nodes so that the proposed architecture functions, should offer two functionalities, one that executes the abstractions mining algorithm proposed in [6], and one that executes our pruning technique. As for the master node, here there is a need for a functionality that serves for the master's broker facilities, i.e., calling the leaf nodes by also passing them the subcollections, getting final results fron the root node and sending them to the standalone component.

We deployed these functionalities as services, which offer respective methods. As our architecture dictates, the execution of our distributed abstractions mining facility forms a choreography, with one service calling another, until the standalone component gets the final results from the root node.

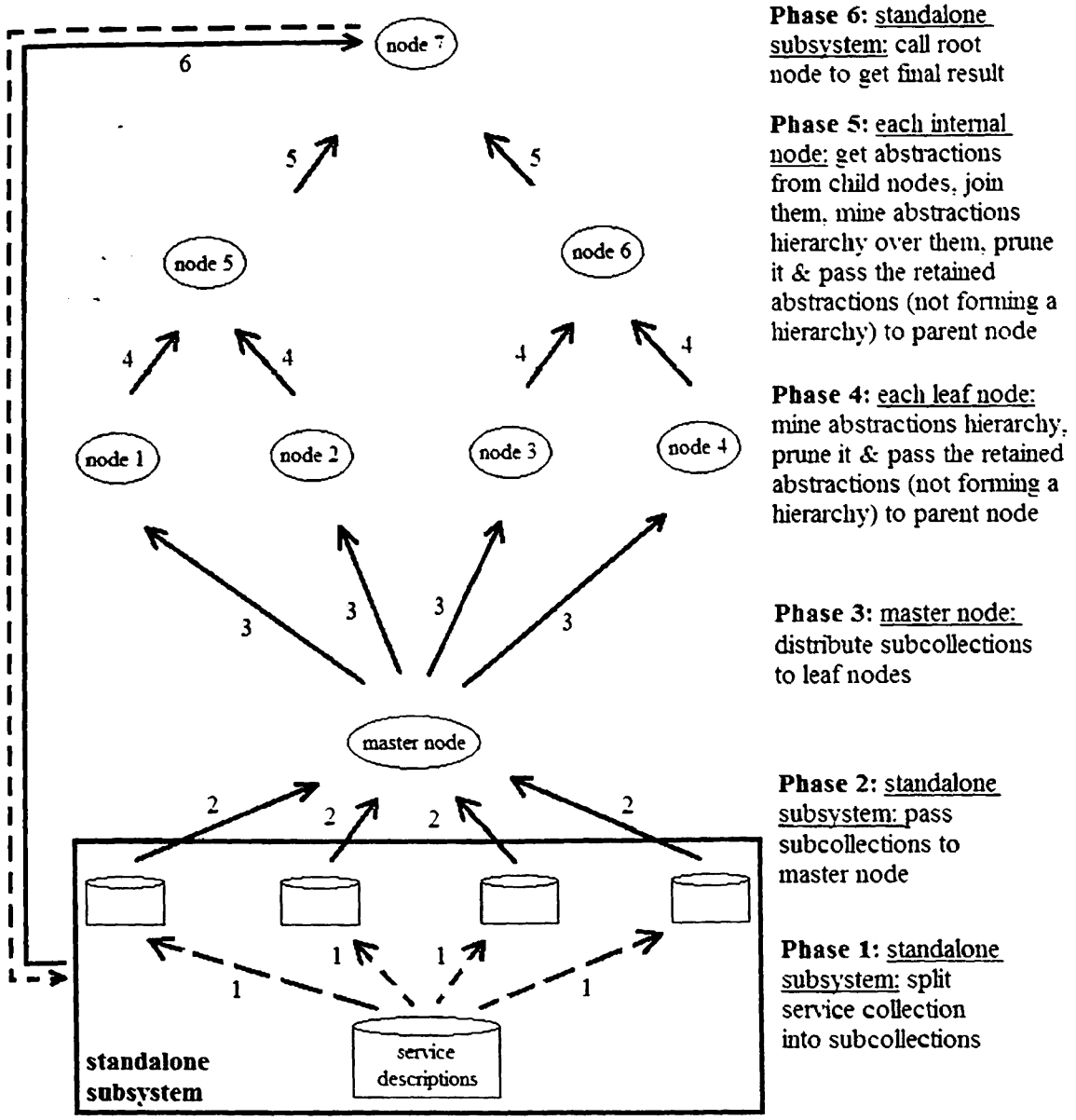


Figure 4.1 General architecture of distributed abstractions mining facility.

4.3 Phase 1. Standalone Subsystem: Service Collection Splitting

As we mentioned in the previous subsection, at Phase 1, our standalone AoSBM component divides the input service collection into a number of subcollections, equal to the number of leaf nodes. The purpose of this step is that each subcollection will later be passed, via the master node, to a leaf software node. At this point one would wonder how this function is aware of the

available master node, the rest software nodes, and their exact tree structure, so as to know what the leaves are. The fact is that the available nodes and the master node are provided as input by the user, which must define this input with a simple text file, as the one presented in Figure 4.2; The information needed for each available node is its binding URI, i.e., the URI at which the respective service is deployed. Each URI must be written in a separate line, starting from the first line, with no blank lines between URIs. The first line is where the master node's URI is provided, whereas the other lines are for the tree's computational nodes.

```
http://192.168.20.0:8082
http://171.1.30.1:8083
http://192.168.20.3:8082
http://155.161.27.2:8088
http://90.151.20.1:8090
http://192.172.10.5:8082
http://192.12.20.4:8082
http://92.128.34.1:8092
```

Figure 4.2 An example of a user-defined file of available nodes.

An issue that we must make clear is that the URIs in the text file correspond to the hardware components (real computers) that are available to the user. The hypothesis is that each URI provided by the user, corresponds to a service deployer (server), which can serve all services we have mentioned, i.e., services for abstractions mining and pruning, so all services must be bound to this server. This means that each node appearing in the hierarchy of Figure 4.1 regards a specific role (service), i.e., regards a software and not a hardware component. Therefore, a hardware component may correspond to more than one nodes in the nodes hierarchy, each time with a different service. Later on this subsection, we analyze how our system automatically configures the tree of software components, by taking as input this text file, and another input, specifically, the number of hardware components that will be used.

It is possible that the user may not want all components in the text file to be used (e.g., for experimental purposes or because some of them are not on line any more, etc.). Let c be the number of components, excluding the first component that regards the master node, and p be the user-defined preferable number of components to be used in our facility. If $p <= c$, then the first p components of the text file (excluding the very first one) are going to be used, otherwise

an error message is thrown. Of course, the very first component is used in any case, as it regards the master node, but if this component is supplied with other services too, besides the master service, then it can also play a role in the nodes hierarchy and, if the user wants so, he must include one more copy of its URI in the text file.

Phase 1 starts by reading the text file of available nodes. Based on that, as well as on the input number of nodes to use, a software components tree, like the one in Figure 4.1, is constructed. A valid number of nodes to be used, must be equal to or bigger than 2. The tree structure constructed is a full, balanced, binary tree, i.e., each internal node has exactly two children, and balanced, as, for each internal node, its children's depths differ by at most one. A valid number of service descriptions contained in the input collection, must be equal to or bigger than the number of nodes that are dictated to be used. Subsequently, the input collection of service descriptions is divided in as many equal parts as the number of leaves in the nodes tree. This means that each part (subcollection) consists of the same number of service descriptions, except for the case in which the division leaves a remainder; in such case, the remaining service descriptions are inserted to some of the created subcollections arbitrarily. The division takes place in random manner.

The reason why we chose a balanced tree can be inferred from Figure 4.3, in which two different tree configurations are compared, with the textual descriptions beside the nodes indicating the data inputs and outputs to and from the nodes, during execution. For ease of presentation, Figure 4.3 does not take into consideration the pruning facility. The fact that Phase 1 divides the input service descriptions collection randomly into equal-size groups leads to the fact that the leaf nodes, that work concurrently, consume approximately the same time for their processing. Left-deep alternative shows a non-balanced tree, whereas balanced alternative shows a balanced one. The hypothesis is that in both occasions, each leaf node takes 10 service desriptions as input. At time t_1 all leaves have (approximately) concurrently finished their processing. Similarly, at time t_2 nodes 5 on both occasions along with node 6 of balanced alternative, finish concurrently. But then we observe that in left-deep alternative, two serial activities remain, one with an input of 26 abstractions and one with an input of 34 abstractions, whereas in balanced alternative, only one activity with an input of 34 abstractions remains. The conclusion is that the balanced tree configuration leverages more parallelism of activities.

Another issue concerning the configuration is how the input nodes the user defined via providing the text file along with the number of nodes he prefers to be used, are configured. A straightforward solution one could imagine is simply using each software component provided by the user, to be represented by one hardware component in the tree configuration. For example, if a user defined that 7 hardware components should be used, namely 1, 2,..., 7, then we could use the exact configuration illustrated in balanced alternative of Figure 4.3, i.e., having the hardware components forming this hierarchy, each participating with its respective role (service), as depicted in Figure 4.1.

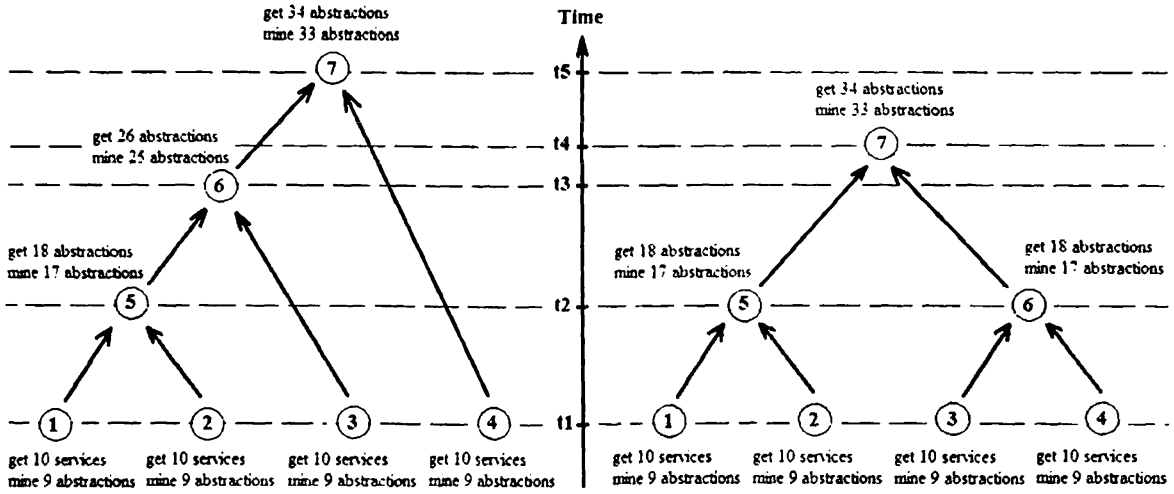


Figure 4.3 Comparing two different tree configurations for seven software nodes.

Instead, we chose to spend computational resources more effectively, by having some hardware components assigned to more than one software component node in the tree configuration. For example, in balanced alternative of Figure 4.3, for the bottom level of nodes we should use four different hardware components to exploit the fact that they work concurrently, however, for the above level it would be a waste to use other hardware components, since the hardware components that correspond to their children are free of work at this point, so they can be reused.

Figure 4.4 shows how our facility would assign hardware components to the configuration of software components for the balanced alternative of Figure 4.3. Only four computational sources would be needed. Figure 4.5 illustrates a realistic example of the configuration our

system would produce in case the user defines the text file of Figure 4.2 as the available nodes, and number 4, as the number of nodes to be used by our facility.

Following, we detail the algorithm we use to configure the user-input hardware components, i.e., to produce the corresponding software components tree. Figure 4.6 presents our `Node` class, which serves for the basic construct of our tree. It has four attributes, namely `url`, for the respective service's url, `parent`, for the node's ancestor in the tree's hierarchy, and `child1`, `child2`, for the node's descendants in the tree's hierarchy. The class constructor does not set the `parent` attribute, as parent value is not known at the point the constructor is used in our tree construction algorithm. Actually, the `setParent()` operation is used for this purpose, later on. Ancestor information is necessary for finding the path from the leaf nodes to the root. As we mentioned before, this path is passed by the master node to the leaf nodes, so that they become aware of the service calls chain till the completion of the distributed abstractions mining process.

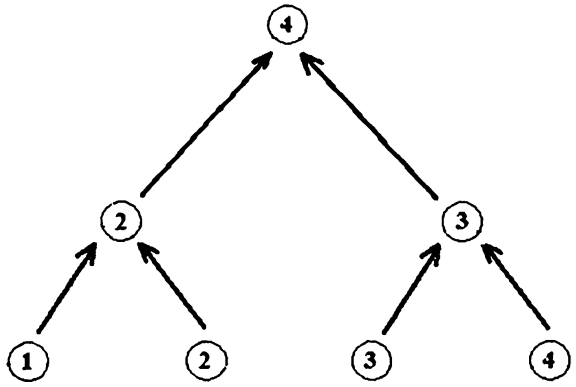


Figure 4.4 Configuration for four hardware components.

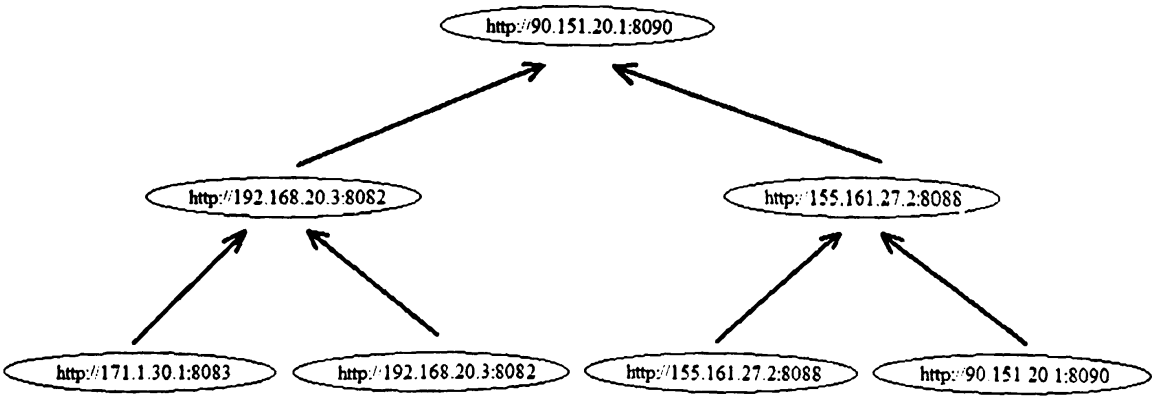


Figure 4.5 Configuration for four hardware components, based on Figure 4.2.

Figure 4.7 presents the algorithm, in pseudocode, that constructs the software components tree. Function `createSoftwareComponentsTree()` takes as input a list containing the URLs of the software components which the user demands to be used, by providing the aforementioned file and the number of components to be used. Produces as output a `Node` object, standing for the root of the tree, from which we can navigate through the entire tree. Generally, the tree is constructed bottom-up, starting from the leaf nodes and creating ancestors till the root is created. The construction is done by levels, each one in an iteration of the while loop.

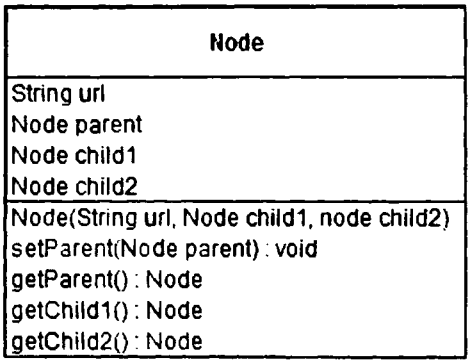


Figure 4.6 The structure used for software components tree node.

Initially the function fills two queues. The `currentLevel` queue represents the upper tree level constructed for the moment and, initially, at the first `foreach` loop, it is filled with the leaf nodes that are created. The `internalNodesURLs` queue stores the URLs of the internal software components and, initially, it is filled with all URLs in `softwareComponentsURLs`, but the first; this can be deduced from the fact that we reuse components, and since such a tree like ours would always have one less internal node than the number of leaf nodes. However, the choice to leave out the first service of the list is arbitrary, i.e., we could choose any other as well. Each time an ancestor is constructed, an element of this list is removed.

The `while` loop's functionality is as follows:

- The nested `while` loop removes `currentLevel` nodes per pair, constructing an ancestor over them, whose URL is obtained by removing an element from `internalNodesURLs` queue. Ancestors are inserted in nextLevel queue.

- The `if` clause deals with the rest one node of `currentLevel` node that could probably remain. In such case, an ancestor is constructed over the remaining node from `currentLevel` and one node from `nextLevel`. This way we achieve the tree to be balanced.
- Finally, `nextlevel` becomes the `currentLevel` and the loop is going for a new repetition, until no more elements remain in `internalNodesURLs`.

At the end, the function returns the only element of `currentLevel` at that point, which is the tree root node.

```
Node createSoftwareComponentsTree(List<String> softwareComponentsURLs) {
   Queue<Node> currentLevel = new LinkedList<Node>();
   Queue<String> internalNodesURLs = new LinkedList<String>();

   foreach scURL in softwareComponentsURLs
      currentLevel.add( new Node(scURL, null, null) );
   foreach scURL in softwareComponentsURLs except for the first
      internalNodesURLs.add(scURL);

   while (internalNodesURLs is not empty) {
      Queue<Node> nextLevel = new LinkedList<Node>();
      while (currentLevel.size() >= 2) {
         Node node1 = currentLevel.remove();
         Node node2 = currentLevel.remove();
         Node internalNode = new Node(internalNodesURLs.remove(), child1, child2);
         nextLevel.add(newInternalNode);
      }
      if (currentLevel.size() == 1) {
         Node node1 = currentLevel.remove();
         Node node2 = nextLevel.remove();
         Node internalNode = new Node(internalNodesURLs.remove(), child1, child2);
         nextLevel.add(newInternalNode);
      }
      currentLevel = nextLevel;
   }

   Node rootNode = currentLevel.remove();
   return (rootNode);
}
```

Figure 4.7 The software components tree configuration in pseudocode.

Figure 4.8 presents an example of an application of this algorithm to a set of seven service URLs, hypothetically the user dictates to be used. The figure is divided in 7 frames (steps), each one illustrating the tree's part constructed (at the left side) and the queues' contents (at the right side) up to the step's start, while the circular and rectangular drawings around queues' elements stand for the removals of these elements from the queues, during the step. After a removal of three elements (i.e., two children and a future ancestor), the ancestor is constructed over the two children, which is illustrated in the next frame's left side.

In Figure 4.8, we suppose that the two foreach loops of the algorithm of Figure 4.7 have constructed the leaf nodes and have set the queues, as presented in frame 1. The shaped drawings in frame 1, as well as the rest of the frames detail the actions performed by the external while loop of the algorithm. Below we give a concise description of these actions:

- 1st iteration of external while loop (steps 1-4)
  - Step 1: Ancestor 2 is constructed over the leaf nodes 1 and 2.
  - Step 2: Ancestor 3 is constructed over the leaf nodes 3 and 4.
  - Step 3: Ancestor 4 is constructed over the leaf nodes 5 and 6.
  - Step 4: Ancestor 5 is constructed over the leaf node 7 and the internal node 2.

- 2nd iteration of external while loop (steps 5-6)
  - Step 5: Ancestor 6 is constructed over the internal nodes 3 and 4.
  - Step 6: Ancestor 7 is constructed over the internal nodes 5 and 6.

- 3rd iteration of external while loop (step 7)
  - Step 7: loop terminates

When the configuration is completed, the standalone component not only is aware of the exact path of calls for each leaf-to-root choreography, but also knows which the root is, so that it can call it at the end (phase 6) to get the final result.

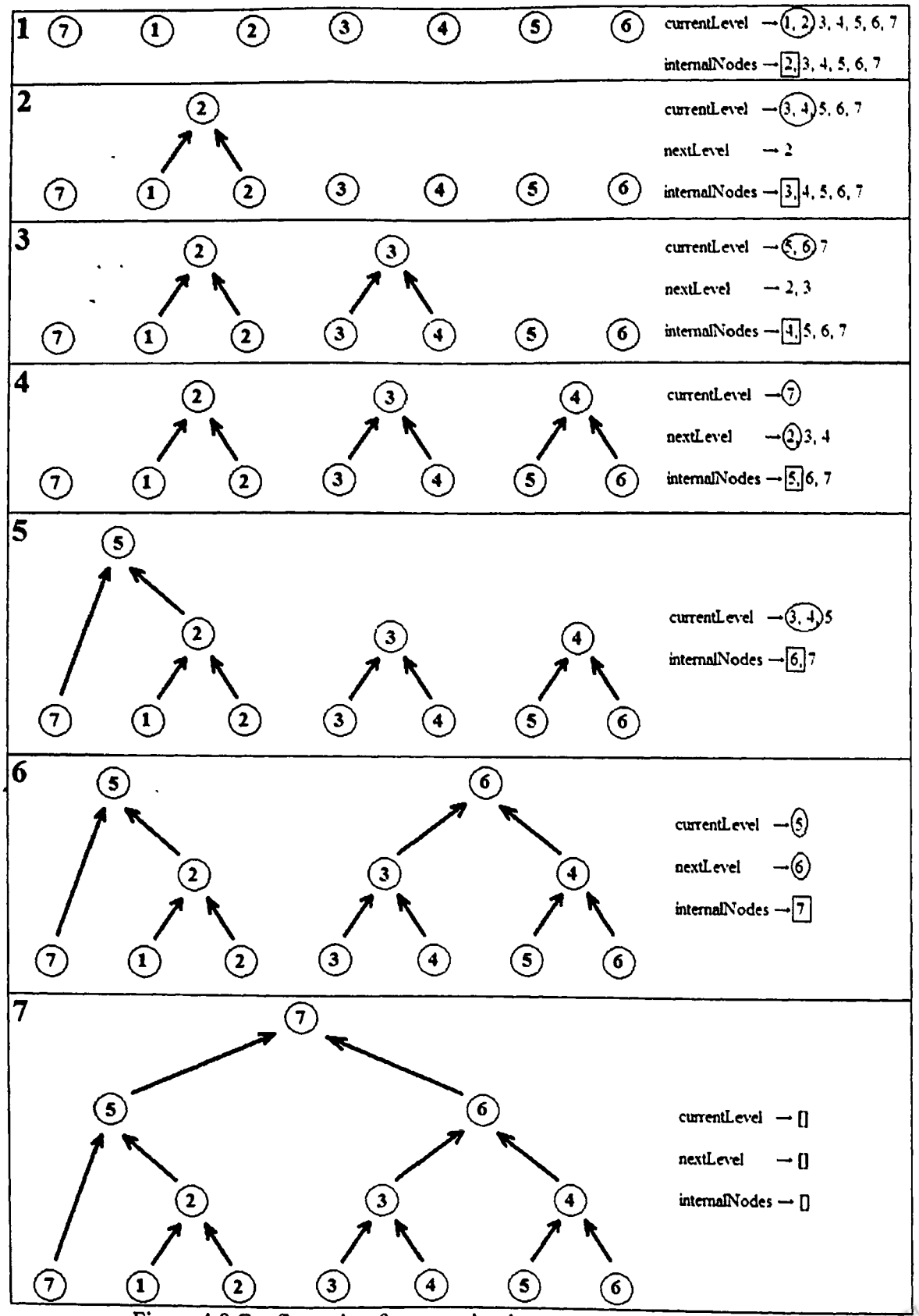


Figure 4.8 Configuration for seven hardware components, in 7 steps.

4.4 Phase 2. Standalone Subsystem: Pass Subcollections to Master Node

When our standalone AoSBM component finishes Phase 1, it makes a call to the master node's service, which is responsible for distributing the subcollections to service nodes. The input data that come along with this call, are not only the subcollections but also information regarding the configuration of nodes. In fact, Phase 2 constructs:

- A mapping that assigns a subcollection to each leaf node (URI). The assignment of subcollections to leaf nodes is random.
- A mapping that assigns to each leaf node (URI) the path to the root of the tree, i.e., a list of URIs of the nodes comprising the path from the leaf to the root, at the nodes tree.

These two mappings comprise the input data passed to master node. Thus, the master node is aware of the addresses of the leaf nodes, with which it must communicate, which subcollection to pass to each of them, and, finally, which calls path to pass to each of them. The technique of passing to a node the rest of the calls path is applied to the entire choreography, in particular, each node, when called, takes as an input the rest of the calls path to the root, and, when calling the next node in the path, it abstracts the first node from the calls path list and passes the list as input to the calling node.

4.5 Phase 3. Master Node: Distribute Subcollections to Leaf Nodes

At this phase, the master node calls each leaf node by passing input data consisting of:

- a subcollection of service descriptions
- a calls path (the path from this node to the root of the nodes tree).

4.6 Phase 4. Each Leaf Node: Mine Abstractions Hierarchy, Prune it & Call Parent Node

During this phase, each leaf node called by the master node, mines an abstractions hierarchy out of the subcollection of service descriptions using the algorithm proposed in [6]. Subsequently, a pruning technique is applied towards the produced hierarchy, with the purpose of retaining only a part of the most informative - representative abstractions.

Prior to further analyzing our pruning approach, we first have to examine some important aspects regarding the algorithm of [6].

4.6.1 Important aspects regarding the abstractions mining algorithm of [6]

First of all, the hierarchy produced over a set of services. Figure 4.9 illustrates a realistc example of a possible outcome of the application of this algorithm to a set of 13 services. The squared objects represent concrete services, while the circular ones represent functional abstractions. Solid arrows represent the parent - child relationships between objects, and one can use them to deduce the intermediate steps followed by the algorithm to come to this result. Dashed arrows also indicate steps of the abstractions hierarchy production process, but they refer to concrete services that are abstracted, which are not part of the abstractions hierarchy. As we can see, there is not a singlehierarchy produced over the 13 services, but there are actually three independent hierarchies. As we have mentioned in subsection 3.5.2, there are two circumstances in which the distance calculated between two interfaces is set to ∞. In such a case, an abstraction over these two interfaces cannot be constructed in any case, i.e., even if their distance (∞) is the lower among all pairs' distances. This can lead to a final result of having more than one independent hierarchies constructed, as depicted in Figure 4.9.

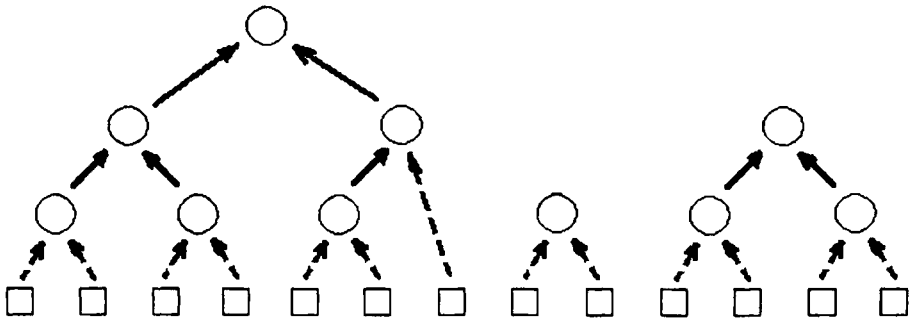


Figure 4.9 A possible abstractions hierarchy over thirteen services.

Another aspect we must clarify concerns the mappings part of an abstraction's structure. Figure 4.10 illustrates an example of an abstraction's mappings data and how they are related to the representative interface data, which are also part of an abstraction. The `InterfacesMapping` field of `FunctionalAbstraction` consists of information about the two mapped interfaces which are abstracted, and information about the mapped operations of the two interfaces, namely `OperationsMappings`. In particular, the `OperationsMappings` field contains `OperationsMapping` objects. Each such object is much like the `InterfacesMapping` object, i.e., consists of the two mapped operations, etc. This nested mappings structure finishes at message types mappings. For reasons of clarity,

Figure 4.10 presents a simplified view that stops at mapped operations' names. The dashed lines represent the mappings between operations. As we see, each operation in the `RepresentativeInterface` field is positioned relatively to the respective pair of mapped operations that abstracts (depicted by the numbering at the left and also by the consecutive lines).

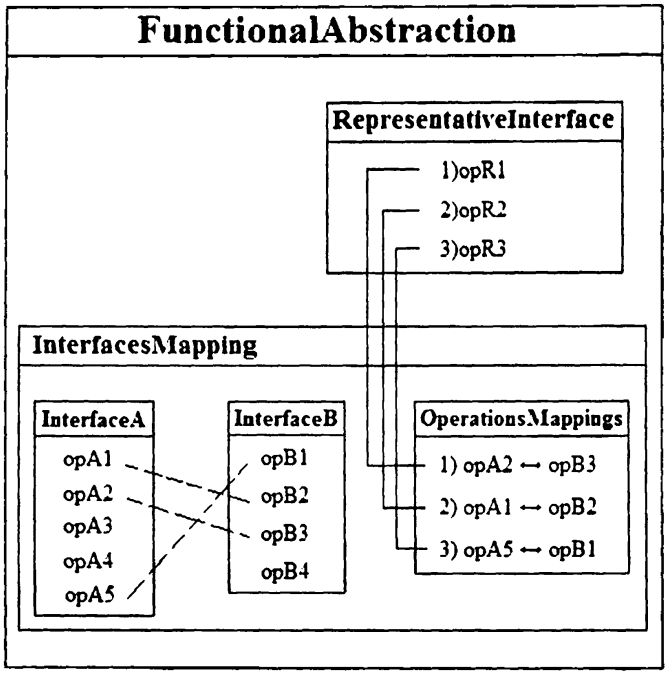


Figure 4.10 The mappings aspect of the abstraction structure.

4.6.2 The concept of pruning

Our pruning algorithm retains only a part of the constructed abstractions, on the purpose of reducing the number of abstractions that will be sent to the next software component. Thus, it has to retain the most informative abstractions whereas throwing out the less informative/useless ones.

The major concept is that, for each abstraction, either itself or its children should be retained, but not both of them. The criterion we use for this choice is the calculated distance between the two interfaces which an abstraction abstracts, compared to a user-given threshold. Let A be an abstraction over two interfaces, namely $A1$ and $A2$, $d(A) = d(A1, A2)$ the distance between the two interfaces, and *disThres* the distance threshold defined by the user. Then:

- If *d(A) < disThres*, only the parent is retained.
- If *d(A) >= disThres*, only the children are retained.

Moreover, the algorithm has to take care of the mappings between the interfaces of the abstractions, and how this will be sustained during the pruning process. For this reason we have inserted a new field to the `FunctionalAbstraction` object, namely `interfacesMappings`, which is a list of `InterfacesMapping` objects. To sustain compatibility with the adaptation facility of the AoSBM, we kept the standard existing `interfacesMapping` field, but, after the termination of our algorithm, the only actually valid field regarding interfaces mappings is `interfacesMappings`. In case children abstractions are pruned, the sustained parent abstraction has its `interfacesMappings` field filled with all the mappings between the concrete interfaces that are abstracted by it. The association between the representative interface of the parent and the mappings, regarding the places of the corresponding matched operations, is kept as described, for all mappings.

Yet, as indicated in Figure 4.11, there are different types of abstractions, regarding the children they do or do not have, with our pruning algorithm performing a different series of actions for each of them. Following, we examine the three types of abstractions along with the respective actions performed when the criterion mentioned before is applied to them:

a) Abstraction having no children (Figure 4.11, shape (a)). This is a leaf of the abstractions hierarchy. It abstracts concrete services. Since it has no children we cannot apply the aforementioned criterion, thus the abstraction is retained anyway. No change in interfaces mapping information (`interfacesMapping` field) is needed, except for it is added to `interfacesMappings` list.

b) Abstraction having a single child (Figure 4.11, shape (b)). It abstracts a concrete service and an abstract one. In this case the criterion is applied, considering the concrete service as a child too. We distinguish between two cases:

   1) If *d(A) < disThres*, the child is pruned. We construct and add to abstraction's `interfacesMappings` list an `InterfacesMapping` object that maps the

concrete service's interface to itself. We also add the single child's `interfacesMapping` field.

2) If *d(A) >= disThres*, we cannot prune the parent, because it holds information for the concrete service object, which won't be stored in any other abstraction retained (since the retained abstractions do not form a hierarchy). Thus, both abstractions are retained. Concerning the parent abstraction retained, all information about the single child is abstracted from it, relieving it of redundant data. The eliminated information contains the `interfacesMapping` object and the record in `representedInterfaces` object that respects to the single child. We construct and add to parent abstraction's `interfacesMappings` list an `InterfacesMapping` object that maps the concrete service's interface to itself.

c) Abstraction having two children (Figure 4.11, shape (c)). It abstracts two abstract services. In such case the criterion is applied with the standard actions taking place:

1) If *d(A) < disThres*, children are pruned. The `InterfacesMapping` objects that correspond to all the abstracted concrete services are added to parent's `interfacesMappings` list.

2) If *d(A) >= disThres*, the parent is pruned.

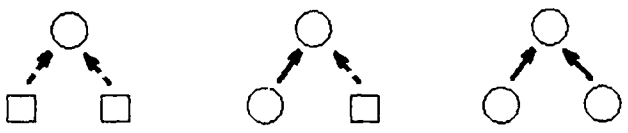


Figure 4.11 The three different types of abstractions regarding the objects they abstract.

4.6.3 Our pruning algorithm

As mentioned, the application of the abstractions mining algorithm proposed in [6] may lead to more than one hierarchies produced. We prune each one of them, with the goal of retaining only a part of the initially produced abstractions. We give the user the choice of defining how big this part will be, by providing as input the proportion of the abstractions to be retained, with

respect to the total number of initially produced abstractions. Thus, we apply to each abstraction in the hierarchy, a technique which decides whether this abstraction will be retained or not, according to the previously described criteria (cases (a), (b), (c)) and according to one more criterion, the user-input number of abstractions that should be retained.

Our dicision to apply both the two aforementioned criteria in the pruning process, and not just one of them, lies on the fact that the two criteria's roles are supplementary. The idea behind this is that we need to retain only a part of the initial mined hierarchy of abstractions, but that part, instead of being chosen arbitrarily, it could be chosen by the application of a quality criterion, like the distance threshold. In this way we retain few but informative abstractions.

Figures 4.12 and 4.13 present our pruning algorithm in pseudocode. For briefness, we use the abbreviations H, for the `Hierarchy` class and FA, for the `FunctionalAbstraction` class.

Function `prune()` is the main function of our approach, taking as input three arguments, a list of abstraction hierarchies and the aforementioned two thresholds, namely the retention threshold and the distance threshold. The `hierarchies` argument contains the complete result of the application of [6]'s abstractions mining technique, i.e., a list of abstraction hierarchies. The other two arguments, that represent a proportion, must be in the form of a real number between 0 and 1. The output of the function is a list of functional abstractions, i.e., the retained abstractions, which do not form a hierarchy, but just a list of independent elements. The `reprIfacesNum` variable stores the total number of represented interfaces of all hierarchies. The `foreach` loop prunes each hierarchy, and adds the respective retained abstractions to `result` variable, which is returned as a result when the function terminates. The loop calculates the number of abstractions that are going to be retained for a specific hierarchy (`retNum`) and calls `pruneHier()` function to prune the hierarchy.

Because `pruneHier()` prunes recursively, per level of the abstractions hierarchy, it is called each time taking as input a hierarchy's level of abstractions, so the first time it is called takes as input the first level, which has only one element; the root. Also, `retNum` is decreased by one, as we consider that the root abstraction has already been retained.

```
List<FA> prune(List<H> hierarchies, double retThres, double disThres) {
   List<FA> result = new List<FA>();
   int reprIfacesNum = getNumberOfRepresentedInterfaces(hierarchies);
   foreach h in hierarchies {
      int retNum = (int) ((h.reprIfacesNum / reprIfacesNum) * retThres);
      List<FA> firstLevel = new List<FA>();
      firstLevel.add(h.root);
      retNum --;
      List<FA> retainedFAs = pruneHierarchy(h,firstLevel,retNum,disThres);
      result.add(retainedFAs);
   }
   return result;
}
```

Figure 4.12 Our pruning algorithm in pseudocode - pruning the set of hierarchies.

```
List<FA> pruneHier(H h, List<FA> level, double retNum, double disThres) {
   if(level is empty) return h.toList();        // termination condition
   else {
      List<FA> nextLevel = new List<FA>();
      foreach fa in level {
         List<FA> children = fa.children();
         double distance = fa.distance();
         if(retNum == 0 || distance < disThres)
            children are pruned
         else {
            fa is pruned
            retNum --;
            nextLevel.add(children);
         }
      }
      pruneHier(h, nextLevel, retNum, disThres);      //recursive call
   }
}
```

Figure 4.13 Our pruning algorithm in pseudocode - pruning an hierarchy.

The `pruneHier()` function prunes a hierarchy and works, as said, recursively, taking as input, each time it is called, the hierarchy itself and a specific level of the hierarchy, i.e., a list of abstractions of the same level. The other inputs are the number of abstractions to be retained

(retNum) and the distance threshold (disThres). The function's output is a list of independent functional abstractions. The function checks each abstraction of the level's list and prunes it or its children, depending on the values of the two thresholds, retNum and disThres (Figure A.1 in Appendix contains a more detailed view of the algorithm; the function calls indicated by b1(), b2(), c1() and c2(), represent the respective actions described by the three cases (a), (b), (c) previously in this subsection). In cases children are retained, they are added to the nextLevel list. After all abstractions of the level are examined, pruneHier() is recursively called by passing nextLevel as argument. Recursion is terminated when the level list is empty. At this point, the hierarchy h consists of abstractions having no bonds between them, i.e. it is just a set of independent abstractions. This is converted to a list and returned.

Figure 4.14 depicts an example of our pruning algorithm applied to a functional hierarchy like the one in the figure, which abstracts 15 services. It consists of 8 basic steps, illustrating the algorithm's steps. The numbers below the abstractions' circles stand for the distance between their children's interfaces. We consider that retNum=6 and disThres=0.2. The grayed circles imply that the respective abstractions are retained for the time being, while the intersected gray lines represent abstractions that have been pruned. Unlike grayed circles, which do not represent something permanent (as at a next step the abstraction may be pruned), the intersected gray lines stand for permanent deletion of the respective abstractions. A call to the pruneHier() function and 3 recursive calls of it will take place till the process terminates.

Hereafter, we analyse each step of the application of our algorithm to the hierarchy of the Figure 4.14, in a brief form. Specifically, we mention the operation that is called, i.e,. pruneHier(), the input level, which is a list of abstractions that will be processed, eg [h, i, j], and the number of retained abstractions, which is set to 6, thus before the first call of pruneHier(), it will be reduced to 5, and each time an abstractions is retained, it will be reduced by one. The steps that we mention, correspond to the ones illustrated in Figure 4.14.

- pruneHier() call, input level: [n], retNum: 5
  step 2: check n → 0.38 > 0.2, thus n is pruned, retNum = 4

- `pruneHier()`, 1st recursive call, input level: [l, m], retNum: 4
  step 3: check l → 0.30 > 0.2, thus l is pruned, retNum = 3
  step 4: check m → 0.33 > 0.2, thus m is pruned, retNum = 2
- `pruneHier()`, 2nd recursive call, input level: [h, i, j, k], retNum: 2
  step 5: check h → 0.28 > 0.2, retNum = 1
  step 6: check i → 0.11 < 0.2, thus b, c are pruned
  step 7: check j → 0.22 > 0.2, thus j is pruned, retNum = 0
  step 8: check k → retNum == 0, thus f, g are pruned
- `pruneHier()`, 3rd recursive call, input level: [a], retNum: 0
  check a → has no children, thus nothing happens
- `pruneHier()`, 4th recursive call, input level: [], retNum: 0
  level is empty, thus return list of abstractions

4.7 Phase 5. Each Internal Node: Get Independent Abstractions from Children Nodes, Join them, Mine Abstractions Hierarchy Over them, Prune it & Call Parent Node

This phase is executed by the internal nodes of the software components tree. The first internal nodes entering this phase are those having children that are leaves. When these leaves finish their processing, they call their parent nodes, also passing them the results of their processing, i.e, a list of independent abstractions. Thus, the internal node's service deployed for our distributed organization, takes as input two lists of independent abstractions, each one from the respective child node. It also takes as input the two thresholds used in our approach, namely the retention threshold and the distance threshold.

During this phase, the two input lists are simply merged into their union, which is the input to the abstractions mining algorithm of [6]. From this point, the steps followed are mainly the same as those followed by leaf nodes at phase 4. The only difference is that the input to the algorithm of [6] is not a list of services but a list of abstractions.

4.8 Phase 6. Standalone Subsystem: Call Root Node to Get Final Result

The standalone component calls a specific method of the root node and gets the final result. The standalone component is not triggered by the root to do that, on the contrary, it periodically checks if the root has finished, and, when it finds that the root has finished, it calls the root and gets the result.

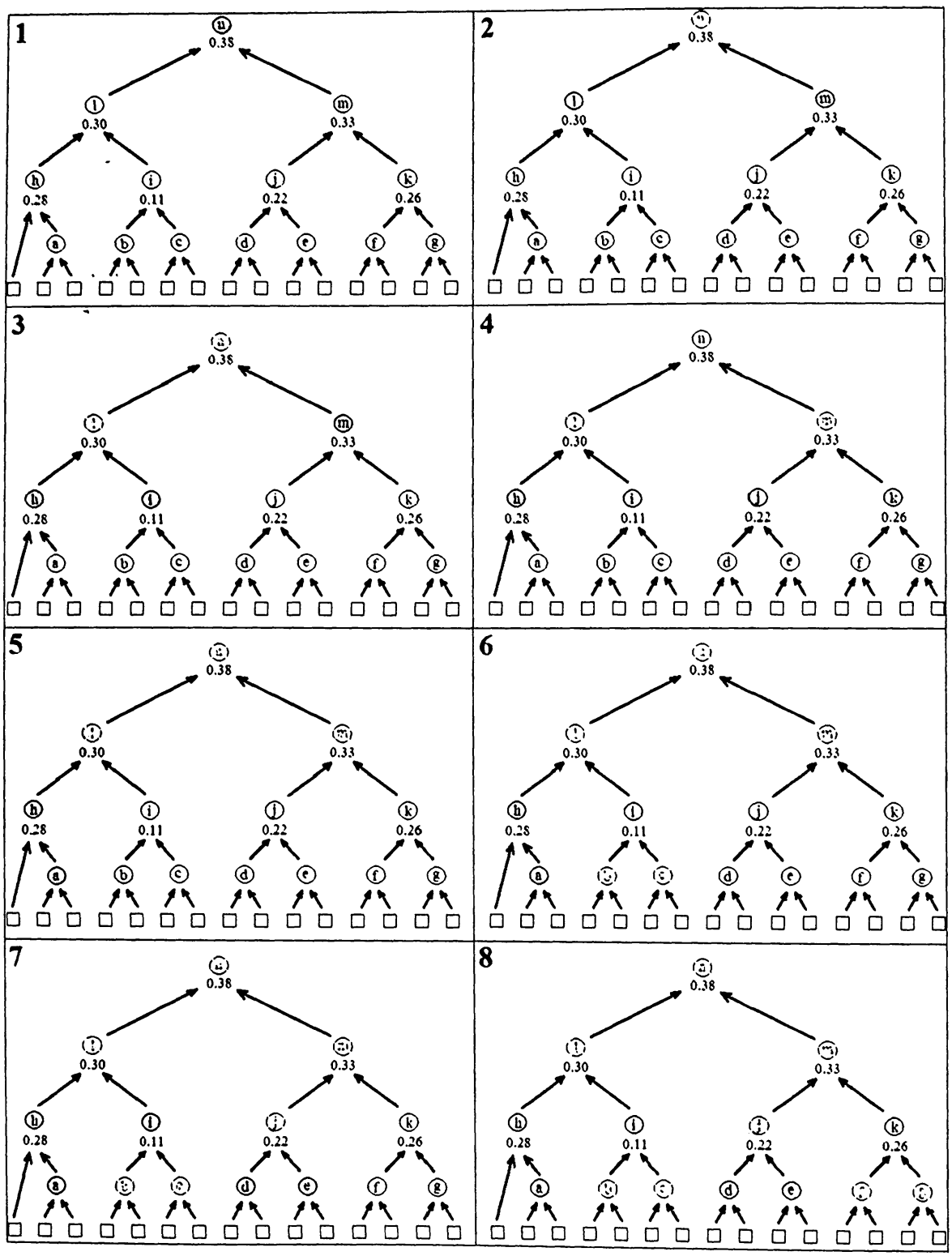


Figure 4.14 An example of our pruning algorithm applied to a hierarchy, with `retNum=6` and `disThres=0.2`.

4.9 Software Components and their Interaction

Hereafter, we summarize the software facilities we had to develop for our approach, and present the corresponding classes and the interaction between them. We also describe the structure of the data transmitted in this interaction.

Mainly, the developed facilities include (Figure 4.15):

1) A standalone facility which takes the user-inputs, such as the available hardware components, the number of available hardware components to be used, the services collection to be registrated in service base and abstracted, the proportion of functional abstractions to be retained and the distance threshold between the interfaces an abstraction abstracts, that will be used in the distributed abstractions mining process. The standalone facility firstly configures the components tree. Secondly, it registrates the service collection and stores it into the service base. Subsequently, it divides the collection into a list of subcollections (they are equal-sized and the division is done arbitrarily), and calls the master components, passing it the subcollections. The master component distributes the subcollections to the tree's leaf components. The tree components perform the distributed abstractions mining and, when they have finished, the resulted list of functional abstractions lies in the hardware component that respects to the root software component of components' tree. This software component offers an operation which just returns the final result. That operation is called by the standalone facility, the result is obtained and stored in the abstractions base. Class `DistributedFAMiningLauncher` represents the standalone component, which offers `launchDistributedFAMining()` operation, with the functionality that was described.

2) A master facility, responsible for the distribution of the subcollections to the leaf components. This facility is developed as a service, named `MasterService`, which offers a respective operation, `manageDistributedFAMining()`. The operation is called by the `launchDistributedFAMining()` operation of

`DistributedFAMiningLauncher` component and takes as input a `MasterData` object, which includes:

- A list of the subcollections of the initial service collection, as divided by the standalone component.
- An integer, representing the number of abstractions that should be retained.
- A real number, representing the distance threshold between the children of an abstraction.

The operation calls the `LeafService` service of every leaf node, passing it a respective subcollection of service interfaces.

3) A leaf facility which executes the abstractions mining algorithm of [6] and then applies our pruning algorithm, thus retaining a list of independent abstractions. This facility is developed as a service, namely `LeafService`, offering a respective operation, `mineAndPruneFunctionalHierarchy()`, which performs the aforementioned steps. The operation takes as input a `LeafData` object, which includes:

   - A list of service interfaces (subcollection).
   - An integer, representing the number of abstractions that should be retained.
   - A real number, representing the distance threshold between the children of an abstraction.
   - A list of component URLs, standing for the rest of the calls path till the root of the tree.

   The operation is called by the `MasterService`'s operation. When it finishes, it calls the next service in the calls list, passing it the list of independent abstractions. This next service is an `InternalService`, and the operation of it that is called is the `mergeFAsAndMineAndPruneFunctionalHierarchy()` operation.

4) An internal facility which merges the two lists of independent functional abstractions that takes as input from the respective leaf facilities. Following, it performs what the leaf facility performs, i.e., mining an abstractions hierarchy over the list of abstractions

and then prune it. The only difference is that the leaf facility starts mining abstractions over concrete interfaces, whereas the internal facility starts mining abstractions over abstractions. To apply this facility, we developed a service named `InternalService`, which offers two operations:

- The `mergeFAsAndMineAndPruneFunctionalHierarchy()` operation performs the aforementioned steps. It takes as input two `InternalData` objects, each one from the respective child component. An `InternalData` object comprises:

  - A list of independent functional abstractions.
  - An integer, representing the number of abstractions that should be retained.
  - A real number, representing the distance threshold between the children that an abstraction abstracts.
  - A list of component URLs, standing for the rest of the calls path till the root of the tree.

  When finished, the operation calls the next service in the calls list, specifically the `mergeFAsAndMineAndPruneFunctionalHierarchy()` method of the next service. This process is repeated till the component called is the root of the tree. When the operation of the root component finishes, it stores the result in a structure that `getFinalResult()` operation can access.

- The `getFinalResult()` operation is called to return the final result

4.10 Data transmission and optimization

The technology we chose to use for the services' implementation, deployment and call leaded to a serious issue concerning the efficiency of the overall process. Specifically, the data transmitted during a service call, i.e., the arguments to the respective operation called, are wrapped into an XML structure. This results to large XML structures being constructed and

transmitted, because in an XML structure, the net information (excluding tags) is many times less than the total one.

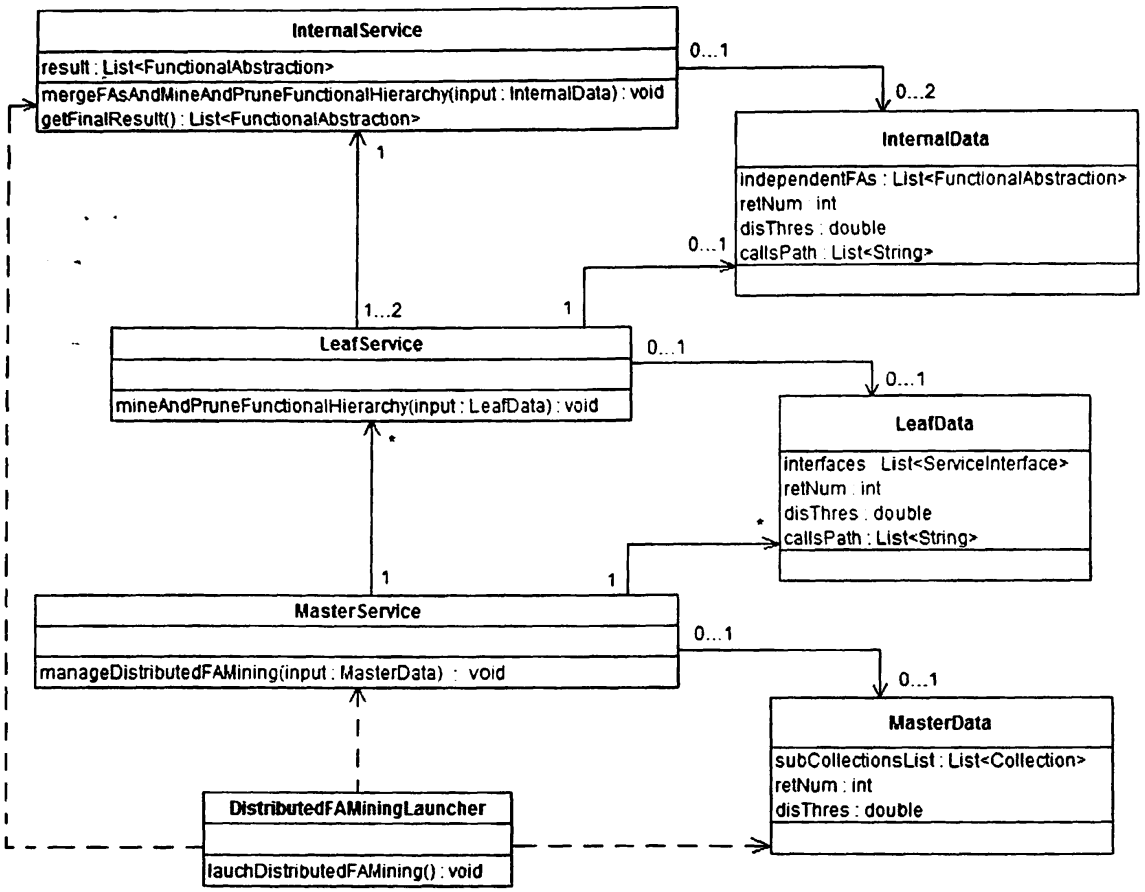


Figure 4.15 Distributed abstractions mining: software components and their interaction.

The impact of this is a significant memory waste in both sides of caller and called services. Also, there is a time overhead concerning the coding and decoding information into XML structures, as well as a time waste in transmitting all this information.

An action we take for this is to adapt the specifical part of our code which transforms our structures into XML form, so that small XML tags be formed.

Another action is to eliminate redundant information from data structures before they are wrapped and transmitted. Specifically:

- The standalone component, just before calling the master component and pass it the list of subcollections of service interfaces, eliminates every service interface structure from service instances objects; since the registration and storage of the initial collection of services has already been done at that point, the information concerning the service instances can be retrieved from the service base anyway.

- A leaf component, after mining and pruning a functional hierarchy over the concrete service interfaces passed to it by the master node, has to call an internal component (its parent) and pass it the list of independent abstractions it has retained. Just before passing the list, it scans every functional abstraction object and performs the following eliminations:

    - From every represented interface object, only data base key information is kept; represented interfaces are concrete interfaces, which are stored in the service base, thus only their data base keys are needed to retrieve them.

    - From every mappings object, i.e., interfaces/operations/messages/messagetypes mappings, the respective mapped elements objects, i.e, mapped interfaces/operations/messages/messagetypes, are eliminated from everything else but their data base key, for the same reason as mentioned before.

4.11 Random Choice Technique For Name Extraction

We added an extra option on the abstractions mining algorithm of [6]. The extra option regards the technique used by the algorithm to extract the name of a representative object, i.e., a representative interface, operation, message or message type, out of the names of the represented objects that it abstracts. Specifically, the algorithm exploits the Longest Common Substring (LCS) technique for this, i.e., the representative name is constructed from the longest common substring of the names of the two represented objects. Our added technique sets the representative name by randomly choosing between the two represented names (RC). In our experiments we applied both techniques.

CHAPTER 5 QUERY ENGINE

CHAPTER 5. QUERY ENGINE

We developed a query engine, to serve as a more developer-friendly tool for querying the service base, than SBQL was. Additionally, we developed a Web service that provides access to the query engine, allowing the service base to be used in a distributed setting.

5.1 Query Engine

As presented, AoSBM provides a query language, named SBQL, and a corresponding facility, which executes SBQL queries using the mined abstractions stored in the abstractions base. Based upon this facility, we developed a more friendly one, which simplifies querying. Also, we made all querying facilities available in a distributed setting by designing and realizing a REST API, named `QueryEngineService`, which exposes them as a service. Following, we provide further details concerning the design and the functionalities offered by the `QueryEngineService`.

Overall, the `QueryEngineService` API provides several operations that can be used for service lookup (Figure 5.1). In general, these operations accept as input constraints that should be satisfied by the discovered services and produce as output information concerning the discovered services. In general, we divide the operations that are offered by the `QueryEngineService` API in two different categories:

- The operations of the first category allow to use the AoSBM as a typical service registry that does not employ abstractions in the service lookup process. Specifically, the first category consists of operations for which the input constraints are matched against the service model information that is stored in the AoSBM.

- The operations of the second category enable abstraction-driven service discovery. In particular, the second category comprises operations for which the input constraints are matched against the abstractions model information that is stored in the AoSBM. To support typical and more experienced developers, each category provides operations for simple and more advanced lookup queries.

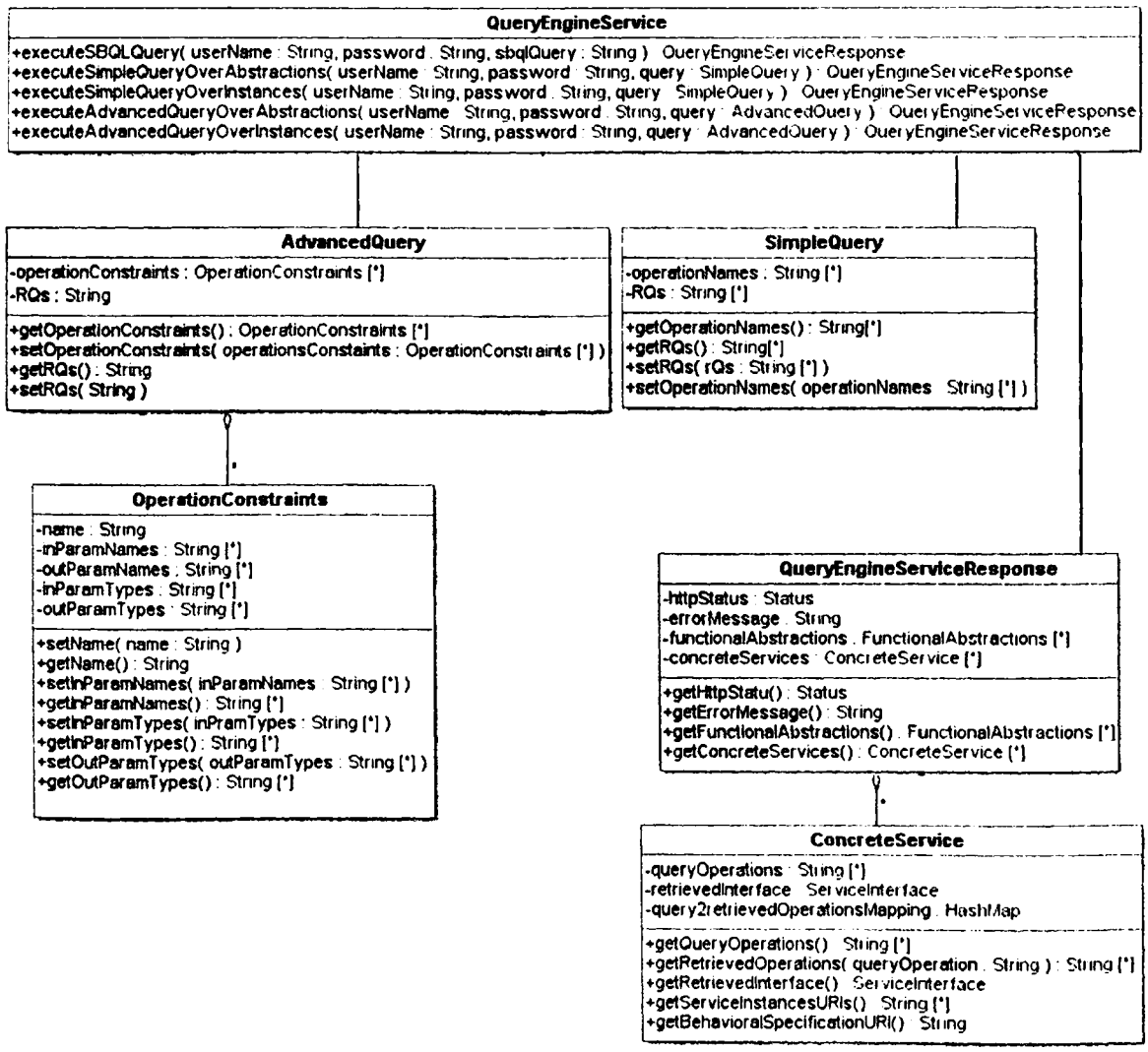
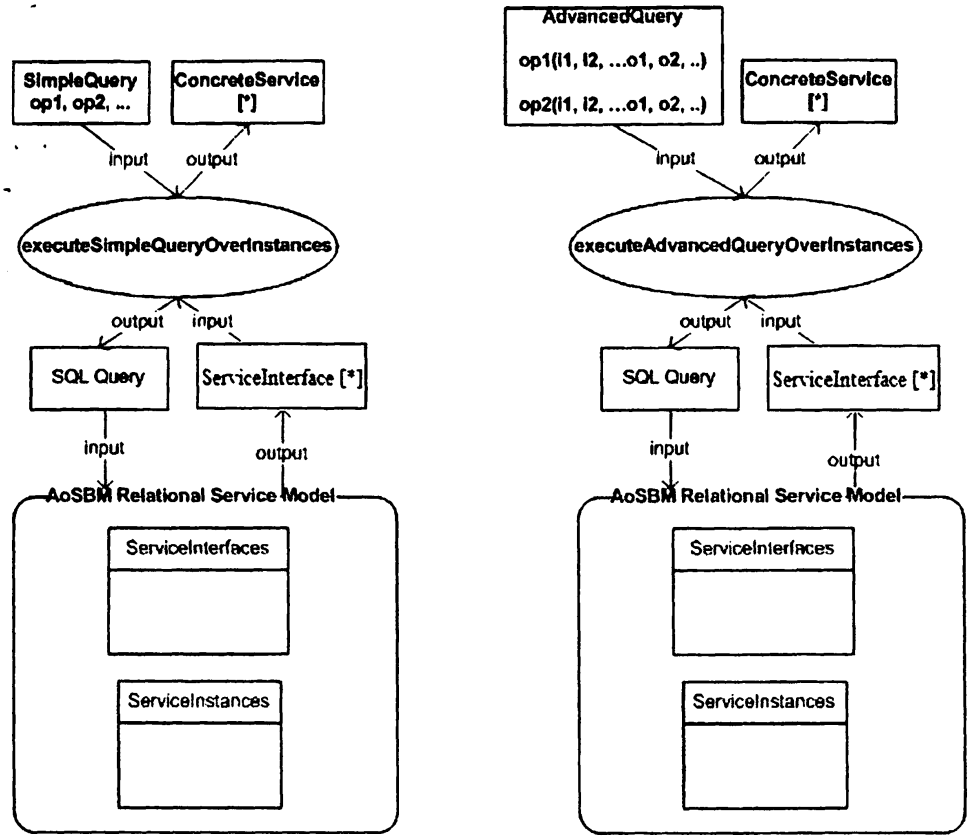


Figure 5.1 Design of the `QueryEngineService`.

5.2 Service Lookup over the Service Model

To enable service lookup over the service model information that is stored in the AoSBM we provide the following alternative options:



(A) (B)

Figure 5.2 Lookup operations over the service model.

1) The `executeSimpleQueryOverInstances`, takes as input a `SimpleQuery` object and produces as output a `QueryEngineServiceResponse` object. The code snippet in Figure 5.3 gives an example of how to call the `executeSimpleQueryOverInstances` and navigate through the results. The `SimpleQuery` object contains the following information:

   - A list of operation names that should match with corresponding names of the operations of the services that will be returned as a result. Specifically, for each

required operation name, a discovered service must provide at least one operation, whose name comprises the required operation name.

- A specification of requirements that concern the services that are used by the discovered services. These requirements may comprise, for instance, the names of the operations that are called by the discovered services.

The `QueryEngineServiceResponse` object that is produced as output includes a list of `ConcreteService` objects. Each `ConcreteService` object contains information about a discovered service. Specifically, a `ConcreteService` object includes:

- The list of the required operation names of the input `SimpleQuery` object.
- The full specification of the `ServiceInterface` that is offered by the discovered service.
- A mapping between the required operations and the operations of the interface that is offered by the discovered services.

To facilitate the work of the developer the `ConcreteService` object provides operations that provide easy access to the URIs of the discovered services and to the behavioral specification of the discovered services. These operations reveal the developer from the need to navigate in the `ServiceInterface` object structure.

The execution of the `executeSimpleQueryOverInstances` operation takes place in three main steps (Figure 5.2 (A)):

- Based on the given `SimpleQuery` object, an SQL query is generated over the relations of the AoSBM service model.
- The generated query is executed and a list of `ServiceInterface` objects is reconstructed, based on the information that is retrieved from the AoSBM relational store.

- Finally, the `QueryEngineServiceResponse` that encapsulates the reconstructed `ServiceInterface` objects is constructed and returned to the developer.

```
List<String> operations = new ArrayList<String>();
operations.add("request");
operations.add("get");
String RQs = null;
SimpleQuery simpleQuery = new SimpleQuery(operations, RQs);
Response response = client.executeSimpleQueryOverInstances(simpleQuery);

QueryEngineServiceResponse qeResponse = null;
try {
   InputStream in = (InputStream) response.getEntity();
   JAXBContext context = JAXBContext.newInstance(QueryEngineServiceResponse.class);
   Unmarshaller unmarshaller = context.createUnmarshaller();
   qeResponse = (QueryEngineServiceResponse) unmarshaller.unmarshal(in);
} catch (WebApplicationException e) { e.printStackTrace(); }
  catch (JAXBException e) { e.printStackTrace();  }

List<ConcreteService> queryResults = qeResponse.getConcreteServices();
for( int i = 0; i < queryResults.size(); i++) {
   ConcreteService concService = queryResults.get(i);
   String LTS_URI = concService.getBehavioralSpecificationURI();
   String ENC_URI = concService.getEnactementURI();

   List<String> queryOperations = simpleQuery.getOperationNames();
   for( int j = 0; j < queryOperations.size(); j++) {
      String queryOp = queryOperations.get(j);
      List<Operation> retrievedOpers = concService.getRetrievedOperations(queryOp);

      if (retrievedOpers != null) {
         for( int k = 0; k < retrievedOpers.size(); k++)
            String retrievedOperName = retrievedOpers.get(k).getName();
      }
   }

   List<String> retrievedURIs = concService.getServiceInstancesURIs();
   for( int j = 0; j < retrievedURIs.size(); j++)
      String URI = retrievedURIs.get(j);
}
```

Figure 5.3 Executing a simple query over the service model.

2) The `executeAdvancedQueryOverInstances`, takes as input an `AdvancedQuery` object and produces as output a `QueryEngineServiceResponse` object. The code snippet in Figure 5.4 gives an example of how to call the `executeAdvancedQueryOverInstances`. The `AdvancedQuery` object contains the following information:

   - A list of `OperationConstraints` objects, which contain functional constraints that should be satisfied by the discovered services. Specifically, an `OperationConstraints` object contains the following information:

     - An operation name that should match with corresponding names of the operations of the services that will be returned as a result. Specifically, a discovered service must provide at least one operation, whose name comprises the required operation name.

     - A list of input (resp. output) parameter names that should match with corresponding input (resp. output) parameter names that should match with corresponding input (resp. output) parameter names of the operations of the discovered services. For each required input (resp. output) parameter name, a discovered service must provide at least one operation with an input (resp. output) parameter name that includes the required input (resp. output) parameter name.

     - A list of input (resp. output) parameter types that should match with corresponding input (resp. output parameter types of the operations of the discovered services. For each required input (resp. output) parameter type, a discovered service must provide at least one operation with an input (resp. output) parameter type that matches with the required input (resp. output) patameter type.

       We assume that the lists have equal number of elements and that elements stored in the same list position correspond to the same required parameter.

A list element may be null in case there are no requirements on the name, or the type of the parameter.

- A specification of requirements that concern the services that are used by the discovered services. These requirements may comprise, for instance, the names of the operations that are called by the discovered services.

As in the case of the `executeSimpleQueryOverInstances`, the `QueryEngineServiceResponse` object that is produced as output from the `executeAdvancedQueryOverInstances` includes a list of `ConcreteService` objects. The execution of the operation takes place as follows (Figure 5.2(B)):

- Based on the given `AdvancedQuery` object, an SQL query is generated over the relations of the AoSBM service model.
- The generated query is executed and a list of `ServiceInterface` objects is reconstructed, based on the information that is retrieved from the AoSBM relational store.
- Finally, the `QueryEngineServiceResponse` that encapsulates the reconstructed `ServiceInterface` objetcs is constructed and returned to the developer.

5.3 Service Lookup over the Abstractions Model

To enable service lookup over the abstractions model information that is stored in the AoSBM we provide the following alternatives:

1) The `executeSBQLQuery`, takes as input a SBQL query and produces as output a `QueryEngineServiceResponse` object. The code snippet in Table 5.3 gives an example of how to call the `executeSBQLQuery` and navigate through the results. The `QueryEngineServiceResponse` object that results from the operation contains a list of `FunctionalAbstraction` objects that contain information regarding the discovered functional abstractions that satisfy the given SBQL query. The developer may navigate through the information that is included in each

`FunctionalAbstraction` object to get the specification of the abstract interface that characterizes the functional abstraction, the specification of the represented service interfaces, the mappings between the abstract interface and the represented service interfaces, etc.

```
String opN1 = "request";
List<String> iN1 = new ArrayList<String>();
iN1.add("parameters");
List<String> iT1 = new ArrayList<String>();
iT1.add("string");
List<String> oN1 = new ArrayList<String>();
oN1.add("parameters");
List<String> oT1 = new ArrayList<String>();
oT1.add("string");
String opN2 = "get";
List<String> iN2 = new ArrayList<String>();
iN2.add("parameters");
List<String> iT2 = new ArrayList<String>();
iT2.add("string");
List<String> oN2 = new ArrayList<String>();
oN2.add("parameters");
List<String> oT2 = new ArrayList<String>();
oT2.add("string");

OperationConstraints opC1 = new OperationConstraints(opN1, iN1, iT1, oN1, oT1);
OperationConstraints opC2 = new OperationConstraints(opN2, iN2, iT2, oN2, oT2);
List<OperationConstraints> operConstraints = new ArrayList<OperationConstraints>();
operConstraints.add(opC1);
operConstraints.add(opC2);

AdvancedQuery advancedQuery = new AdvancedQuery(operConstraints, null, null);
Response response = client.executeAdvancedQueryOverInstances(advancedQuery);
QueryEngineServiceResponse qeResponse = null;
try {
   InputStream in = (InputStream) response.getEntity();
   JAXBContext context = JAXBContext.newInstance(QueryEngineServiceResponse.class);
   Unmarshaller unmarshaller = context.createUnmarshaller();
   qeResponse = (QueryEngineServiceResponse) unmarshaller.unmarshal(in);
} catch (WebApplicationException e) { e.printStackTrace(); }
  catch (JAXBException e) { e.printStackTrace(); }
List<ConcreteService> queryResults = qeResponse.getConcreteServices();
```

Figure 5.4 Executing an advanced query over the service model.

2) The `executeSBQLQuery`, takes as input a SBQL query and produces as output a `QueryEngineServiceResponse` object. The code snippet in Figure 5.6 gives an example of how to call the `executeSBQLQuery` and navigate through the results. The `QueryEngineServiceResponse` object that results from the operation contains a list of `FunctionalAbstraction` objects that contain information regarding the discovered functional abstractions that satisfy the given SBQL query. The developer may navigate through the information that is included in each `FunctionalAbstraction` object to get the specification of the abstract interface that characterizes the functional abstraction, the specification of the represented service interfaces, the mappings between the abstract interface and the represented service interfaces, etc.

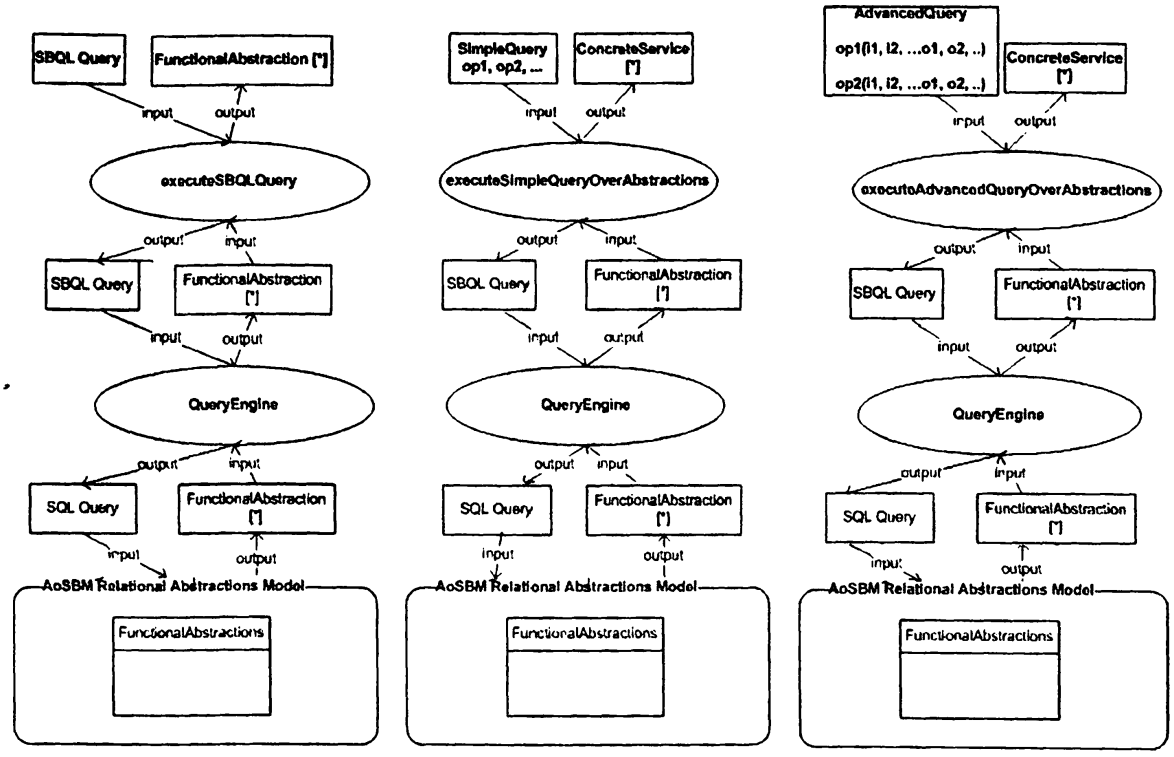


Figure 5.5 Lookup operations over the abstractions model.

The execution of the `executeSBQLQuery` operation takes place in three main steps (Figure 5.5 (A)):

- Based on the given SBQL query, an SQL query is generated over the relations of the AoSBM abstraction model.
- The generated query is executed and a list of `FunctionalAbstraction` objects is reconstructed, based on the information that is retrieved from the AoSBM relational store.
- Finally, the `QueryEngineServiceResponse` that encapsulates the reconstructed `FunctionalAbstraction` objects is constructed and returned to the developer.

```
String sbqlQuery =
" let        $db = db('localhost/mySB')" + "\n\n" +
" for        $c in $db/servicecollections" + "\n\n" +
" for        $fa in $c/hierarchies/abstractions" + "\n\n" +
" for        $ri in $fa/representativeinterfaces" + "\n\n" +
" for        $nfa in $c/hierarchies/abstractions" + "\n\n" +
" for        $prl in $nfa/qproperty" + "\n\n" +
" where      $ri/rsi name like '%forecast%' and" + "\n\n" +
"            $prl/qp name = 'Availability' and" + "\n\n" +
"            $prl/qp value = 'High'" + "\n\n"
" return     Abstractions.fullObject";

Response response = client.executeSBQLQuery(sbqlQuery);
QueryEngineServiceResponse qeResponse = null;
try {
   InputStream in = (InputStream) response.getEntity();
   JAXBContext context = JAXBContext.newInstance(QueryEngineServiceResponse.class);
   Unmarshaller unmarshaller = context.createUnmarshaller();
   qeResponse = (QueryEngineServiceResponse) unmarshaller.unmarshal(in);
} catch (WebApplicationException e) { e.printStackTrace(); }
  catch (JAXBException e) { e.printStackTrace(); }

List<FunctionalAbstraction> abstractions = qeResponse.getFunctionalAbstractions();
if (abstractions != null && abstractions.size() != 0) {
   for( int i = 0; i < abstractions.size(); i++)
      String representativeName = abstractions.get(i).getInterface().getName();
}
```

Figure 5.6 Executing a SBQL query over the abstractions model.

3) The `executeSimpleQueryOverAbstractions`, takes as input a `SimpleQuery` object and produces as output a

`QueryEngineServiceResponse` object. The code snippet in Figure 5.7 gives an example of how to call the `executeSimpleQueryOverAbstractions` and explore the results. The `SimpleQuery` object contains the following information:

- A list of operations names that should match with corresponding names of abstract operations (i.e., the operations of the abstract interfaces that characterize the discovered functional abstractions), offered by the discovered functional abstractions. For each required operation name, a discovered functional abstraction must provide at least one operation, whose name comprises the required operation name.

- A specification of requirements that concern the services that are used by the services that are represented by the functional abstractions. These requirements may comprise, for instance, the names of the operations that are called by the represented services.

The `QueryEngineServiceResponse` object that is produced as output includes a list of `ConcreteService` objects. Each `ConcreteService` object contains information about the services that are represented by the discovered functional abstractions. Specifically, a `ConcreteService` object includes:

- The list of the required operation names of the input `SimpleQuery` object.
- The full specification of the `ServiceInterface` that is offered by the represented service.
- A mapping between the required operations and the operations of the interface that is offered by the represented service.

The execution of the `executeSimpleQueryOverAbstractions` operation takes place as follows (Figure 5.5(B)):

- Based on the given `SimpleQuery` object, a SBQL query is generated over the relations of the AoSBM abstractions model.

- The generated query is executed and a list of `FunctionalAbstraction` objects is reconstructed, based on the information that is retrieved from the AoSBM relational store.
- The `ConcreteService` objects that contain information about the services that are represented by the reconstructed `FunctionalAbstraction` objects are created and encapsulated in the `QueryEngineServiceResponse` object.

4) The `executeAdvancedQueryOverAbstractions`, takes as input an `AdvancedQuery` object and produces as output a `QueryEngineServiceResponse` object. The code snippet in Figure 5.8 gives an example of how to call the `executeAdvancedQueryOverAbstractions`. The `AdvancedQuery` object contains the following information:

- A list of `OperationConstraints` objects, which contain functional constraints that should be satisfied by the abstract interfaces of the discovered functional abstractions. Specifically, an `OperationConstraints` object contains the following information:

  - An operation name that should match with corresponding names of the abstract operations, offered by the discovered functional abstractions. Specifically, a functional abstraction must provide at least one abstract operation, whose name comprises the required operation name.

  - A list of input (resp. output) parameter names that should match with corresponding input (resp. output) parameter names of the abstract operations of the discovered functional abstractions. For each required input (resp. output) parameter name, a discovered functional abstraction must provide at least one operation with an input (resp. output) parameter name that includes the required input (resp. output) parameter name.

- A list of input (resp. output) parameter types that should match with corresponding input (resp. output) parameter types of the abstract operations of the discovered functional abstractions. For each required input (resp. output) parameter type, a discovered functional abstraction must provide at least one operation with an input (resp. output) parameter type that matches with the required input (resp. output) parameter type.

- A specification of requirements that concern the services that are used by the services that are represented by the discovered functional abstractions. These requirements may comprise, for instance, the names of the operations that are called by the represented services.

The `QueryEngineServiceResponse` object that is produced as output includes a list of `ConcreteService` objects that contain information about the services that are represented by the discovered functional abstractions.

The execution of the `executeAdvancedQueryOverAbstractions` operation takes place as follows (Figure 5.5(C)):

- Based on the given `AdvancedQuery` object, a SBQL query is generated over the relations of the AoSBM abstractions model.
- The generated query is executed and a list of `FunctionalAbstraction` objects is reconstructed, based on the information that is retrieved from the AoSBM relational store.
- The `ConcreteService` objects that contain information about the services that are represented by the reconstructed objects are created and encapsulated in the `QueryEngineServiceResponse` object.

```
List<String> operations = new ArrayList<String>();
operations.add("request");
operations.add("get");
String RQs = null;
SimpleQuery simpleQuery = new SimpleQuery(operations, RQs);

Response response = client.executeSimpleQueryOverAbstractions(simpleQuery);
QueryEngineServiceResponse qeResponse = null;
try {
   InputStream in = (InputStream) response.getEntity();
   JAXBContext context = JAXBContext.newInstance(QueryEngineServiceResponse.class);
   Unmarshaller unmarshaller = context.createUnmarshaller();
   qeResponse = (QueryEngineServiceResponse) unmarshaller.unmarshal(in);
} catch (WebApplicationException e) { e.printStackTrace(); }
  catch (JAXBException e) { e.printStackTrace(); }

List<ConcreteService> queryResults = qeResponse.getConcreteServices();
for( int i = 0; i < queryResults.size(); i++) {
   ConcreteService concService = queryResults.get(i);
   String LTS_URI = concService.getBehavioralSpecificationURI();
   String ENC_URI = concService.getEnactementURI();

   List<String> queryOperations = null;
   if (simpleQuery != null)
      queryOperations = simpleQuery.getOperationNames();

   for( int j = 0; j < queryOperations.size(); j++) {
      String queryOp = queryOperations.get(j);
      List<Operation> retrievedOpers = concService.getRetrievedOperations(queryOp);

      if (retrievedOpers != null) {
         for( int k = 0; k < retrievedOpers.size(); k++)
            String retrievedOperName = retrievedOpers.get(k).getName();
      }
   }

   List<String> retrievedURIs = concService.getServiceInstancesURIs();
   for( int j = 0; j < retrievedURIs.size(); j++)
      String retrievedURI = retrievedURIs.get(j);
}
```

Figure 5.7 Executing a simple query over the abstractions model.

```
String opN1 = "request";
List<String> iN1 = new ArrayList<String>();
iN1.add("parameters");
List<String> iT1 = new ArrayList<String>();
iT1.add("string");
List<String> oN1 = new ArrayList<String>();
oN1.add("parameters");
List<String> oT1 = new ArrayList<String>();
oT1.add("string");
        . .
String opN2 = "get";
List<String> iN2 = new ArrayList<String>();
iN2.add("parameters");
List<String> iT2 = new ArrayList<String>();
iT2.add("string");
List<String> oN2 = new ArrayList<String>();
oN2.add("parameters");
List<String> oT2 = new ArrayList<String>();
oT2.add("string");

OperationConstraints opC1 = new OperationConstraints(opN1, iN1, iT1, oN1, oT1);
OperationConstraints opC2 = new OperationConstraints(opN2, iN2, iT2, oN2, oT2);

List<OperationConstraints> opConstraints = new ArrayList<OperationConstraints>();
opConstraints.add(opC1);
opConstraints.add(opC2);

AdvancedQuery advancedQuery = new AdvancedQuery(opConstraints, null, null);

Response response = client.executeAdvancedQueryOverAbstractions(advancedQuery);
QueryEngineServiceResponse qeResponse = null;
try {
   InputStream in = (InputStream) response.getEntity();
   JAXBContext context = JAXBContext.newInstance(QueryEngineServiceResponse.class);
   Unmarshaller unmarshaller = context.createUnmarshaller();
   qeResponse = (QueryEngineServiceResponse) unmarshaller.unmarshal(in);
} catch (WebApplicationException e) { e.printStackTrace(); }
  catch (JAXBException e) { e.printStackTrace(); }

List<ConcreteService> queryResults = qeResponse.getConcreteServices();
```

Figure 5.8 Executing an advanced query over the abstractions model.

CHAPTER 6. EVALUATION

6.1 Overview

This chapter presents the results of the experiments we have performed in order to evaluate our [illegible] described in [illegible].

We have performed experiments with our distributed [illegible] on real [illegible] large scale hardware deployments and [illegible] to the algorithms, we [illegible] experiments so as to evaluate both the performance and the quality of [illegible].

We have also performed a set of experiments to [illegible] the [illegible] and impact on the [illegible] in relation to the scaling of the number of [illegible] and [illegible]. These experiments are [illegible] the algorithm [illegible] on the [illegible] synthetic data [illegible] i.e., [illegible]. In this way we managed to scale [illegible] so as to be able to evaluate the [illegible] execution [illegible].

CHAPTER 6. EVALUATION

6.1 Overview

This chapter presents the results of the experiments we have performed in order to evaluate our distributed service base system.

We have performed experiments with our distributed service base system, based on real data. Using a set of hardware components and various inputs to our algorithm, we executed a set of experiments so as to evaluate both the performance and the quality of our method.

We have also performed a set of experiments to measure the mined abstractions' impact on the queries' execution time, in relation to the scaling of the number of services and abstractions. These experiments are not based on any abstractions mining technique, neither on the algorithm of [6] nor on our distributed one, but, on the contrary, we loaded synthetic data in the service base, i.e., synthetically created services and abstractions. In this way we managed to scale up to rather large number of services/abstractions, so as to be able to evaluate the actual scalability of abstractions-based querying in general, related to the queries execution time.

6.2 Performance and Quality Assessment

To evaluate our approach we used the *OWLS-TC* benchmark[3], which is a collection of services annotated with OWL-S semantic descriptions. The benchmark also contains a set of queries. The *OWLS-TC* benchmark was the source of our experimental data set, i.e., we used our system to registrate the benchmark's service descriptions and mine abstractions with our distributed approach. The *OWLS-TC* benchmark was also the source of our queries set, which was the means to evaluate the query execution performance and the quality of the answers returned.

- We chose the latest version, 4.0, of *OWLS-TC*. We did not exploit the OWL-S semantic descriptions, on the contrary, we isolated the two folders, containing the WSDL service descriptions and the queries.
- For the input data set, we isolated the *htdocs->wsdl* folder, which contains 1076 WSDL files, each one specifying a service description.
- For the queries data set, we isolated the *htdocs->queries* folder, which contains 42 queries over the described services. We used the 1.1 version of the set of queries. The queries are in the form of OWLS files, i.e., files describing the desirable features of the services.

6.2.1 Description of the Input Data Set

The benchmark's service descriptions cover a wide thematical domain. In particular, there are descriptions for services referring to communication, economy, education, food, geography, medicine, simulation, travel and weapon.

We had to modify the service descriptions slightly, so that they could be parsed by the AoSBM's WSDL parser. Figure 6.1 depicts a WSDL snippet comprising some type declarations and a message. We present this snippet to illustrate the issues that made the initial WSDL files unsuitable for being parsed by the AoSBM's parser. Mainly, there were two issues:

- Self-nesting for complex types is not supported. As an example, the complex type `PersonType` in Figure 6.1 contains an element of the same type, i.e., `PersonType`, which cannot be parsed.

[3] http://www.semwebcentral.org/projects/owls-tc

- To refer to a declared type from outside the <wsdl:types> declaration element, this type must be declared, apart from its <xsd:complexType> or <xsd:simpleType> declaration element, also in a <xsd:element> element. For example, in the message declaration (<wsdl:message> element), the first part refers to a Person type, for which there is a corresponding <xsd:element> element. On the other side, the second part refers to a CompanyType type, for which there is no such element.

To make the document of Figure 6.1 parsable we had to substitute the PersonType type of the third element of the declared complex type, with another type, e.g., xsd:string. Also, we had to add the following <xsd:element> declaration line:

```
<xsd:element name = "Company" type = "CompanyType"/>
```

For all the 1076 WSDL files that we used, we had to perform such modifications for making the files parsable.

```
<wsdl:types>
   <xsd:element name = "Person" type = "PersonType"/>
   <xsd:complexType name = "PersonType">
      <xsd:sequence>
         <xsd:element name = "name" type = "xsd:string"/>
         <xsd:element name = "age" type = "xsd:integer"/>
         <xsd:element name = "wife" type = "PersonType"/>
      </xsd:sequence>
   </xsd:complexType>
   <xsd:simpleType name = "CompanyType">
      <xsd:restriction base = "xsd:string"/>
   </xsd:simpleType>
</wsdl:types>

<wsdl:message name = "sendInfo">
   <wsdl:part name = "person" type = "Person"/>
   <wsdl:part name = "company" type = "CompanyType"/>
</wsdl:message>
```

Figure 6.1 Example of a WSDL document.

6.2.2 Description of the Input Queries

The benchmark, as mentioned, contains a set of OWLS queries. The notion of these queries is to combine naming, structural and behavioral properties of the searched services, however our query engine is not tailored to that. Thus, we had to adjust the queries to our query engine abilities, i.e., form queries into structures that could be passed as input to our query engine service's operations. We used the `AdvancedQuery` structure to form each query, and the respective operation to execute the queries. Below we detail the process we followed to produce the advanced queries.

Foreach OWLS query file, we formed a text file containing information able to form an advanced query. Each OWLS document of the queries folder describes a service offering a single operation, with a number of input and output parameters and this information is included in a compact form in the `<process:AtomicProcess>` element of the OWLS document. Figure 6.2 presents an example of such a query definition. For each OWLS document, we extracted the `<process:AtomicProcess>` element and wrote a text file serving as a form of advanced query. This was done automatically by an application we developed for this purpose, however we manually applied an additional modification over the text file. We illustrate our preprocessing with an example: Figure 6.3 shows an advanced query in text form, particularly the query derived from Figure 6.2 definition, after the manual modification phase. As can be inferred, the purpose of our manual intervention, was to keep only meaningful words, i.e, throw out parts such as "#_" and common words, such as "PROCESS" (or "METHOD", "OPERATION", "RETURN", "GET", "SET", etc in other circumstances). We also separate the semantical terms by adding an underscore ("_") character between them. Thus, each operation or parameter name consists of a set of semantical terms.

```
<process:AtomicProcess rdf:ID="DVDPLAYERMP3PLAYER_PRICE_PROCESS">
   <service:describes rdf:resource="#DVDPLAYERMP3PLAYER_PRICE_SERVICE"/>
   <process:hasInput  rdf:resource="#_MP3PLAYER"/>
   <process:hasOutput rdf:resource="#_PRICE"/>
   <process:hasInput  rdf:resource="#_DVDPLAYER"/>
</process:AtomicProcess>
```

Figure 6.2 Example of an operation's definition in a OWLS document.

```
Operation:
PLAYERS_PRICES
Input Message Types (2)
MP3_PLAYER
DVD_PLAYER
Output Message Types (1)
PRICE
```

Figure 6.3 The advanced query corresponding to the example of Figure 6.2, in plain text form.

Based on such a query in text form, we produce a series of advanced queries, each one corresponding to a combination of elements. Each advanced query comprises two elements; an operation name and a parameter name (either input or output, but not both). Specifically, each advanced query comprises, as the operation name, a term of the actual operation name and, as the input (resp. output) parameter name, a term of the actual input (resp. output) parameter name. Additionally, we produce advanced queries, each one comprising just the operation name (a term of the actual operation name) and nothing else. In every produced advanced query, the parameter's type is set to null. Figure 6.4 shows the set of advanced queries produced from Figure 6.3 in text form. Each line represents an advanced query in a brief form, in particular, the word in the left stands for the operation name, while the word in the right stands for the message type name (inpur or output). Each line represent an advanced query. The first two advanced queries comprise only the operation name, while the others comprise the operation name and a parameter name.

```
PLAYERS
PRICES
PLAYERS - MP3
PLAYERS - PLAYER
PLAYERS - DVD
PLAYERS - PLAYER
PLAYERS - PRICE
PRICES - MP3
PRICES - PLAYER
PRICES - DVD
PRICES - PLAYER
PRICES - PRICE
```

Figure 6.4 A representation of the advanced queries produced from text query of Figure 6.3.

6.2.3 Experimental Setup

We executed our standalone component on an Intel Dual-Core, 2.00 GHz, 3 GB RAM. The operating system was Windows 7 Professional. For the service base we employed MySQL Server 5.5.

For our distributed approach, we used as hardware components the nodes of a cluster of AMD Dual-Core, 2.2 GHz, 4 GB RAM computers, running Ubuntu 13.04.

In our experiments, we used our distributed abstractions mining tool to produce abstractions over the 1076 service descriptions that we mentioned before. Note that our algorithm takes four inputs, apart from the collection of service descriptions and the available nodes:

- the number of available nodes to be used
- the retention threshold
- the distance threshold
- the name extraction technique (LCS or RC)

We experimented on a big variety of combinations of the input values, and chose to present the most representative ones. We organized our presentation in three sets. For each set, we executed our tool by keeping constant the two of the first three inputs listed above, while varying the other two. Thus, each time, a different database instance was produced. Subsequently, we posed queries over these instances, using our query engine service. We measured our system's mining and querying efficiency, as well as the quality of the retrieved answers.

Measurement of the quality of the query results

We analysed how, based on a OWLS query, we extract a set of advanced queries, like the one depicted in Figure 6.4. For each OWLS query, we posed all these advanced queries to the system, using the `executeAdvancedQueryOverAbstractions()` operation of our query engine service. We also posed the same queries using the `executeAdvancedQueryOverInstances()` operation, reminding that this operation matches queries against concrete services instead of service abstractions. In both cases, we collected the answers from each advanced query and formed a union of them. Let *retrievedAnswers* be the union of the answers returned by the

`executeAdvancedQueryOverAbstractions()` operation, and *relevantAnswers* be the union of the answers returned by the `executeAdvancedQueryOverInstances()` operation. We calculated the precision and recall values, as the measures of quality of the answers to the OWLS query. In particular, we applied the following well-known formulas:

$$Precision = \frac{| relevantAnswers \cap retrievedAnswers |}{| retrievedAnswers |}$$

$$Recall = \frac{| relevantAnswers \cap retrievedAnswers |}{| relevantAnswers |}$$

We applied the steps mentioned in the previous paragraph to each of the 42 OWLS queries using a tool we developed for this purpose. Finally, we aggregated all queries measurements, producing an average value and a standard deviation of them, which are the values we actually present in this chapter. In some cases, the value of a query's quality metric is not defined mathematically, thus leading us to exclude this value from the calculation of the average value and the standard deviation.

Measurement of the query execution time

Similar to our query results quality presentation, we present the average execution time of the 42 queries, as a representative value for the entire query workload. Additionally, we present the measurements of what we call the "search" time, which is a part separated from the total query execution time. Basically, the query execution consists of two discrete phases:

- In case of querying over abstractions it comprises, in the following time order:
  - the phase of SQL querying, till the `abstractions'` table keys are found, i.e., the relevant functional abstractions are found
  - the phase of SQL table joining to compose the `FunctionalAbstraction` objects, that will be finally returned.

- In case of querying over instances (concrete services), it comprises, in the following time order:

- the phase of SQL querying till the `serviceinterfaces`' table keys are found, i.e., the relevant concrete service interfaces are found
- the phase of SQL table joining to compose the `ServiceInterface` objects, that will be finally returned.

In both cases, we call "search" time, the time consumed for the first phase, while we call "non search" time, the time consumed for the second phase.

The presented time durations were calculated in the following way: we executed our distributed abstractions mining tool once per case, as we relied on an experiments-dedicated Ubuntu computers cluster. On the other side, we executed the 42 queries 20 times per case, and present the average query time, as we relied on a typical Windows platform.

6.2.4 Findings

Hereafter, we present our experimental findings. For each investigated impact (set of experiments), we give the results of the two different representative name extraction techniques by presenting a pair of charts, namely:

a) for the technique which uses the longest common substring, (LCS), and
b) for the technique which just picks randomly one of the names of the two abstracted interfaces, (RC)

For each technique, we measured the distributed abstractions mining execution time, the precision and recall of the query results, as well as the query execution time.

1st set of experiments: the impact of the number of nodes

We investigated the impact of the number of hardware components participating in the distributed approach. We fixed the abstractions retention threshold to **0.9** and the distance threshold to **0.2**. For the number of hardware components to be used, we gave the values **2, 4, 8**.

- Figure 6.5 depicts **the impact of the number of nodes on the abstractions mining execution time.**

The findings show that the time decreases when the number of nodes increases, almost proportionally. An increase in the number of used nodes reduces proportionally the time spent by the leaf nodes, but on the other hand, introduces more steps (tree nodes) to the root. However, the additional steps are not so much, so as to overturn the time gained during the leaf level processing.

We also clearly observe that LCS is much faster than RC. A fact regarding the two methods, LCS and RC, is that LCS extracts rather small representative names. On the contrary, RC retains the names intact, this causing a serious delay in the name distance calculations.

- Figure 6.6 depicts **the impact of the number of nodes on the precision of the query results.**

For RC, we observe that the precision decreases when the number of nodes increases. An increase in the number of used nodes introduces more pruning steps, this causing less abstractions retained. Thus the quality of the results degrades. For LCS, a constant absolute precision is observed, with its values being equal to 1. We would expect it to decrease, however, this finding may be due to our small data set.

Comparing the precision values for the two methods, we find, as expected, that LCS gives better results. This happens beacause the names of all the represented objects (interfaces, operations, e.g.) of the abstractions that are mined using LCS, are relevant to the respective representative objects' names. RC extracts the representative name by randomly choosing one of the two represented ones, thus meaning that, if an abstraction matches a user's query, not all of its represented interfaces, which will be also retrieved, are relevant to the query.

- Figure 6.7 shows **the impact of the number of nodes on the recall of the query results.**

Quite medium recall values, as well as a values' decrease with the increase of the number of nodes is observed for both methods. As also mentioned in the case of precision values, the results quality is expected to degrade as the number of nodes increases, due to the introduction of more pruning steps, which causes less abstractions to be retained.

We also find that RC's values are generally bigger than those of LCS; LCS extracts too cropped representative names, that cannot be easily matched with the names the user includes in his query. On the contrary, RC does not crop the initial names, therefore, despite the small relativeness of some of the represented services with the representative one, it gives more chances to query-relevant services for being retrieved.

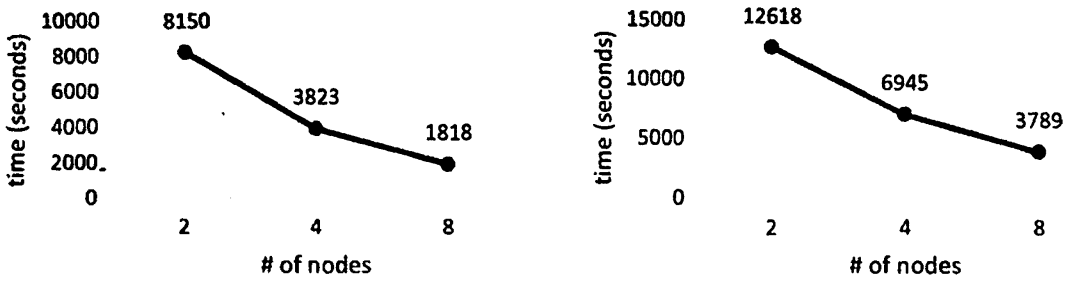
Regarding LCS, we see that the recall values in cases of 4 and 8 used nodes do not much differ. An explanation for this could be that the numbers of retained abstractions in cases of 4 and 8 nodes used, do not much differ, and, particularly they are both quite small (if they were both big, they would probably differ more). This is also justified by the query execution time behavior, which we discuss in the next paragraph. Since we expect LCS to mine abstractions with quite low children distance threshold, we conclude that most of the abstractions mined by LCS have a distance threshold value which is smaller than the 0.2 value that we set.

- Figure 6.8 shows **the impact of the number of nodes on the query execution time.**

We generally observe the time values decreasing with the increase in the number of nodes; less abstractions retained means less time consumed for searching, as the searching is applied to the abstractions and not to the concrete services.

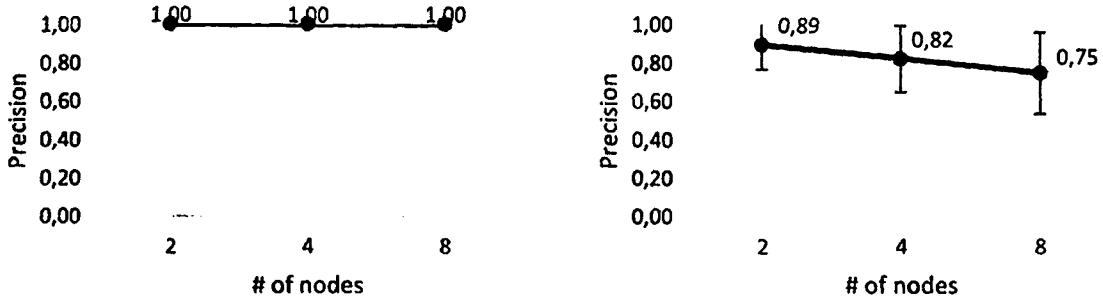
We also find LCS faster than RC. Actually, LCS should be faster, due to the small extracted names; it should be faster in terms of search time, because sql name comparisons will delay, and it should be quicker also in terms of the non-search time, i.e., the composition time, because smaller names would need to be retrieved from the database.

We find RC leading to rather expected results, with the time values almost constantly decreasing with the increase in the number of nodes. Nonetheless, LCS would behave the same, if it was not for the observed misconduct in the case of 8 nodes used and, actually, this is confirmed by all the three presented charts; even the search time chart shows a decrease, but not to the expected degree. This probably relates to the previous paragraph's respective finding, which, as explained, may occur due to the little difference between the retained abstractions, in cases of 4 and 8 used nodes.



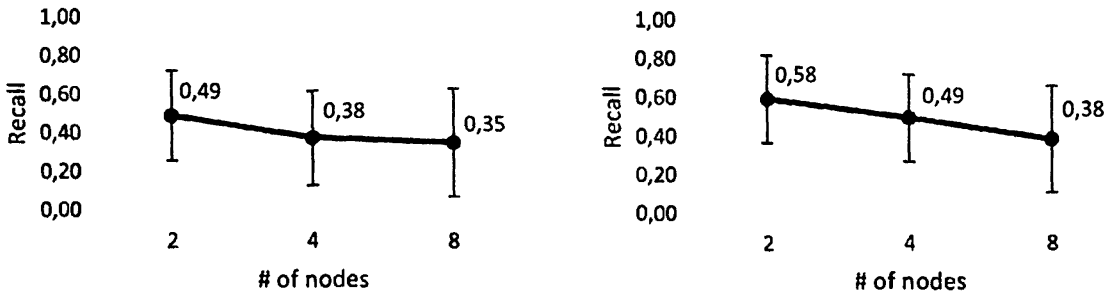
(a) LCS (b) RC

Figure 6.5 The impact of the number of nodes on the abstractions mining execution time.



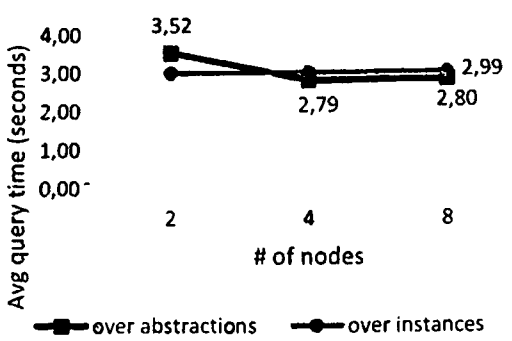
(a) LCS (b) RC

Figure 6.6 The impact of the number of nodes on the precision of the query results.

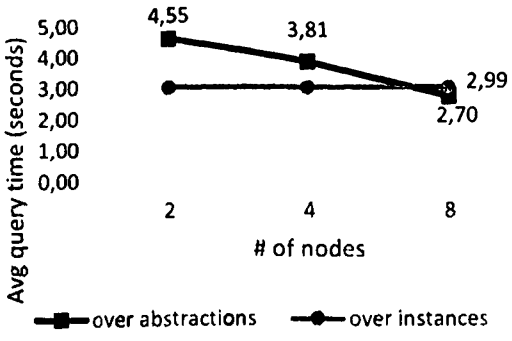


(a) LCS (b) RC

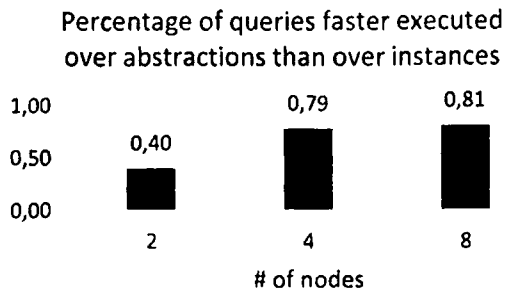
Figure 6.7 The impact of the number of nodes on the recall of the query results.



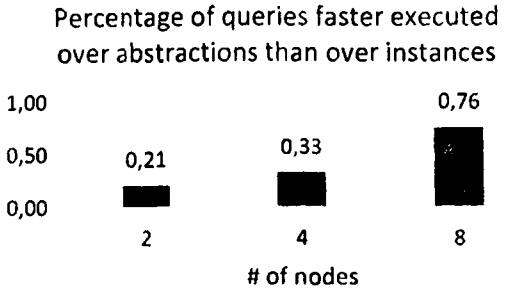
(a1) LCS



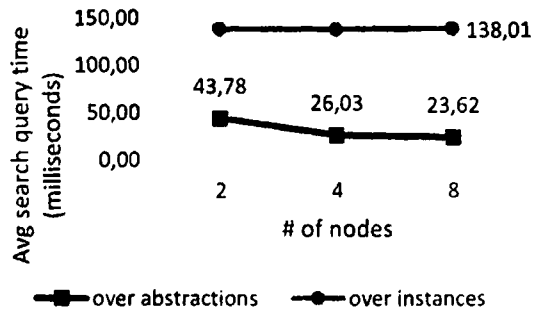
(b1) RC



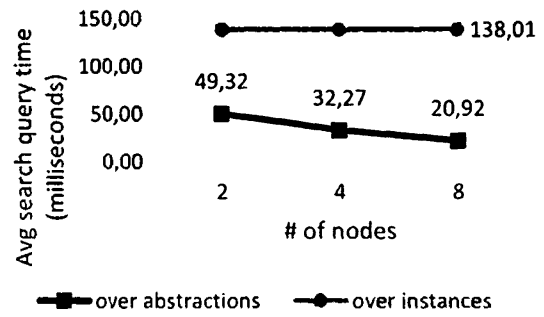
(a2) LCS



(b2) RC



(a3) LCS



(b3) RC

Figure 6.8 The impact of the number of nodes on query execution time.

2nd set of experiments: the impact of the abstractions retention threshold

We investigated the impact of the abstractions retention threshold. We fixed the number of nodes to 8 and the distance threshold to **0**. Actually, the two thresholds interact during the overall distributed mining process and the pure impact of each of them is affected by the other. Setting the distance threshold to 0 leads to the distance threshold not affecting the overall

distributed mining process, since every time the distance between the two abstracted interfaces of an abstraction is examined, it will be found equal or bigger than 0, thus the child abstractions will be retained and the parent abstraction will be thrown away. For the retention threshold, we gave the values **0.33, 0.50, 0.66, 0.83, 1.00**.

- Figure 6.9 shows **the impact of the abstractions retention threshold on the abstractions mining execution time.**

  There is found an increase in the consumed time as the abstractions retention threshold increases. For our distributed process, bigger proportion of abstractions' retention means more abstractions retained at each node, thus more abstractions passed to the parent node. This results to each node having to process more abstractions, i.e., the overall processing time increases.

  Another point is that the curves' gradient is not proportional to the increase of the abstractions retention threshold, and this can be due to the fact that a part of the consumed time regards the initial abstractions mining performed by the leaf nodes, which is the same in all cases. Also, the pruning process adds an overhead, thus in cases of smaller retention threshold, i.e, more pruning, the overhead will be bigger.

  LCS is much faster than RC, for the reasons we mentioned earlier.

- Figure 6.10 depicts **the impact of the abstractions retention threshold on the precision of the query results.**

  In case of RC, we observe a slight increase in the precision values, as the retention threshold increases. This happens because, bigger proportion of abstractions retention means more abstractions retained, thus the results' quality upgrades.

  We find again LCS giving better results than RC, in particular, the observed precision values are equal to 1, independently of the change in the value of the abstractions retention threshold.

- Figure 6.11 depicts **the impact of the abstractions retention threshold on the recall of the query results.**

  The two methods' behavior is much similar in this occasion; as explained in the previous bullet, an increase in the number of retained abstractions means better quality of the query results, the recall values increase with the increase of the retention threshold.

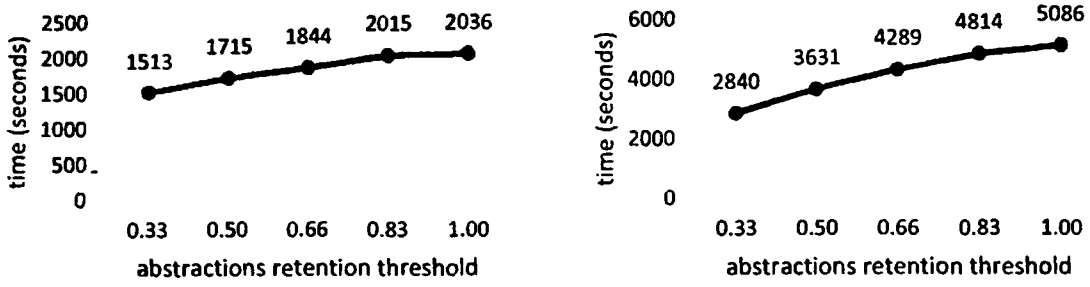
  The precision values in case of RC are generally bigger than those in case of LCS, an expected finding that we explained earlier..

- Figure 6.12 shows **the impact of the abstractions retention threshold on the query execution time.**

  First of all, we observe that, for both methods, the pure search query time behaves in a rather expected manner, increasing, at an almost constant rate, as the retention threshold's value increases. This is explained by the fact that there are more abstractions retained, thus more abstractions to be searched.

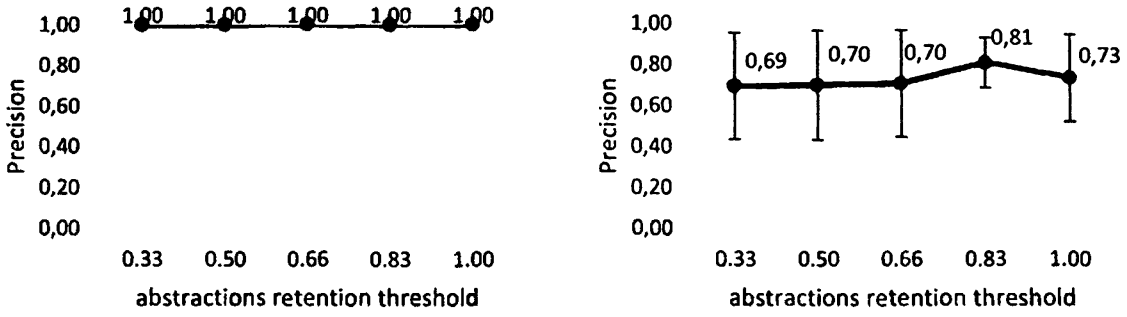
  Secondly, the indicated by the (b3) chart change rate, is confirmed by the (b2) one but not quite confirmed by the (b1) chart. Additionally, the (a3) chart is not quite confirmed by neither the (a1) chart nor the (a2) chart. Another point is that, according to (b1) and (b2) charts, when the threshold's value exceeds 0.33, the query execution time exceeds the respective one consumed by querying over instances. Later on, we explain some of the reasons that cause this kind of findings.

  Another finding is that, as expected, LCS leads to a considerably faster query execution time than RC, both in terms of total time and search time.



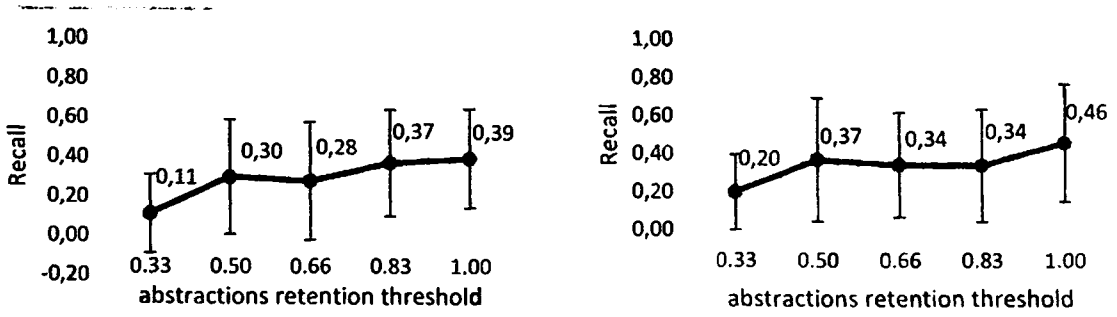
(a) LCS (b) RC

Figure 6.9 The impact of the abstractions retention threshold on the abstractions mining execution time.



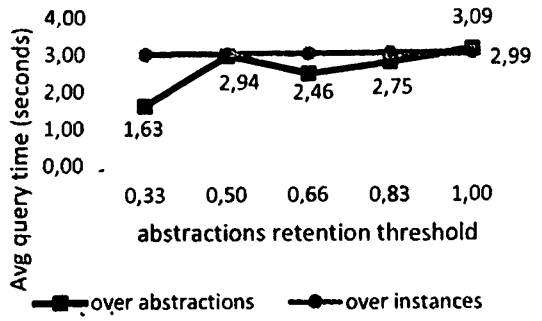
(a) LCS (b) RC

Figure 6.10 The impact of the abstractions retention threshold on the precision of the query results.

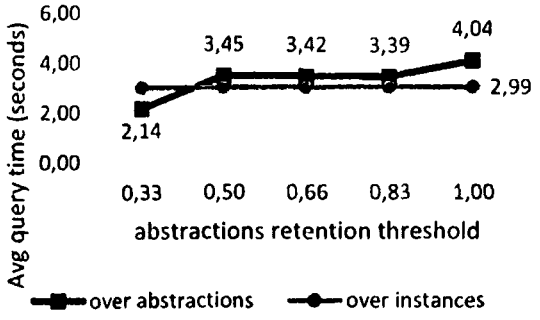


(a) LCS (b) RC

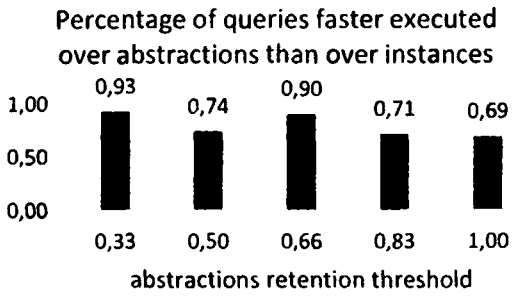
Figure 6.11 The impact of the abstractions retention threshold on the recall of the query results.



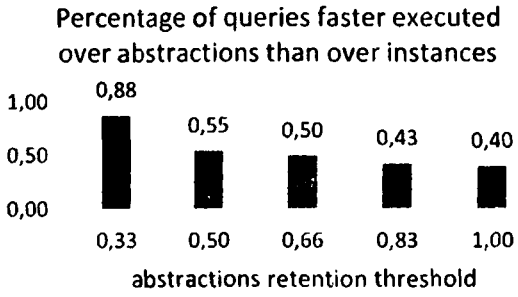
(a1) LCS



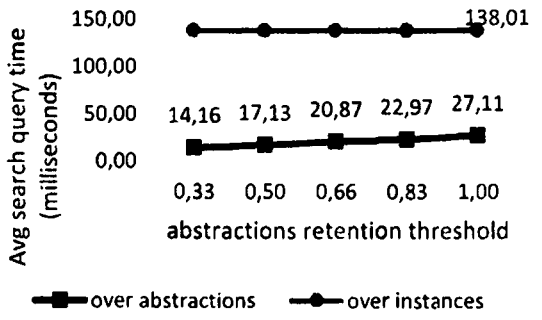
(b1) RC



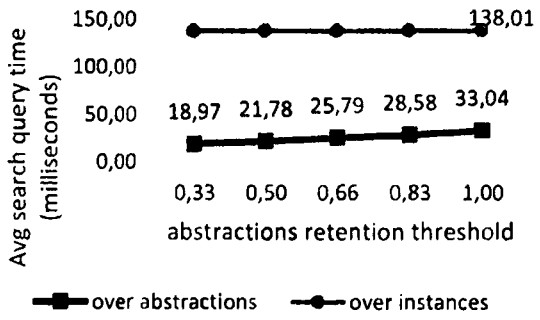
(a2) LCS



(b2) RC



(a3) LCS



(b3) RC

Figure 6.12 The impact of the abstractions retention threshold on query execution time.

3rd set of experiments: the impact of the distance threshold

We investigated the impact of the represented interfaces distance threshold. We fixed the number of nodes to 8 and the abstractions retention threshold to **0.83**. For the distance threshold, we gave the values **0.00, 0.16, 0.33, 0.50, 0.66**.

- Figure 6.13 depicts **the impact of the distance threshold on the abstractions mining execution time.**

Both methods behave normally; the execution time decreases as the distance threshold increases. An increase in the distance threshold's value causes more children abstractions being pruned (or else, more parents retained), thus meaning less abstractions retained.

Again, we find that LCS is significantly faster than RC.

A more close look at the findings concerning LCS reveals that, above a specific value of the distance threshold, no significant changes in the time values happen. Actually, this is expected to be observed not only in case of the abstractions mining execution time, but in all cases of our metrics. The reason for that is that there should normally be a limit in the distance threshold's value, above which almost all parent-children abstraction structures will have their children pruned. This, in conjunction with the fact that there is a priority in the application of the two thresholds criteria, i.e., the distance threshold is firstly applied, leads, especially in cases of big distance threshold values, to the pruning of an abstractions hierarchy terminate, even if the number of retained abstractions is not even close to the number of abstractions that are dictated to be retained. Thus, when the distance threshold's value reaches that limit, the number of retained abstrations will be abruptly reduced to a very small number, not likely changed any more.

We observe, as we would expect, that this limit is lower for LCS than for RC; RC chooses entire names as representative names, and this leads to more heterogeneous abstractions, as the abstractions mining process goes on to higher levels, i.e., abstractions not actually representing the services they are supposed to represent. On the contrary, LCS uses parts of all services' (and operations', messages', etc.) names to compose representative names, thus mining more homogeneous abstractions. Therefore, most abstractions mined by LCS have a lower children distance threshold, than those mined by RC.

- Figure 6.14 depicts **the impact of the distance threshold on the precision of the query results.**

  Concerning RC, we observe a small decrease in the precision values, as the distance threshold increases, which is an expected finding.

  Again, we observe the precision values for LCS being constantly 1, not affected by the change in the distance threshold.

- Figure 6.15 depicts **the impact of the distance threshold on the recall of the query results.**

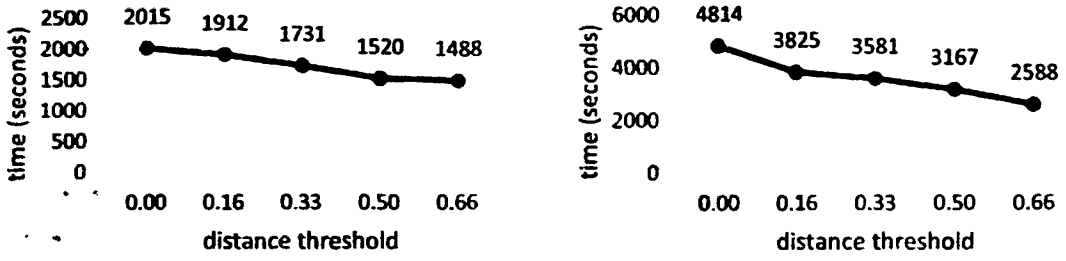
  The two methods behave quite differently in this case; while for RC we observe a slight decrease in the recall values as the distance threshold increases, for LCS we observe a significant decrease. Moreover, for LCS, we observe that when the distance threshold's value reaches a limit, around 0.50, there is an abrupt reduction of the recall values to a disappointing level. We also see that, above this limit, no significant changes to the recall values happen. This phenomenon, regarding the distance threshold's limit and why it is lower for LCS, was explained earlier.

- Figure 6.16 shows **the impact of the distance threshold on the query execution time.**

  We find that, for both methods, the pure search query time behaves as expected, decreasing as the retention threshold's value increases. The (a3) and (b3) charts' indications are almost confirmed by the rest of the charts, except for the fact that the little changes in (a3) and (b3), correspond to practically no changes in the other charts.

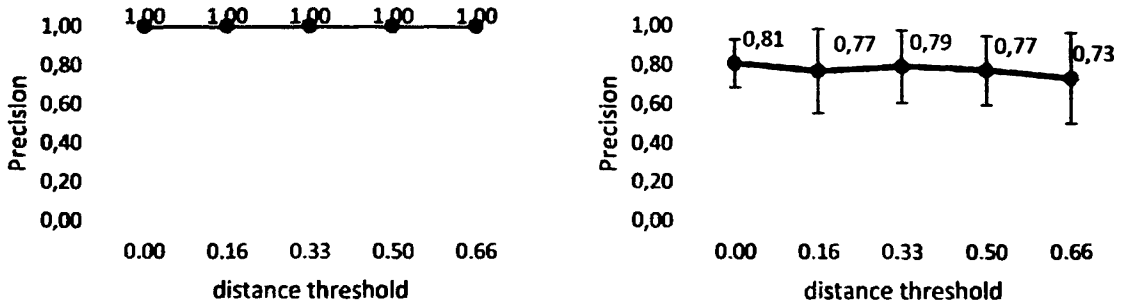
  Concerning LCS, we observe that, above value 0.50 of the distance threshold, all metrics' values are only slightly affected by the threshold's further increase. This is not observed in RC's charts, confirming our expectations regarding the distance threshold's limit and why it is lower for LCS. In this case we conclude that, for LCS, this limit must

be inside the distance threshold values' range that we experimentally used, while, for RC, must be outside.



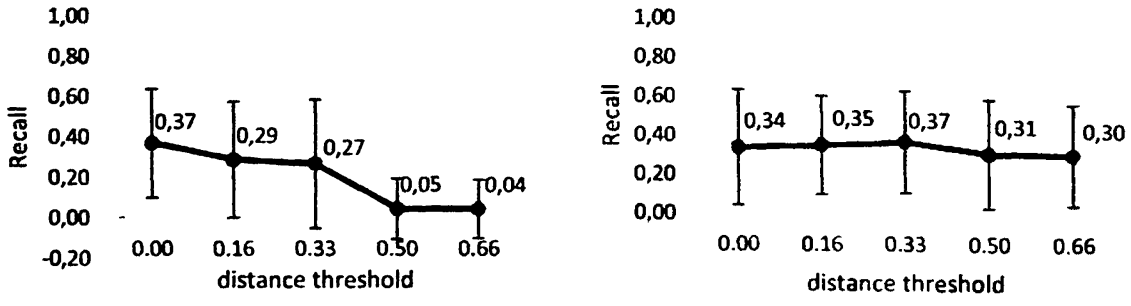
(a) LCS (b) RC

Figure 6.13 The impact of the distance threshold on the abstractions mining execution time.



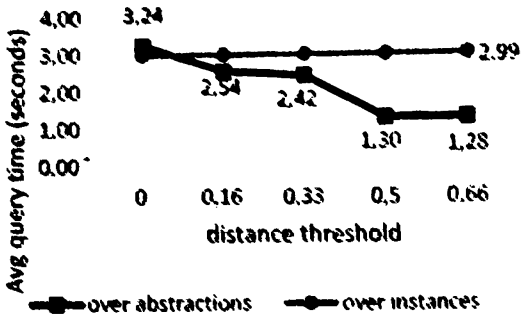
(a) LCS (b) RC

Figure 6.14 The impact of the distance threshold on the precision of the query results.

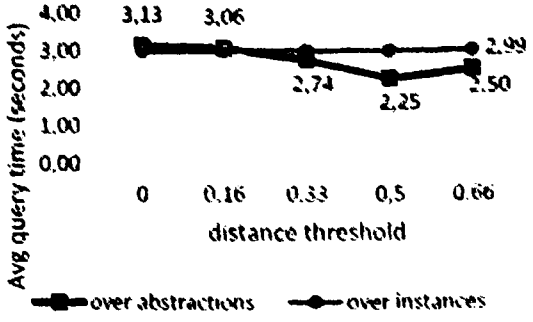


(a) LCS (b) RC

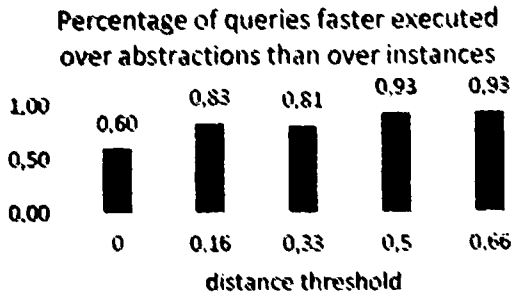
Figure 6.15 The impact of the distance threshold on the recall of the query results.



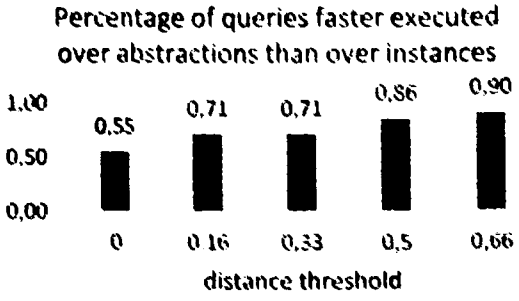
(a1) LCS



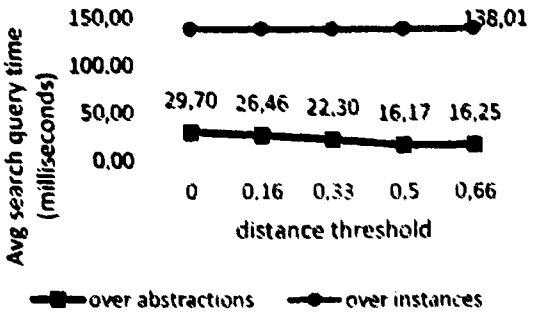
(b1) RC



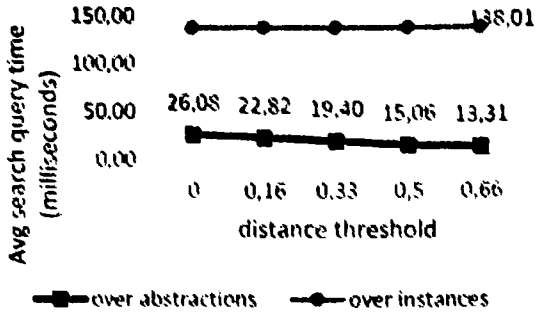
(a2) LCS



(b2) RC



(a3) LCS



(b3) RC

Figure 6.16 The impact of the distance threshold on query execution time.

Additional explanations concerning the experimental findings

We have mentioned that the query process can be divided in two phases, the search and the non-search (composition) phase.

Thus, the main difference between the two querying methods is the additional composition of the representative interface and the interfaces mappings, in the case of querying over

abstractions. Approximately, this time overhead is proportional to the total number of concrete services that the returned abstractions represent.

In the small-scale experiments performed with OWL-TC benchmark, we found that, in both querying cases, the proportion of the time consumed during the first phase is quite small, compared with the overall consumed time. We measured these proportions for all queries in all experiments, and we present the average values. As Figure 6.17 points out, only a 4.6% of the overall time is consumed during the first phase in case of querying over instances. An even much smaller percentage, 0.8% of the overall time, is consumed during the first phase in case of querying over abstractions. As we can see, the first phase, in case of querying over instances, proportionally lasts almost 6 times more than the respective one in case of querying over abstractions. Figure 6.18 depicts how many times bigger the non-search time is, than the search time, in the two querying cases.

All the aforementioned findings show that, the quicker search time that we expect for the querying over abstractions method, comes along with an overhead to the non-search time. This implies that the clear difference observed in search times, between the two querying methods, may not be observed in total times, and perhaps, in some cases, the total time consumed by the querying over abstractions method may exceed the total time consumed by the querying over instances method.

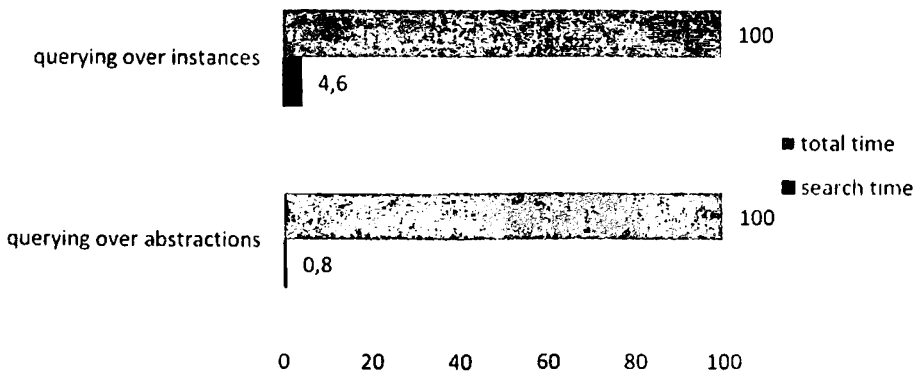


Figure 6.17 Comparing the pure search time with the total time consumed for querying.

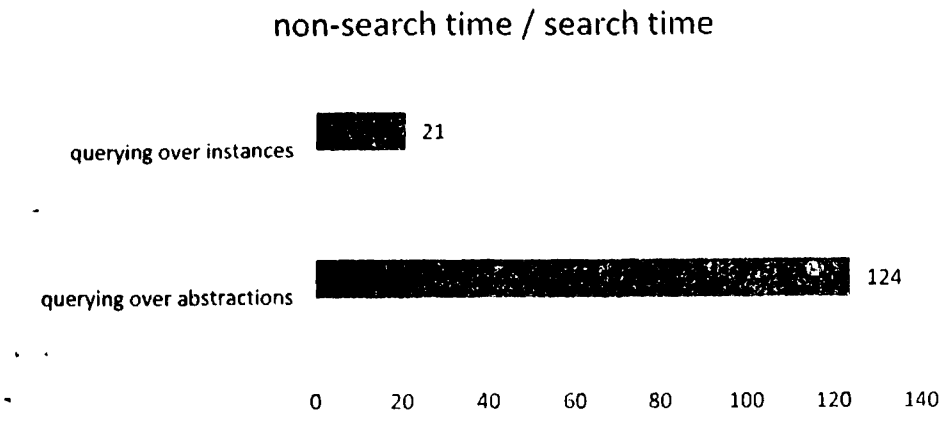


Figure 6.18 The quotient of the non-search time / total time, in the two querying cases.

In our experiments, we observed some unexpected findings, e.g., querying over abstractions being slower than querying over instances. The explanation of these phenomena lies in the combination of three main facts:

- The time overhead of the querying over abstractions method, which we described earlier.
- The high deviation of the number of services returned by each query. For instance, the querying over instances method returns 161 services on average, with a standard deviation of 283 services.
- The fact that not all 42 queries are uniformly affected by a change in a threshold's value of our experiments. By this we mean that the proportion of resulted services that each query produces, can significantly change, owing to a change in a threshold's value. The problem for us is that, measuring the mathematical average query execution time of the 42 queries, we consider each query execution having the same importance, not taking into consideration the very different number of services each query returns. The 42 queries are not equally affected, in fact, what the most affected queries are, plays a critical role; if the queries that usually (in executions with other threshold values) retrieve big numbers of services are affected the most, this may lead to unexpectable findings. Thus, the average value can be easily affected by a few overweighted queries.
- The small scale of our data set. An important issue comes from the experiments towards scalability, presented in section 6.3. We observe that, no matter what the abstractions/services ratio is, the more the number of registered services decreases, the

much closer the querying over abstractions time becomes to the corresponding time of querying over instances. This implies that, in the case of the 1076 services, which we use in these experiments, the difference in the two querying methods' execution times could be quite small or non-existent.

6.3 Scalability Assessment

Description of the Input Data Set

The input data we have used in these sets of experiments were synthetically produced data. We developed a data generator for this purpose. The generator takes as input the number of services and the number of abstractions, and creates, for each of our service base's relations, a text file consisting of synthetic records, according to the input parameters. We also manually changed some records to fit our query needs. Specifically, in the `ServiceInterface`'s text file, we changed some records' name to *weather*, in `Operation`'s text file we changed some records' name to *getHumidity* some others' to *getTemperature*. We did so because, the query we used for our experiments comprises the aforementioned terms.

Description of the Input Queries

We used our `SimpleQuery` class to query the service base. In particular, we created a simple query to search for services offering 2 operations, one whose name contains the term *getTemperature* and one whose name contains the term *getHumidity*.

Experimental Setup and Findings

We executed our experiments on an Intel Dual-Core, 2.00 GHz, 3 GB RAM. The operating system was Windows 7 Professional. For the service base we employed MySQL Server 5.5.

We organized our experiments in two sets, which are briefly described below. In both sets we compare querying over service abstractions with querying over concrete service descriptions, while varying different parameters. Further details concerning the experiment setup for each set of experiments (operations per service, in/our parameters per operation, required disk space) are given in Table 6.1. We performed each experiment 10 times and we report the average

execution times. Figure 6.19 gives the results that we obtained; the reported numbers are the average values.

- **1st set of experiments: The impact of scaling the number of services and the number of abstractions**

  We varied the number of service abstractions from $5 * 10^3$ to 10^6 and the number of service descriptions from $5 * 10^4$ to 10^7; hence, each service abstraction represented 10 services.

- **2nd set of experiments: The impact of scaling the number of services**

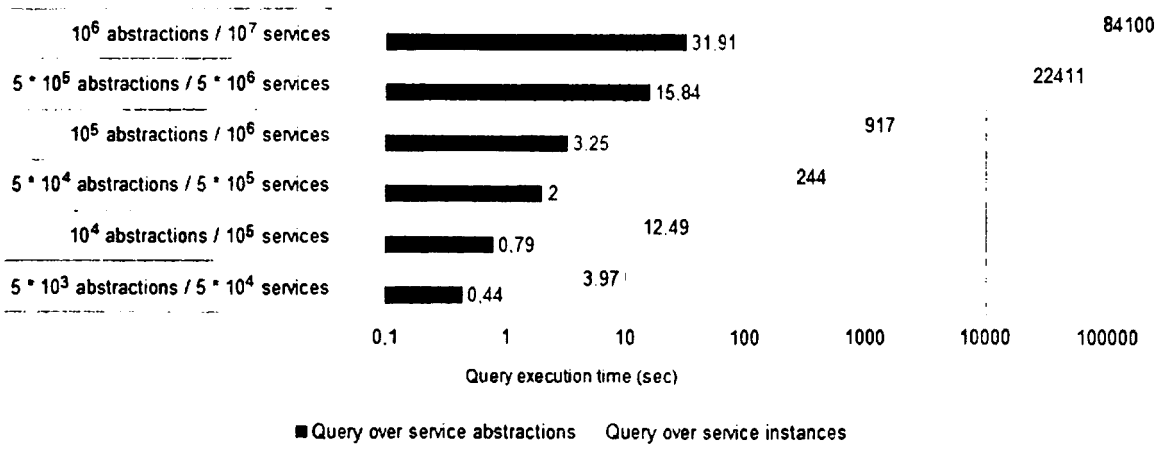
  The number of stored abstractions was 10^4, while the number of service descriptions ranged from $5 * 10^4$ to 10^7; thus, the number of represented services per abstraction varied from 5 to 1000.

Table 6.1 Experimental setup for query execution performance towards scaling.

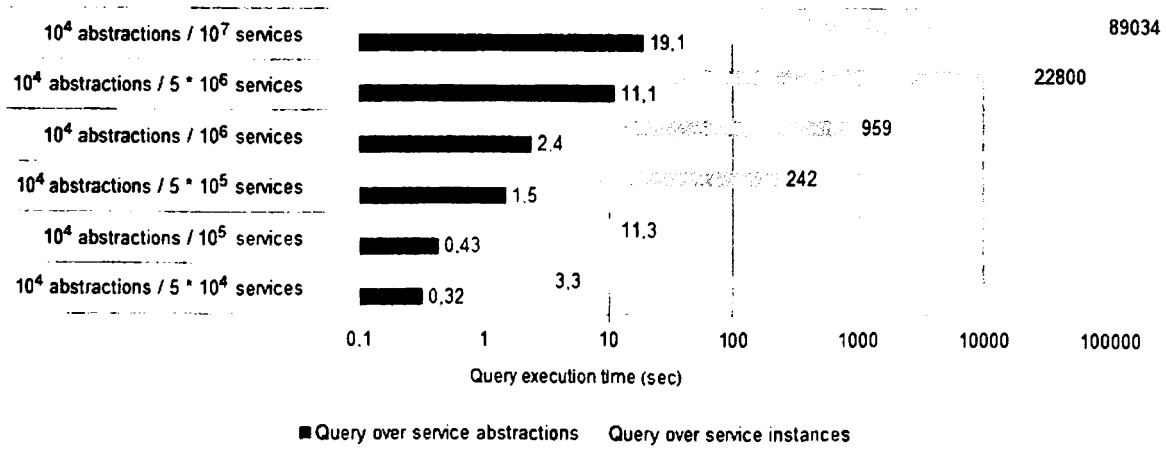
1st set of experiments data set properties						
# service abstractions	$5 * 10^3$	10^4	$5 * 10^4$	10^5	$5 * 10^5$	10^6
# concrete services	$5 * 10^4$	10^5	$5 * 10^5$	10^6	$5 * 10^6$	10^7
# represented services per abstraction	10					
# operations per service	3					
# in parameters per operation	2					
# out parameters per operation	2					
overall disk space (MB)	159	325	1700	3451	17920	37478
2nd set of experiments data set properties						
# service abstractions	10^4					
# concrete services	$5 * 10^4$	10^5	$5 * 10^5$	10^6	$5 * 10^6$	10^7
# represented services per abstraction	5	10	50	100	500	1000
# operations per service	3					
# in parameters per operation	2					
# out parameters per operation	2					
overall disk space (MB)	171	325	1587	3205	16486	33382

Concerning the query over concrete services, in both sets of experiments (Figure 6.19 (a) and (b)), we observe that the execution time increases with the number of concrete service descriptions stored in the service base. Concerning the query over abstractions, in the 1st set of

experiments, the execution time increases with the number of stored service abstractions. On the other hand, in the 2nd set of experiments, the execution time increases with the size of the result; the number of represented services for the service abstractions that are returned by the query varies from 5 to 1000. In both sets of experiments querying over service abstractions is much faster than querying over concrete service descriptions. More specifically, in the 1st set of experiments, querying over service abstractions is 88% to 99% faster than querying over concrete service descriptions. Similarly, in the 2nd set of experiments, querying over service abstractions is 90% to 99% faster than querying over concrete service descriptions.



(a) 1st set of experiments



(b) 2nd set of experiments

Figure 6.19 Querying over abstractions vs. querying over instances.

6.4 Conclusion

Performance and Quality

There normally exists an inherent trade-off in trying to improve the consumed time, both in cases of abstractions mining execution and query execution, while retaining the quality of the query results in a satisfying level. Our experimental findings quite justify our efforts to confront this issue.

In general, we found that we can tune the three inputs of our tool, namely the number of nodes, the abstractions retention threshold and the distance threshold, so as to reduce the abstractions mining and query execution times, while obtaining query results of good quality. In fact, the abstractions mining execution time significantly decreases, the query execution time generally decreases, and the query results quality is bearably degraded. Some observed unexpected findings, mainly regarding the query execution time values, are due to the limited number and variety of our input service descriptions, and most of all, the high diversity of the applied queries.

We briefly denote the three thresholds' impact:

- ***Number of nodes***

  An increase in the number of used nodes:

  - enhances parallelism, thus the abstractions mining execution time is significantly reduced
  - introduces more pruning steps, which causes less abstractions being retained, thus
    - ➢ the query execution time decreases
    - ➢ the query results' quality decreases

- ***Abstractions retention threshold:***

  An increase in the abstractions retention threshold:

  - retains more abstractions to be processed by each internal node, so the abstractions mining execution time increases

- increases the final number of retained abstractions, thus
  - the query execution time increases
  - the query results' quality increases

- ***Distance threshold:***

  An increase in the distance threshold's value
  - retains less abstractions to be processed by each internal node, so the abstractions mining execution time decreases
  - decreases the final number of retained abstractions, thus
    - the query execution time decreases
    - the query results' quality decreases

  However, there is normally a limit in the distance threshold's value, above which, no significant changes happen.

Based on the aforementioned findings, we come to the conclusion that, given a fixed number of nodes, the combination of values of the two thresholds, that should be specified in order to achieve better results, has to be quite small for the distance threshold, preferably in the range [0.1, 0.3], and quite high for the abstractions retention threshold, preferably in the range [0.7, 0.9]. The exact values' specification depends a lot on the collection of service descriptions that have to be processed, especially on the level of thematical diversity among services. A practical approach to this issue, is that the administrators create a sample collection out of their entire collection of service descriptions. As a preprocessing step, our system could be tried with the sample data set and using a set of different combinations of input values, thus finding the combination which best fits the administrators' needs.

We briefly denote the differences between the LCS and the RC method:

- ***Distributed abstractions mining execution time:***

  The critical part concerns the names' distance calculations. The LCS method extracts smaller names, thus the mining process is quicker than the RC method. Moreover, the more calculations are executed, the bigger the difference between the two methods' execution time becomes. We have more calculations when the number of nodes

increases, the abstractions retention threshold increases and the distance threshold decreases.

- ***Query results precision:***
  The LCS method mines abstractions with their representative names being more relevant to the corresponding represented ones, than in case of RC method happens. Thus, an application of the LCS method leads to a better precision of the query results, in fact, a constant absolute precision is observed.

- ***Query results recall:***
  The LCS method extracts too cropped representative names, that cannot be easily matched with the names the user includes in his query. The RC method, when applied, leads to a better recall of the query results since, despite the small relativeness of some of the represented services with the representative one, it gives more chances to query-relevant services for being retrieved.

- ***Query execution time:***
  Because of the small representative names, the LCS method leads to quicker query execution, than the RC method does.

Additionally, the two methods differ in that the aforementioned limit in the distance threshold value, will generally be smaller in the case of LCS method. That happens because the abstractions mined by the LCS method are more homogeneous, i.e., there is a bigger relevance between the representative and the represented interfaces.

Overall, we find the LCS method much more efficient that the RC method. In particular, the RC method gives better results only in terms of the recall of the query results, while the LCS method leads to much quicker execution times and much better precision of the query results.

Scalability

Our substantial experimental finding is that, the more the number of registrated services increases, the much more quicker querying over abstractions becomes, compared with querying over concrete services. Regarding an ultra large scale of services, the results are rather impressive.

CHAPTER 7. CONCLUSION AND ADDITIONAL CHALLENGES

7.1 Conclusion

Service-Oriented Computing (SOC), despite emerging as a very promising trend for application development, has failed to be widely used. The main reason for that is the limited efficiency and effectiveness of the current search technologies; structured queries, which mainly concern a developer, are not offered, while search time is high, since answering a query requires matching it against all the services, thus meaning that search time scales with the number of services.

Abstraction-Oriented Service Base Management (AoSBM) introduces a clustering technique; the summaries that characterize the clusters are called service abstractions. A service abstraction represents a group of services that have similar functional properties (operations, inputs, outputs, etc.). The lookup queries are matched against service abstractions, thus the query execution time scales with the number of service abstractions, instead of scaling with the number of service descriptions.

We built upon the notion of service abstractions and the abstractions mining algorithm used in AoSBM to facilitate the organization of large unstructured collections of service descriptions and the execution of service lookup queries. More specifically, we developed a service

discovery facility that we called service base. The main constituents of the service base are the following:

- A novel, scalable distributed abstractions mining facility that makes the clustering of large collections of service descriptions feasible.
- A user-friendly query engine facility that enables the execution of service lookup queries over abstractions. Moreover, we developed a Web service that provides access to the query engine and allows using the service base in a distributed setting

We experimentally tested our system for scalability, performance and quality of query results.

- For scalabilty, using an ultra large scale of synthetic data, we found that, the more the number of registrated services increases, the much faster the querying over abstractions is, than querying over concrete services.

- For performance and quality of query results, using a benchmark of 1076 real-world service descriptions, we found that our approach suggests a considerable solution to the aforementioned issues, besides the normal trade-off in trying to improve both performance and quality.

7.2 Additional Challenges

The baseline system (AoSBM) accompanied with our distributed abstractions mining approach is aimed at improving the overall process of organizing Web services on the purpose of quicker and qualitative service retrieval. This approach, as illustrated in our experiments, improves the query execution time, while returning satisfactory results in terms of their quality. However, our experimental findings reveal a number of challenges and potential future improvements regarding the abstractions mining execution time, the query execution time and the quality of the query results. Hereafter, we present some of them, starting from the more straightforward ones, i.e., those that can be realized without radical changes.

The division of the services collection into subcollections, prior to their distribution to the computer nodes during the distributed abstractions mining process, could not be arbitrary.

Specifically, a fast clustering technique could be applied to the initial collection, which would divide it into clusters of similar services. There are many chances that this will improve the quality of the retrieved query results, both in terms of precision and recall, since it would increase the homogeneity of the abstractions mined by the leaf computer nodes.

A challenge for improving the query execution time comes from the experimental findings; As indicated in the evaluation chapter, the pure search query time metric, behaves as expected; in all sets of our experiments, the querying over abstractions method's values have an almost constant rate of increase or decrease, and are many times less than the respective values of the querying over instances method. However, the behavior of the total query execution time metric, is not enough tailored to this. We explained that this misconduct is due to the additional load of the querying over abstractions method, which is the composition of the representative interface and the interfaces mappings. Thus, for users not concerning about software component adaptation, and moreover, can search themselves to find the results' elements mapping their query elements, this additional load could be alleviated.

Moreover, the querying over abstractions method has the load of the composition of the represented interfaces, i.e., the interfaces of the concrete services, stored in the service base after the service descriptions are parsed. We could additionally store the entire WSDL documents as plain text, so that we could retrieve them, rather than reconstructing them, i.e., the abstraction object could be composed of a string, representing the WSDL file, instead of the represented interfaces' list.

Another challenge emerges from the storage architecture of our system. We employed a standalone database for storing both services and abstractions. We could make this architecture distributed. In this way, a set of computational nodes would be utilized to store the mined abstractions to their own instances of the database. This would decrease the distributed abstractions mining execution time, because it would eliminate the time it is currently consumed for the storage of the mined abstractions to the standalone database. However, the main reason for such an approach, is a possible great improvement in the query execution time, since the distribution of a query, as well as the results' joining, is quite easy; the same query will be posed to each node, and the final result would simply be the union of the individual ones.

An important aspect we should also consider, is the calculation of the distance between two service interfaces, proposed by the authors in [6], and particularly, the part that calculates the distance between the two names, either regarding interfaces names, operations names, messages names or message types names; it relies on the syntactical diferrence between the two words, based on experimental findings showing that such a syntactical difference usually indicates a respective semantical difference. However, we could even improve the algorithm by calculating the semantical difference between the two words. A simple approach comprises a database storing information about words, their synonyms and antonyms, and perhaps a ranking of them. Then, the names distance calculation could leverage this information to calculate the semantic distance between the two names. Of course, a name usually contains not a single, but several semantically different words, so a preprocessing step which separates them would be needed. In cases one or more words of a name are not contained in the database, the syntactical difference calculation could be additionally applied. Another solution similar to this, comprises service descriptions containing semantic information, as have been proposed in [18, 21, 23].

Another approach based on the proposal of the previous paragraph, regards the representative name extraction. As the names would consist of a number of separated words, the pairs having the biggest semantical similarity could be chosen as constructs for the representative name; for each pair, the one of the two words could be finally chosen, or perhaps none of them would be chosen, but their synonym which best expresses their meaning, and could be found in the aforementioned database.

REFERENCES

[1] Al-Masri, E., Mahmoud, Q.H.. "Investigating Web services on the World Wide Web". 17th International Conference on World Wide Web (WWW). 2008

[2] Athanasopoulos, D., Zarras, A., Issarny, V.. "Service substitution revisited". 24th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2009

[3] Athanasopoulos, D., Zarras, A., Issarny, V.. "Towards the maintenance of service oriented software". 3rd CSMR Workshop on Software Quality and Maintenance (SQM). 2009

[4] Athanasopoulos, D., Zarras, A., Issarny, V., Vassiliadis, P.. "Hiding Design-Decisions in Service-Oriented Software via Service Abstraction Recovery". Technical Report inria-00491349 - version 2. INRIA, 2010. Available HTTP://HAL.ARCHIVES-OUVERTES.FR/.

[5] Athanasopoulos, D., Zarras, A., Vassiliadis, P.. "Service Selection for Happy Users: Making User-Intuitive Quality Abstractions". ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE). 2012

[6] Athanasopoulos, D., Zarras, A., Vassiliadis, P., Issarny, V.. "Mining Service Abstractions (NIER Track)". 33rd International Conference on Software Engineering (ICSE). 2011

[7] Atkinson, C., Bostan, P., Hummel, O., Stoll, D.. "A Practical Approach to Web Service Discovery and Retrieval". 29th International Conference on Software Engineering (ICSE). 2007

[8] Cilibrasi, R.L., Vitnyi, P.M.B.. "The Google Similarity Distance". IEEE Transactions on Knowledge and Data Engineering. Vol. 19, No. 3, pp. 370-383. March 2007

[9] Dong, X., Halevy, A., Madhavan, J., Nemes, E., Zhang, J.. "Similarity Search for Web Services". Proceedings of VLDB. 2004

[10] Elgazzar, K., Hassan, A.E., Martin, P.. "Clustering WSDL documents to bootstrap the discovery of Web services". IEEE International Conference onWeb Services (ICWS). 2010

[11] Fan, J., Kambhampati, S.. "A Snapshot of Public Web Services". SIGMOD Record 34(1). 24-32. 2005

[12] FIA. "The Cross-ETP Vision Document. Technical Report". Future Internet Assembly. 2009

[13] Issarny, V., Georgantas, N., Hachem, S., Zarras, A., Vassiliadis, P., Autili, M., Gerosa, M.A., Hamida, A.B.. "Service-Oriented Middleware for the Future Internet: State of the Art and Research Directions". Journal of Internet Services and Applications 2, 1. 23-45. 2011

[14] Kim, S.M., Rosu, M.C.. "A Survey of Public Web Services". Proceedings of the 13th International World Wide Web Conference (WWW). 312-313. 2004

[15] Li, Y., Liu, Y., Zhang, L.J., Li, G., Xie, B., Sun, J.. "An Explaratory Study of Web Services on the Internet". Proceedings of the IEEE International Conference on web Services (ICWS). 380-387. 2007

[16] Liang, Q.A., Lam, H.. "Web service matching by ontology instance categorization". IEEE International Conference on Services Computing (SCC). 2008

[17] Liu, W., Wong, W.. "Web service clustering using text mining techniques". International Journal of Agent-Oriented Software Engineering. Vol. 3, No. 1, pp. 6-26. 2009

[18] Luo, J., Montrose, B.E., Kim, A., Khashnobish, A., Kang, M.H.. "Adding OWL-S support to the existing UDDI infrastructure". IEEE International Conference on Web Services (ICWS). 2006

[19] Munkres, J.. "Algorithms for the Assignment and Transportation Problems". Journal of the Society for Industrial and Applied Mathematics. 5(1):32-38. 1957.

[20] Nayak, R., Lee, B.. "Web service discovery with additional semantics and clustering". IEEE / WIC/ ACM International Conference on Web Intelligence (WI). 2007

[21] Paolucci, M., Kawamura, T., Payne, T.R., Sycara, K.P.. "Importing the semantic web in UDDI". Revised Papers for International Workshop on Web Services, EBusiness, and the Semantic Web (WES 2002), in conj. with CAiSE 2002, LNCS 2512. 2002

[22] Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.. "Service-oriented computing: State of the art and research challenges". IEEE Computer 40(11). 2007

[23] Sivashanmugam, K., Verma, K., Sheth, A.P., Miller, J.A.. "Adding semantics to Web services standards". IEEE International Conference on Web Services (ICWS). 2003

[24] W3C. "XML Schema Part 2: Datatypes Second Edition". W3C, Technical Report. October 2004. Available at HTTP://WWW.W3.ORG/TR/XMLSCHEMA-2/.

[25] Wu, J., Chen, L., Xie, Y., Zheng, Z.. "Titan: a System for Effective Web Service Discovery". 21st International Conference on World Wide Web (WWW). 2012

APPENDIX

APPENDIX

```
List<FA> pruneHier(H h, List<FA> level, double retNum, double disThres) {
   if(level is empty) return h.toList();          // termination condition
   else {
      List<FA> nextLevel = new List<FA>();
      foreach fa in level {
         List<FA> children = fa.children();
         double distance = fa.distance();
         if(children.size() == 1) {
            if(retNum == 0)   b1();
            else {
               if(distance < disThres)   b1();
               else {
                  retNum --;
                  b2();
                  nextLevel.addAll(children);
               }
            }
         }
         else if(children.size() == 2) {
            if(retNum == 0)   c1();
            else {
               if(distance < disThres)   c1();
               else {
                  retNum --;
                  c2();
                  nextLevel.addAll(children);
               }
            }
         }
      }
      pruneHier(h, nextLevel, retNum, disThres);      //recursive call
   }
}
```

Figure A.1 A detailed version of our algorithm for pruning an abstractions hierarchy