THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE
ET DE L'UNIVERSITÉ DE SFAX

Délivré par *l'Université Toulouse III - Paul Sabatier*
et la Faculté des Sciences Économiques et de Gestion - Sfax

Discipline : Informatique

Présentée et soutenue par

Sofien KHEMAKHEM

Le Vendredi 08 Juillet 2011

Un environnement de spécification et de découverte pour la réutilisation des composants logiciels dans le développement des logiciels distribués

---

JURY

Président

M. Rafik BOUAZIZ — Professeur à la FSEG, Sfax, Tunisie

Rapporteurs

M. Djamal BEN SLIMANE — Professeur des Universités IUT Lyon 1

M. Faiez GAEGOURI — Professeur à l'ISIM, Sfax, Tunisie

Examinateur

Me. Michelle SIBILLA — Professeur à l'Université Paul Sabatier, Toulouse III

Directeurs de thèse

M. Khalil DRIRA — Directeur de Recherche au LAAS-CNRS

M. Mohamed JMAIEL — Professeur à l'ENIS, Sfax, Tunisie

Laboratoire d'Architecture et d'Analyse des Systèmes
***
LAAS-CNRS

Unité de Recherche en Développement et Contrôle d'Applications Distribuées
***
ReDCAD

Abstract

Components are developed as important and big autonomous and customizable software units. The successfulness of the reuse is important and depend on first the efficiency of the search procedure, second in the success of integration of the discovered component in system engineering.

In order to find components that best meet their functionalities and QoS requirements, the discovery process need to know both the QoS information for the components and the reliability of this information. The problem, however is that the current approaches lack both a well-defined semantics to of diverse components and the used discovery processes are inherently restricted to the exact querying. Those problems may provide an incomplete answer and may lead to low precision and recall.

When we integrate the discovered component, two things should be considered. one is which solution will be integrated if we have many solutions for the same discovered component. second how resole the collision problem in the matching of different data types.

We propose a discovery ontology to describe functional and non-functional properties of software components and an integration ontology to describe its internal structure. We propose also an enhanced Search Engine for Component based software engineering(SEC++), a persistent component that acts as an intelligent search engine, which is based on the subsumption mechanism and a function that calculates the semantic distance between the query and the components descriptions. We also describe how user-specified preferences for components in terms of non-functional requirements (e.g., QoS) can be incorporated into the component discovery mechanism to generate a partially ordered list of services that meet developer-specified functional requirements.

When integrating the discovered component, our search engine SEC++ interrogates the integration ontology to choose the component solving method which adapts with the current environment. We also develop a convertor component for conversion between two different types to solve the type collision problem. We propose a shared ontology-supported components composition, which provides a novel solution if no individual component is

found.
Our results are encouraging, in fact they are a great improvement over the SEC, SEC+ and other retrieval systems.

Key words: Component discovery, QoS, Ontology, Components composition, Component integration.

To my family: Ines, Ayoub and Lina

Acknowledgements

The research that has gone into this thesis has been thoroughly enjoyable. That enjoyment is largely a result of the interaction that I have had with my supervisors, colleagues and the people who have tested and used the resulting software.
I feel very privileged to have worked with my supervisors, Khalil DRIRA and Mohamed JMAIEL. To each of them I owe a great debt of gratitude for their patience, inspiration and friendship. Mohamed JMAIEL was always there to listen and to give advice. Khalil DRIRA taught me how to ask questions and express my ideas. He showed me different ways to approach a research problem and the need to be persistent to accomplish any goal. His truly scientist intuition has made him as a constant oasis of ideas and passions in science, which exceptionally inspire and enrich my growth as a student, a researcher and a scientist want to be. I am indebted to him more than he knows. Khalil was for me like a brother, he also listen to me whenever I was a scientific or even personal problem.
The ReDCAD Laboratory have provided an excellent environment for my research. I spent many enjoyable hours with laboratory members. Without this rich environment I doubt that many of my ideas would have come to fruition. I am particularly grateful to Moez KRICHEN, Nadhmi MILEDI, Bechir ZALILA, Hatem HADJKACEM, Mohamed HADJKACEM, Mehdi Ben ABDERRAHMEN and Afef MDHAFFAR for their worthy contribution. I am also indebted to my colleagues Kais SMAOUI, Bilel GARGOURI, Maher JAOUA, Moez BELHARETH and Saber MERZOUKI. Special thanks, tribute and appreciation to all those their names do not appear here who have contributed to the successful completion of this study.
A special thanks to Professor Abdelmajid BEN HAMADOU, Professor Slimane EL-GABSI, Professor Ahmed HADJKACEM, Professor SAMIR JMAL, Professor Faiez GARGOURI and Professor Abdelfettah BELGHITH.
Thanks also to my family who have been extremely understanding and supportive of my studies. I feel very lucky to have a family that shares my enthusiasm for academic pursuits. This thesis is dedicated to my wife and children(Ayoub and Lina) who have always stood by me and dealt with all of my absence from many family occasions with a smile.

Contents

1 Introduction **1**

I State of the art **7**

2 Software component survey **9**

2.1 Introduction 9
2.2 Component based software development 9
2.3 Component description 11
2.3.1 Description levels and Description aspects 12
2.3.2 Generation techniques for component description 13
2.3.2.1 Statically generated description 13
2.3.2.2 Dynamically generated descriptions 14
2.4 Component classification in repository 17
2.4.1 Attribute-based classification 17
2.4.1.1 The basic attribute-based classification 17
2.4.1.2 The elaborated classification 19
2.4.2 Method-based classification 20
2.4.3 Classification techniques 21
2.5 Component discovery 23
2.5.1 Comparison distance and search style 23
2.5.2 The discovery techniques 24
2.5.2.1 Probabilistic techniques 25
2.5.2.2 Learning techniques 26
2.5.3 Discovery algorithm 28
2.5.3.1 Unification based discovery 29
2.5.3.2 Decision tree-based discovery 30
2.5.4 Interface type 30

2.6 Synthesis 31
2.7 Discussion 33
2.8 Conclusion 36

3 Ontology survey **39**
3.1 Introduction 39
3.2 Ontology definition 39
3.3 Langages for representing ontologies 40
3.3.1 Resource Description Framework 41
3.3.1.1 RDF Core 41
3.3.1.2 RDF Schema, RDF(S) 42
3.3.1.3 Problems in RDF(S) 44
3.3.2 Darpa Agent Markup Language 45
3.3.3 Ontology Web Langage 46
3.3.3.1 The three sublanguages of OWL 46
3.3.3.2 Problems in OWL 48
3.3.4 Ontology Web Language for Web Services 49
3.4 Survey of ontology editors 49
3.5 Conclusion 53

II Contributions **55**

4 Approaches for component discovery and integration **57**
4.1 Introduction 57
4.2 The discovery and the integration approaches 57
4.2.1 Classification of component Non-Functional properties 60
4.2.1.1 The non functional properties characteristics 60
4.2.1.2 Static/Dynamic Non-Functional Properties 64
4.2.1.3 Non-Functional Properties Domain 64
4.3 The atomic component discovery approach 65
4.3.1 The discovery ontology 65
4.3.1.1 Definition of the discovery ontology structure 67
4.3.1.2 Definition of the discovery ontology properties 68
4.3.2 The first version of the search engine: SEC 70
4.3.2.1 The Matching algorithm 70
4.3.2.2 The semantic Web toolkit: Jena 74
4.4 The composite component discovery approach 75

4.4.1 The shared ontology for composition 76
4.4.2 Shared ontology implementation 79
4.4.3 The second version of the search engine: SEC+ 79
4.5 The integration approach 82
4.5.1 The integration ontology 82
4.5.1.1 Integration type 83
4.5.1.2 Zero-updating code in the integration process 84
4.5.1.3 Composition Description Language 85
4.5.1.4 Integration as a Generic Problem solving Method . . . 85
4.5.1.5 Integration ontology construction 89
4.5.2 The third version of the search engine: SEC++ 91
4.5.3 Lifecycle of Constituent Component 93
4.6 Conclusion 93

5 Experimental evaluation of SEC+ 95
5.1 Introduction 95
5.2 Evaluation 96
5.2.1 Experiments using SEC 97
5.2.2 Experiments using SEC+ 98
5.3 Application scenarios 100
5.3.1 Mapping the discovery ontology into integration ontology 100
5.3.2 How to implement integration ontology 102
5.3.3 Mapping the discovery ontology into shared ontology 104
5.4 Conclusion 107

6 Conclusion 109

Publications 113

Bibliographie 115

List of Figures

2.1 The approach structure . . . 11
2.2 Search style . . . 24
2.3 Work structure (See table II for corresponding numbers to references) . . 35

3.1 RDF graph . . . 42
3.2 OWL Layer . . . 48
3.3 The General Process of Engaging a Web Service . . . 49

4.1 Different steps of our approach . . . 58
4.2 Discovery and integration ontologies . . . 60
4.3 RDF Graph structure in OWL-S Profil form . . . 67
4.4 The search engine SEC and its different version . . . 69
4.5 Search step . . . 73
4.6 An ontology-supported system for component composition . . . 77
4.7 Discovery and integration ontologies: The new version . . . 78
4.8 unionOf vocabulary relationship . . . 78
4.9 Functional aspect interface . . . 80
4.10 Non-Functional aspect interface . . . 81
4.11 Result interface . . . 81
4.12 Static Composition . . . 83
4.13 Dynamic Composition . . . 84
4.14 The integration result . . . 88
4.15 Production result . . . 89
4.16 Ontology construction . . . 91
4.17 Input-Output Convertor . . . 92
4.18 Output-Matching-Service . . . 93
4.19 State Transition Model of Constituent Component . . . 94

5.1 Comparison between SEC and SEC+ . . . 99

5.2 Comparison between SEC, FIM and SEC+ 100
5.3 mapping the discovery ontology individual instance into integration ontology 101
5.4 mapping the discovery ontology into shared ontology 104
5.5 mapping the discovery ontology individual instances into shared ontology 105
5.6 The scenario 106

List of Tables

2.1 comparison of the main approaches techniques and methods. 33
2.2 the corresponding numbers to references. 37

3.1 table of ontology editors 52

5.1 Queries description 97
5.2 Recall and precision of SEC - Bad query (Q8) filtered 98
5.3 Recall and precision of SEC+ - Bad query (Q8) filtered 98
5.4 The average of the Recall and the precision of SEC,FIM and SEC+ . . . 100

1

Introduction

Component-based and service-oriented software architectures are likely to become widely used technologies in the future distributed system development. Component reuse is a critical requirement for the development process of component-based and service-oriented software.

Component are developed as important and big autonomous and customizable software units. The successfulness of the reuse is important and depend first on the efficiency of the search procedure, and second on the outcome of the integration step.

Reuse is cost effective only when the developer can find and handle(ie. possibly adapts, extends and integrates) a component quickly, and when the component solves a significant problem that would be expensive to solve with software built and debugged from scratch.

Nowadays, many industrial and academic research results have been developed to solve issues for component-oriented technologies, such as component discovery, description, and component integration. Component discovery and integration, becomes a critical success factor of component-based software engineering. However, component discovery and integration are still a highly complex but critical tasks in component-oriented technologies. Several key challenges in component discovery and integration need to be addressed:

- How to facilitate the discovery of components? In real world, there are usually multiple components which offer seemingly similar features but with some variation (e.g., different component interfaces, different attributes, different quality, etc). If we cannot locate possible components with respect to a request that serve as replacements to one another, we can not execute the constituent components properly.

- How to facilitate the integration of the discovered component in composite component? When two or more heterogeneous components are composed, two problems should be considered. One is which component will be integrated if we have many components having the same target. second is the type collision in the matching of different data types. For example, type collision happens when a 'double' type output parameter of a component is matched with a 'string' type input parameter. The third thing to be considered is how to extract input parameters when a component has two or more output results. This problem does not need to be considered if all the results of the component match. However, if only some of returned output results match, a process to extract them is needed.

The search step will become an important step in the development process. The search step may fail if the explored component repositories are not appropriately structured. This step may also fail if we use only exact query. This may provide incomplete answers since queries are often overspecified and may lead to low precision and recall.

Recall is defined as the ratio of the number of correct solutions retrieved to the number of correct solutions that exist. It indicates the ability of the system to retrieve all relevant components. Ideally, recall should be high, meaning solutions should not be missed. Precision is defined as the ratio of correct solutions retrieved to the total number of results retrieved. High precision is the result of retrieving few irrelevant or invalid solutions. It indicates the ability of the system to present only relevant components Morel et Alexander (2004).

Most of the existing component discovery mechanisms Damiani *et al.* (1999), Ostertag *et al.* (1992), Vitharana *et al.* (2003) retrieve component descriptions that contain particular keywords from the user's query. In the majority of the cases, this leads to low quality of the retrieved results. The first reason for this is that query keywords might be semantically similar but syntactically different from the terms in component descriptions, e.g. 'buy' and 'purchase' (synonyms).

Another problem with keyword-based component discovery approaches is that they cannot completely capture the semantics of the user's query because they do not consider the relations between the keywords (e.g. if the query is "order food", the relation between

these keywords could indicate a need for a restaurant).

An envisioned approach to overcome these limitations is to use ontology-based component discovery. In this approach, ontologies are used for classification of the components based on their properties. This enables retrieval based on components types rather than keywords. This is our approach in this work.

Several semantic discovery approaches Penix et Alexander (1999), Rosa *et al.* (2001) use only the exact and/or synonym matching. This can decrease the reuse of software components, reduce the precision of the search engine and provide a large number of non-necessary appropriate components. Also there is a lack of support for component selection based on non functional attributes such as Quality of Service (QoS). Some approaches to incorporation non functional attributes in component discovery lack support for dealing with depend or independent domain. Also there is a of lack support for dealing with Dynamic or Static non-functional attributes.

From the point of the integration process, static non-functional properties may compose well as they tend not to change during the system execution. The dynamic non-functional properties are influenced by the execution environment, which includes computational resources.

To alleviate these problems, elaborate and implement an ontology to semantically describe the functional and the non-functional aspect of components. This description can improve the quality of the search and can enhance both the recall and the precision. In several cases, the non-functional constraints play a decisive role in the choice of the most powerful component. To improve the re-use of software a component we use an approximate comparison between the specified query and the components semantic description. This comparison is based on the semantic distance and the subsumption notion.

In order to improve the precision and the recall of the discovery process we extend in our approach the existing approaches to component discovery by semantically describing software components and incorporating non functional aspects specifications into the component description as well as into the query. Doing so, we develop a tool, which helps the developer to select the adequate component, and an ontology which contains the semantic description and non functional information of components. This tool, called SEC+ (An enhanced Search Engine for Component Based Software Development), which is implemented as a software component, can be integrated in several development environments such as Eclipse and Jbuilder. The first step in component selection is to determine a set of components which offer the requested functionality in terms of operation name, inputs,Output, Precondition and poscondition parameters. For the operation name we

calculate the sematic distance between components names and query operation, then we regard if the set of obtained candidate meet the requested functional properties of the developper (in terms of IOPE's). In general, some components will match all the requested IOPE parameters, while others will not. To distinguish between them, we categorize them based on the degree of match Paolucci *et al.* (2002), Back et Wright (1998), Li et Horrocks (2003).

The second step in the component discovery process further refines the set of candidate component based on developer specified dynamic/static and indpendant/dependant domain non-functional attributes. The set of non-functional attributes may impact the component quality offered by a component. However, different aspects of QoS might be important in different applications and different classes of components might use different sets of non-functional attributes to specify their QoS properties.

To select the adequate component which can easily integrate in the current work, SEC++ interrogate an integration ontology. The integration ontology describe the more general internal structure of each component specified in the discovery ontology. To obtain discovery ontology instances we use adapters Fensel et al. (2003) to specify mappings among the knowledge components of a PSM. The adapters are used to achieve the reusability, since they bridge the gap between the general description of a PSM and the particular domain where it is applied.

Before selecting the appropriate component, the developer can have an idea of the various methods used to resolve the component. For example if the candidate is The towers-of-Hanoi Eriksson *et al.* (1995) which is interesting as a case study of tradeoff of space and time resources with more task-specific knowledge. The towers-of-Hanoi can be used in several domains such as psychological research computer data backups WIKIPEDIA (2007). It demonstrates several possible task-level indices that can be used to select candidate problem-solving methods from library. These indices characterize different dimensions of the problem and of its potential solutions.

There is a several solution to solve this problem such as *Recursive task decomposition*, *iterative and piece-oriented methods* and *chronological-backtracking method*. Chronological backtracking is the general method that can provide solutions for several versions of the component described in the discovery ontology. We can completely avoid backtracking, and can guarantee an optimal, solution. In general however, we might have more than three pegs, and we might start or end with any state; the domain definition of a legal move might be different, too. Although the task-specific methods are more usable than in chronological backtracking with respect to alternative problem variants, they are

not reusable across different tasks.

Common factors to consider in the selection of a problem-solving method for the choosing component in the discovery step are:

1. Input and output of the component. What information is available at run time? What is the run-time output.

2. Method flexibility. Is the component likely to be modified during development and maintenance? What flexibility in terms of reconfiguration of the method for modified component required.

3. Computational and space complexity: What are the resources available in terms of time and space?

If the developer select the component which resolved with the desired method, He can use the two components: Output-Matching-Service and Input-Output-convertor. Output-Matching-Service and Input-Output-convertor are component types used in matching parameters. Output-Matching-Service is a service that extracts what it needs from component output parameters, and Input-Output-convertor is a component that converts the output parameter type of a selected component to the input parameter type of a component to be extracted.

This research contributes to the body of component composition by proposing an ontology-supported and component-oriented approach to organizational knowledge management and components composition. We introduce an integrated a shared ontology for component composition to improve the reuse and to search a composite component if there is no individual component result. We have applied the proposed shared ontology to a corporate Mathematical service application. The prototype shows that the developed system can support semantic, dynamic, and automated component composition effectively.

We conducted various experiments to evaluate the effectiveness of SEC+. Our results are encouraging, in fact both precision and recall improved significantly compared to the results obtained with other approaches

The remainder of this thesis is structured in two parts as follows:

In Part 1, and precisely in Chapter 1 we analyze related work in the areas of component description, Component discovery and component classification.Chapter 2 presents and compare first the Web ontology languages next the ontology editors. The description of

the languages will consist in a short introduction to their functionalities and their instructions. The comparison between ontology editors will be resume in a table.

In part 2,we are going to present our approaches, which are the atomic component discovery approach, composie component discovery approach and the integration ontology.

In Chapter 4, we evaluate the performance of the SEC+ by measuring the criteria Recall and Precision. Retrieval performance experiments were performed both with and without the semantic distance and the subsumption. We conclude in this chapter that SEC+ has a high Recall and precision compared to many search engines. To demonstrate the benefits of the proposed composition and integrated ontology, we have applied it to the Matrix operations components

Conclusions and future work are given in Chapter 5 to show how our contributions can be reused in the advancement of software component technology.

Part I

State of the art

2

Software component survey

2.1 Introduction

In the first part of in this Chapter, we are going to present a comparative study of classification and discovery approaches for software components. In the second part we will present respectively the ontology definition, the languages of representing ontologies and a survey of ontology editors and compare first the Web.

2.2 Component based software development

Component-based and service-oriented software architectures are likely to become widely used technologies in the future distributed system development. Component reuse is a crucial requirement for the development process of component-based and service-oriented software. Components are developed as important and big autonomous and customizable software units. The successfulness of the reuse is important and depends on the efficiency of the search procedure. The search step is essential in the development process, since the developer is generally faced with a significant number of various component types.

The search step may fail if the explored component repositories are not appropriately structured or if the required and the provided services are not correctly compared. The use of a component repository, having a well-defined structure, is crucial for the efficiency of the CBD approach. This allows the developer to easily seek and select the component which perfectly meets his/her needs.

Through this study, we analyze the key factors that are necessary for obtaining a well-organized software component repository and software components having a pertinent description for the search procedures (see figure 2.1). These factors act not only on the precision of the specified request but also on the component resulting from the search process.

For component description, two generation approaches of description are distinguished: manual generation Erdur et Dikenelli (2002) and automatic generation relying on different methods such as introspection Sessions (1998), Neil et Schildt (1998), trace assertion Whaley *et al.* (2002) and invariant detection Perkins et Ernst (2004).

The second part identifies and describes five categories of methods for representing component classification. The first is the adhoc method, called also behavioral method Podgurski et Pierce (1992), Atkinson et Duke (1995). The second is based on the semantic characteristics of the component Penix et Alexander (1999). The third uses the facet classification Damiani *et al.* (1999), Ostertag *et al.* (1992), Vitharana *et al.* (2003), Ferreira et Lucena (2001). The fourth method is based on the lattice Fischer (2000). Finally, the fifth method applies the notion of ontology Erdur et Dikenelli (2002), Meling *et al.* (2000) to describe and classify components. Different techniques are used to organize components in repository: the cluster technique Nakkrasae *et al.* (2004), the thesaurus technique Liao *et al.* (1997) and the subsumption technique Napoli (1992).

The third part addresses the component discovery techniques related to classification methods. A successful adequation between the description and the classification methods should provide a powerful discovery service. This allows the developer to easily discover the appropriate component that meets his/her needs. The most popular discovery techniques are based on: genetic algorithms Xie *et al.* (2004), neural networks Nakkrasae *et al.* (2004), symbolic learning Utgoff (1989) and probabilistic information retrieval Yunwen et Fischer (2001). These techniques use the decision tree algorithm Ruggieri (2004), Vasiliu *et al.* (2004) or unification of the component description in the comparison phase Yao et Etzkorn (2004).

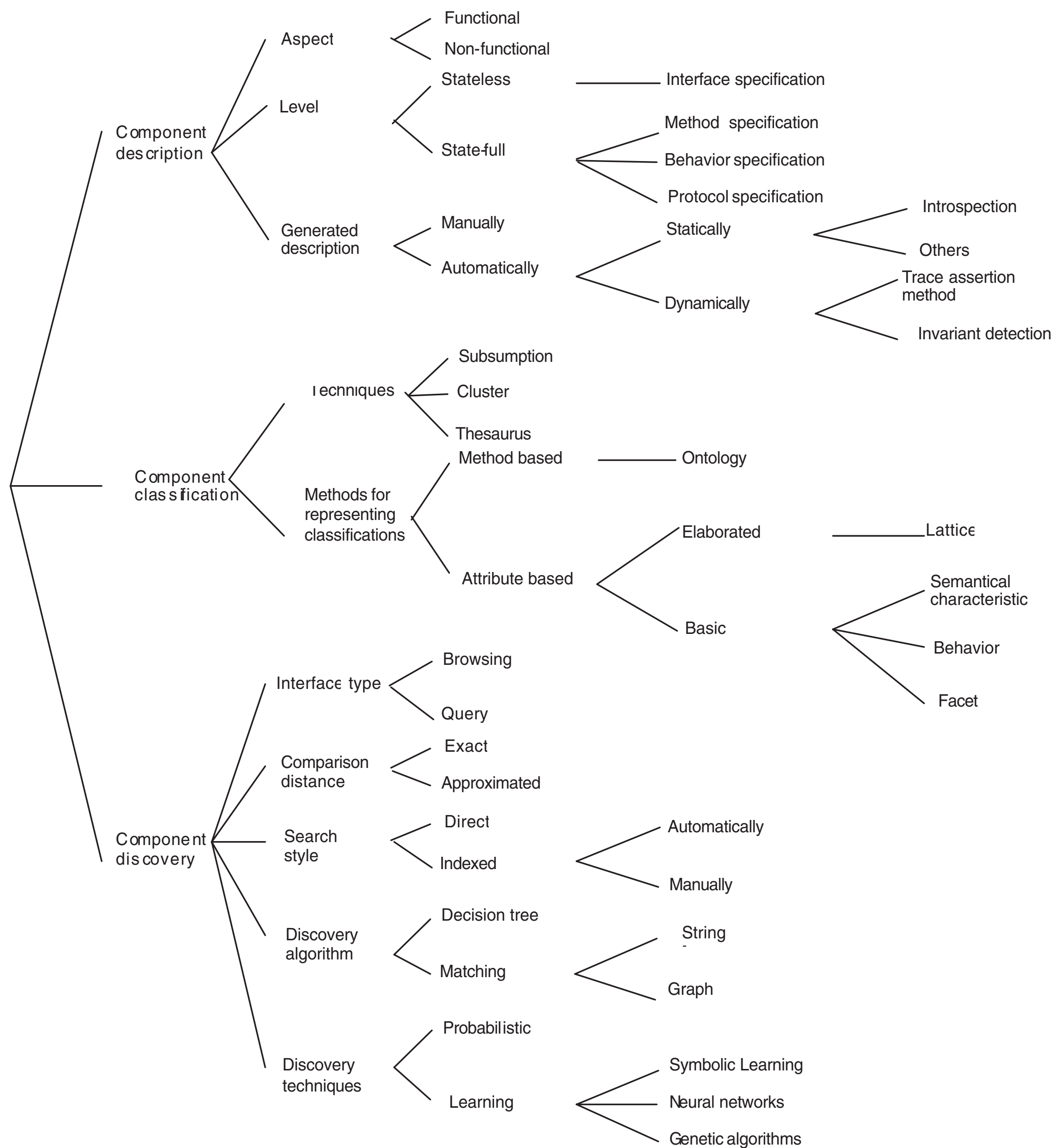

Figure 2.1: The approach structure

2.3 Component description

The description of a component constitutes an effective means which makes it possible for a user to obtain a complete and precise vision of the component.

Both functional and non-functional aspects of component description are handled by the different existing approaches. Descriptions can be generated manually or automatically and may consider two kinds of representing description levels. Two classes of component description are distinguished: Stateless and Statefull behavioral categories.

2.3.1 Description levels and Description aspects

1- Stateless: includes the service signature, its attributes, a component identification and the exceptions. Different approaches propose IDL (Interface Description Langage) as a specification language Fetike et Loos (2003) and particularly address COTS components.

2- Statefull: At this level, descriptions encompass component internal characteristics. Three sub-levels are identified:

- Methods specification: The method specification allows the developer to understand the component functionalities in detail. It describes not only the methods signature but also the method body. The majority of used languages are formal such as the Oslo University Notation (OUN) Ryl *et al.* (2001) and the LARCH notation Penix et Alexander (1999).

- Component behavior specification: it is usually defined in terms of pre-condition, post-condition of the operations, and invariants. This level of specification was described by several languages such as XML, Eiffel style in Cicalese et Rotenstreich (1999), LARCH in Zaremski et Wing (1995) and linear temporal logic (LTL) in Nakajima et Tamai (2001).

- Protocol specification: the protocol describes the component states when we execute component methods. Finite State Machines Yellin et Strom (1997), Petri Net Bastide *et al.* (1999) and π-calculus Canal *et al.* (2000) are the most often used. This level of specification is applied, not only for classification, but also for checking, substitution, composition of components Farías et Y.Guéhéneuc (2003) and analysis of compatibility between protocols Yellin et Strom. (1997).

In the specification of a software component, two different aspects are considered:

1. Functional aspect: It identifies the functionalities of the component that should provide. The methods of a component are an example of this type of information. The approach of Sofien *et al.* (2002) specifies the static part of the functional aspect through the service interface and the dynamic part through the invocation interface.

2. Non-functional aspect: It specifies the component properties. They include properties of safety, and fault tolerance as well as quality of service. The approach presented in Sun (2003) classifies the non-functional information into two categories: dynamic constraints and static constraints. This distinction is related to the degree of constraint change at the run-time in different operating system and application server.

2.3.2 Generation techniques for component description

Recent works propose tools which automatically generate specifications based on program runs Whaley *et al.* (2002), Ammons *et al.* (2002). These tools allow programmers to benefit from formal specifications with much less effort. Other works specify the component description manually. These approaches are hard to apply if there is a large number of components in the repository.

The component can be specified at design-time by developers via interfaces. Such specifications may also be generated for already implemented components. The IDL specification for object and WSDL for Web services are two examples of description which may be generated after component implementation.

Several works specify the component description manually via an interface. This description is stored as elements of databases Braga *et al.* (2001), as an XML file, as ontologies Paez et Straeten (2002) or as elements of knowledge base.

In Erdur et Dikenelli (2002), components are specified in XML and descriptions are published by local or remote repositories. Domain ontologies are used for reusable component retrieval and OQL queries are used for discovering the appropriate component.

A component specification can be also generated automatically against its implementation either dynamically, by running the component, or statically by examining the program source. Dynamic approaches are simpler to implement and are rarely blocked by inadequacies of the analysis, but they slow down the program and check only finitely many runs Ernst (2000).

2.3.2.1 Statically generated description

A component description can be generated statically by analyzing the component code. Several mechanisms are employed and supported by many tools such as Agora Seacord *et al.* (1998), PEEL Henninger (1997) and Bandera Corbett *et al.* (2000)

The static extraction of component description from implementation code was addressed by several other tools such as PEEL Henninger (1997), LCLint Evans *et al.* (1994), Bandera Corbett *et al.* (2000) and Inscape Perry (1989).

Henninger (1997) presents a re-engineering tool, called PEEL (Parse and Extract Emacs Lisp), that translates Emacs Lisp files into individual, reusable, components in a frame-based knowledge representation language named Kandor Devanbu *et al.* (1991). Kandor

representations can be viewed as a set of attribute/value slots which contain information about a given component.

In Strunk *et al.* (2005), the specification extraction is made from SPARK annotations Barnes (2003) to a PVS specification Rust (1998). A function is extracted from each sub-program in SPARK ADA. Type restrictions over input types are extracted from precondition annotations, and PVS function bodies are extracted from postcondition annotations.

Johannes and Amer Henkel et Diwan (2003) develop a tool which discovers algebraic specifications from Java classes. Algebraic specifications can describe what Java classes implement without revealing implementation details. In this approach They start by extracting the classes signatures automatically using the *Java reflection API*. They use the signatures to automatically generate a large number of terms, using heuristics to guide term generation. Each term corresponds to a legal sequence of method invocations on an instance of the class. The terms are then evaluated and compared with their outcomes. These comparisons yield equations between terms. Finally, equations are generalized to axioms and term rewriting is used to eliminate redundant axioms.

The work of Corbett *et al.* (2000) proposes an integrated collection and transformation components, called Bandera which can extract the Java code source into finite-state models. Each state represents an abstraction of the state of the program's and each transition represents the execution of one or more statements transforming this state.

The paper Evans *et al.* (1994) describes LCLint, a tool that accepts programs as input (written in ANSI C) and various levels of formal specification. It is intended to be used in developing new code and in helping to understand, to document, and to re-engineer legacy code.

Inscape Perry (1989) uses a specification language that can specify pre-conditions and post-conditions of a procedure, as well as obligations on the caller following return from the call (such as closing a returned file).

2.3.2.2 Dynamically generated descriptions

Three generation methods are distinguished: the trace assertion method, the invariant detection method and the introspection.

Trace assertion method detection techniques The trace assertion method is initially defined by D.L. Parnas Bartussek et Parnas (1978). It is a formal state machine-based method for specifying module interfaces. A module interface specification

regards a module as a black-box, identifying all module access programs, and describing their externally visible effects Janicki et Sekerinski (2001).

Traces describe the visible behavior of objects. A trace contains all events affecting the object. It is described as a sequence of events.

The trace assertion method is based on the following postulates Janicki et Sekerinski (2001):

-*Information hiding* (black box) is fundamental for any specification.

-*Sequences* are natural and powerful tools for specifying abstract objects.

-*Explicit equations* are preferable over implicit equations. Implicit equations might provide shorter and more abstract specification, but are much less readable and more difficult to derive than the explicit ones.

-*State machines* are powerful formal tools for specifying systems. For many applications they are easier to use than process algebras, and logic-based techniques.

Whaley et al Whaley *et al.* (2002) employs dynamic techniques to discover the component interfaces. It proposes using multiple FSM submodels to model the class interface. Each submodel contains a subset of methods. A state-modifying method is represented as state in the FSM, and allowable pairs of consecutive methods are represented as transitions of the FSMs. In addition, state-preserving methods are constrained to execute only under certain states.

The work of Stotts et Purtilo (1994) suggests another technique called IDTS (Interactive Derivation of Trace Specs) Parnas et Wang (1989), for deriving Parnas' trace specifications from existing code modules. The algebraic specification is also used to automatically generate a specification from modules. It can be seen as a complementary approach for the trace assertion method. The main difference between the two techniques is the use of implicit equations in algebraic specifications, and explicit equations only in trace assertions.

Invariant detection Dynamic invariant detection methods discovers specifications by learning general properties of a program execution from a set of program runs.

Invariants provide valuable documentation of a program's operation and data structures which help developers to discover the appropriate component in a given repository.

The approach presented in Ernst (2000) describes a tool which detects dynamic invariants by starting with a specific space of possible program invariants. It executes the program on a large set of test inputs, and infers likely invariants by ruling out

those which are not violated during any of the program runs. Unlike static specification, this approach has the advantage of being automatic and pervasive, but it is limited by the fixed set of invariants considered as hypothesis.

The paper of Hangal et Lam (2002) introduces DIDUCE, a tool which helps developers to specify the behavior of programs by observing its executions. DIDUCE dynamically formulates invariants hypothesis assured by the developer. It supposes the strictest invariants at the beginning, and gradually relaxes the hypothesis when violations are detected in order to allow new behavior.

Considerable research has addressed static checking of formal specifications Naumovich *et al.* (1997), Leino et Nelson (1998). This work could be used to verify likely invariants discovered dynamically. For example Jeffords and Heitmeyer Jeffords et Heitmeyer (1998) generate state invariants from requirement specifications, by finding a fixed point of equations specifying events causing mode transitions. Compared to code analyzing, this approach permits operation at a high level of abstraction and detection of errors early in the software life cycle.

Introspection Introspection is the ability of a program to look inside itself and return information for its management.

Introspection is provided for Java programs. It describes the capacity of Java components to provide information about their own interfaces. Introspection is implemented for Java components. Introspection determines the properties, the events, and the methods exported by a component. The introspection mechanism is implemented by the *java.beans.Introspector* class; it relies on both the *java.lang.reflect* reflection mechanism and a number of JavaBeans naming conventions. Introspector can determine the list of properties supported by a component, for example, if a component has a "getColor" method and a "setColor" method, the environment can assume you have a property named "Color" and take action appropriately. Bean developers can also override introspection and explicitly tell the development environment which properties are available.

The introspection mechanism does not rely on the reflection capabilities of Java alone, however any bean can define an auxiliary BeanInfo class that provides additional information about the component and its properties, its events, and its methods. The Introspector automatically attempts to locate and load the BeanInfo class of a Bean.

The introspection mechanism is used for many component models and in many approaches. For example all JViews components advertise their aspects using a set of AspectInfo class specializations, similar to BeanInfo introspection classes and

COM type libraries. The work presented in Seacord *et al.* (1998) describes a search engine for the retrieval of reusable code components. The introspection is used by Agora system and Varadarajan *et al.* (2002) respectively for registering code components, through its interface and for discovering the syntactic interface of a component at run-time.

2.4 Component classification in repository

During the development process, the developer faces handling a significant number of component types. The use of a component repository, having a clear structure, is crucial for the effectiveness of the CBD approach. This allows the developer to easily search and select the component which perfectly meets his/her needs. Several approaches tried to improve software components classification by developing methods to represent the classification of components based on their description. In existing work, two types of classification are distinguished: The attribute-based classification and the method-based classification.

2.4.1 Attribute-based classification

This classification is based on components attributes. It has two forms: an elaborated form, which uses the components attributes to make a relation between components, and a basic form, which uses the attribute types to organize the repository.

2.4.1.1 The basic attribute-based classification

The basic attribute-based classification uses the component attributes to classify components. In the basic representation we distinguish three methods: The semantical characteristic-based method, the behavior-based method and the facet-based method.

The behavior-based method This classification method is based on the exploitation of results provided by the execution of the component. These results are collections of answers which represent the dynamic behavior of the component. A relation of a behavioral nature must be used to classify software components.

In Pozewaunig et Mittermeir (2000) this technique is applied to functions. To facilitate the search process, the repository is divided into segments. Each segment contains all the functions having the same types of input and output parameters. For example, a segment contains all the functions having an input and output parameter of type integer. The developer request is presented in the form of a program which calls systematically each function of the concerned segment and collects the output of each function to compare it with the required value. Only the functions which check the value indicated in the request are provided.

In Podgurski et Pierce (1992); Atkinson et Duke (1995) components are identified by classes. The behavior is defined as the response of the objects to sequences of external messages. The comparison is made between the expected and the provided results. In Atkinson et Duke (1995), the selected behavior may come from a class or from a union of two classes.

The facet-based method Facet classification approaches Damiani *et al.* (1999); Vitharana *et al.* (2003) represent the type of information to describe software components. Each facet has a name which identifies it and a collection of well-controlled terms known as vocabulary to describe its aspects. For example, the facet *component-type* can have the following values: COM, ActiveX, Javabean, etc. In the search procedure, the user query is specified by selecting a term for each facet. The set of the selected terms represents the task to be executed by the component.

Ferreira et Lucena (2001) uses the component external description to organize the repository. Different facets are defined, among which: the applicability, the specialization domain and the hardware platform. This approach handles several technologies of components: EJB and CORBA components. Zhang *et al.* (2000) distinguishes three granularity levels for a component in a Metacase environment:

- Project level component: like projects for developing information systems.
- Graph level component: like use case diagrams.
- Unit level component: like class, state and transition diagrams.

A facet formed by a n-uplet is designed for each type of component. A hierarchical relation between the three types of facets is considered. The component description is limited to the type of the component, its properties and the name of its superior.

The approach presented in Franch *et al.* (1999) is the only one which introduces non-functional constraints. The components are ADA packages. Each facet includes the name of the non-functional constraints, a list of values and constraints

called Nfconstraints. An interface can have several implementations (components): the interface which minimizes the number of connections between components can be implemented using several methods like hashing, AVL trees, etc. The comparison distance is approximate since the developer chooses the component to which he/she applies the correction necessary to adapt it to his/her needs. In this approach, there is no specification phase since the facets are introduced in the implementation level as ADA package.

The semantical characteristic-based method The component semantic characteristic is represented by a pair (attribute, value). It represents the functional aspects of software components. The identification of these characteristics and the classification procedure are fixed and verified by an expert of the domain. The similarity between two components is measured based on the number of common characteristics. The search process is based on a syntactic comparison of the set of characteristics.

In Penix et Alexander (1999) the retrieval is achieved using feature-based classification scheme. When applying feature-based classification by hand, repository components are assigned a set of features by a domain expert. To retrieve a set of potentially useful components, the designer classifies the problem requirements into sets of features and the corresponding class of components is retrieved from repositories. Queries can be generalized by relaxing how the feature sets are compared.

2.4.1.2 The elaborated classification

The elaborated classification uses the component properties (attributes and/or methods) to form a relation. This relation can have a graph representation or a hierarchical form and is restricted by constraints. We divide the elaborated classification into attribute-based classification and method-based classification.

This type of representation is essentially used in the lattice method. The latter uses component attributes and establishes relations between them. The concept of lattice was initially defined by R. Wille Wille (1982). This concept is the representation of a relation, R, between a collection of objects G (Gegentande) and a collection of attributes M (Merkmale). The triplet (G, M, R) is called concept. The artificial intelligence is the first discipline which uses this technique for representation and acquisition of knowledge. Wille Wille (1982) considers each element of lattice as a formal concept and the graph (Hasse diagram) as a relation of generalisation/specialisation. The lattice is seen as a hierarchy of concepts. Each concept is seen as a pair (E, I) where E is a sub-set of the application

instance and I is the intention representing the properties shared by the instances.

Granter and Wille Granter et wille (1996), and Davet and Priesly Davey et Priesly (1990) apply the technique of lattice to establish the relation between objects and their attributes. This idea was applied by Fischer (2000) and Davey et Priesly (1990) to classify software components. The relation R is represented with a tree whose leaves are the components and the nodes are the joint attributes. In the search phase, the user chooses one or more attributes, according to his/her needs. The system notifies the associated components.

2.4.2 Method-based classification

This classification is handled using ontologies. For each component method this approach defines its relation with its synonyms, its Hyperonymes and its Hyponymes.

Ontology is defined by Gruber as an explicit specification of a conceptualization or a formal explicit description of concept(denoting sometimes a class) in a speech domain Natalya et Deborah (2001). The properties of each concept describe the characteristics and the attributes, also called slots or roles. The restrictions apply to the slots and are called facets. The objects of classes constitute the knowledge base. Several disciplines developed and standardized their own ontology with a well-structured vocabulary as in e-commerce Fensel *et al.* (2001) and in medicine Humphreys et Lindberg (1993).

In software engineering and particularly in the specification and the search-related domains for software components, ontology is also used. This type of description can facilitate organization, browsing, parametric search, and in general provides, more intelligent access to components.

Braga *et al.* (2001) uses ODL notations as a tool for the component external specification. Term, ontology term and component are among the used concepts. Term contains the slots names and descriptions. For each term, it defines its relation with its synonyms, its Hyperonymes and its Hyponymes in the concept ontology term. In the class component, a slot called type is defined. The comparison distance in this approach is exact.

The software components organization in Paez et Straeten (2002) is based on a multidimensional classification. A dimension is defined by a set of facets. Each facet describes an aspect of the component. The dimension implementation, for example, contains the following facets: programming language, the execution platform, etc. In dimension reuse, the facets are: the history of the use of the component, protocol, environment and components frequently used by the component. Another dimension like ScrabbleGU,

contains facets in which are defined the signatures of the methods. The notation used for the specification is Q-SHIQ.

2.4.3 Classification techniques

Classifying reusable components and easily retrieving them from existing repositories are among objectives of reuse systems design Mili *et al.* (1995). In literature, we distinguish two classification levels. The lower level hierarchy and the higher level hierarchy. The first is created by a subsumption test algorithm Napoli (1992) that determines whether one component is more general than another; this level facilitates the application of logical reasoning techniques for a fine-grained, exact determination of reusable candidates. The higher level hierarchy provides a coarse-grained determination of reusable candidates and is constructed by applying the clustering approach to the most general components from the lower-level hierarchy.

Classification by clustering techniques has been used in many areas of research, including image processing and information retrieval. Applying a clustering algorithm to the most general components of the lower-level hierarchy leads to the generation of the higher-level hierarchy of the component library.

Many methods are employed to classify the components by clustering. Such methods include fuzzy subtractive clustering algorithm Chiu (1996), neural network techniques, decision tree algorithm and fusion algorithm.

The work of Nakkrasae *et al.* (2004) employs Fuzzy Subtractive Clustering (FSC) which is a fast one-pass algorithm for estimating the number of clusters and their centers in a set of data to preprocess the software components. Once the software component groups are formed, classification process can proceed in order to build a repository containing cluster groups of similar components. The center of each cluster will be used to construct the coarse grain classification indexing structure. Three levels of component description are used: behavior, method and protocol specification. An approximate comparison query and components is employed.

In similar domain, where components are used to implement documents, Zhang et al. proposes a fusion algorithm Jian Zhang et Wang (2001) which clusters the components in different result sets. Clusters that have high overlap with clusters in other result sets are judged to be more relevant. The components that belong to such clusters are assigned the highest score. The new score is used to combine all the result sets into a single set.

A heuristical approach is used by Willet (1988), Carpineto et Romano (2000) and Daudjee et Toptsis (1994) to cluster the set of components. In Willet (1988) components are used to implement documents and heuristical decisions are used not only to cluster the component set but also to compute a similarity between individual component clusters and a query. As a result, hierarchical clustering-based ranking may easily fail to discriminate between documents that have manifestly different degrees of relevance for a certain query. Carpineto et al. applies in Carpineto et Romano (2000) the same approach to a web page. Daudjee et Toptsis (1994) uses heuristical clustering scheme. The scheme clusters software components also contains functional descriptions of software modules. It is automatic and classifies components that have been represented using a knowledge representation-based language. The facet method is used for representing the classification. This representation takes the form of verb-noun pairs where the verb is the action or operation performed and the noun is the object upon which the operation is performed

The work of Pozewaunig et Mittermeir (2000) adopts decision trees to classify and to cluster the repository into partitions with respect to the signatures of all reusable components. In the traditional approach, a partition contains assets which conform with the signature only. However, to allow a higher level of recall, this approach uses generalized signatures by extending the approaches of Novak (1997). The component description is limited to the specification of component methods.

The thesaurus is also used to organize the components into a repository. It provides knowledge about the relationships between index terms; it adds conceptual meaning to simple keyword matching. Similarity between the query posed by the user and the candidate searched for is computed by a model in which similarity between facets gives a measure of conceptual closeness (or distance). After computing the conceptual distances, the result is multiplied with facet weight (which is user-assigned).

Liao *et al.* (1997) develop a Software Reuse Framework (SRF) which is based on a built-in hierarchical thesaurus. Its classification process may be made semi-automatic. SRF is a domain-independent framework that can be adapted to various repositories and also provides four search levels to assist users with different levels of familiarity with repositories.

Llorens et al. Llorens *et al.* (1996) implements "Software Thesaurus" (ST), a tool whose objective is to develop software while reusing objects produced previously in other software projects. This tool is defined by a new repository metamodel which supports the classification and retrieval of essential software objects defined by current object oriented methodologies using GUI.

In other similar works Carpineto et Romano (1996), Carpineto et Romano (1994), the thesaurus is integrated into a concept lattice either by explicitly expanding the original context with the implied terms or by taking into account the thesaurus ordering relation during the construction of the lattice.

2.5 Component discovery

To improve component discovery, we must well classify the component repository as mentioned in previous section. This classification facilitate the discovery process and decrease the search time.

In this section, we study the component discovery related works that include: comparison distance, search style, discovery techniques, interface type and discovery algorithm.

The comparison distance between the specified query and the component description can be approximate or exact. We distinguish also two kinds of search: directed search and indexed search. We divide the discovery techniques into probabilistic and the learning techniques and we show that the majority of discovery algorithms are based on the decision tree and the unification of component descriptions.

2.5.1 Comparison distance and search style

The search procedure of software components is a delicate task especially when it handles a repository containing a significant number of software components. Indeed, the search procedure explores the structure of the repository to discover the seeked components. In literature, we distinguish two kinds of search (figure 1): directed search and indexed search. In the direct search, the developer negotiates directly with the component repository. In the indexed technique, the search process is conducted manually Fischer (2000) or automatically Seacord *et al.* (1998) according to a pre-defined process. CodeFinder Henninger (1997) and CodeBroker Yunwen et Fischer (2001) use automatic indexing. In CodeFinder, the indices are terms and sentences, whereas in CodeBroker, the indices are comments. The access to the repository is automatically managed by an agent.

The indexed search style is the mostly used in many discovery algorithms such as Decision tree algorithm. In this algorithm the repository is indexed by a decision tree, which is a tree data structure consisting of decision nodes and leaves. A leaf contains a class value and the node specifies a test over one of the attributes.

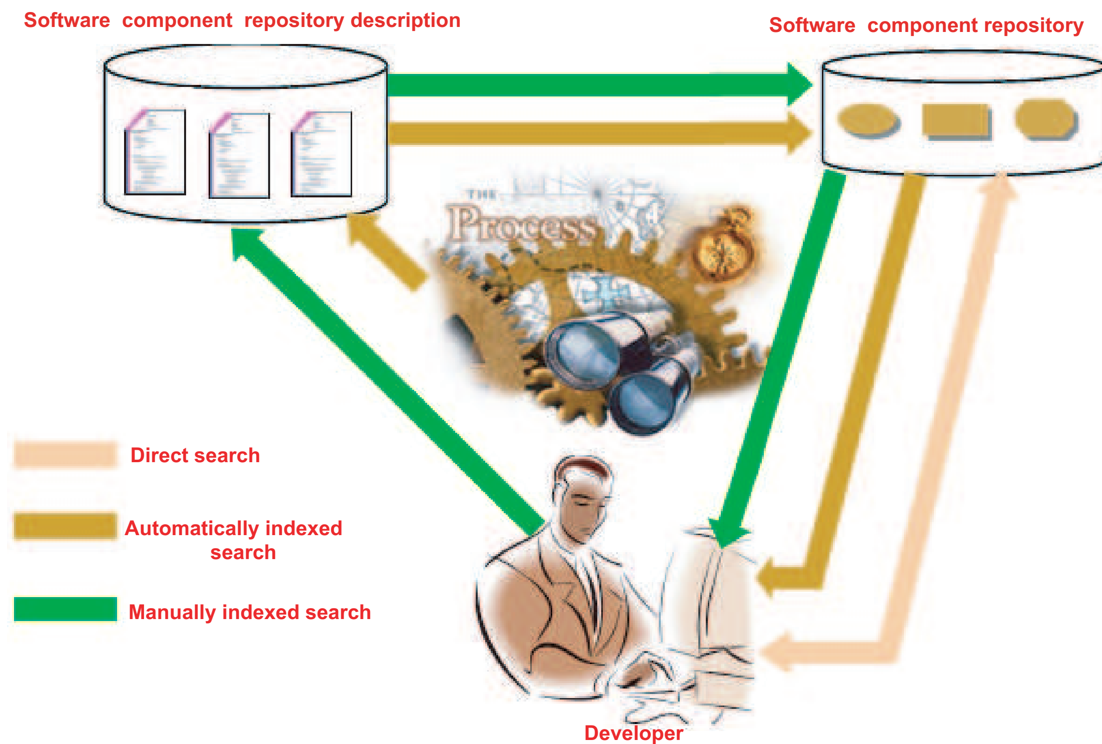

Figure 2.2: Search style

The definition and the use of a comparison distance make it possible to quantify the result of the comparison between the query requirements (Q) and the component properties. This distance is represented by a Vector Space Models in Li (1998), several evaluation functions in Cheng et Jeng (1997) and a probabilistic calculation in Yunwen et Fischer (2001). Hence, search can provide an "exact" ($P = 1$) or approximate ($P < 1$) result, where P is the probabilistic calculation . In the first case, the developer can re-use the component as such in the application. In the second case, the developer has to adapt the component to the task specified in the query.

The approximate comparison is used in many discovery techniques such as probabilistic and learning techniques.

2.5.2 The discovery techniques

Existing software repositories that provide search facilities adopt different retrieval methods. Based on a variety of technologies, they can be divided into probabilistic and learning techniques.

2.5.2.1 Probabilistic techniques

In probabilistic techniques, components indexation and selection can be seen as information retrieval problems. The goal is to estimate the relevance probability of a given component description to a user with respect to a given query. Probabilistic assumptions about the distribution of elements in the representations within relevant and irrelevant documents are required.

The CodeBroker agent Yunwen et Fischer (2001) uses both free-text information retrieval techniques and signature matching to retrieve task relevant components. It uses the probability-based information retrieval technique defined in Robertson et Walker (1994), in order to compute the concept similarity between queries extracted from doc comments of emacs programs and documents of components in the repository.

In Callan *et al.* (1992) the probabilistic retrieval model is a type of Bayesian network. They consist of two component networks; the first for documents and the second for queries. The links in the documents networks are weighed by conditional probabilities defining the probability that the document is related to the concept. Queries are related to different concepts by the user interface. Document selection is achieved using recursive inference to propagate belief values through the inference net, and then retrieving documents with the highes rank.

In Sofien *et al.* (2006) a persistent component, called SEC it developed. It can be loaded in development environments during project creation. It contains the search process and manages access to the repository of component descriptions. It executes the specified query, retrieves and presents components using a probabilistic technique. In addition, it sorts the resulted components according to the degree of similarity with the query. Four degrees of similarity have been considered:

- **Exact**: If component C and query Q are

equivalent concepts, this is the Exact match ; denoted, $C \equiv R$.

It means that for each couple of the request and the description, there is identity of types.

- **PlugIn**: If query Q is sub-concept of component C, this is the PlugIn match ; denoted, $Q \sqsubseteq C$.

It means that for each element of the query there is a similar element in component description

- **Subsume**: If query Q is super-concept of component C, this is the Subsume match ;

denoted, $C \sqsubseteq Q$.

- **Disjoint**: The last case is the Disjoint match; for which, $C \sqcap Q \sqsubseteq \perp$.

It means that there is no element of the component description that corresponds to an element of the query.

Similarly Fuhr and Pfeifer use in Fuhr et Pfeifer (1994) a probabilistic technique based on three concepts: abstraction, reductive learning and probabilistic assumptions for information retrieval. The three concepts may relate to: documents, queries, and terms.

2.5.2.2 Learning techniques

More recently, information science researchers presented new artificial-intelligence based inductive learning techniques to extract knowledge or identify patterns in examples or data. They include neural networks, genetic algorithms and symbolic learning. We provide below an overview of these three classes of techniques, along with a representative technique for each one.

The neural network is used for structuring a repository of reusable component according to the semantical similarities of the stored software components in order to facilitate the search and to optimize the retrieval of similar repetitive queries. Neural networks are considered as content-addressable or associative memories in some approach in support of imprecise querying.

The work of Clifton et Wen-Syan (1995) can be considered as instances of information retrieval methods. In this approach, conventional abstractions are used to describe software. Clifton and Li use design information as abstraction and propose neural network technology to accomplish the match.

The approach of Eichmann et Srinivas (1992) uses neural network to extend and to improve the traditional methods where the query contains exact information about the component in the repository. The motivations behind using neural networks are to use relaxation, retrieving component based on approximate/best matches, to optimize the retrieval of similar repetitive queries and to retrieve component from large repository, using the fast associative techniques that are natural and inherent in this tools.

Zhiyuan (2000) proposes a neural associative memory and bayesian inference technology to locate components in a repository. For each component, there are ten

facets (type, domain, local identifier, etc.). The neural associative memory memorizes the relationship between components and facet values. During the search, the described component representation is mapped to facets. The value of each facet is fed into its dedicated associative memory to recall the components that have the same value for this facet. After one processing step, all the components having this value will be recalled. In this approach, the comparison distance is exact and the information type is functional.

Nakkrasae *et al.* (2004) proposes two computational approaches to classify software components for effective archival and retrieval purposes, namely, fuzzy subtractive clustering algorithm and neural network technique. This approach uses a formal specification to describe three properties of components: structural, functional, and behavioral properties. Components specification are represented in a matrix form to support classification in the component repository. Subsequent retrieval of the desired component uses the same matrix to search the appropriate matching. The specification level in this approach is behavioral, the information type is functional and the comparison distance is approximate.

Genetic algorithms are based on the principle of genetics Michalewicz (1992). In such algorithms a population of individuals (a component repository) undergoes a sequence of unary (mutation) and higher order (crossover) transformations. These individuals strive for survival: a selection (reproduction) scheme, biased towards selecting fitter individuals, produces the individuals for the next generation. After a number of generations, the program converges - the best individual represents the optimum solution Chen (1995). In our case the individual represents the component and the best individual is the desired one.

The approach Xie *et al.* (2004) uses facet presentation to model query and component. Genetic algorithm, which is based on facet weight self-learning algorithm can modify dynamically the weight of the facet in order to improve retrieval accuracy. This algorithm is integrated into FWRM's that contains three main implementation parts: Facet-Weight optimization system, component retrieve system and resource.

In Chen et Kim (1995), Chen and Kim developed a hybrid system, called GANNET for information retrieval. The system performs concept optimization for user-selected documents using genetic algorithms. They use the optimized concepts to perform concept exploration in a large network of related concepts through the Hopfield net parallel relaxation procedure.

symbolic machine learning In symbolic machine learning, knowledge is represented in the form of symbolic descriptions of the learned concepts, e.g., production rules or

concept hierarchies. It is used essentially for information retrieval. The problem of component retrieval can be converted into information retrieval, the information represents the component description.

In literature, several symbolic learning algorithms have been developed. Quinlan's ID3 decision tree building algorithm and its descendants Quinlan (1986) are popular algorithms for inductive learning. ID3 takes objects of a known class, specified in terms of properties or attributes, and produces a decision tree containing these attributes that correctly classifies all the given objects. To minimize the number of tests, necessary to classify an object, it uses an information-economics approach. Its output can be summarized in terms of IF-THEN rules.

In Hsinchun et Linlin (1994), Hsinchun and Linlin adopted ID3 and the incremental ID5R Utgoff (1989) algorithm for information retrieval. Both algorithms were able to use user-supplied samples of desired documents to construct decision trees of important keywords which could represent the user queries.

For large-scale real-life applications, neural networks and, to some extent, genetic algorithms have some limitations. In fact, they suffer from requiring extensive computation time and lack of interpretable results. Symbolic learning, on the other hand, efficiently produces simple production rules or decision tree representations. The effects of the representations on the cognition of searchers in the real-life retrieval environments (e.g., users' acceptance of the analytical results provided by an intelligent system) remain to be determined Chen (1995). The importance of sample size has been stressed heavily in the probabilistic techniques Fuhr et Pfeifer (1994).

2.5.3 Discovery algorithm

Well organized repositories can be queried by developers according to a search process. To perform process and to deliver the component that meets the developer's need many algorithms have been proposed. Most of them are based on decision trees and unification of component descriptions. We distinguish two forms of unification: string unification and graph unification. This unification make easy the selection of the appropriate component by using one of the discovery techniques mentioned above.

2.5.3.1 Unification based discovery

String unification can be used to order components and hence to organize repositories hierarchically. Theses hierarchies can then be exploited to optimize the search process or to compute a navigation structure. The unification in Cheng et Jeng (1997) is a unification of logic expressions. It uses the order-sorted predicate logic (OSPL) to specify components. The relationship between two components is based on the sort information and a logical subsumption test applied to the specification body. The search process assesses the equivalence class for each of the predicates and functions and develops a unified hierarchy of components.

The discovery algorithm based on graph unification consists in transforming the query and the component specification into graph representation. After this step a discovery technique is used to compare between the resulted graphs.

AIRS (AI-based Reuse System) Ostertag *et al.* (1992) represents a component using a set of (feature; term) pairs. Each feature has a feature graph that the system traverses in search of conceptually close features with respect to the user query. This represents the distance (and thus the user effort) required to modify the retrieved candidate to meet the user's needs. The number of features used to represent all components is fixed.

Manuel *et al.* (2000) use conceptual graphs for the representation of the component(document) and the query. A conceptual graph is a network of concepts and relation nodes. The concept nodes represent entities, attributes, or events (actions). The relation nodes identify the kind of relationship between two concept nodes. The retrieval mechanism consists in comparing two conceptual graph representations. It is composed of two main parts:

1. Find the intersection of the two (sets of) graphs,

2. Measure the similarity between the two (sets of) graphs

The work of Yao et Etzkorn (2004) uses conceptual graphs to describe a component. In the retrieval process the query is translated into a conceptual graph in order to enhance both retrieval precision and recall by deploying the same representation technique on both sides: user query side and component side.

2.5.3.2 Decision tree-based discovery

A decision tree is a tree data structure consisting of decision nodes and leaves. A leaf contains a class value. A decision node specifies a test over one of the attributes, which is called the attribute selected at the node. For each possible outcome of the test, a child node is present Ruggieri (2004). In particular, the test on a discrete attribute *A* has h possible outcomes $A = d1, \ldots, dh,$ where $d1, \ldots, dh$ are the known values for attribute *A*.

The literature contains several decision tree algorithms. The survey Lim *et al.* (2000) compares twenty-two decision tree algorithms, nine classical and modern statical algorithms, and two neural networks algorithms. These algorithms are compared with respect to the classification accuracy, the training time, and the number of leaves.

In software engineering several approaches use the decision tree to classify and discover web services Vasiliu *et al.* (2004), Chirala (2004), software components Fox *et al.* (1998) and objects Olaru et Wehenkel (2003).

2.5.4 Interface type

As a supporting tool for reusable component selection, a reuse repository system has three constituents: a component repository, a discovery process, and an interface for software developers to interact with. Most of repositories have a conversational interface which is implemented either as in command line interpreter or as in graphical user interface (GUI). To find a reusable component, developers either type command lines or use direct manipulation to search or browse component repositories.

The Agora system is a web-based search approach that searches only on component interfaces, covering solely the component connectiveness problem. Agora query interface supports basic operators, + and - , as well as advanced search capabilities with boolean operators. Users can search for and retrieve components through a web interface.

In Mori *et al.* (2001) the user issues a search request with a requirement specification through a web browser. Then the trader passes this information to the inference engine (called PigNose). PigNose responds with a list of views if signature match is successful. The trader receives the result and displays it on the user's web browser.

In Ferreira et Lucena (2001), Ferreira and Lucena propose a GUI for component selection. The selection is based on the desired application domain name and its respective specialization, the automation task to be fulfilled, and the position of the desired functionality in

the automation hierarchy.

As defined in Group (2006), "browse" means reading superficially or at random. It consists in inspecting candidate components for possible extraction, but without a predefined criterion.

In general, people who search an information prefer browsing to searching because they do not need to commit resources at first and can incrementally develop their requirements after evaluating the information along the way Thompson et Croft (1989). Mili et et al. (1999) claim that browsing is the predominant pattern of component repository usage because many software developers often cannot clearly formulate queries.

However, browsing is not scalable; for large repositories, following the right link in a browsing interface requires developers to have a good understanding of the whole system, which is hard for less experienced developers.

The work in Pozewaunig et Mittermeir (2000) interests specifically on fine grained search. The principe is to exploit test cases as initial knowledge source for representing component functionalities. Augmented test cases (data points) are then classified using a decision tree algorithm. The resulting hierarchical indexing structure supports interactive browsing without the need for extensive user training.

Yunwen et Fischer (2001) proposes an agent called code broker that locates software components in a given component repository: context-aware browsing. Without any explicit input from software developers, this approach automatically locates and presents a list of software components that could be used in the current work.

2.6 Synthesis

In this section, we will summarize the comparison of the main approaches, techniques and methods (see table I). We will use a tabular like notation. In the first column we present the different methods of component classification representation. For each method we point out the search style in the second column, the information aspect in the third column, the comparison distance in the fourth column and the component specification level in the fifth column.

For the search styles:

C1 denotes the direct search,

C2 denotes the manually indexed search,

C3 denotes the automatically indexed search.

For the information aspects:

C4 denotes the functional aspect,

C5 denotes the non functional aspect.

For the comparison distance:

C6 denotes exact comparison,

C7 denotes approximate comparison.

For the specification level:

C8 denotes the external specification,

C9 denotes the interface specification,

C10 denotes the method specification,

C11 denotes the behavior specification,

C12 denotes the protocol specification.

Regarding the search style, the majority of approaches, within each method, use a manually indexed search. Although this method is slow, it has advantages for developers and especially for beginners. It allows them to understand the repository structure and to learn about its content. The search interfaces could provide meaningful messages to explain search and support progressive refinementShneiderman (1997).

The description of the non-functional aspects is, generally, neglected. Both functional and non-functional aspects should be considered during the specification, the design, the implementation, the maintenance and the re-use. In the phase of re-use, and if the search is based only on the functional aspects, the selected component may not satisfy the non-functional constraints of the environment. In several cases, the non-functional constraints play a decisive role in the choice of the most powerful component Rosa *et al.* (2001).

Table 2.1: comparison of the main approaches techniques and methods.

Methods	Search style			Inf. aspect		Comp. dist.		Specification level				
	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12
Behavior	•			•		•			•		•	
Semantic charac-teristic		•		•		•		•	•			
Lattice		•		•		•			•			
Ontology		•		•		•		•				
Facet		•		•		•		•	•			

The exact distance comparison is the most used to compare the component specified using a query with the discovered components. This type of comparison decreases the re-use of the software components. An approximate comparison not only makes it possible to understand the component functionalities by developers but also to adapt it to the application.

In the classification representation methods, there are few approaches that specify the software components with more than two levels. This allows users to understand many details, and to have higher probability to find the component matching exactly the desired functionalities. The specification details complicate the formulation of the research query. There is a tradeoff between the specification detail of the component and the difficulty of query formulation.

2.7 Discussion

In summary, we notice a similarity between the facet technique and the semantic characteristics technique except that classification with facets uses a fixed number of facets per domain and is more flexible. Moreover, one facet can be modified without affecting the others. The facet technique has also the following advantages:

- The maintenance of classification by facet is not complicated. It is achieved by updating the list of the facets,

- It has a high level of description,

- The list of terms for each facet provides a common standard vocabulary for the repository administrator and the user.

However, the developer can face problems at the time of the query formulation and in the classification. Contrarily to the behavior-based technique, it is difficult to specify the query and to combine the good terms to describe the task in the facet technique. This technique requires the repository structure understanding, the terms, and the significance of each facet Curtis (1989). Software components classification problems can appear when the component has many states. Component behavior depends on its current state, which multiplies the possibilities of its classification.

These problems are not presented in the ontology-based technique. The latter facilitates the fusion of the repositories having the same ontology Fensel *et al.* (2001), as well as the component insertion. This is not the case for the facet technique where the fusion of two repositories is done manually by adding component per component from one repository to another. Moreover, the ontology-based technique needs a heavy and painful process. Even if the two repositories would use the same terminologies (for example the same facets and same terms), the user must interpret each facet and each term while making the "mapping" in the concepts of the other repository.

The comparison in the behavior-based technique is done between the specified behavior and the behaviors of each component. The search procedure becomes very slow for a repository having a significant number of components.

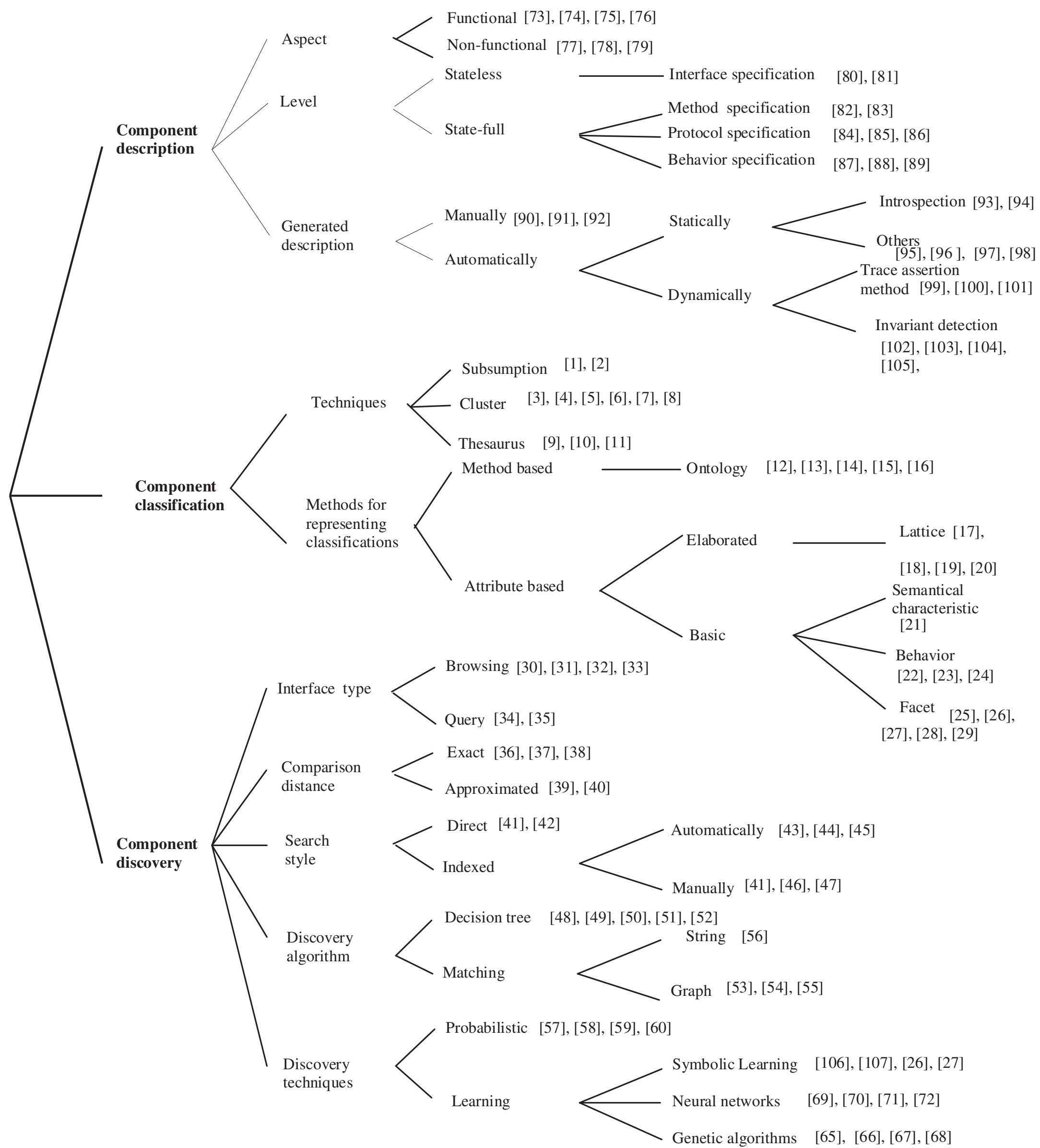

Figure 2.3: Work structure (See table II for corresponding numbers to references)

2.8 Conclusion

In this chapter Sofien *et al.* (2011), we studied different approaches which aim to improve the repository reuse. We identified three key factors that enable the repository reuse successfulness: the description, the classification and the discovery of components. A comparison between the approaches was developed. The comparison is based on search style, information type, comparison distance and specification level. We highlighted the interest of the non-functional constraints in component description, the advantage of the approximate comparison and the tradeoff to be achieved between the level of specification detail and the degree of difficulty to formulate a query.

We can conclude that to have a good search result, one must consider a tradeoff between the component specification detail and the degree of difficulties to formulate a query. It is also important to consider the non-functional aspect into component description, to use an approximate comparison and to follow a manually indexed search.

Table 2.2: the corresponding numbers to references.

Number	Reference	Number	Reference
[1]	[Napoli 1992]	[2]	[Cheng and Jeng 1997]
[3]	[Nakkrasae et al. 2004]	[4]	[Jian Zhang and Wang 2001]
[5]	[Willet 1988]	[6]	[Carpineto and Romano 2000]
[7]	[Daudjee and Toptsis 1994]	[8]	[Pozewaunig and Mittermeir 2000]
[9]	[Llorens et al. 1996]	[10]	[Carpineto and Romano 1996]
[11]	[Carpineto and Romano 1994]	[12]	[Natalya and Deborah 2001]
[13]	[Braga et al. 2001]	[14]	[Paez and Straeten 2002]
[15]	[Fensel et al. 2001]	[16]	[Humphreys and Lindberg 1993]
[17]	[Wille 1982]	[18]	[Granter and wille 1996]
[19]	[Davey and Priesly 1990]	[20]	[Fischer 2000]
[21]	[Penix and Alexander 1999]	[22]	[Pozewaunig and Mittermeir 2000]
[23]	[Podgurski and Pierce 1992]	[24]	[Atkinson and Duke 1995]
[25]	[Damiani et al. 1999]	[26]	[Vitharana et al. 2003]
[27]	[Ferreira and Lucena 2001]	[28]	[Zhang et al. 2000]
[29]	[Franch et al. 1999]	[30]	[Ferreira and Lucena 2001]
[31]	[Thompson and Croft 1989]	[32]	[Mili and et al. 1999]
[33]	[Yunwen and Fischer 2001]	[34]	[Ostertag et al. 1992]
[35]	[Manuel et al. 2000]	[36]	[D. Eichmann 1992]
[37]	[Braga et al. 2001]	[38]	[Zhiyuan 2000]
[39]	[Nakkrasae et al. 2004]	[40]	[D. Eichmann 1992]
[41]	[Fischer 2000]	[42]	[Seacord et al. 1998]
[43]	[Seacord et al. 1998]	[44]	[Yunwen and Fischer 2001]
[45]	[Henninger 1997]	[46]	[Paez and Straeten 2002]
[47]	[Braga et al. 2001]	[48]	[Ruggieri 2004]
[49]	[Vasiliu et al. 2004]	[50]	[Chirala 2004]
[51]	[Fox et al. 1998]	[52]	[Olaru and Wehenkel 2003]
[53]	[Ostertag et al. 1992]	[54]	[Manuel et al. 2000]
[55]	[Yao and Etzkorn 2004]	[56]	[Cheng and Jeng 1997]
[57]	[Yunwen and Fischer 2001]	[58]	[Robertson and Walker 1994]
[59]	[Callan et al. 1992]	[60]	[Khemakhem et al. 2006]
[61]	[Clifton and Wen-Syan 1995]	[62]	[D. Eichmann 1992]
[63]	[Zhiyuan 2000]	[64]	[Nakkrasae et al. 2004]
[65]	[Michalewicz 1992]	[66]	[Chen 1995]
[67]	[Callan et al. 1992]	[68]	[Chen and Kim 1995]
[69]	[Quinlan 1986]	[70]	[Chen and She 1994]
[71]	[Utgoff 1989]	[72]	[Fuhr and Pfeifer 1994]
[73]	[Khemakhem et al. 2002]	[74]	[Zhiyuan 2000]

Number	Reference	Number	Reference
[75]	[Nakkrasae et al. 2004]	[76]	[Daudjee and Toptsis 1994]
[77]	[Sun 2003]	[78]	[Franch et al. 1999]
[79]	[Rosa et al. 2001]	[80]	[Fetike and Loos 2003]
[81]	[Erdur and Dikenelli 2002]	[82]	[Ryl et al. 2001]
[83]	[Penix and Alexander 1999]	[84]	[Yellin and Strom 1997]
[85]	[Bastide et al. 1999]	[86]	[Canal et al. 2000]
[87]	[Cicalese and Rotenstreich 1999]	[88]	[Zaremski and Wing 1995]
[89]	[Cicalese and Rotenstreich 1999]	[90]	[Zaremski and Wing 1995]
[91]	[Paez and Straeten 2002]	[92]	[Erdur and Dikenelli 2002]
[93]	[Seacord et al. 1998]	[94]	[S.Varadarajan et al. 2002]
[95]	[Henninger 1997]	[96]	[Evans et al. 1994]
[97]	[Perry 1989]	[98]	[Corbett et al. 2000]
[99]	[Janicki and Sekerinski 2001]	[100]	[Whaley et al. 2002]
[101]	[Stotts and Purtilo 1994]	[102]	[Hangal and Lam 2002]
[103]	[Naumovich et al. 1997]	[104]	[Leino and Nelson 1998]
[105]	[Jeffords and Heitmeyer 1998]		

3
Ontology survey

3.1 Introduction

In this Chapter, we are going to present respectively the ontology definition, the languages of representing ontologies and a survey of ontology editors.

3.2 Ontology definition

The term "ontology" comes from the philosophy field which is concerned with the study of being or existence. In philosophy, one can talk about an ontology as a the nature theory of existence. In computer science, ontology is a technical term denoting an artifact that is designed for a purpose, which is to enable the modeling of knowledge about some domain, real or imagined Gruber (2008). Ontology had been adopted by early Artificial Intelligence (AI) researchers, who recognized the applicability of the work from mathematical logic and argued that AI researchers could create new ontologies as computational models that enable certain kinds of automated reasoning. In the 1980's the AI community came to use the term ontology to refer to both a theory of a modeled world and a com-

ponent of knowledge systems. Some researchers, drawing inspiration from philosophical ontologies, viewed computational ontology as a kind of applied philosophy.

In computer sciences, an ontology specifies a set of representational primitives with which to model a knowledge domain discourse. The representational primitives are: classes (or sets), attributes (or properties), and relationships (or relations among class members). The definitions of the representational primitives include information about their meaning and constraints on their logically consistent application. In the context of database systems, ontology can be viewed as a level of abstraction of data models, analogous to hierarchical and relational models, but intended for modeling knowledge about individuals, their attributes, and their relationships to other individuals. Ontologies are typically specified in languages that allow abstraction away from data structures and implementation strategies; in practice, the languages of ontologies are closer in expressive power to first-order logic than languages used to model databases. For this reason, ontologies are said to be at the "semantic" level, whereas database schema are models of data at the "logical" or "physical" level. Due to their independence from lower level data models, ontologies are used for integrating heterogeneous databases, enabling interoperability among disparate systems, and specifying interfaces to independent, knowledge-based services. In the context of the Semantic Web standards, ontologies are called out as an explicit layer. There are now standard languages and a variety of commercial and open source tools for creating and working with ontologies.

3.3 Langages for representing ontologies

Ontologies are not all built the same way. A number of possible languages can be used, including that have evolved specifically to support ontology construction. The Open Knowledge Base Connectivity (OKBC) model and languages like KIF (and its emerging successor CL – Common Logic) are examples that have become the bases of other ontology languages. There are also several languages based on a form of logic thought to be especially computable known as description logics. These include Loom and DAML+OIL, which is currently being evolved into the Web Ontology Language (OWL) standard. When comparing ontology languages, what is given up for computability and simplicity is usually language expressiveness, which isn't always a bad deal. A language need only be as rich and expressive as is necessary to represent the nuance and intricacy of knowledge that the ontology's purpose and its developers demand. The wide array of information residing on the Web has given ontology use an impetus, and ontology languages increasingly rely on W3C technologies like RDF Schema as a language layer, XML Schema for data typing,

and RDF to assert data Aranda (2005).

3.3.1 Resource Description Framework

The Resource Description Framework is a framework for representing information in the Web. RDF is developed by W3C and provides meaning to data in a machine understabdable format allowing dor more sophisticated data interchange or searching.

If we look at the W3C web page we can see this definition: "The Resource Description Framework" (RDF) integrates a variety of applications from library catalogs and world-wide directories to syndication and aggregations from library catalogs and world-wide directories to syndication and aggregation of news, software, and content to personal collections of music, photos, and events using XML as an interchange syntax. The RDF specifications provide a lightweight ontology system to support the exchange of knowledge on the Web.

RDF will allow us to put information and meaning to our data. RDF is extremely flexible for accomplishing that objective because it will allow us to put the information in one context with enough extra information that an information agent will be capable to process understand.

If RDF is a way for describing data the RDF Schema is a domain-neutral way of describing the metadata that can then be used to describe the data for a domain-specific vocabulary. RDF Schema provides the ressources necessary to describe the objects and properties of a domain-specific schema.

3.3.1.1 RDF Core

The core RDF is a set of triples consisting in RDF Subject, RDF Verb or Predicate and RDF Object. The first principal component of RDF is the subject. The subject can be seen as a name or an object. The subject is the resource being described and can be identified by an URI. The second principal component is the verb or property of the subject. The verb is a characteristic of the subject and for example, it can be color, size or another property applicable to a resource. Properties can also be multiple resources, values of properties can be other resources. The third and last component of the RDF triples is the object. This object is the value associated to this resource, for example can be red, big or another value applicable to a defined property. In every RDF triple wae can see always:

Every RDF triple is made of subject, property and object. Every triple represents one fact. Every RDF triple can be joined with other RDF triples and will not loose their initial meaning. A subject is an URI

RDF can be represented in a graph way, like in the figure /refRDF graph, a directed labeled graph and is the way that RDF Core Working Group decided as default method for describing RDF data models.

There are three different kinds of nodes in a directed graph for representing RDF data models:

URIref: node consist in Uniform Resource Identifier, that is , an identifier for the node. Can reference to data, not only to Web resources.

Blank nodes: Nodes that do not have URI

Literals: Formed by three components, a character string, an optional language tag and data type.

Figure 3.1: RDF graph

3.3.1.2 RDF Schema, RDF(S)

RDF Schema defines a simple modelling language on top of RDF. In RDF you can represent the data, with their properties but you can not represent the description of these properties or describe relationships between these properties and other resources. To solve this problem W3C specified RDF Schema. It is introduced as a layer on top of the basic RDF Model

RDF Schema is a domain-neutral way for describing metadata. This metadata can be used to describe the data for a domain specific vocabulary. RDF(S) helps us to create and define new objects and properties, With RDF(S) we will define classes and properties that may be used to describe classes, properties and other resources Manola et Miller (2004) Resources may be divided into groups called classes. The members of a class are known as instances of the classes are themselves resources. They are often identified by RDF URI References and may be described using RDF properties. The rdf:type property may

be used to state that a reource is an instance of a class.

`rdfs:Resource` All things described by RDF are called resources, and are instances of the class rdfs:Resource. This is the class of everything. All other classes are subclasses of this class. rdfs:Resource is an instance of rdfs:Class.

`rdfs:Class` This is the class of resources that are RDF classes. rdfs:Class is an instance of rdfs:Class.

`rdfs:Literal` rdfs:Literal is an instance of rdfs:Class. rdfs:Literal is a subclass of rdfs:Resource. 2.4 rdfs:Datatype

`rdfs:Datatype` This is the class of datatypes. All instances of rdfs:Datatype correspond to the RDF model of a datatype described in the RDF Concepts specification [RDF-CONCEPTS]. rdfs:Datatype is both an instance of and a subclass of rdfs:Class. Each instance of rdfs:Datatype is a subclass of rdfs:Literal.

`rdf:XMLLiteral` The class rdf:XMLLiteral is the class of XML literal values. rdf:XMLLiteral is an instance of rdfs:Datatype and a subclass of rdfs:Literal.

`rdf:Property` rdf:Property is the class of RDF properties. rdf:Property is an instance of rdfs:Class.

`rdfs:range` rdfs:range is an instance of rdf:Property that is used to state that the values of a property are instances of one or more classes.

`rdfs:domain` rdfs:domain is an instance of rdf:Property that is used to state that any resource that has a given property is an instance of one or more classes.

A triple of the form: P rdfs:domain C

`rdf:type` rdf:type is an instance of rdf:Property that is used to state that a resource is an instance of a class.

A triple of the form: R rdf:type C

states that C is an instance of rdfs:Class and R is an instance of C.

`rdfs:subClassOf` The property rdfs:subClassOf is an instance of rdf:Property that is used to state that all the instances of one class are instances of another.

A triple of the form:

C1 rdfs:subClassOf C2

states that C1 is an instance of rdfs:Class, C2 is an instance of rdfs:Class and C1 is a

subclass of C2. The rdfs:subClassOf property is transitive.

`rdfs:subPropertyOf` is an instance of rdf:Property that is used to state that all resources related by one property are also related by another.

A triple of the form: P1 rdfs:subPropertyOf P2

states that P1 is an instance of rdf:Property, P2 is an instance of rdf:Property and P1 is a subproperty of P2. The rdfs:subPropertyOf property is transitive.

`rdfs:label` This is an instance of rdf:Property that may be used to provide a human-readable version of a resource's name.

A triple of the form: R rdfs:label L

states that L is a human readable label for R.

`rdfs:comment` rdfs:comment is an instance of rdf:Property that may be used to provide a human-readable description of a resource.

A triple of the form:

R rdfs:comment L

states that L is a human readable description of R.

3.3.1.3 Problems in RDF(S)

When designing a basic ontology with RDF(S) It will make sense of possibility to create infinite layers of classes. It is possible to observe that `rdfs:Class` is a subclass of `rdfs:Resource` and `rdfs:Resource` is at the same time an instance of `rdfs:Class`. The problem comes when the next layer, the Logical layer, tries to extend the previous layer, the metamodel layer. These problems are described in Pan et Horrocks (2001) and the result is that RDF(S) has no clear semantics:

1. The class `rdfs:Class` is an instance of itself. That means that you can find the Russell's paradox. The paradox arises when considering the set of all sets that are not members of themselves. Such a set appears o be a member of itself if it is not member of itself, hence the paradox.

2. The class `rdfs:Resource` is a super class ans ibstance of rdfs:Class at the same time, which means that the superset (rdfs:Resource) is a member of the subset (rdfs:Class).

3.The properties rdfs:subClassOf, rdf:type, rdfs:range and rdfs:domain are used to define

both the other RDF(S) modeling primitives and the ontology, which makes their semantics unclear and makes very difficult to formalize RDF(S)

3.3.2 Darpa Agent Markup Language

Unlike RDF and topic maps DAML is not a data model; instead, it is a schema language that can be used to constrain and describe data following the RDF data model. To put it another way: DAML is an RDF schema language. RDF already has a schema language, called RDF Schema [RDF-Schema], and DAML is an extension of this language. Note that DAML also extends the RDF syntax, and that DAML files cannot necessarily be parsed with RDF parsers. DAML strengthens the RDF schema language, and adds a little bit of semantics on top. The semanics are mainly things topic maps already have, apart from the ability to specify that a relationship is transitive. This ability is really a poor man's inference engine, and any inference engine, for RDF or for topic maps, will provide capabilities far beyond what this property provides. " " OIL (Ontology Inference Layer) is an initiative funded by the European Union programme for Information Society Technologies as part of some of its reasearch projects. The work has been done by participants in these projects, and the resulting specification is a specification published by the reseach project. OIL is obviously a semantic web technology, and according to the OIL FAQ OIL is intended to solve the findability problem, support e-commerce, and enable knowledge management. OIL is very similar to DAML in that it, too, is an extension of RDF Schema, and the capabilities of the two languages are very similar. They are not entirely the same, however, despite the fact that the latest release of DAML is called DAML+OIL. The proponents of OIL claim that OIL has some desirable properties and capabilities that DAML does not, but these are not very relevant to the issue discussed in this paper, and will therefore not be discussed here. To compare with topic maps, there is no standardized schema language for topic maps, although one is under development ([TMCL]). As for the semantics added by DAML to RDF, topic maps already have most of these. Stating that two association types or occurrence types are the same is done by merging them in topic maps. There is no need for an inverse of relationship, since all relationships are multidirectional in topic maps. The ability to say that a relationship is transitive, however, is missing from topic maps, and would make a useful addition.

3.3.3 Ontology Web Langage

OWL Efforts toward the creation of the Semantic Web are gaining momentum. Soon it will be possible to access Web resources by content rather than just by keywords. A significant force in this movement is the development of a new generation of Web markup languages such as OWL Dean *et al.* (2002) and its predecessor DAML+OIL Braga *et al.* (2001). These languages enable the creation of ontologies for any domain and the instantiation of these ontologies in the description of specific Web sites. Among the most important Web resources are those that provide services. By “service” we mean Web sites that do not merely provide static information but allow one to effect some action or change in the world, such as the sale of a product or the control of a physical device. The Semantic Web should enable users to locate, select, employ, compose, and monitor Web-based services automatically. To make use of a Web service, a software agent needs a computer-interpretable description of the service, and the means by which it is accessed. An important goal for Semantic Web markup languages, then, is to establish a framework within which these descriptions are made and shared. Web sites should be able to employ a set of basic classes and properties for declaring and describing services, and the ontology structuring mechanisms of OWL provide the appropriate framework within which to do this.

Comparing to RDF(S) OWL adds more vocabulary for describing properties and classes: among others, relations between classes (e.g. disjointness), cardinality (e.g. "exactly one"), equality, richer typing of properties, characteristics of properties (e.g. symmetry), and enumerated classes.

3.3.3.1 The three sublanguages of OWL

OWL provides three increasingly expressive sublanguages designed for use by specific communities of implementers and users.

- OWL Lite supports those users primarily needing a classification hierarchy and simple constraints. For example, while it supports cardinality constraints, it only permits cardinality values of 0 or 1. It should be simpler to provide tool support for OWL Lite than its more expressive relatives, and OWL Lite provides a quick migration path for thesauri and other taxonomies. Owl Lite also has a lower formal complexity than OWL DL.

- OWL DL supports those users who want the maximum expressiveness while retain-

ing computational completeness (all conclusions are guaranteed to be computable) and decidability (all computations will finish in finite time). OWL DL includes all OWL language constructs, but they can be used only under certain restrictions (for example, while a class may be a subclass of many classes, a class cannot be an instance of another class). OWL DL is so named due to its correspondence with description logics, a field of research that has studied the logics that form the formal foundation of OWL.

- OWL Full is meant for users who want maximum expressiveness and the syntactic freedom of RDF with no computational guarantees. For example, in OWL Full a class can be treated simultaneously as a collection of individuals and as an individual in its own right. OWL Full allows an ontology to augment the meaning of the pre-defined (RDF or OWL) vocabulary. It is unlikely that any reasoning software will be able to support complete reasoning for every feature of OWL Full.

Each of these sublanguages is an extension of its simpler predecessor, both in what can be legally expressed and in what can be validly concluded. The following set of relations hold. Their inverses do not.

- Every legal OWL Lite ontology is a legal OWL DL ontology.
- Every legal OWL DL ontology is a legal OWL Full ontology.
- Every valid OWL Lite conclusion is a valid OWL DL conclusion.
- Every valid OWL DL conclusion is a valid OWL Full conclusion.

Ontology developers adopting OWL should consider which sublanguage best suits their needs. The choice between OWL Lite and OWL DL depends on the extent to which users require the more-expressive constructs provided by OWL DL. The choice between OWL DL and OWL Full mainly depends on the extent to which users require the meta-modeling facilities of RDF Schema (e.g. defining classes of classes, or attaching properties to classes). When using OWL Full as compared to OWL DL, reasoning support is less predictable since complete OWL Full implementations do not currently exist. OWL Full can be viewed as an extension of RDF, while OWL Lite and OWL DL can be viewed as extensions of a restricted view of RDF. Every OWL (Lite, DL, Full) document is an RDF document, and every RDF document is an OWL Full document, but only some RDF documents will be a legal OWL Lite or OWL DL document. Because of this, some care has

to be taken when a user wants to migrate an RDF document to OWL. When the expressiveness of OWL DL or OWL Lite is deemed appropriate, some precautions have to be taken to ensure that the original RDF document complies with the additional constraints imposed by OWL DL and OWL Lite. Among others, every URI that is used as a class name must be explicitly asserted to be of type owl:Class (and similarly for properties), every individual must be asserted to belong to at least one class (even if only owl:Thing), the URI's used for classes, properties and individuals must be mutually disjoint. The details of these and other constraints on OWL DL and OWL Lite are explained in appendix E of the OWL Reference.

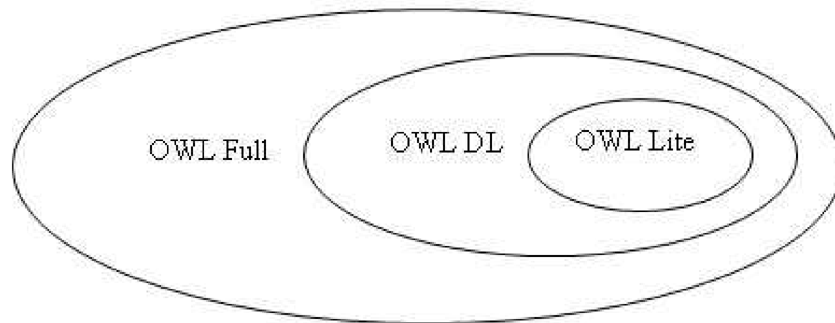

Figure 3.2: OWL Layer

3.3.3.2 Problems in OWL

One intelligent agent can reason more things in RDF(S) and then obtain more answers than OWL agent. That implies if using the other layers basis to make new layer, ontology layer extends the RDF layer. Then is possible also that some problems will be extended. Classes with the unserlying principles of RDF(S) resulting paradoxes in same syntax and extended semantics layering of OWL on top of RDF(S)Schneider et Fensel (2002).

We have seen that OWL offers many features for modelling a domain, providing classes, relationships, properties or it is also possible to apply restrictions to the elements previously created. It is possible to specify these restrictions with first order predicates that will provide more elements in order to allow the information agents automatic reasoning Smith *et al.* (2004)

3.3.4 Ontology Web Language for Web Services

OWL-S Lee *et al.* (2001) is a Web Services ontology that specifies a conceptual framework for describing semantic web services. OWL-S is also a language that enriches Web Services descriptions with semantic

information from OWL ontologies. OWL-S is characterized by three modules: (1) a Profile that describes capabilities of Web Services as well as additional features (e.g. inputs, outputs, preconditions and effects) of web services hence crucial in the web service discovery process.; (2) a Process Model that provides a description of the activity of the Web Service provider from which the Web Service requester can derive the interaction; (3) a Grounding that is a description of how abstract information exchanges described in the Process Model are mapped onto actual messages that the provider and the requester exchange.

In the figure 3.3 is possible to see the architecture of OWL-S. In this figure it is shown the main modules of the ontology for Web Services. These elements are described in the next sections.

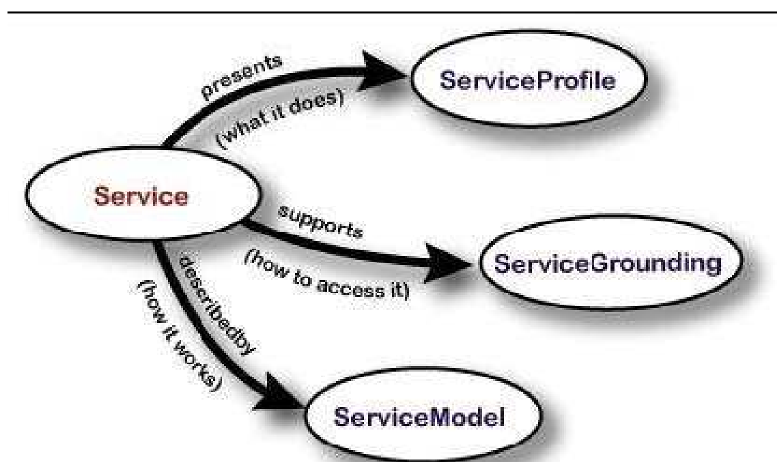

Figure 3.3: The General Process of Engaging a Web Service

3.4 Survey of ontology editors

This state of the art covers tools that have ontology editing capabilities. The software tools may be useful for modeling ontology schemas alone or together with instance data. Ontology browsers without an editing focus and other types of ontology building tools are

not included. The editing tools are not necessarily production level development tools, and some of them may offer only limited functionality and support for user. Concise descriptions of each software tool were compiled and then reviewed by the organization currently providing the software for commercial, open, or restricted distribution. The descriptions are factored into a five different categories covering important functions and features of the software. These categories appear in Table 3.1 summarizing the results.

Tool	Source	Modeling Features/Limitations	Import/Export Formats	Graph View	Information Extraction
OilEd	University of Manchester Information Management Group	DAML constraint axioms; same-class-as; limited XML Schema datatypes; creation metadata; allows arbitrary expressions as fillers and in constraint axioms; explicit use of quantifiers; one-of lists of individuals; no hierarchical property view.	RDFS; SHIQ	Browsing Graphviz files of class subsumption only	No
OntoBuilder	Institute for Medical Information, Statistics and Epidemiology University of Leipzig	Manages compilation of domain terms, their description, and contexts using natural language.	No	No	No
OntoEdit	Ontoprise GmbH	F-Logic axioms on classes and relations; algebraic properties of relations; creation of metadata; limited DAML property constraints and datatypes; no class combinations, equivalent instances.	RDFS; F-Logic; DAML+OIL (limited); RDB	Yes, via plug-in	No

Tool	Source	Modeling Features/Limitations	Import/Export Formats	Graph View	Information Extraction
Protégé-2000	Stanford Medical Informatics	Multiple inheritance concept and relation hierarchies (but single class for instance); metaclasses; instances specification support; constraint axioms ala Prolog, F-Logic, OIL and general axiom language (PAL) via plug-ins.	RDF(S); XML Schema; RDB schema via Data Genie plug-in; (DAML+OIL backend due 4Q'02 from SRI)	Browsing classes	global properties via GraphViz plug-in; nested graph views with editing via Jambalaya plug-in.
No					
Taxonomy Builder	Semansys Technologies	General taxonomy of elements assigned data types and substitution groups. Predefined XBRL relation types via links.	XML; XML Schema	No	No

Table 3.1: table of ontology editors

The ontology editor chosen for this thesis is the Protégé ontology editor and acquisition system. Protégé provides an intuitive interface for developing ontologies by supporting multiple design panes for hierarchical design, property design, restriction construction, comment and definition development, and disjoint function construction. Protégé supports a number of ontology languages, including OWL. The Protégé OWL plugin allows for a supported development of OWL ontologies through its use of the rules and syntax of the OWL language as well as support for reasoning . The ontology interface, includes OWL Classes, Properties, Forms, Individuals, and Metadata tabs. The OWL Classes tab provides the basic ontology development interface. This interface includes an Asserted Hierarchy toolbox for creating hierarchies, a Comment box to include additional descriptions of entities, Asserted Conditions hierarchy which displays the restrictions of each class, Annotations which include additional annotation development, Properties which display the properties that are defined in the Properties tab, and Disjoints toolbox which aids in defining classes as disjoint. This robust and intuitive interface provides an outstanding tool for creation of ontologies while the backend ontology language rule and syntax control mechanisms allow for easy development and checking of not only the design of an ontology, but also the syntax necessary for the ontology to communicate its knowledge with other systems.

3.5 Conclusion

In this chapter, we have seen different ontology languages. Each language has it purpose and are more suitable for solving determinate problems. RDF(S) fits better in simples cases, OWL is better to develop business ontologies. With the OWL-S extension we have seen a very powerful ontology language that offers tools not only to describe data, also to describe not only the functional aspects of software components but also the non functional aspects.

After a comparison between ontology editors we have chosen Protege2000 because it is very easy to use to develop the discovery ontology and the integration ontology. This is mainly because of its screen interface and also because it is highly configurable and you can download many plug-in from the Protege Web site.

Part II

Contributions

4

Approaches for component discovery and integration

4.1 Introduction

In this Chapter, we are going to describe our approaches for component discovery, composition and integration. We will detail in each approaches the used ontology and process. The figure 4.1 describes the steps which we use from query specification to component integration.

4.2 The discovery and the integration approaches

We describe the semantics of components to express knowledge about functional and non-functional aspects of a component. This knowledge comprises:

- The structural aspects that specify the component's internal structure. The developer uses these aspects to determine if interaction exists between component oper-

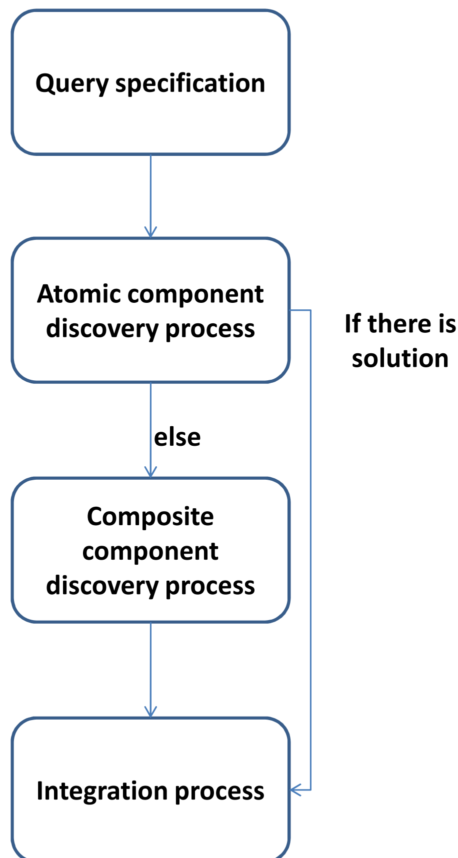

Figure 4.1: Different steps of our approach

ations and other components used to build the current project.

- The functional aspects that identify the functionalities of the component is expected to provide through many features. These features include methods that are used to adapt the behavior of the component to his context. The adaptation is made by specializing and customizing. The other kinds of features are used by the application specific part of a component-based software.

Generally this type of information is specified by the component's methods.

- The non functional aspect specifies the component constraints related to communication or computation. The non functional aspect includes features such as performance, availability, reliability, security, adaptability and dependability. We distinguish static and dynamic categories of non functional features. Static features, such as security-related constraints, do not change during component execution. Dynamic features, such as performance-related properties, depend on the deployment environment.

All these features represent different and complementary views of a component. The feature set used to describe a component, depends on the developer action: discovery and integration. The discovery of a component is made by sending a query to the repository manager. Once a set of components has been selected, additional features are specified to select a component before integration. For the discovery action, the query includes functional and/or non functional features. For integration action, the structural features have to be specified.

The underlying approach for SEC+ is based on the following ontologies Sofien *et al.* (2006):

- The discovery ontology that specifies functional and non functional features.
- The integration ontology that describes the problem solving method (PSMs) used to specify the component's structural features.

As illustrated in figure 4.2, the main information contained in constraints, interface and model are respectively the non-functional properties, the functional information (operation names, input, output, precondition and postcondition) and the internal structure of the component. We use RDF language to describe the discovery ontology. One step further, the elements in the discovery ontology link to the corresponding properties in the integration ontology for example, the interface concept in the discovery ontology corresponds to Tasks concept in the integration ontology.

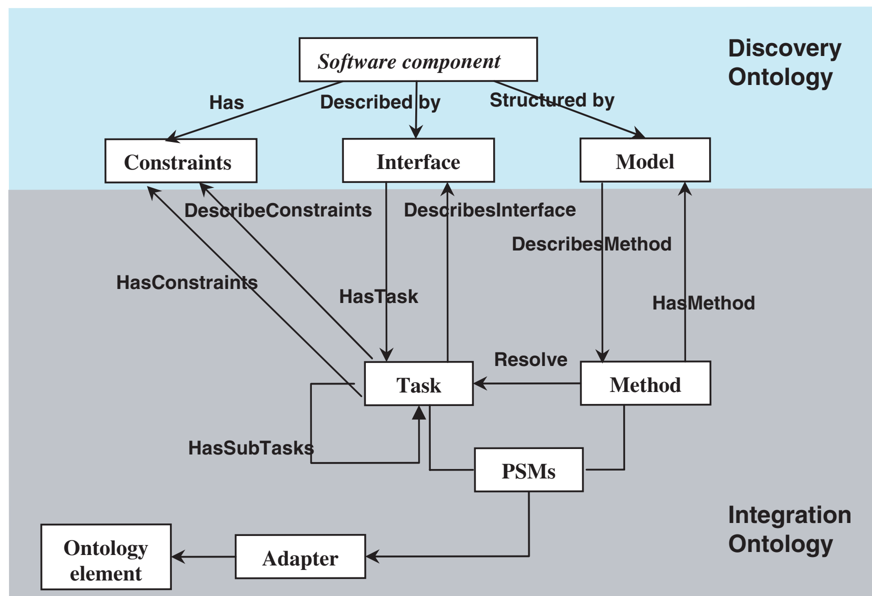

Figure 4.2: Discovery and integration ontologies

4.2.1 Classification of component Non-Functional properties

The non-functional properties of a component cover a wide range of the aspects of the component, and may have different attributes. The aim of this section is to investigate these non-functional properties from the angles of the discovery and integration process, and classify them into different categories. The classification of the component non-functional properties provides knowledge on how to treat these properties during the discovery and integration process.

4.2.1.1 The non functional properties characteristics

The ISO-9126 standard		
Characteristics	Subcharacteristics	Definitions
Functionality	Suitability	This is the essential Functionality characteristic and refers to the appropriateness (to specification) of the functions of the software.
	Accurateness	This refers to the correctness of the functions, an ATM may provide a cash dispensing function but is the amount correct?

	Interoperability	A given software component or system does not typically function in isolation. This subcharacteristic concerns the ability of a software component to interact with other components or systems.
	Compliance	Where appropriate certain industry (or government) laws and guidelines need to be complied with, i.e. SOX. This subcharacteristic addresses the compliant capability of software.
	Security	This subcharacteristic relates to unauthorized access to the software functions.
Reliability	Maturity	This subcharacteristic concerns frequency of failure of the software.
	Fault tolerance	The ability of software to withstand (and recover) from component, or environmental, failure.
	Recoverability	Ability to bring back a failed system to full operation, including data and network connections.
Usability	Understandability	Determines the ease of which the systems functions can be understood, relates to user mental models in Human Computer Interaction methods.
	Learnability	Learning effort for different users, i.e. novice, expert, casual etc.
	Operability	Ability of the software to be easily operated by a given user in a given environment.
Efficiency	Time behavior	Characterizes response times for a given thruput, i.e. transaction rate.
	Resource behavior	Characterizes resources used, i.e. memory, cpu, disk and network usage.
Maintainability	Analyzability	Characterizes the ability to identify the root cause of a failure within the software.
	Changeability	Characterizes the amount of effort to change a system.
	Stability	Characterizes the sensitivity to change of a given system that is the negative impact that may be caused by system changes.
	Testability	Characterizes the effort needed to verify (test) a system change.

Portability	Adaptability	Characterizes the ability of the system to change to new specifications or operating environments.
	Installability	Characterizes the effort required to install the software.
	Conformance	Similar to compliance for functionality, but this characteristic relates to portability. One example would be Open SQL conformance which relates to portability of database used.
	Replaceability	Characterizes the plug and play aspect of software components, that is how easy is it to exchange a given software component within a specified environment.

Functionality is the essential purpose of any product or service. For certain items this is relatively easy to define, for example a ship's anchor has the function of holding a ship at a given location. The more functions a product has, e.g. an ATM machine, then the more complicated it becomes to define it's functionality. For software a list of functions can be specified, i.e. a sales order processing systems should be able to record customer information so that it can be used to reference a sales order. A sales order system should also provide the following functions:

- Record sales order product, price and quantity.
- Calculate total price.
- Calculate appropriate sales tax.
- Calculate date available to ship, based on inventory.
- Generate purchase orders when stock falls below a given threshold.

The list goes on and on but the main point to note is that functionality is expressed as a totality of essential functions that the software product provides. It is also important to note that the presence or absence of these functions in a software product can be verified as either existing or not, in that it is a Boolean (either a yes or no answer). The other software characteristics listed (i.e. usability) are only present to some degree, i.e. not a simple on or off. Many people get confused between overall process functionality (in which software plays a part) and software functionality. This is partly due to the fact that Data Flow Diagrams (DFDs) and other modeling tools can depict process functionality (as a set of data inout conversions)

and software functionality. Consider a sales order process, that has both manual and software components. A function of the sales order process could be to record the sales order but we could implement a hard copy filing cabinet for the actual orders and only use software for calculating the price, tax and ship date. In this way the functionality of the software is limited to those calculation functions. SPI, or Software Process Improvement is different from overall Process Improvement or Process Re-engineering, ISO 9126-1 and other software quality models do not help measure overall Process costsbut only the software component. The relationship between software functionality within an overall business process is outside the scope of ISO 9126 and it is only the software functionality, or essential purpose of the software component, that is of interest for ISO 9126.

Following functionality, there are 5 other software attributes that characterize the usefulness of the software in a given environment. Each of the following characteristics can only be measured (and are assumed to exist) when the functionality of a given system is present. In this way, for example, a system can not possess usability characteristics if the system does not function correctly (the two just don't go together).

Reliability Once a software system is functioning, as specified, and delivered the reliability characteristic defines the capability of the system to maintain its service provision under defined conditions for defined periods of time. One aspect of this characteristic is fault tolerance that is the ability of a system to withstand component failure. For example if the network goes down for 20 seconds then comes back the system should be able to recover and continue functioning.

Usability Usability only exists with regard to functionality and refers to the ease of use for a given function. For example a function of an ATM machine is to dispense cash as requested. Placing common amounts on the screen for selection,

Efficiency This characteristic is concerned with the system resources used when providing the required functionality. The amount of disk space, memory, network etc. provides a good indication of this characteristic. As with a number of these characteristics, there are overlaps. For example the usability of a system is influenced by the system's Performance, in that if a system takes 3 hours to respond the system would not be easy to use although the essential issue is a performance or efficiency characteristic.

Maintainability The ability to identify and fix a fault within a software component is what the maintainability characteristic addresses. In other software quality models

this characteristic is referenced as supportability. Maintainability is impacted by code readability or complexity as well as modularization. Anything that helps with identifying the cause of a fault and then fixing the fault is the concern of maintainability. Also the ability to verify (or test) a system, i.e. testability, is one of the subcharacteristics of maintainability.

Portability This characteristic refers to how well the software can adopt to changes in its environment or with its requirements. The subcharacteristics of this characteristic include adaptability. Object oriented design and implementation practices can contribute to the extent to which this characteristic is present in a given system.

4.2.1.2 Static/Dynamic Non-Functional Properties

Static non-functional properties can be evaluated by examining the internal structure of a software component. These properties are stable in different environments provided the internal structure of component is unchanged. The examples of static non-functional properties are reliability, maintainability, portability, scalability, reusability, presentation, usability, security, priority, and parallelism constraints, etc. Dynamic non-functional properties, on the other hand, can be measured by observing the component behavior at run-time. These component properties are tightly associated with the deployment environment. Examples of dynamic properties are throughput, turnaround time, capacity, availability, result, etc.

From the point of the integration process, static non-functional properties may compose well as they tend not to change during the system execution. The dynamic non-functional properties are influenced by the execution environment, which includes computational resources such as the CPU time, the memory, the disk bandwidth; communication resources such as the network bandwidth; the software resources such as the lock, the pool, the buffer, the semaphores, and the interactions with other components. Most of these factors are not known in advance, thereby the composition of these properties becomes difficult.

4.2.1.3 Non-Functional Properties Domain

Different non-functional properties are emphasized in different application domains. For example, security is most important in the banking domain, while safety and reliability are highly demanded in health care systems. In different application domains, the same non-functional properties may (domain independent) or may not (domain dependent).

For example, the reusability is an example of a domain independent property, while the throughput is an example of a domain dependent property. The system reusability depends on the component with the minimum value of reusability. For a project with two components, if the two components are in a sequence, then the system throughput depends on the component with the minimum throughput; if the two components are in parallel, then the system throughput is the sum of the throughputs of the two components. Reliability is another example of domain dependent property. For a system with two components, if the two components are in serial configuration, the system is reliable if all of these two components are reliable. On the other hand, if the two components are in parallel or redundant configuration, then the system is reliable if at least one component is reliable. Obviously, the domain independent system properties are more convenient to deal with than the domain dependent system properties from the angle of the integration process, because the latter need further information from the specific application domains.

4.3 The atomic component discovery approach

In this section we will describe the discovery ontology mentioned above and our search engine SEC (Search Engine for Component based software development). SEC use the query specification to discover the appropriate component.

4.3.1 The discovery ontology

The ontology describes the subject matter using the notions of concepts, instances, relations, and axioms Gruber (1993). The discovery ontology contains:

- Concepts are organized in taxonomies through which inheritance mechanisms can be applied. A concept contains slots which are restricted by facets. In our case, we specify a component as a concept that contains many slots such as the *performance* slot that describes the component performance using one of the three values(*low, medium, high*)

- Relations represent a type of interaction between concepts. They are formally defined as subsets of a cartesian product of n sets, that is: $R : C_1 \times C_2 \cdots \times C_n$. Examples of binary relations include: subclass-of and connected-to. For example the relation between component and method is a connected-to relation. The concept method contains the input types, output types, precondition and signature slots.

- Functions constitute a special case of relations in which the n-th element of the relationship is unique for the n-1 preceding elements. Formally, functions are defined as: $F : C_1 \times C_2 \cdots \times C_n$. Examples of functions are father-of and rank-of-a-component that calculates the rank of a component depending on the "used rate" and kindness match. The "used rate" computes the rate of the component utilization Sofien *et al.* (2002). The kindness (efficiency) of a matching is related to the subsumption notion. The subsumption idea has to be related to suitable matching notions for components provided and query specification in order to refine the selection result.

```
1
2 ... <rdfs:Class rdf:about="&kb;Component"[3 lines] <rdf:Property
3 rdf:about="&kb;Authors"[5 lines] <rdf:Property
4 rdf:about="&kb;component_model" [4 lines] <rdf:Property
5 rdf:about="&kb;Communicating_ Component"[4 lines] <rdf:Property
6 rdf:about="&kb;Libelle"[4 lines] <rdf:Property
7 rdf:about="&kb;Location"[4 lines] <rdf:Property
8 rdf:about="&kb;Has_Dynamic_NF _Aspect"[4 lines] <rdf:Property
9 rdf:about="&kb;Has_Interface" [4 lines] <rdf:Property
10 rdf:about="&kb;Has_Static_NF _Aspect"[4 lines]
11
12 <rdfs:Class rdf:about="&kb;Dynamic_NF _Aspect"[3 lines]
13 <rdf:Property rdf:about="&kb;Availability" [4 lines] <rdf:Property
14 rdf:about="&kb;Capacity" [4 lines] <rdf:Property
15 rdf:about="&kb;Performance" [4 lines] <rdf:Property
16 rdf:about="&kb;Turnaround_Time" [4 lines] <rdf:Property
17 rdf:about="&kb;Inverse_of_Has_ Dynamic_NF_Aspect"[4 lines]
18
19 <rdfs:Class rdf:about="&kb;Interface"[3 lines] <rdf:Property
20 rdf:about="&kb;Inverse_of_Has_ Interface"[4 lines] <rdf:Property
21 rdf:about="&kb;Method"[4 lines] <rdf:Property
22 rdf:about="&kb;Output"[4 lines] <rdf:Property rdf:about="&kb;Input"
23  a:minCardinality="1"
24  rdfs:label="input">
25 <rdfs:domaine rdf:resource="kb;Interface"/> <rdfs:range
26 rdf:resource="&kb;TYPE"/> </rdf:Property> .... </rdf:RDF>
```

Listing 4.1: The discovery ontology

- Instances are used to represent elements.

The discovery ontology has been developed using PROTEGE2000 Informatics (2001) and mapped through the Resource Description Framework which is an XML-based language Lassila et Swic (1999) and which also represents the component descriptions in the repository. Our XML document, generally, contains the following key concepts: Component, Interface, Dynamic_NF_Aspect that represents the dynamic non functional aspect, and Static_NF_Aspect that represents the static non functional aspect. Each Concept contains

many slots which describe component features. The slot, generally, contains the following key elements: *mincardinality*, *maxcardinality*, *label*, *domain* and *range*. As indicated in the listing 4.1, the slot *input* has as *mincardinality* value "1" and as *label* the value "input". The *domain* represents the concept of the slot, in our case the concept of *input* is "interface". The *range* indicates the value type of the slot, in our case the *input* value(s) is/are one or more instances of the concept *Type*. These instances are: integer, double, float, date, string, class, etc..

4.3.1.1 Definition of the discovery ontology structure

The first stage in the definition of an ontology consists in defining its structure, i.e., the classes which characterize the ontology. The class root of any ontology is `owl :Thing`

The figure 4.3 present the RDF ontologie structure. This ontology dose'nt show the relations between classes.

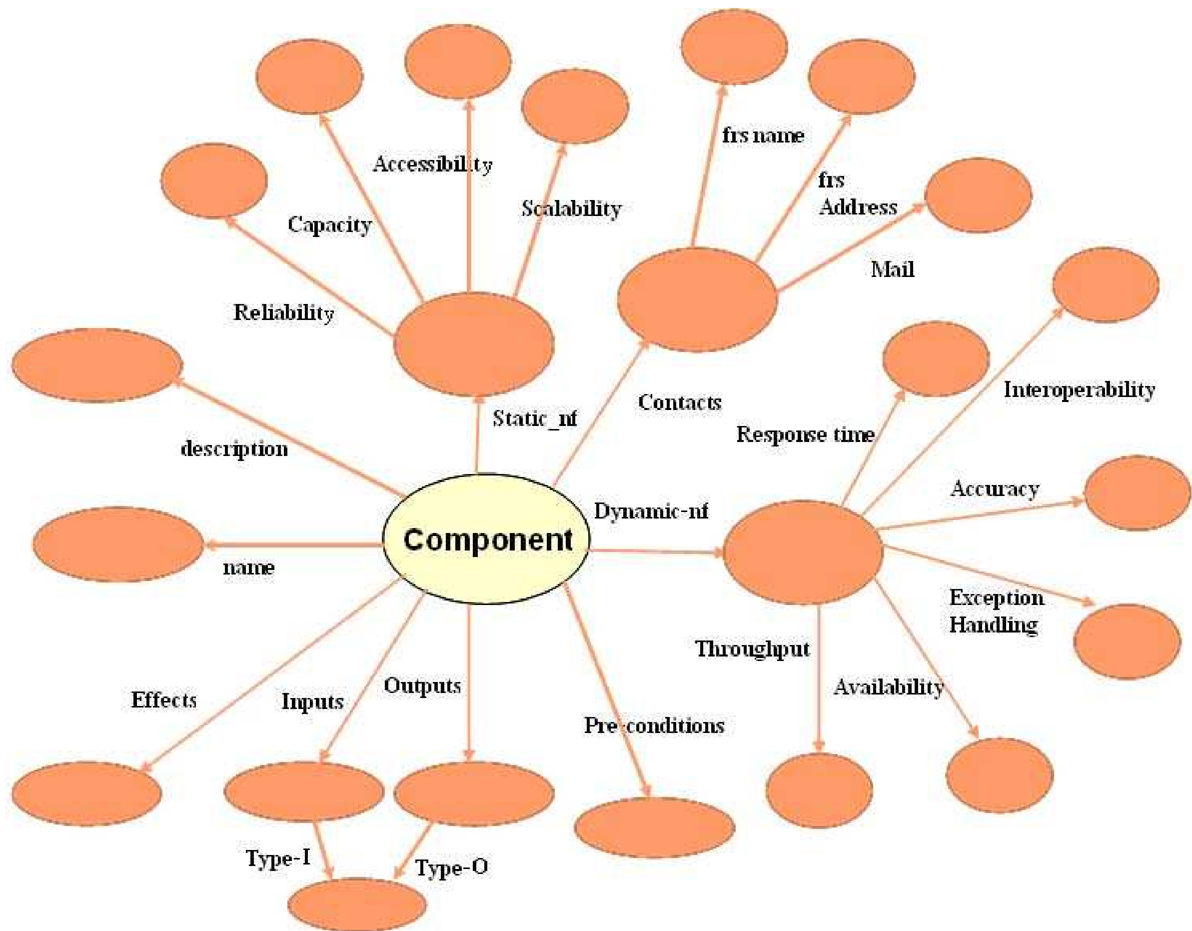

Figure 4.3: RDF Graph structure in OWL-S Profil form

4.3.1.2 Definition of the discovery ontology properties

The properties in the class are enable to specify the class information. In fact, some of them can establish relation between classes.

The component class properties has five properties.

Input Describes the input of the method

Output Describes the output of the method

Precondition Describes the preconditions of the method

Effect Describes the expected result of the method

method_label Describe the label of the method

Using our search engine, we can discovery software component based on the inputs and preconditions that need to be satisfied and outputs and effects that need to be produced. The search process compare also the components methods names and the specified method name in the query, produces results that closely match a user's requirement. Also we use the non-functional aspect to filter the selection. We identify six properties

Availability Users must be able to access the system twenty fours hours per day

Accuracy This refers to the correctness of the component

Capacity/Performance Is a measure of how quickly the component responds to stimuli, and how well it utilizes resources in providing that response.

turnaround time Is time between component start execution and completion of output

Exception handling Designed to handle the occurrence of some condition that changes the normal flow of execution.

Throughput is a measure of how many operations can be performed in a given amount of time under a given operating load.

In our case we specify each non functional properties by one of the three values(low, medium, high).

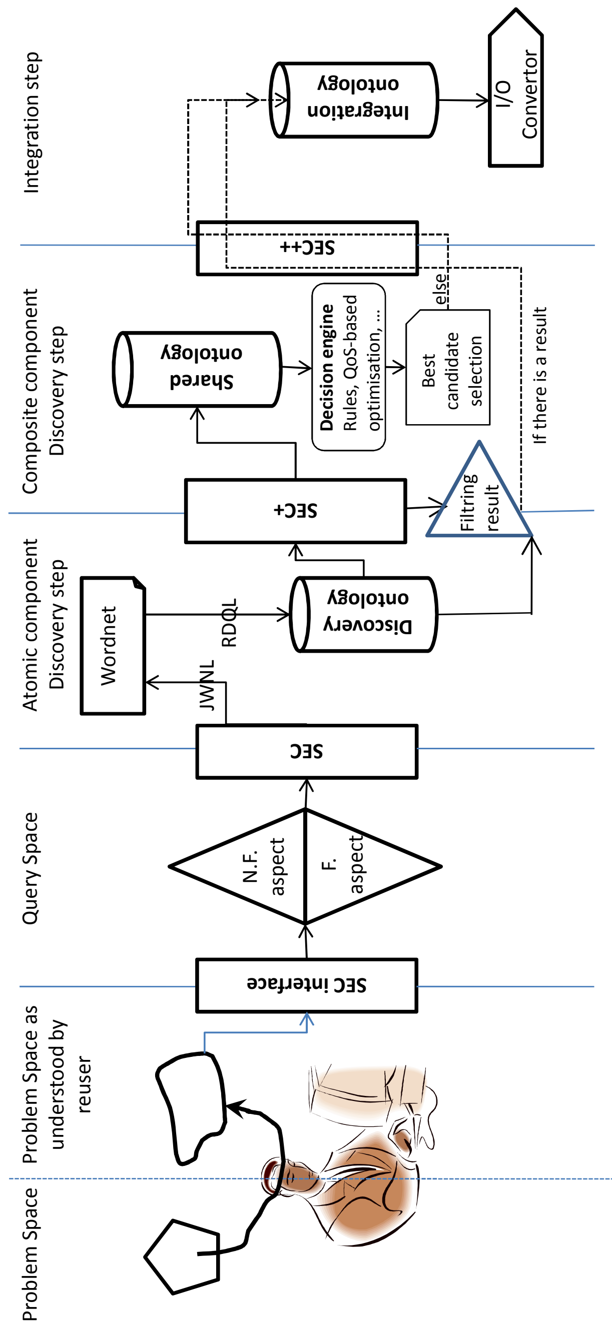

Figure 4.4: The search engine SEC and its different version

4.3.2 The first version of the search engine: SEC

We implemented the search engine progressively. In a first version, called SEC(Search Engine for Component based software development), we only discover atomic component. The second version SEC+ extends the first one by adding a composition process to discover a composite component. The last version, SEC++ , introduce the integration process. Figure 4.4, shows the architecture of the search engine and the components added in each step. It also presents the different steps from the query specification until the component integration.

The SEC component manages locating software components. It executes the specified query, retrieves and presents relevant components. SEC requires no loading from software developers in development environments. In current development practices, the developer chooses the non-functional features and the functional features that meet his needs. A query will be formulated and then executed automatically in order to deliver the appropriate component.

4.3.2.1 The Matching algorithm

The matching algorithm we used in SEC is based on the algorithm *matchComp* (see listing 4.2)which calculates the similarity degree between component attributes and the query parameters. The algorithm defines a flexible matching mechanism based on the subsumption mechanism and a function that calculates the semantic distance in the WordNet hierarchy between the component method names and the method name specified in the query.

```
1
2 double matchComp (method_name_Comp, method_name _query) maxScore =
3 10; if (method_name_WS is identical to method_name _query)
4   case subsume (Input_Comp[], Output_Comp
5   [ ],Input_query[ ], Output_query[ ]) =
6
7 "Exact" : score = maxScore; "Subsume" : score = 8; "Plugin": score
8 =6;
9
10
11 else if (method_name_Comp and method_ name_query are synonymous)
12        case subsume (Input_Comp[],
13        Output_Comp[ ],
14        Input_query[ ],
15        Output_query[ ]) =
16
17 "Exact" : score = 8; "Subsume" : score = 6; "Plugin": score =4;
18
19 else if (method_name_Comp and method_ name_query have hierarchical
```

```
20 relations)
21       case subsume (Input_Comp[],
22       Output_Comp[ ],Input_query[ ],
23       Output_query[ ]) =
24
25 "Exact" : score=6/(distance between them ) "Subsume" : score = 6/(
26 distance between them )
27 *1.2;
28 "Plugin": score =6/( distance between them )
29 *1.5;
30
31 else score = 0; return score
```

Listing 4.2: Matching algorithm code

The subsumption mechanism is the degree of correspondence between the inputs/outputs of the query and of the component. We consider four degrees of correspondence:

-**Exact** If Inputs/Outputs of a component are equal (or unified) to the Inputs / Outputs of a query. Also equal included subsume between respective individual input or output of the component and the query.

- **PlugIn** If Inputs/Outputs of a Component is a subset of Inputs / Outputs of a query.

- **Subsume** If Inputs/Outputs of a query is a subset of Inputs / Outputs of a Component .

- **Disjoint** If Inputs/Outputs of an Advertisement do not match with Inputs/Outputs of a query.

```
1
2 /* {JWNL initialisation code} */
3
4 public IndexWord ACCO;
5
6 String propsFile = "D:\\workspace\\Test\\file_properties.xml";
7
8 JWNL.initialize(new FileInputStream(propsFile));
9
10 String wn=jTextField.getText();
11
12 ACCO = Dictionary.getInstance().getIndexWord(POS.NOUN, wn);
```

Listing 4.3: JWNL initialisation code

Intuitively, the term similarity degree is a function of the term semantic distance in the WordNet hierarchy and is a function of degree of correspondence between the inputs/outputs of the query and of the component: Components methods names that are located close to each other in the WordNet semantic hierarchy have similar meanings and therefore are assigned a higher similarity degree than others that are further apart in the

WordNet hierarchy. In each case we compare the degree of correspondence between the inputs/outputs of the query and of the components. In all cases Exact match are clearly preferable to PlugIn match which are considered the second best, Subsume match are considered to be third best.

```
1
2 /*extract of the source code that access WordNet}*/
3
4
5 int nbr=word.getSenseCount(); for(int vi=0;vi<nbr;vi++) {
6         PointerTargetNodeList hypernyms =
7
8
9
10   PointerUtils.getInstance().getDirectHypernyms(word.getSense(vi+1));
11
12     if (hypernyms.isEmpty())
13     {
14         System.out.println("empty hypernyms");
15     }
16     else
17     {
18         String var=hypernyms.toArray()[0].toString();
19         ch += position(var)+",";
20
21                 }
22         }
23 //  Position method
24 public String position (String a)
25     {
26         String b="";
27         int i=a.indexOf(" Words: ");
28         int j=a.indexOf(" -- ");
29         b=a.substring(i+7,j);
30         return b;
31     }
```

Listing 4.4: Extract of the source code that access WordNet

More specically, if components methods names and the specified method name in the query are identical they are assigned respectively 10 if there is Exact match, 8 if there is a Plugin match and 6 if there is Subsume match. If they are synonymous (regardless of the words' senses), their similarity degree is respectively 8 if there is Exact match, 6 if there is a Plugin match and 4 if there is Subsume match. Otherwise, if two words are in a hierarchically semantic relation, i.e., they are hypernyms, hyponyms or siblings to each other, their similarity is inversely proportional to the shortest path in the WordNet hierarchy linking them semantically. The identifier similarity score between two such terms is calculated by dividing 6 by respectively their (semantic_distance) if there is exact match, (semantic_distance)*1,25 if there is Plugin match and (semantic_distance)*1,5 if

there is Subsume match.

The components resulting from the search will be sorted according to the degree of the similarity with the query.

Once SEC is running, the developer specifies a query by selecting the adequate criteria as indicated in figure 4.5. To query the model we use the RDQL (RDF Data Query Language) which is a query language for RDF models. An approximate comparison between the specified query and the synonyms description of components in the discovery ontology is made by the *compare_query_description()* function. If there is a positive result, the *search_component(ref_component[])* function retrieves the appropriate component(s), where *ref_component[]* is the list of the component references to retrieve. Then, the developer uses an application programme interface (API) to integrate the desired component into the current project. Finally, to facilitate the component integration, the developer uses the integration ontology.

WordNet JWNL (2003) is used as a thesaurus for synonyms, hyponyms and hypernyms. However, the thesaurus has to be initialized for each domain for which it is used. If additional knowledge or a different domain is needed, then the user has to interactively input the corresponding terminology.

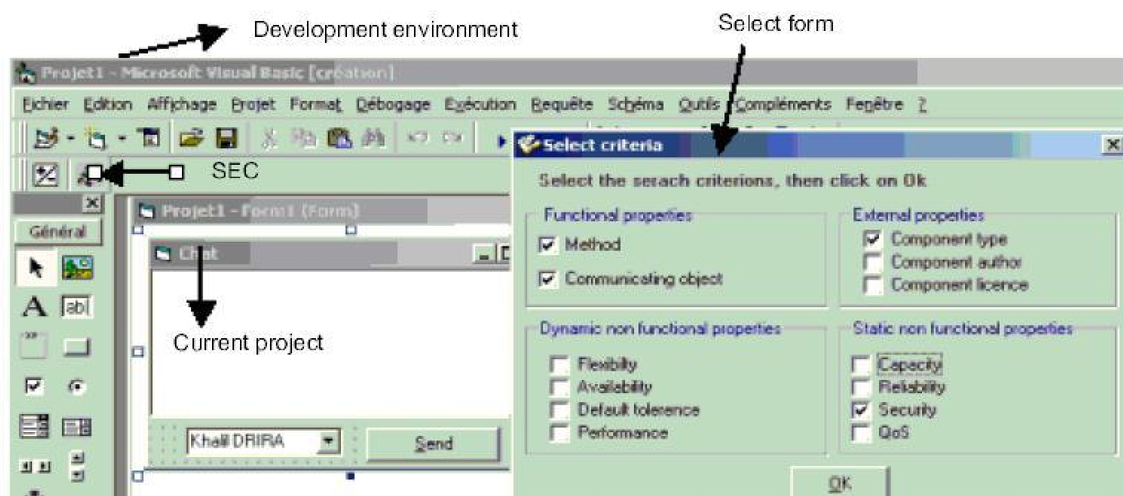

Figure 4.5: Search step

Our search engine SEC uses Jena Clifton et Wen-Syan (1995) to parse and negotiate the discovery ontology. Jena provides an easy and a robust API and the possibility to be used remotely following a client-server interface. To create an RDQL query, we put the RDQL in a string, and pass it to the constructor of the query. It's usual to explicitly set the model to use as the source for the query, unless otherwise specified with a FROM clause in the

RDQL itself. Once a Query is prepared, a QueryEngine can be created, and the query executed. Listing 4.5 is a query to find components name which the turnaround_time rate is less than 0.75 second.

```
1
2 String queryString= "SELECT ?Libelle WHERE (?Libelle
3 info:Turnaround_Time ?Turnaround_time) AND ?Turnaround_Time <= 0.7
4 USING info FOR <http://somewhere/componentInfo#>";
5
6 Query query = new Query(queryString);
7 //Need to set the source if the query does not.
8 query.setSource(model); QueryExecution qe = new QueryEngine (query);
9
10 QueryResults results = qe.exec(); for (iterator iter = results;
11 iter.hasNext();) {
12   resultBinding res = (ResultBinding) iter.Next();
13   ...Process result here ...
14 } results.close();
```

Listing 4.5: Query Code extract

4.3.2.2 The semantic Web toolkit: Jena

Jena is a leading Semantic Web toolkit McBride (2002) for Java programmers. Jena1 was first released in 2000 and has had over 10,000 downloads. Jena2, with a revised internal architecture and many new features, was released in August 2003, and has had over 7,000 downloads. This section presents Jena2, concentrating on the key architectural. The heart of the Semantic Web recommendations is the RDF Graph as a universal data structure. An RDF graph is simply a set of triples (S, P, O), where P names a binary predicate over (S, O). Jena2 similarly has the Graph as its core interface around which the other components are built.

The main contribution of Jena1 McBride (2002) was the rich Model API for manipulating RDF graphs. Around this API, Jena1 provided various tools, including I/O modules for: RDF/XML, N3, and N-triple; and the query language RDQL refee. Using the API the user can choose to store RDF graphs in memory or in persistent stores. Jena1 provided an additional API for manipulating DAML+OIL. User feedback on Jena1 suggested better integration between the DAML+OIL support and the RDF support to permit, for example, the storing of DAML models within databases. It also had proved too difficult to add further implementations of the rich Model API to Jena1. In response to these issues, Jena2 has a more decoupled architecture than Jena1. The two key architectural goals of Jena2 are:

- Multiple, flexible presentations of RDF graphs to the application programmer. This allows graph data to be accessed and manipulated through higher-level interfaces.
- A simple minimalist view of the RDF graph to the system programmer wishing to manipulate data as triples. This is particularly useful for RDFS and OWL reasoning.

The first is layered on top of the second: any triple source can back any presentation API. Both the architectural goals provide extension points for system programmers. The presentation layer is the basis of both the existing Model API and the new Ontology APIs for OWL, DAML+OIL and RDFS. The graph layer allows the development of new triple sources, both materialized triples, for example from database or in-memory triple stores, and virtual triples generated dynamically as a result of some processing, such as inference or access to legacy data sources. Jena2 provides inference support for both the RDF semantics and the OWL semantics. Jena supports a Semantic Web query language, RDQL, that can be used either on top of materialized graphs, or on the virtual results of RDFS or OWL reasoning. Complete queries can be passed into the underlying graph layers, so database-backed graphs can take advantage of SQL optimization. A third presentation interface, the RDF WebAPI, provides web clients with query-based access to RDF graphs. This querybased access is also available at both the system and application programmer interfaces, and acts as a further unifying theme of the architecture.

The heart of the Jena2 architecture is the RDF graph, a set of triples of nodes. This is shown in the Graph layer. This layer, following the RDF abstract syntax, is minimal by design: wherever possible functionality is done in other layers. This permits a range of implementations of this layer such as inmemory or persistence triple stores.

The EnhGraph layer is the extension point on which to build APIs: within Jena2 the functionality offered by the EnhGraph layer is used to implement the Jena1 Model1 API and the new Ontology functionality for OWL and RDFS, upgrading the Jena1 DAML API. I/O is done in the Model layer, essentially for historical reasons. The Jena2 architecture supports fast path query that goes all the way through the layers from RDQL at the top right through to an SQL database at the bottom, allowing user queries to be optimized by the SQL query optimizer. We give some more detail on the three layers below.

4.4 The composite component discovery approach

This approaches is used there is no atomic component discovered in the discovery approaches. We tend to discover a composite component that response to developer's query.

For this we develop a shared ontology and a composition process.

4.4.1 The shared ontology for composition

This approach exploits the advantages of semantic composition approaches, powered by ontologies at both component discovery and integration levels. Building on top of that, we introduce an ontology-based semantic approach. First, the semantic component specification provides a mechanism to enrich atomic components with more semantics than the syntactical method. Second, mapping atomic components and other relevant concepts into a centralized shared ontology offers a knowledge repository for software components. The objective of semantic enhancement is to support ontological heuristics in order to enable automated and dynamic component composition (see Figure 4.6). When our enhanced search engine SEC+ receives a query from a consumer, it first searches the discovery ontology. Our approach enhances the discovery ontology with a shared ontology. This centralized ontology represents relevant components and concepts in a specific domain, constructed by mapping and integrating individual integration ontologies for software components. Here, the ontological heuristics serves as guidelines to respond to a developer request. After using ontological heuristics on the shared ontology, SEC++ generates a number of alternative solutions to component composition. These alternatives are then evaluated by a decision engine using a set of criteria specified by the developer. Such criteria may include QoS-based optimization of component composition, business rules and strategies. A selected optimal composition scheme is then executed.

As for the integration ontology, we employ problem solving method to develop a local ontology for component. In the integration ontology we try to divide the component process into tasks. Tasks are either solved directly (by means of primitive methods), or are decomposed into subtasks (by means of decomposition methods). We use the Unified Problem-Solving Method Language (UPML) Fensel et al. (2003) to describe the components of PSMs (task, method and adapter). Similarly, the component model subclass is especially beneficial for composition. The proposed approach utilizes the component model class in two ways. For base components, a component model keeps information about composability, which specifies when the component can be used in a composite component. For composite components, a component model maintains alternative composite solutions incrementally for reuse. This semantic enrichment provides a self-learning capability of component composition.

Local integration ontologies are consolidated in a server by ontology mapping and integration. As a result, all relevant concepts and components in a domain are in the shared

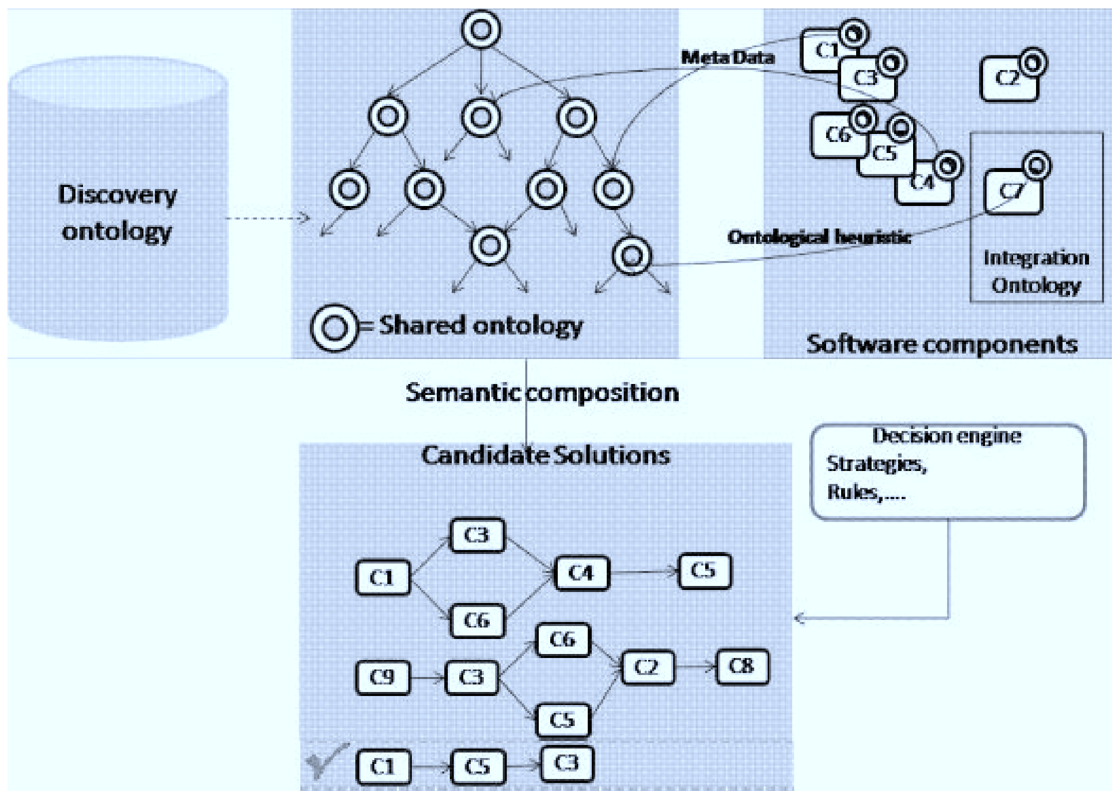

Figure 4.6: An ontology-supported system for component composition

ontology, local integration ontology for a component is mapped into the shared ontology, appearing as a node in the ontology tree. How to organize all components into the repository depends on domains and application requirements. For example, for calculating Matrix we can maintain semantic relationships (e.g., hierarchical and sibling relationships) between Matrix operations. The shared ontology also represents other application-specific concepts for mapping and integrating components. The mapping and integration not only unite component descriptions and concepts but also add more semantics. Moreover, the shared ontology enables ontological heuristics, thus facilitating dynamic component composition. For example, we can study composability of components based on some generic concepts. As a simple example, when composing component C2 that calculates the determinant of a real matrix by receiving the output parameters of a component C1 that calculates the sum of two matrix which have a natural type. At first glance, these two components cannot be composed. However, the relationship between real and natural is revealed in the type ontology: natural is included in real. The RDF+OWL document shows how OWL uses unionOf vocabulary to represent this relationship. Similarly when composing two components which conducted at different periodical levels: annual and quarterly. These two components cannot be composed if time period is a parameter. However, the relation between annual and quarterly is revealed in the time ontology (see

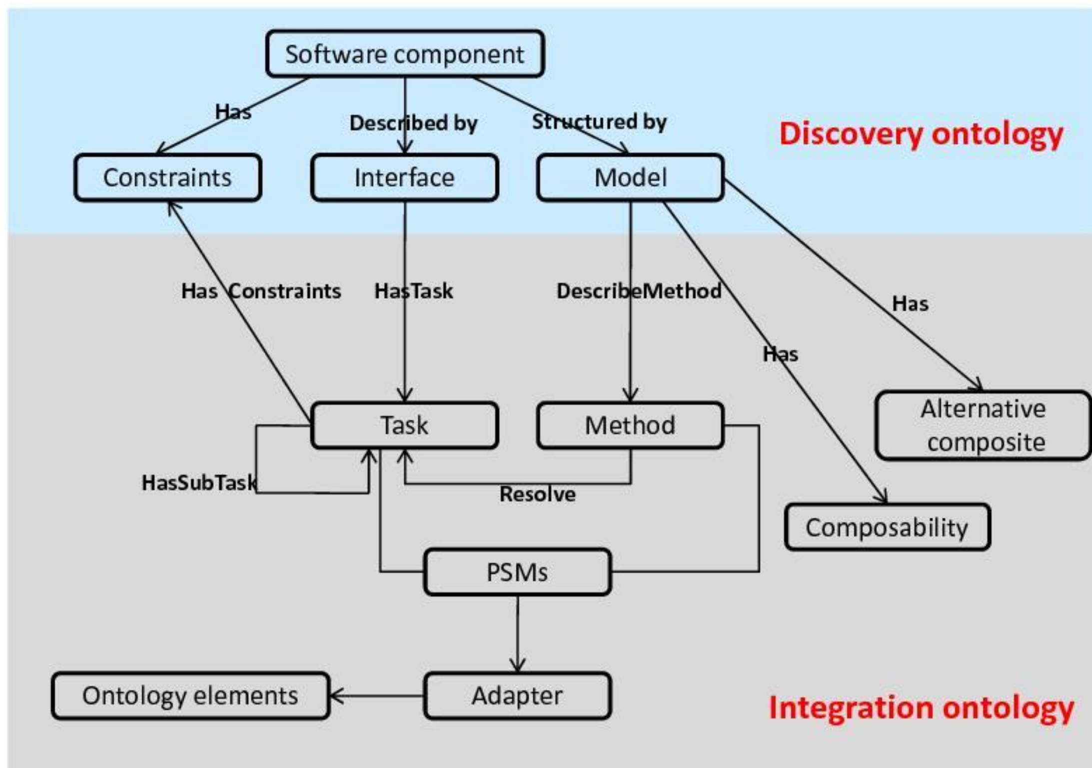

Figure 4.7: Discovery and integration ontologies: The new version

Figure 4.8). In an industrial context, a specific team would be responsible for the description of local integration and shared ontologies. This team mainly considers functional features should focus more on the analysis part (e.g., determine the domain and scope of ontologies, and enumerate important terms in ontologies), while the technical part takes charge of the design and implementation (e.g., define classes and class hierarchy, define properties of classes, define facets of the slots, and create instances). The developed ontologies should be reviewed periodically. Our proposed ontology represents an enhanced approach to organizational knowledge management. The shared ontology incorporates systematically relevant knowledge into a centralized repository.

```
  <owl:onProperty rdf:resource="http://www.daml.org/Process#
- <owl:toClass>
  - <owl:subClassOf>
    - <owl:unionOf rdf:parseType="owl:collection">
        <rdfs:Class rdfs:about="#FirstQuarter" />
        <rdfs:Class rdfs:about="#SecondQuarter" />
        <rdfs:Class rdfs:about="#ThirdQuarter" />
        <rdfs:Class rdfs:about="#FourthQuarter" />
      </owl:unionOf>
```

Figure 4.8: unionOf vocabulary relationship

4.4.2 Shared ontology implementation

As mentioned the composability property of the component model class can have values denoting possible ways for component composition. Taking the *binary_operation_Matrix* component. The inputs to this component include two matrix *M1* and *M2*. The outputs are sum, product and determinant. The input and output its composability contains a list of possible parameter flows (from inputs to outputs): *M1* to *determinant*, (*M1* and *M2*) to *product*, *M2* to *determinant* and so forth, each of which can be a part of an alternative path in a composite component. Another way to exploit composability is first to attach composability to other properties with concrete meanings, then associate composability with composition rules; for example, assuming composability is a property Another way to use composability is first to attach composability to other properties with concrete meanings, then associate composability with rules of composition; for example, assuming composability is a property of time. If a a component *C1* is time period based, while *C2* is time point based, these two component should not be composed together. As a result, the value of the composability property for the time of *C1* can be $\neg$*timepoint*.

All designed local integration ontologies are mapped together following Matrix operation organisation, appearing as nodes or subclasses in the shared ontology, as each component described in discovery ontology. After organizing components into a shared knowledge repository, we can adds other concepts relevant to Matrix operations, either domain-specific or generic, such as Type and operation. The semantics obtained so far are limited to hierarchical and sibling relationships. Ontological mapping and linkage supplement a richer set of semantics, which can be performed through the value type constrained.

4.4.3 The second version of the search engine: SEC+

SEC+ is a novel version of SEC (Search Engine for Component based software development) Sofien *et al.* (2006) which guaranties the composition step. It is an extension of SEC Sofien *et al.* (2007) by adding a component composition based on ontological heuristics. SEC+answer the developer query if no single component can provide all required information, but composing some of them can fulfill the request (see figure 4.4). We developed a persistent component, called SEC+, that contains the search process and the composition process. It can be loaded in the development environment.

The first screen shot of SEC+ contains two tags. The First contains the different functional attributes, the second is devoted to the non functional attributes.

For the tag dedicated to the functional aspects (See figure 4.9), the user must specify the name of the desired services as well as the the inputs/outputs which he considers useful to this service. The selection of the inputs/outputs is carried out through two listboxes.

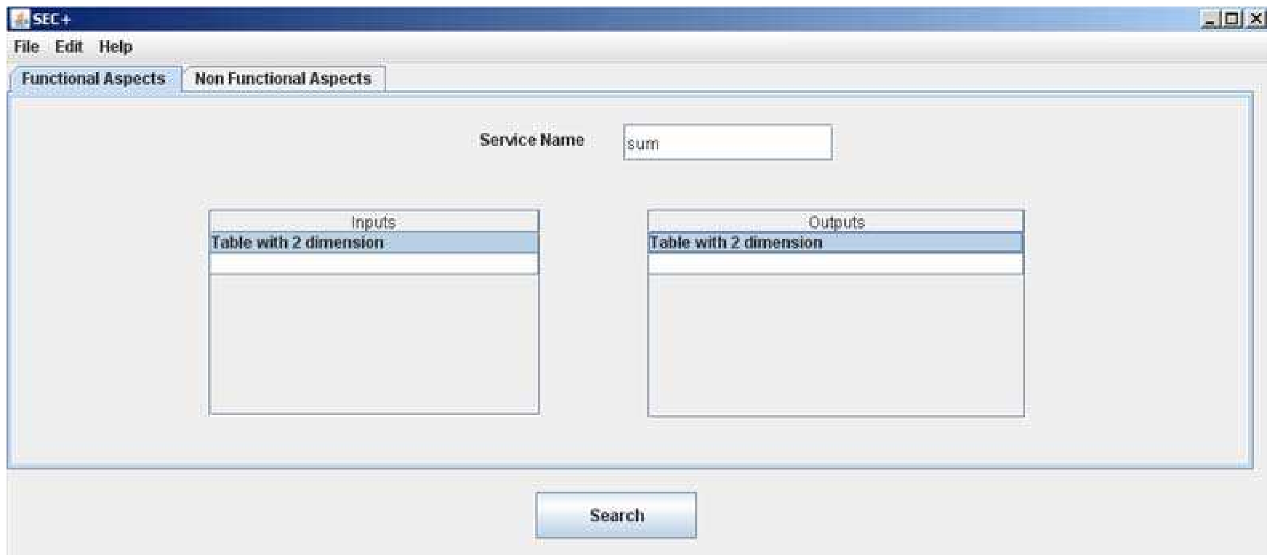

Figure 4.9: Functional aspect interface

On the second tag, the user chooses a list of the non functional attributes (See figure 4.10)which he considers useful for the desired service. For each selected attribute, the user must indicate the desired level (High, Meduim, Low) as well as the relative weight. The weight varies between 1 and 3 (By default = 1). If the user judges that a nonfunctional attribute is important, it can reinforce the relative weight and rate it at 2 or 3 according to the degree of importance of the attribute.

The discovered components, will be presented in a drop-down list (See figure 4.11), sorted according to their weights. When the user selects a component in the list, the functional and non-functional aspects (Name, Description, inputs, ouputs, Pre-conditions, non functional aspects with their levels) relative to this component will be set up to help him. By comparing the details of each component in the list, the user can choose the appropriate component which is near to his need.

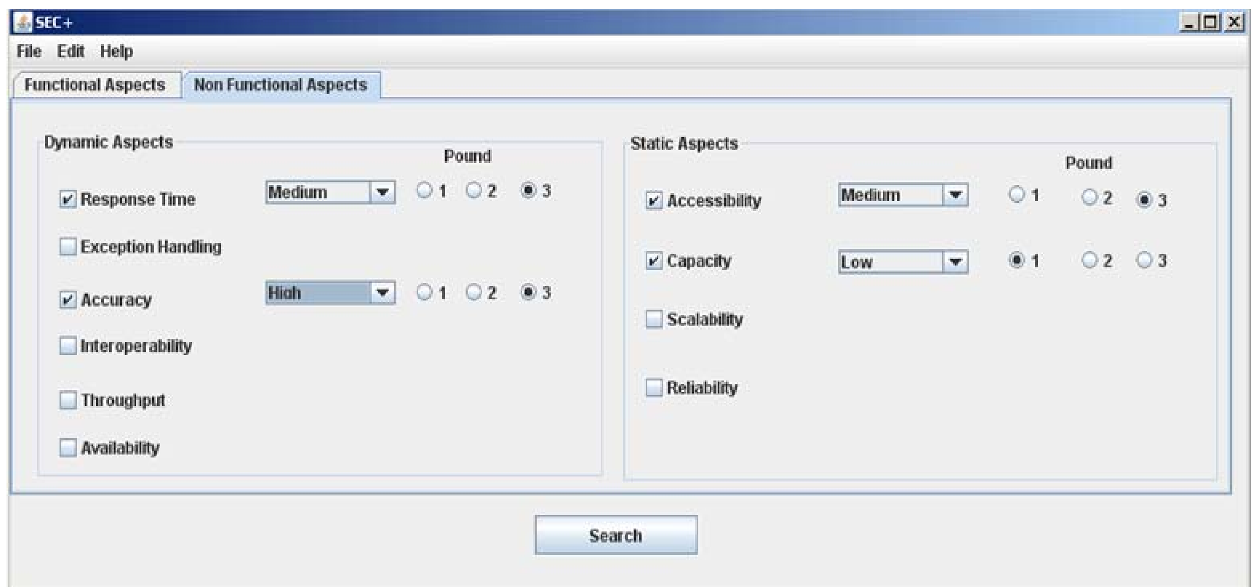

Figure 4.10: Non-Functional aspect interface

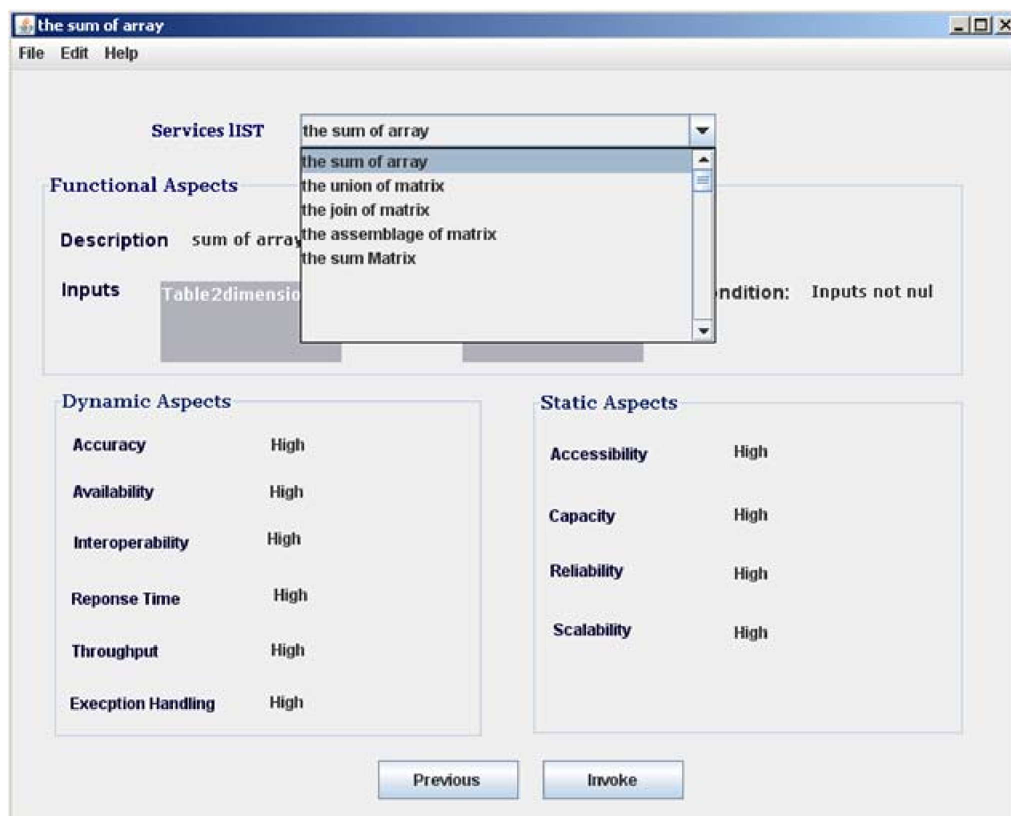

Figure 4.11: Result interface

4.5 The integration approach

4.5.1 The integration ontology

The integration ontology purpose is to separate component's functional features from its internal specification. Many approaches have proposed process-based languages such as BPEL(Business Process Execution Languages) Andrews *et al.* (2003) and OWL (Ontology Web Language) et al. (2002). These languages describe a component's internal structure using a predefined set of workflow-like patterns (sequence, parallel split, choice, etc.). But these languages lack an explicit, declarative decoupling between a component's functional features (what) and its structural description (how)Gomez-Perez *et al.* (2004).

In the integration ontology we try to divide the process into tasks (see figure 4.7). Tasks are either solved directly (by means of primitive methods), or are decomposed into subtasks (by means of decomposition methods) whose interaction can be modelled as a workflow pattern Aalst *et al.* (2003). We use the Unified Problem-Solving Method Language (UPML)Fensel et al. (2003) to describe the components of PSMs 1 (1998)(*task, method and adapter*).

There are 3 main features that distinguish our approach from others. Firstly, it counts with a Zero-updating code in the integration process. It enables the construction of application systems out of existing components independently developed in various domains without any modification of components. Integration mismatches which will occur can be solved by automatic mediation. Secondly, it separate component's functional features from its internal specification. Finally, it provides a simple Composition Description Language (CDL), which represents the binary relations among components as output of the integration process. It is intended as a mechanism to describe the internal connection between components.

From the process perspective, we believe that a complete integration process should include the following activities at least: (1)Composite Component Definition: The activity for building the composite component schema or an executable composite component. (2) Component Deployment: The activity for deploying executable composite components in the component execution engine. (3) Component Execution: The activity for performing tasks of the composite component in the execution engine. However, only integration-related activities are not enough. Component's dynamics nature

must be considered by integrating component discovery activities. We will detail how to combine component integration and component discovery in the following.

4.5.1.1 Integration type

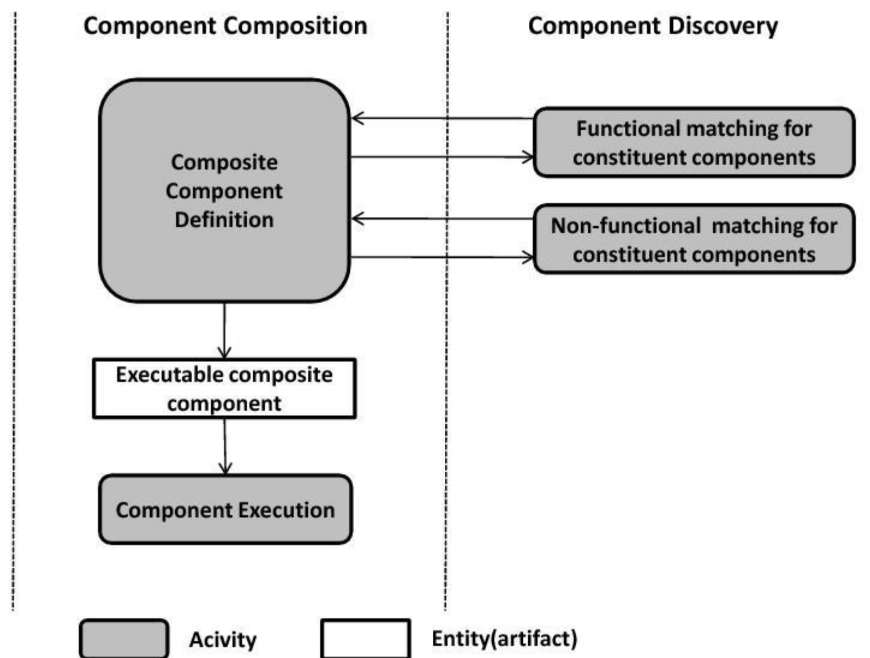

Figure 4.12: Static Composition

We identify two type of integration

Static component integration Static integration is the style that components to be integrated are decided at design phase. If we consider only static integration, component discovery will be needed in the "component definition" step(see Figure 4.12) when the component developer want to obtain constituent components to have a composite component. functional matching (matching by inputs, outputs , precondition, component name, etc.) and non-functional matching (matching by non-functional conditions, such as cost, performance, etc..) will both be utilized according to the developer's needs. That is the only difference compared with the standard component integration process. In static component integration, developers can produce an executable composite component or an abstract composite component schema.

Dynamic component integration Dynamic integration is the style that components to be integrated are decided at run-time. If we want to dynamically discover the best available components that response the needs of the developer, dynamic integration process must be ready. All three phases in dynamic integration (See Figure 4.13) must rely on component discovery:

Composite component Definition Before invoking the composite component, we need to produce the abstract composite component flow (i.e. the integration schema). To construct the flow, interface matching is needed to choose the interfaces of the constituent components used in this integration.

Component Deployment When we want to deploy the composition, all required component bindings must be ready. Semantic matching is required for selecting most suitable concrete constituent components. The concrete component can be replaced according to the user's requirements.

component Execution If some constituent component leaves or malfunctions, interface match- ing or semantic matching can be re-performed.

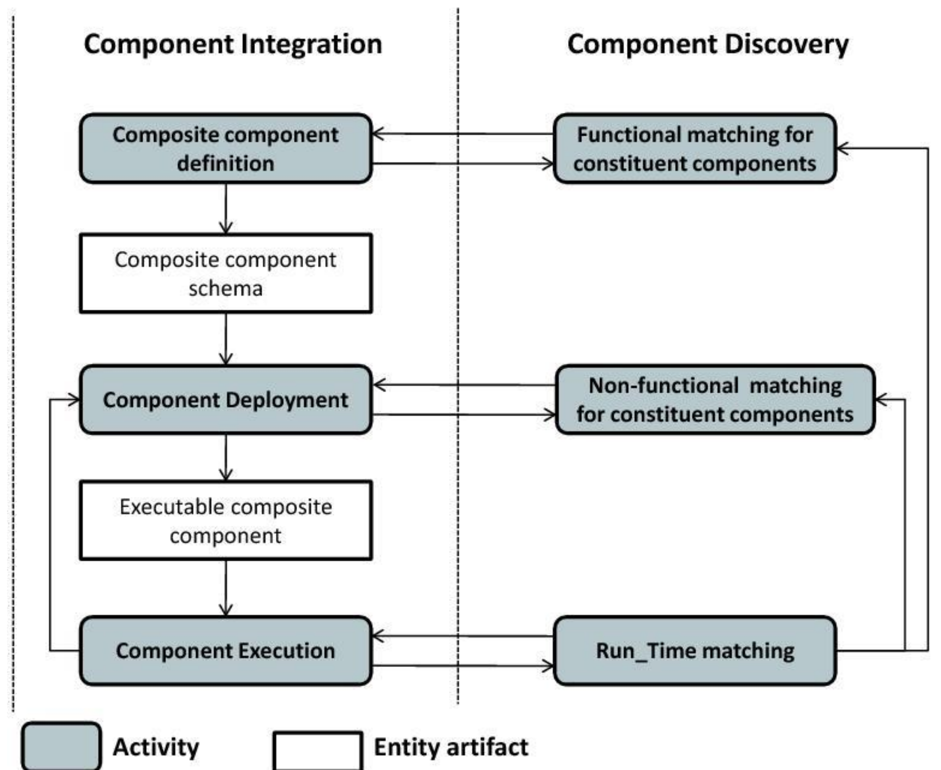

Figure 4.13: Dynamic Composition

4.5.1.2 Zero-updating code in the integration process

It enables the construction of application out of existing components without any components updating. In general, software components are developed in various domains and heterogeneous for an integration to accommodate their singularities. Moreover integration is an inter-domain problem that must contain different syntax and semantics of

component knowledge. It seems that integration cannot be solved without modification of components or wrapping methods used as glue parts. We can conclude that automatic mediation of mismatches is possible with knowledge about the specification of components. This solution can removed the zero-updating code most obstacles in integration process.

4.5.1.3 Composition Description Language

The integration is an independent task or phase of domain-specific

application system where an integrator produces an integration list, and an integration schematic, as outcome of the process. To cope with mismatches integrator can automatically insert mediators to solve interpretational difficulties between components. As a result of this integration list is generated that represents binary relations among components. For the representation of integration list, a Composition Description Language (CDL) is used. The CDL is not intended to describe the functional capability of the the integrated components, but to depict the internal connection between components. An integrator can use CDL to represent and describe the physical connection of an assembled component. CDL can describe the internal organization of integrated component effectively because it is supported by the uniform connection mechanism.

4.5.1.4 Integration as a Generic Problem solving Method

The integration approach presented so far can be easily adapted into UPML since it is very similar to this framework in the way it conceptualizes generic tasks. In the following, we describe the detail formalization of our generic approach to integration based on UPML. The integration ontology provides the common terminology used by tasks, PSMs and domain model. The task requires the terminology to specify integration requirements, the PSM uses it to specify the integration list, and the domain model makes use of it to specify component properties and characteristics of the component repository. The specification of the task integration request consists of description of components. Since all knowledge constructs are constituted in the unit of a component, it is possible to construct an integration ontology from the unified view of components. In addition to this feature for ontology description, the inference operations provided in the ontology are also performed in the unit of a component, which means, that all inference operations for component discovery, integration and verification are accomplished with the component description as the arguments of operations. Consequently, the integration ontology can be shared in other parts of UPML. Domain Model The role of the domain model of UPML is

to provide the specific domain knowledge to the generic task and PSM. The task in UPML is virtually specified by composition requester in terms of its input and output roles, preconditions and post-conditions, competence, and assumptions. The integration request specification is the description of overall architectural structure consisting of the conceptual components. The method details the control of the reasoning process to achieve a task. It also describes both the decomposition of the general tasks into subtasks and the coordination of those subtasks to achieve the required result (control flow)Gomez-Perez *et al.* (2004). The UPML, however, doesn't define a set of program elements to specify a method's control flow.

In the following, we describe the detail formalization of our generic approach to integration based on UPML.

Problem-solving-method Many approaches have traditionally modeled the internal structure of software components as a process Grüninger et Menzel (2003) carrying out a set of actions to execute the process. This idea breaks the process (or service) into activities whose interactions are modeled as workflow patterns which basically describe the coordination of those activities during the process execution. Because of this, some researchers have proposed process-based languages such as BPEL (Business Process Execution Language) and OWL-S (for Semantic Web Services). These languages specify the internal structure of service using a predefined set of workflow-such as patterns (sequence, choice, parallel split, and so forth). This approach's main drawback is its lack of an explicit, declarative decoupling between its structural description (how) and the process's functional features (what). In fact the functional features is linked directly to the parameters used in the process's internal structure. So, the process is designed to carry out a particular operation (for example, `to book`) in a particular domain (such as `flight booking`). With this approach, reusing processes among domains becomes difficult, and component or service integration in a project, must be programmatically solved. For example, a component that deals with `theater booking` shares some operations with a `flight booking` component (check credit card, select seat, confirm booking, and so forth). Processes that execute such operations should be quasi reusable among both components, and we must differentiate between the description of those operations and how they are solved.

Domain Model The domain model consists of three elements: meta-knowlcdge.domain knowledge and properties itself. The domain knowledge of the domain model is the knowledge base of the domain that is necessary not only to define the task in the given application domain and but also to carry out the inference steps of the

chosen problem solving method. For composition tasks, the component repository is the knowledge base containing all knowledge about components developed in the diverse domains. The integration repository is consisted of each integration description using the composition description language based on integration ontology

Adapter The adapter specifies mappings among a PSM's knowledge components, adapting a task to a method and refining tasks and methods to generate more specific components. So, adapters can achieve reusability at the knowledge level because they bridge the gap between a PSM's general description and the particular domain in which it's applied. All necessary information including the goals of the used components and their required interconnection can be specified in the integration request specification.

Task Specification The integration task produces an integration components list, which represents the assembled composite product, following a job order asstaled in integration request specification. The integration request specification and the integration components list are enough to specify input and output *roles* of the *task* specification. The practical requirements and assumptions are already considered in the integration ontology and the integration request specification. The integration request specification as the input role of the task specification is the description of overall architectural structure, which consists of the conceptual components virtually defined by the integration requester which generally is the developer. As a simple example of integration request specification, the component integration task "find a book price and convert it into other currency unit such us Euro according to the current exchange rate" will be specified as in Figure. The integration request specification contains not only the conceptual information to discover the consistent components but also the binding information between components.

The integration request specification contains the **binding information** among components. In order to discover the appropriate components, each component description must specify locally a domain ontology. This local ontology which is named integration ontology can be defined as lightweight ontology with simple keyword hierarchy or heavyweight ontology representing sophisticated axiomatic features to provide sufficient knowledge about a component. The integration ontology is very useful to understand the characteristics of components from different domain and provide the essential knowledge for the reasoning in PSM. From the connection specification, the necessary mediations are so easily deducible that the mediators are automatically inserted at the proper position by PSM.

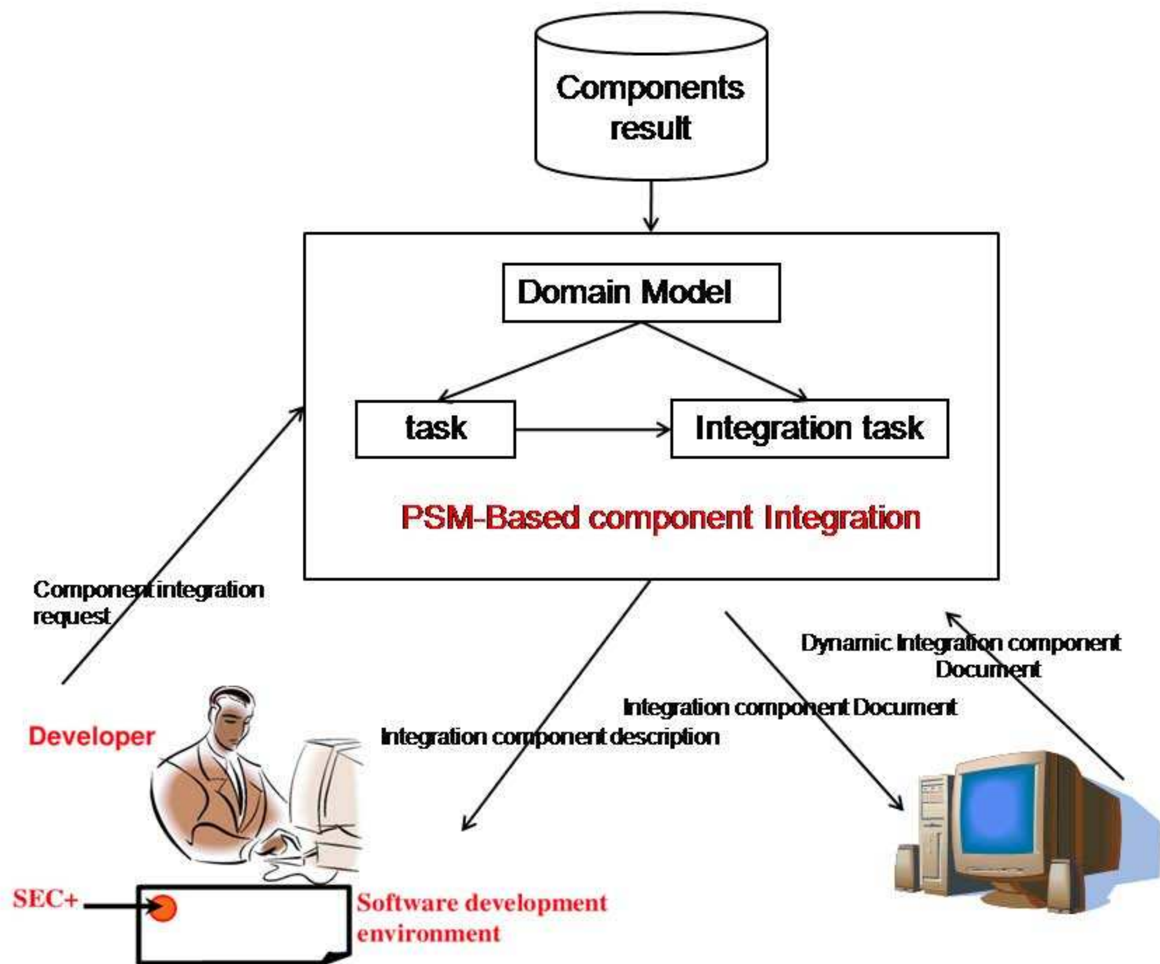

Figure 4.14: The integration result

The PSM-based integration component can be defined as Services that are interconnected using mediators by means of PSMs in a **standard independent fashion** with the aim of offering a solution in the form of functional components, based on its operational requirements. The aim of integrating software components in the PSM-based approach is to produce a solution in the form of functional components or products, by integrating, mixing, or connecting components according to its functional description, in a domain independent fashion, and guaranteeing zero-upadating integration. The PSM-based integration component represent an initiative towards that understands integration as a generic PSM. To do so it provides a run-time environment and an UPML based architecture for integration components. They intend to facilitate the means for the task-driven automatic discovery, integration and execution of components.

Finally in the production part the different WSDL that represent the integration are gener-

ated together with the wrappers that represents the glue among components, in a standard language independent way. Later the result of this production can be translated to any workflow language, i.e. BPEL4WS, BPML/WSCI, etc. Figure 4.15 shows the results of the production phase for the " matrix sum and calculate it determinant ".

MatrixSum component Input roles :Matrix1, Matrix2, author,…. Output roles :Matrix	<integrate_service> <sequence> <item type= "service"> service>http://redcad.tn/MatrixSum.wsdl/</service> <input_binding> ..... </input_binding> <output_binding> ..... </output_binding> </item>
selector input roles: Matrix1, Matrix2, oper_name output roles: Matrix	<item type= “selector”> <service> http://myweb.tn/selector.wsdl/</service> <input_binding> ..... </input_binding> <output_binding> ..... </output_binding> </item>
simple refiner input roles: Oper_name output roles: Matrix	<item type= “refiner”> <service> http://myweb.tn/refiner.wsdl/</service> <input_binding> ..... </input_binding> <output_binding> ..... </output_binding> </item>
MatrixDeterminant component input roles: Matrix,….. output roles: det	<item type= “web_service”> <service> http://enis.tn/MatrixDeterminant.wsdl/</service> <input_binding> ..... </input_binding> <output_binding> ..... </output_binding> </item> </sequence> </integrate_service>

Figure 4.15: Production result

4.5.1.5 Integration ontology construction

Components description ontology is established by extracting semantic information on the actions and objects of components. Figure 4.16 is the relational of established component ontology. component ontology is composed of semantic descriptions of components such as actions and objects, which are domains to which actions are executed, functional and non-functional descriptions of components such as the precondition on input and the post-condition of output, and other information required for describing components.

As shown in 4.16, if semantic annotation is provided for an email sending service using ontology, the action is ’send’ and the object is ’email’. The functional description includes also information on the location of the email transmission service, service provider and input/output parameters for the execution of the component. The modeled component description ontology is described using RDF and coverted to OWL, and ontology input is described using Protégé-2000. In Figure 4.16, rectangles are classes and arrows

are properties. Classes which are Extracted and instances are entered as inputs. The instance of ComponentType class, which describes the type of component to be integrated, should have Atomic, Composite, Output-Matching-Service or Input-Output-convertor as its value. The Atomic means that the component is not a composite but a single service, and the Composite type means a composite component created from the integration of components. In addition, Output-Matching-Service and Input-Output-convertor are service types used in matching parameters. Output-Matching-Service is a service that extracts what it needs from component output parameters, and Input-Output-convertor is a service that converts the output parameter type of a selected component to the input parameter type of a service to be extracted.

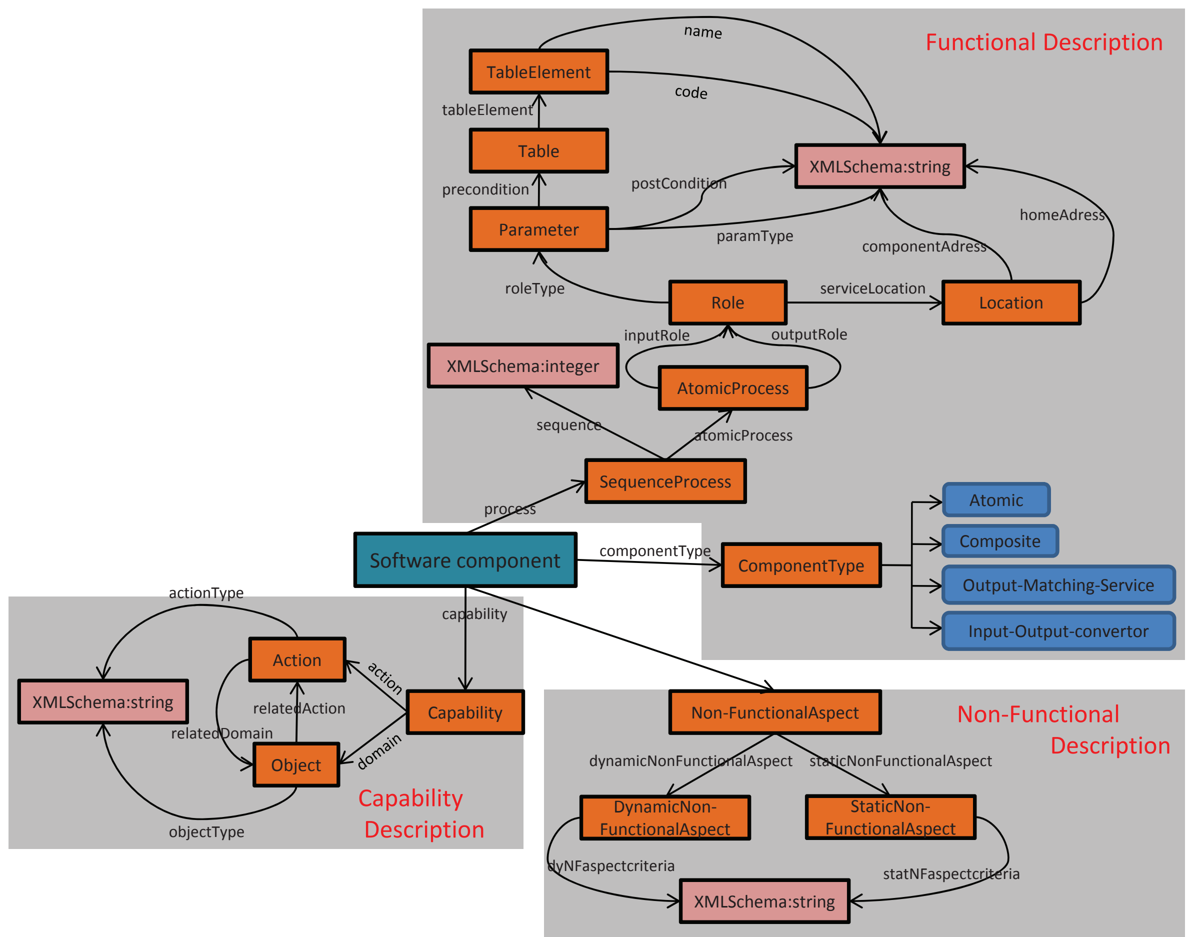

Figure 4.16: Ontology construction

4.5.2 The third version of the search engine: SEC++

When a selected component will be integrated in the current work, two things should be considered. One is the type of the collision Salim *et al.* (2007) in the matching of different types of data. For example, type collision happens when a 'double' type output parameter of a component is matched with a 'string' type input parameter. The other thing to be considered is how to extract input parameters when a service has two or more output results. This problem does not need to be considered if all the results of the component match. However, if only some of returned output results match, we must use a process to extract

them. This work implemented Input-Output-convertor for conversion between two different types to solve the type collision problem, and Output-Matching-Service for extracting necessary output parameters. Figure 4.17 shows the steps of converting 'float' type output parameter C_1O_1 of component C_1 to 'double' type input parameter C_2I_1 of component C_2 through Input-Output-convertor in matching parameters between two different components. For example, when composing exchange component C_2 that exchanges currencies by receiving the output parameters exchange rate (float type), exchange rate (double type) and exchange amount (double type) of component C_1 that calculates exchange rate between two currencies, type collision happens as in figure 4.17. Here, the problem is solved as Input-Output-convertor converts C_1O_1 (float type) of C_1 to C_2I_1 (double type) of C_2. Conversion from string type (not numeric string type) to int or float type is not allowed, so is considered as an exception.

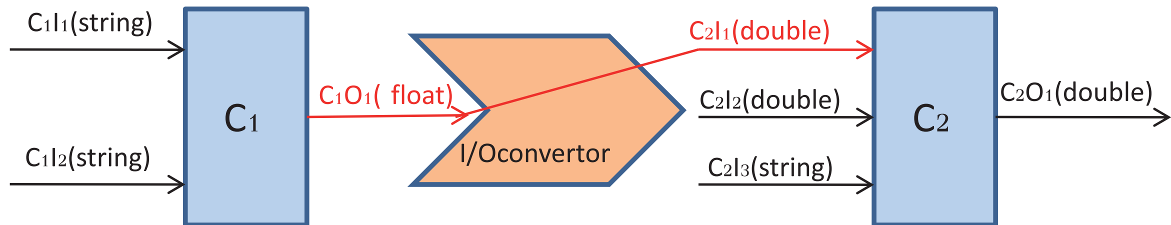

Figure 4.17: Input-Output Convertor

Figure 4.18 shows the steps of extracting only C_1O_1 and C_1O_4 of output parameters C_1O_1 , C_1O_2 , C_1O_3 and C_1O_4 of component C_1 and matching them with input parameter C_1I_1 and C_2I_3 of component C_2. For example, when integrating exchange service C_2 that exchanges currencies by receiving exchange rate (double type) and exchange amount (double type) and ebay book component C_1 that receives input parameters book title as string type and author name as string type and returns output parameters publisher as string type, date of publishing as string type and price as float type. This process has not only parameter extraction problem but also the type collision. In this case, before the execution of Output-Matching-Service, Input-Output-convertor is executed first to extract parameters from the outputs of C_1 to be matched with the input parameters of C_2. In figure 4.18, only C_1O_1 and C_1O_4 of ebay book discovered component C_1 are matched with C_2I_1 and C_2I_3 of exchange service C_2. Thus, Output-Matching-Service is executed to extract C_1O_1 and C_1O_4 among the four output parameters. Because C_1O_1 and C_2I_3 are identical in type they do not need the execution of Input-Output-convertor, but C_1O_4 (book price:float type) and C_2I_3 (exchange amount: double type) requires the execution of Input-Output-convertor for their matching. As mentioned above, Input-Output-convertor is executed after Output-Matching-Service.

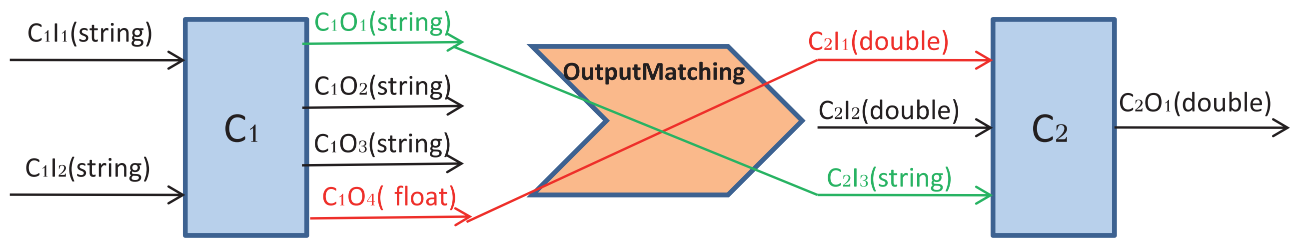

Figure 4.18: Output-Matching-Service

4.5.3 Lifecycle of Constituent Component

The state transitions of a constituent component(see Figure 4.19)is a standard lifecycle of a service component. The detailed descriptions of states and state transitions are described as follows:

- Waiting for Execution: The constituent component is capable of accepting and processing requests (i.e. the component is available) in this state. When component requests are coming, the component will transit to "Component Execution" state.
- Component Execution: In this state, the constituent component will process requests, perform tasks, and send back the component results. Generally, if the execution is performed successfully, the component will transit back to "Waiting for Execution" state. Otherwise, the component will transit to "Unable" state if the event of component termination is received.
- Unable: The component is not capable of accepting any requests (i.e. the service is not available) in this state.
- If the component reaches "Unable" state, the component will transit to the final state (dead state) automatically.

4.6 Conclusion

In this chapter we have presented our approach which is divided in three steps. First we have developed the atomic component discovery process and we have described the discovery ontology that specifies functional and non functional features. Second we have developed the composite component discovery process and the shared ontology used when there is not an appropriate atomic component that response to developer's needs. Finally

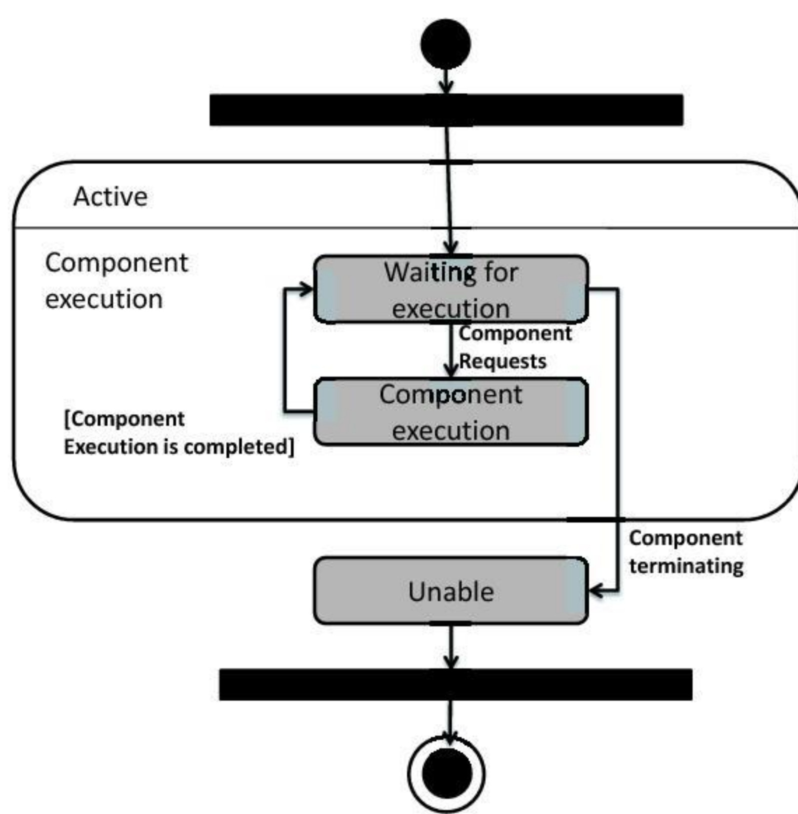

Figure 4.19: State Transition Model of Constituent Component

the integration process which integrate the discovered component into current project.We used the integration ontology to describe the problem solving method (PSMs) used to specify the component's structural features. Those ontologies deliver re-usable components and help the developer to integrate the selected component into the current work. In fact with the integration ontology we guarantee that the selected component is the best adaptable which increase the adaptability.

5

Experimental evaluation of SEC+

5.1 Introduction

Now that the details of the implementation have been described, it is time for a proper test of SEC+. This chapter contains details on the method of evaluation, the selected components and the results gained from the evaluation. In order to measure the retrieval performance, a selection of queries and expected responses were created. This enabled precise measurements of how good the system was at returning the expected results. The goal of this chapter will be to evaluate the performance of the system by mesuring the criteria `Recall` and `Precision` and find out if the problem in chapter 1 has been solved satisfactory. Retrieval performance experiments were performed both with and without the semantic distance and the subsumption notion applied respectively on SEC and the newer version of SEC(SEC+). In this chapter we introduce also three applications scenarios illustrating the application of the approach introduced in the previous Chapter. A first example deals with the case of a mapping an instance of the discovery ontology into integration ontology. In the second example we considers an individual instance of a discovery ontology into shared ontology. In the last example we describe a shared ontology implemented in the corporate mathematical services domain.

5.2 Evaluation

In this section, we report on two sets of experiments: component discovery with subsumption mechanism only implemented in SEC, and with both the subsumption mechanism and the semantic distance between components methods names and specified name of method in the query implemented in SEC+.

```

<?xml version='1.0' encoding='UTF-8'?> <!DOCTYPE rdf:RDF [
     <!ENTITY rdf 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
     <!ENTITY rdf_ 'http://protege.stanford.edu/rdf'>
     <!ENTITY rdfs 'http://www.w3.org/2000/01/rdf-schema#'>
]> <rdf:RDF xmlns:rdf="&rdf;"
     xmlns:rdf_="&rdf_;"
     xmlns:rdfs="&rdfs;">
<rdf_:Component rdf:about="&rdf_;Khalil_component"
     rdf_:Authors="Khalil"
     rdf_:Location="D:\lib_"
     rdfs:label="Khalil_component">
    <rdf_:Comp_Type rdf:resource="&rdf_;Component_type2"/>
    <rdf_:name rdf:resource="&rdf_;Method1"/>
    <rdf_:Sta_NF_attributes rdf:resource="&rdf_;Static_NFA1"/>
    <rdf_:Dyn_NF_attributes rdf:resource="&rdf_;Dynamic_NFA1"/>
</rdf_:Component> <rdf_:Level rdf:about="&rdf_;Level2"
     rdf_:Level_id="Medium"
     rdfs:label="Level2"/>
<rdf_:Level rdf:about="&rdf_;Level3"
     rdf_:Level_id="Low"
     rdfs:label="Level3"/>
<rdf_:Component_Type rdf:about="&rdf_;Component_type1"
     rdf_:Type_Comp="Corba"
     rdfs:label="Component_type1"/>
<rdf_:Component_Type rdf:about="&rdf_;Component_type2"
     rdf_:Type_Comp="COM"
     rdfs:label="Component_type2"/>
........ </rdf_:Component> <rdf_:Dynamic_NF_Aspect
rdf:about="&rdf_;Dynamic_NFA1"
     rdfs:label="Dynamic_NFA1"/>
<rdf_:Level rdf:about="&rdf_;Level1"
     rdf_:Level_id="High"
     rdfs:label="Level1"/>
</rdf:RDF>
```

Listing 5.1: library description extract

Each experimentation is based on three measures of components discovering performance which are recall, precision and response time of the search engine. These experimentations are applied on different set of components (62, 125, 500 and 1000). The listing below 5.1is a part of the used library description (The detail is in the Annex). The test query set contains 10 queries and applied to three development environments (Delphi,

Eclipse and Jbuilder). Among them, 4 queries were created by us, 6 were chosen from questions frequently asked in newsgroups for development environments (see table 1).

Qi	Description
Q1	Linear system resolution
Q2	Sum of matrix
Q3	Matrix symmetry
Q4	Matrix inverse
Q5	Sorting table
Q6	Resolution of second degree equation
Q7	Matrix determinant
Q8	Electronic payment
Q9	Matrix transposee
Q10	Table fusion

Table 5.1: Queries description

Recall is defined as the ratio of the number of correct solutions retrieved to the number of correct solutions that exist, indicates the ability of the system to retrieve all relevant components. Ideally, recall should be high, meaning solutions should not be missed. *Precision* is defined as the ratio of correct solutions retrieved to the total number of results retrieved. High precision is the result of retrieving few irrelevant or invalid solutions, it indicates the ability of the system to present only relevant components Morel et Alexander (2004).

5.2.1 Experiments using SEC

In the experiments that use SEC, we match components descriptions using the subsumption mechanism. We use the used rate and the non functional features to filter the selection.

Table 1 shows the average of the Recall and Precision corresponding to the 10 queries with different set of number of components. All queries have a matching result, only the query Q8 has no results because our ontology doesn't contain the component(s) that feel exactly or approximately the query specification.

SEC has a good recall in fact there is five queries with a recall higher than 70

The SEC achieves a precision of 53.78% at 68.44% recall on average on this set of experiments.

Qi	Recall	Precision
Q1	83%	100%
Q2	40%	100%
Q3	40%	70%
Q4	25%	50%
Q5	80%	100%
Q6	50%	50%
Q7	50%	33%
Q8	0%	0%
Q9	66%	80%
Q10	50%	33%
Average	**53.78%**	**68.44%**

Table 5.2: Recall and precision of SEC - Bad query (Q8) filtered

5.2.2 Experiments using SEC+

These experiments use SEC+, we match components descriptions using the subsumption mechanism and the semantic distance. We enrich the query by adding the pre-condition and the effect in the functional aspect information and we affect dynamic weight for each specified non functional features.

Qi	Recall	Precision
Q1	100%	75%
Q2	85%	83.33%
Q3	100%	100%
Q4	70%	71%
Q5	100%	75%
Q6	75%	100%
Q7	85%	100%
Q8	0%	0%
Q9	66.66%	100%
Q10	100%	66.66%
Average	**86.85%**	**76.41%**

Table 5.3: Recall and precision of SEC+ - Bad query (Q8) filtered

We use in these experiments the same queries tested in SEC. All queries have a matching result, only the query Q8 haven't any results because our ontology doesn't contain the component(s) that feel exactly or approximately the query specification.

On average, SEC+ that uses both subsumption mechanism and the semantic distance

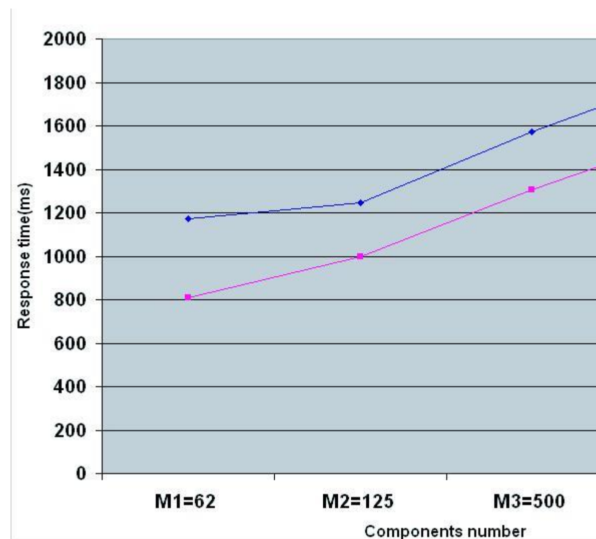

Figure 5.1: Comparison between SEC and SEC+

achieves a precision of 76.14% at 86.85% recall. Compared to performance of experiments that use SEC , precision is increased by 7.97% from 68.44% and recall is increased by 33.07% from 53.78%. Both precision and recall improved significantly compared to the results obtained with SEC.

For all the queries, SEC+ was able to maintain very high precision and recall (almost always 80 percent), see table 5.3. High precision was the result of using an ontology that describes not only the inputs/outputs and methods names of components but also the precondition and effect. To improve precision in SEC+, we use a weight for each specified non functional feature. SEC+ has a good precision compared to other search engines like in Flexible Interface Matching (FIM) Wang et Stroulia (2003) (see table 5.4). FIM was better than many search engines such Larks Sycara *et al.* (2002). High Recall, was the result of using not only Subsumption mechanism such in SEC but also semantic distance that use Wordnet hierarchy. The semantic distance retrieve all components which are synonym, hypernym and hyponym to the specified query. SEC+ improve the reuse

of the set of components and offer more solution for the developers.

SEC+ is not fast compared to SEC. This is du to the fact that SEC does not access to the Wordnet and does not calculate the semantic distance. The difference between the response times is about 200 ms. We consider that if the set of components is more than 500, the difference of response time between SEC and SEC+ remains negligible (see figure 5.1).

Also SEC+ is a good tool for the beginners, in fact it helps them with the discovery

Search engine	Recall	Precision
SEC	53.78%	68.44%
FIM	90%	61,5%
SEC+	86,85%	76,41%

Table 5.4: The average of the Recall and the precision of SEC,FIM and SEC+

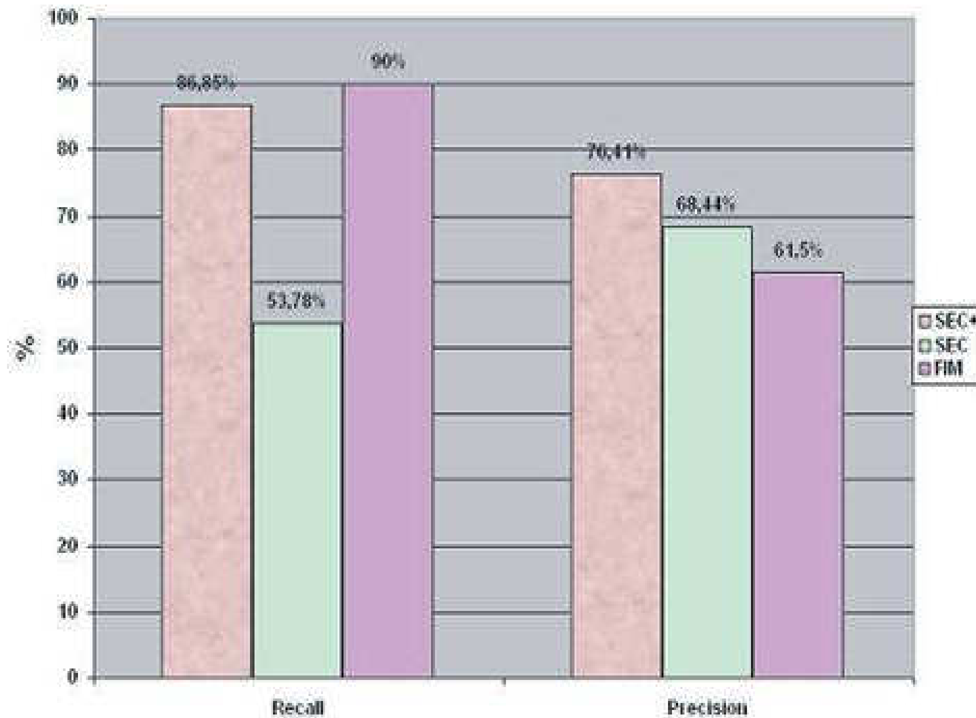

Figure 5.2: Comparison between SEC, FIM and SEC+

ontology to have an idea about the components in the repository and with the integration ontology to construct a project.

5.3 Application scenarios

5.3.1 Mapping the discovery ontology into integration ontology

In this section, we present a prototype system for mapping a discovery ontology of mathematical service into integration ontology, which is developed based on the proposed ontology-supported software component integration Sofien *et al.* (2010a).

We use the matrix operations as a domain for the discovery ontology. There are many

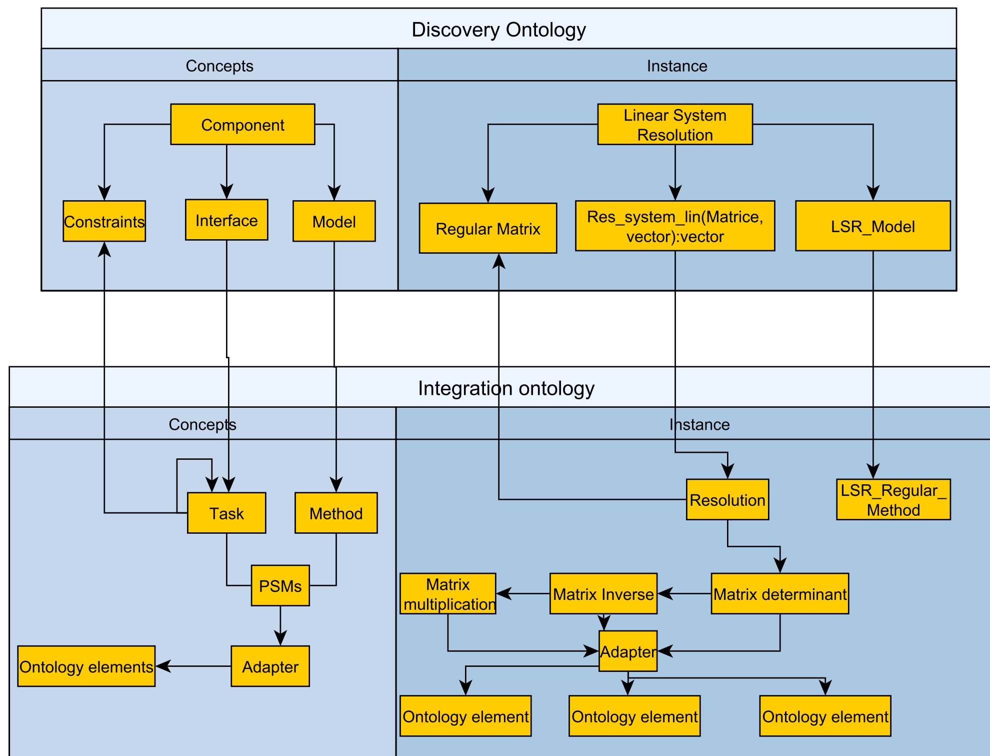

Figure 5.3: mapping the discovery ontology individual instance into integration ontology

operations can be applied to matrix such as linear system resolution, Hill cipher,etc. In the Figure 5.3 we illustrate in the discovery ontology a linear system resolution instance. A linear system resolution is a general system of m linear equations with n unknowns can be written as

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n & = & b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots + a_{2n}x_n & = & b_2 \\ & \ldots = & \ldots \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \ldots + a_{nn}x_n & = & b_n \end{cases}$$

Here x1,x2,...,x3 are the unknowns, a11,a12,...,amn are the coefficients of the system, and b1,b2,...,bm are the constant terms.

Often the coefficients and unknowns are real or complex numbers, but integers and rational numbers are also seen, as are polynomials and elements of an abstract algebraic structure.

We use RDF language to describe the discovery ontology. One step further, the elements in the discovery ontology link to the corresponding properties in the integration ontology.

In our example the concept *Res_sys_lin(Matrix, Vector):Vector* in discovery ontology corresponds with the *Resolution* Tasks concept in the integration ontology.The *LSR_Model* in discovery ontology corresponds with the *LSR_Regular_Method* Model concept in the integration ontology.

Given the *Resolution* Tasks that a system should accomplish a PSM is the specification of the functionality of the problem solving behaviour of the system to be built. It is a description of how the functionality can be achieved and how the requirements can be met. In the integration ontology we decompose a task into subtasks, the *Resolution* Tasks is decomposed into two subtasks: the first is *Matrix_inverse*, the second is *Matrix_Multiplication*. We considers that A is a regular matrix the result vector X is equal to $\mathrm{A}^{-1} * b$.

A^{-1} is the result of the *Matrix_inverse* subtask.

$A^{-1} * b$ is the result of the *Matrix_Multiplication* subtask. each subtasks is generated into PSMs. each PSMs constitute generic inference patters which describe the dynamic behaviour of our systems on an abstract level, which abstracts from details concerned with the implementation of the system. PSMs are independent of the domain they are applied in, but specific for the task which has to be accomplished by them.

Adapters are used to mediate between problem definitions, domain knowledge, and problem-solving methods.

5.3.2 How to implement integration ontology

Problem-solving methods provide reusable architectures and components for implementing the reasoning part of knowledge-based systems. The Unified Problem-solving Method description Language UPML Fensel *et al.* (1999) has been developed to describe and implement our integration ontology components to facilitate their semiautomatic reuse and adaptation. In a nutshell, UPML is a framework for developing knowledge-intensive reasoning systems based on libraries of generic problem-solving components.

We used Protégé-2000 which contains the plugin editor for UPML specifications. Protégé allows developers to create, browse and edit domain ontologies in a frame-based representation, which is compliant with the OKBC knowledge model. From an ontology, Protégé automatically constructs a graphical knowledge-acquisition tool that allows application specialists to enter the detailed content knowledge required to define specific applications. Protégé allows developers to custom-tailor this knowledge-acquisition tool directly by arranging and configuring the graphical entities on forms, that are attached to

each class in the ontology for the acquisition of instances. This allows application specialists to enter domain information by filling in the blanks of intuitive forms and by drawing diagrams composed of selectable icons and connectors. Protégé-2000 allows knowledge bases to be stored in several formats, among which a CLIPS-based syntax and RDF.

Problem-solving methods Tasks Each problem-solving method in an UPML specification can be mapped to a class implementing this problem-solving method. The subtasks of this problem-solving method are mapped to methods of the problem-solving method class. A problem-solving method communicates with other components via roles which are realized by bridges when configuring the whole problem-solving method. Our running example provides a specification for a generic search problem-solving method. This problem-solving method has one input role input, one output role output, and the intermediate roles node, nodes, and successor nodes.roles are translated into instance variables of the search class. Input- and output roles communicate with other components using a setRole and getRole-method, implemented from the general supperclass PSMComponent. The subtasks of the UPML specification are translated into methods of the PSM class. They also communicate with other PSMComponents using the methods getSubTaskRole, setSubTaskRole and excecuteSubtasks, which are implemented from the supperclass PSMComponent. Please note, that nothing is said here, how this subtasks are defined. The execution is delegated to adapters and the configuration can be done while designing the problem-solving method.

Ontologies Domain Models Ontologies are mapped to an ordinary class hierarchy, which defines the basic terminology used in the domain model and the problem-solving method. In the example above, the PSM ontology has to define node, nodes, object, objects etc. Notice, that these ontologies can be application-specific and that details of the actual definition of these classes are not used inside the problem-solving method. So the details of the data structure definitions can be implemented in an application dependent manner.

adapter enable the basic communication infrastructure between several problemsolving method components (providing the subtask-PSM-mapping) and the domain. Because it has to be the most flexible part of the specification (it has to handle all incompatibilities between problem-solving method components) we can only formulate weak requirements. However, a bridge has to at least provide a common interface, such that problem-solving methods and bridges can be plugged together in a flexible way. The API provided by this interface can be structured into two

groups: the first set of methods deals with the configuration of a problem-solving method. The second group of methods handles the execution of subtasks and set handling of roles. A bridge is usually domain- and problem-specific, a general type of bridge is often useful and sufficient. This kind of adapter just performs basic mappings. This adapter can be configured at runtime.

5.3.3 Mapping the discovery ontology into shared ontology

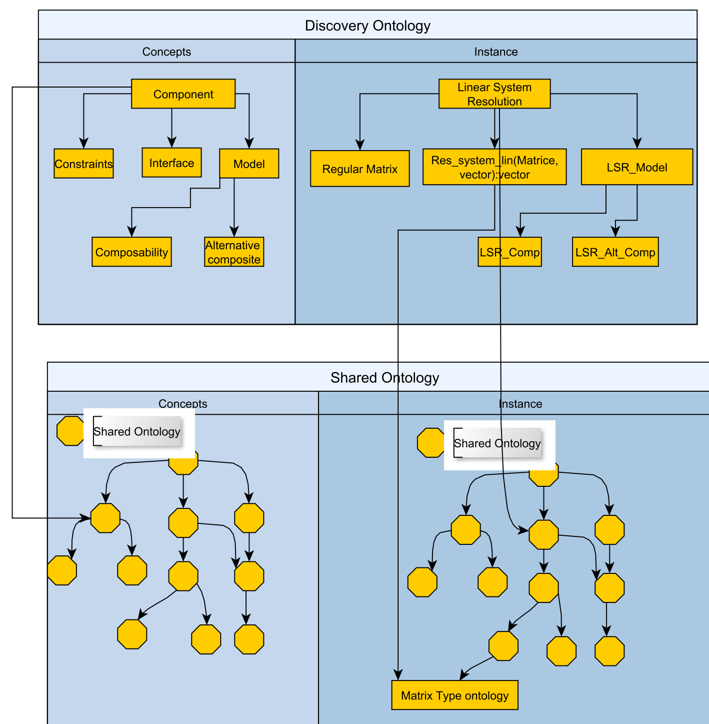

Figure 5.4: mapping the discovery ontology into shared ontology

In this section, we present a prototype system for mapping a discovery ontology of mathematical service into shared ontology, which is developed based on the proposed ontology-supported software component composition Sofien *et al.* (2010b).

Our prototype system employs W3C-recommended standards (i.e., RDF+OWL) for semantic description and ontological engineering. The software utilized for this task is Protégé Protégé 3.4.4. At the discovery ontology description level, the prototype system translates component descriptions and then adds more semantics in the component model class. The composability property of the component model class can have values denoting possible ways for component composition.

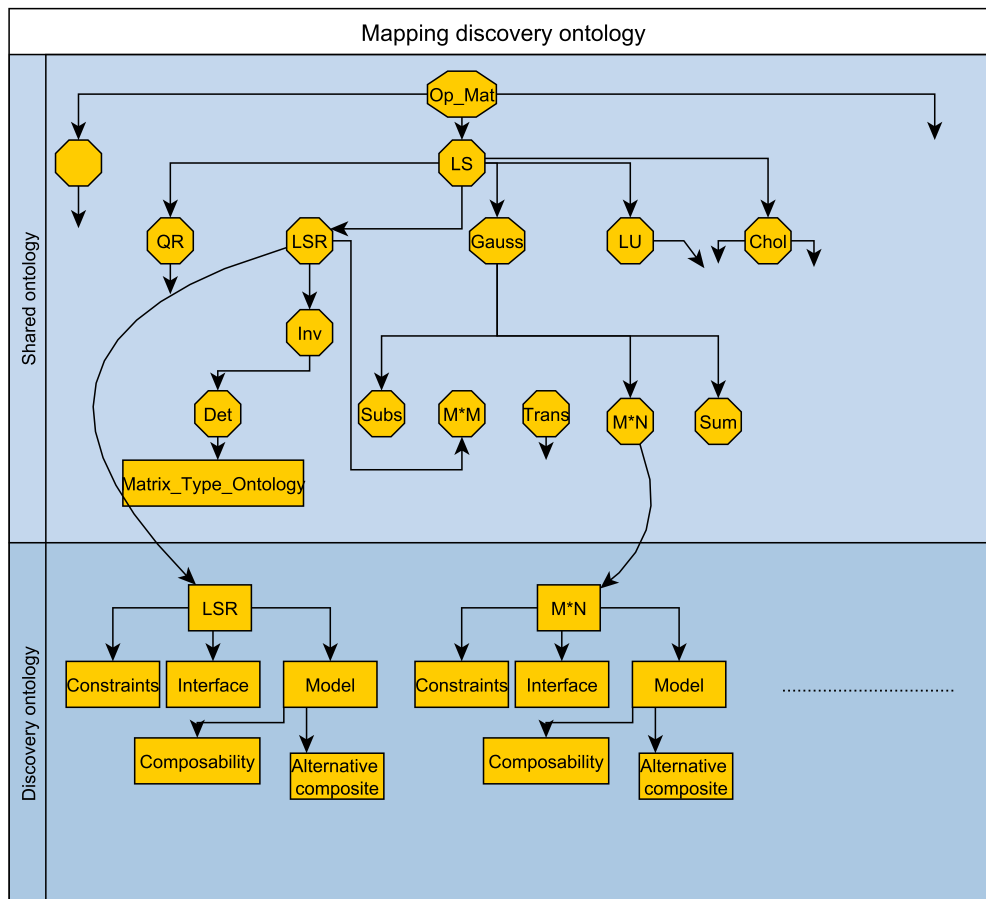

Figure 5.5: mapping the discovery ontology individual instances into shared ontology

Taking the Linear system resolution component as an example, its composability contains a list of possible parameter flows (from inputs to outputs), each of which can be a part of an alternative path in a composite component (Complex Matrix $\longrightarrow$ $ComplexMatrix), (ReelMatrix \longrightarrow RealMatrix)$.

Another way to exploit composability is first to attach composability to other properties with concrete meanings, then associate composability with composition rules.

all individual discovery ontologies are mapped together, appearing as nodes or subclasses in the shared ontology (see Figure 5.4). For example, the LSR component is a subclass of the LS class (see Figure 5.5. The prototype can extract some metadata from individual discovery ontologies and map into the shared one. After organizing those base Matrix Operation services into a shared knowledge repository, the prototype adds other concepts relevant to Matrix operations, either domain-specific or generic, such as Type.

Scenario Calculate the determinant squared of the vector X which is unknown in

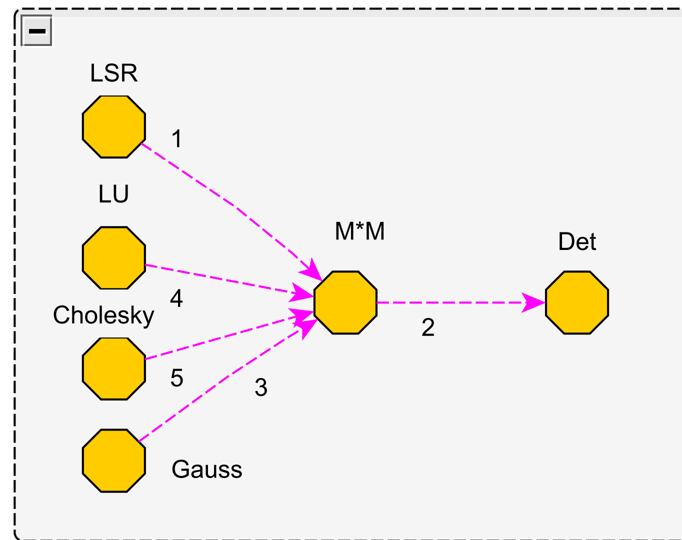

Figure 5.6: The scenario

linear equation

This question is exploratory in nature and cannot be answered by a single query. There are many possible method to resolve linear equation, such as Gauss, LU decomposition, or Cholesky decomposition. The question can be addressed by the ontological heuristics capability of our system. The starting point is the key word squared. From the shared ontology, the prototype system learns that determinant is an output of the Det component. The prototype can select candidate components based on different criteria, including query constraints, business rules, component consumption cost, and so forth.

For example, the LSR component guarantees that the Matrix A is regular. Thus, it is selected first and supplemented with values of input parameters such as the vector b.

As illustrated in the scenario 5.6, an exploratory mathematical question can be investigated through all possible dimensions by performing heuristics along the shared ontology. According to the results of ontological heuristics, an applicable component composition solution can be executed to answer the question. The final answer can come from component execution results along one or more dimensions. Such an ontological heuristics procedure is illustrated in Figure 8. The ontological heuristics paths are displayed at the bottom of the diagram.

The proposed system has theoretical and practical implications for organizational knowledge management. Ontologies are beneficial to knowledge representation, discovery, and sharing, while components facilitate knowledge integration, delivery, and consumption.

However, caution should be taken when extending the framework to other areas. First, we assume that the shared ontology is accepted by all parties within an organization or a community, which is not always the case in a public domain. Similarly, in a situation where component description and domain knowledge cannot be translated easily into concepts represented in ontologies or ambiguities may arise from the conceptualization, task-specific solutions can be undertaken. But the ideal measure of conceptualization should rely on industry wide standardization.

5.4 Conclusion

This chapter introduced the standard methods for evaluating SEC. In order to assess the performance of the search engine, a collection of queries with relevant responses were created. Our results are encouraging, in fact they are a great improvement over the SEC and other retrieval systems. Both SEC+ precision and recall improved significantly compared to the results obtained with SEC due to the the integration of the subsumption mechanism and the semantic distance in the matching algorithm.

To demonstrate the benefits of the proposed composition and integrated ontology, we have applied it to the Matrix operations components and provided a solution to component composition in that domain.

6
Conclusion

In this thesis, we presented the foundations of our discovery and integration approach that allows developers to discover and integrate the appropriate component (or composite component) in the current system engineering. As such, it introduces several features currently missing in current works in software component description, discovery and integration:

1. the use of non functional attributes in the query description and in the component specification.
2. the development of a portable search engine which can be used in several development environment, function that calculate the semantic distance between terms.
3. reasoning approach to validate component internal structure Description
4. The development of two components to solve collision problem in the integration process.

The core of component discovery process is the discovery ontology and the semantic matching. The discovery ontology describes the functional and the non functional aspect

of the software component software. We classify non functional attributes into dynamic/static and indpendant/dependant domain. The non functional attributes serves as a basis for refining the components selection and as adaptation criteria in the integration setp. The semantic matching algorithm is based on a function which calculate the semantic distance between terms and a subsumption notion between components input/ouput data types and query input/output data types.

This thesis extends the search to components integration of the selected component in the current developed project by providing an innovative approach to describing the components internal structure and a mechanism for types matching.

Another main contribution of this thesis is to improve the reuse by selecting a composite component if no individual description component is found in the discovery ontology. We have proposed a shared ontology-supported software component composition. The semantic enrichment shows superiority for automated and on-the-fly component composition. As demonstrated in the usage scenario, new lists of candidate components are generated along the course of problem solving.

We further take advantage of describing the component internal structure by using a Problem Solving Method to integrate the component which has a flexible method. What flexibility in terms of reconfiguration of the method for modified component required. Methods that provide clear models for problem solving help method designers to communicate results, and help developers to understand how methods operate, and how methods can be configured to perform new tasks. Given a repository of such methods, the developer can select an appropriate method, configure it to perform current application tasks.

When a selected component having the best flexible method will be integrated in the current work, two features should be considered. One is the type of the collision in the matching of different types of data. To alleviate this problem we implement Input-Output-convertor for conversion between different types to solve the type collision problem, and Output-Matching-Service for extracting necessary output parameters.

In addition, several benefits are offered by the proposed approach:

- Increased Availability: By means of attaching several candidate components for future integration.
- Increased Usability: The composite component can bind the best available components that fit the end developers needs by interface and semantic component matching through the SEC++.

- Increased adaptability: By means of the component result response to the environment constraints and the developer request. In fact with the integration ontology we guarantee that the selected component is the best adaptable.

Finally, our enhanced search engine SEC++, the discovery ontology, the shared ontology and the integration ontology provide a framework to discover and integrate a component' or a composite component) that feet the developer needs and the environment constraints, and resolving several frequently encountered problems, such as component adaptability and component reuse.

As with all large frameworks, there are a few major difficulties in the implementation of the integration process. We will discuss them here in the context of potential future work that would help resolve these difficulties.

Our future research plan will focus on threefold:

In short term we plain to evaluate the newer version SEC++ and specifically the integration process.

In middle term we plan to develop More efficient ontology structures and searching algorithms. In fact the use of ontological heuristics through the shared ontology tree may consume substantial computational resources, especially when the ontology tree grows very large.

In long term we also plain to extend our search engine to discover a components composition, if no component satisfies the developer query. We plain to design and implement an assembly technique in creating the composite component which feets the developer query. The assembly technique will manipulate and select the appropriate set of components from components description repository. BPEL can be used to orchestrate the selected components, and we can hence select the best-assembled component as the composed component. Depending on how many matching components are available many component assemblies are possible. Therefore, after all the possible assemblies are generated, the assembled components are ranked based on their non functional attributes and the flexibility of the corresponding assembled methods in integration ontology.

Our experimentation highlights the main advantages of our approach.

Fist it improve the precision by using an ontology that describes not only the inputs/outputs and methods names of components but also the pre-condition/effect and a weight for each specified non functional feature. The second advantage is the amelioration of the recall criteria. This amelioration was the result of using not only Subsumption mecha-

nism such in SEC but also semantic distance that use Wordnet hierarchy. The semantic distance retrieve all components which are synonym, hypernym and hyponym to the specified query.

Our future experimentation plan will be applied on a well known components repository

Publications

Journal papers

[J1] Sofien Khemakhem, Khalil Drira, and Mohamed Jmaiel. An integration ontology for components composition. *In International Journal of Web Portals (IJWP)*, 2(3), 2010.

[J2] Sofien Khemakhem, Khalil Drira, and Mohamed Jmaiel. An experimental evaluation of SEC+, an enhanced search engine for component-based software development. *In ACM SIGSOFT Software Engineering Notes*, 33(4), 2008.

[J3] Sofien Khemakhem, Khalil Drira, and Mohamed Jmaiel. SEC+: An enhanced search engine for component-based software development. *In ACM SIGSOFT Software Engineering Notes*, 32(4), 2007.

Conference papers

[C1] Sofien Khemakhem, Khalil Drira, and Mohamed Jmaiel. Ontology-based discovery and integration. *In Third International Conference on the Applications of Digital Information and Web Technologies, ICADIWT 2010*, Istanbul, Turkey, 2010.

[C2] Sofien Khemakhem, Khalil Drira, and Mohamed Jmaiel. SEC: A search engine for component based software engineering. *In 21st Annual ACM Symposium on Applied Computing, SAC 2006*, Dijon, France, April, 2006. (acceptance rate: 18%)

[C3] Sofien Khemakhem, Khalil Drira, and Mohamed Jmaiel. CSCLD: A Component for Software Component Library Discovering . *In Cinquièmes Journées Scientifiques des Jeunes Chercheurs en Génie Electrique et Informatique, GEI 2005*, Sousse, Tunisie, 2005.

[C4] Sofien Khemakhem, Mohamed Jmaiel, Abdelmajid Ben Hamadou, and Khalil Drira. Un environnement de recherche et dintègration de composants logiciels. *In the seventh Maghrebian Conference on Computer Sciences, MCSEAI 2002*, Annaba, Algeria, 2002.

Book chapter

[CH1] Sofien Khemakhem, Khalil Drira, and Mohamed Jmaiel. Description, classification, and discovery for software components: a comparative study. *In Modern Software Engineering Concepts and Practices: Advanced Approaches*, ISBN 9781609602154, IGI Global, 2011, 29p.

Bibliography

AALST, W. M. P. V. D., HOFSTEDE, A. H. M. T., KIEPUSZEWSKI, B. et BARROS, A. P. (2003). Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51.

AMMONS, G., BODIK, R. et LARUS, R. J. (2002). Mining specifications. *In POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 4–16, New York, NY, USA. ACM Press.

ANDREWS, T., CURBERA, F., DHOLAKIA, H., GOLAND, Y., KLEIN, J., LEYMANN, F., LIU, K., ROLLER, D., SMITH, D., THATTE, S., TRICKOVIC, I. et WEERAWARANA, S. (2003). Business process execution language for web services version 1.1. *Technical report*.

ARANDA, C. B. (2005). *Development of a Semantic Web Solution for Directory Services*. Thesis, Norwegian University of Science and Technology, Faculty of Information Technology, Mathematics and Electrical Engineering, Department of Computer and Information Science.

ATKINSON, S. et DUKE, R. (1995). Behavioural retrieval from class libraries. *in Proceedings of the Eighteenth Australasian Computer Science Conference*, *17(1)*:13–20.

BACK, R. et WRIGHT, J. V. (1998). *The Refinement Calculus: A Systematic Introduction*. Springer-Verlag.

BARNES, J. (2003). *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley.

BARTUSSEK, W. et PARNAS, D. L. (1978). Using assertions about traces to write abstract specifications for software modules. *Proceedings of the second conference on European cooperation in informatics, Springer-Verlag*, pages 111–130.

BASTIDE, R., SY, O. et PALANQUE, P. (1999). Formal specification and prototyping of corba systems. *ECOOP'99, Springer-Verlag*, pages 474–494.

BRAGA, R., MATTOSO, M. et WERNER, C. (2001). The use of mediation and ontology technologies for software component information retrieval. *International Symposium on Software Reusability, Toronto, Ontario, Canada*.

CALLAN, J. P., CROFT, W. B. et HARDING, S. M. (1992). The INQUERY retrieval system. *In in Proceedings of the Third International Conference on Database and Expert Systems Applications*, pages 78–83, New York, NY, USA. Springer-Verlag.

CANAL, C., FUENTES, L., TROYA, J. M. et VALLECILLO, A. (2000). Extending corba interfaces with p-calculus for protocol compatibility. *TOOLS'00, IEEE Press*, *19(2)*: 292–333.

CARPINETO, C. et ROMANO, G. (1994). Dynamically bounding browsable retrieval spaces an application to galois lattices. *In Proceedings of RIAO 94: Intelligent Multimedia Information Retrieval Systems and Management*.

CARPINETO, C. et ROMANO, G. (1996). A lattice conceptual clustering system and its application to browsing retrieval. *Machine Learning*, 24(2).

CARPINETO, C. et ROMANO, G. (2000). Order-theoretical ranking. *Journal of the American Society for Information Science*, 51(7):587–601.

CHEN, H. (1995). Machine learning for information retrieval: Neural networks, symbolic learning and genetic algorithms. *Journal of the American Society for Information Science (JASIS)*, 46(3):194–216.

CHEN, H. et KIM, J. (1995). GANNET: a machine learning approach to document retrieval. *Journal of Management Information Systems*, 11(3):7–41.

CHENG, B. et JENG, J. (1997). Reusing analogous components. *IEEE Transactions on Knowledge and Data Engineering*, *9*(2):341–349.

CHIRALA, R. C. (2004). *A Service-Oriented Architecture-Driven Community of Interest Model*. Thesis, Department of Computer Science and Engineering Arizona State University.

CHIU, S. (1996). Method and software for extracting fuzzy classification rules by subtractive clustering. *Fuzzy Information Proc. Society, Biennial Conference of the North American*, pages 461–465.

CICALESE, C. D. T. et ROTENSTREICH, S. (1999). Behavioral specification of distributed software component interfaces. *IEEE Computer*.

CLIFTON, C. et WEN-SYAN, L. (1995). Classifying software components using design characteristics. *Proceedings. The 10th Knowledge-Based Software Engineering Conference (Cat. No.95TB100008)*, pages 139–146.

CORBETT, J. C., DWYER, M. B., HATCLIFF, J., LAUBACH, S., PASAREANU, C. S., ROBBY et ZHENG, H. (2000). Bandera: Extracting finite-state models from java source code. *In Proc. of the Intl. Conf. On Software Engineering (ICSE 2000)*, pages 263–276.

CURTIS, B. (1989). Cognitive issues in reusing software artifacts, software reusability. *ACM Press, New York, NY, 2*.

DAMIANI, E., G.FUGINI, M. et BELLETTINI, C. (1999). A hierarchy-aware approach to faceted classification of objected-oriented components. *ACM Transactions on Software Engineering and Methodology*, *8(3)*:215–262.

DAUDJEE, K. S. et TOPTSIS, A. A. (1994). A technique for automatically organizing software libraries for software reuse. *In CASCON '94: Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, page 12. IBM Press.

DAVEY, B. A. et PRIESLY, H. A. (1990). *Introduction to lattices and order*. Cambridge, UK: Cambridge University Press, 2nd édition.

DEAN, M., CONNOLLY, D., van HARMELEN, F., HENDLER, J., HORROCKS, I., MCGUINNESS, D. L., PATEL-SCHNEIDER, P. F., et STEIN., L. A. (2002). Owl web ontology language 1.0 reference. *http://www.w3.org/TR/owl-ref/*.

DEVANBU, P., BRACHMAN, R. J., BALLARD, B. W. et SELFRIDGE, P. G. (1991). Lassie: A knowledge-based software information system. *Communications of the ACM*, 34(5): 34–49.

EICHMANN, D. et SRINIVAS, K. (1992). Neural network-based retrieval from software reuse repositories. *Neural Networks and Pattern Recognition in Human Computer Interaction*, pages 215–228.

ERDUR, R. C. et DIKENELLI, O. (2002). A multi-agent system infrastructure for software component market-place: An ontological perspective. *ACM SIGMOD*.

ERIKSSON, H., SHAHAR, Y., TU, S. W., PUERTA, A. R. et MUSEN, M. A. (1995). Task modeling with reusable problem-solving methods. *Artif. Intell.*, 79(2):293–326.

ERNST, M. D. (2000). *Dynamically discovering likely program invariants*. Thèse de doctorat, a chercher. Chair-David Notkin.

et AL., M. D. (2002). Web ontology languages (owl) reference version 1.0. *World Wide Web Consortium (W3C)*. www.w3.org/TR/2002/WD-owl-ref-20021112.

EVANS, D., GUTTAG, J., HORNING, J. et TAN, Y. M. (1994). Lclint:a tool for using specifications to check code. *InProceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*.

FARÍAS, A. et Y.GUÉHÉNEUC (2003). On the coherence of component protocols. *Proceedings of the ETAPS Workshop on Software Composition*.

FENSEL, D. et AL. (2003). The unified problem-solving method development language UPML. *Knowledge and Information Systems*, 5(1):83–131.

FENSEL, D., MCGUINESS, D., SCHULTEN, E., KEONG, W., LIM, G. et YAN, G. (2001). Ontologies and electronic commerce. *Intelligent Systems, IEEE*, *16*:8–14.

FENSEL, D., MOTTA, E., HARMELEN, F. V., BENJAMINS, R., CRUBEZY, M., DECKER, S., GASPARI, M., GROENBOOM, R., GROSSO, W., MUSEN, M., ENRIC, PLAZA, E., SCHREIBER, G., STUDER, R. et WIELINGA, B. (1999). The unified problem-solving method development language upml. *Knowledge and Information Systems*, 5:2003.

FERREIRA, V. et LUCENA, J. (2001). Facet-based classification scheme for industrial automation software components. *6th International Workshop on Component-Oriented Programming*.

FETIKE, P. et LOOS, P. (2003). Specifying business components in virtual engineering communities. *Ninth Americas Conference on Information Systems, Tampa, FL*, pages 1937–1947.

FISCHER, B. (2000). Specification-based browsing of software component libraries. *Automated Software Engineering*, *7(2)*:179–200.

FOX, G. C., FURMANSKI, W. et PULIKAL, T. (1998). Evaluating new transparent persistence commodity models: Jdbc, corba pps and oledb for hpc t and e databases. *Proceedings of the International Test and Evaluation Association (ITEA) Workshop on High Performance Computing for Test and Evaluation*, pages 13–16.

FRANCH, X., PINYOL, J. et VANCELLS, J. (1999). Browsing a component library using non-functional information. *Procs. International Conference on Reliable Software Technologies - Ada Europe'99, Santander (Spain)*, pages 332–343.

FUHR, N. et PFEIFER, U. (1994). Probabilistic information retrieval as a combination of abstraction, inductive learning, and probabilistic assumptions. *ACM Trans. Inf. Syst.*, 12(1):92–115.

GOMEZ-PEREZ, A., GONZALEZ-CABERO, R. et LAMA, M. (2004). ODE SWS: A framework for designing and composing semantic web services. *IEEE Intelligent Systems, IEEE Press*, pages 24–31.

GRANTER, B. et WILLE, R. (1996). *formale begriffsanalyse*. Mathematische grundlagen, Berlin, Springer.

GRÜNINGER, M. et MENZEL, C. (2003). The process specification language (psl) theory and applications. *AI Magazine*, pages 63–74.

GROUP, L. P. (2006). Dictionary. URL: http://dictionary.reference.com/.

GRUBER, T. (1993). A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5:199–220.

GRUBER, T. (2008). Ontology. *In* SPRINGER-VERLAG, éditeur : *Encyclopedia of Database Systems*.

HANGAL, S. et LAM, M. (2002). Tracking down software bugs using automatic anomaly detection. *In Proceedings of the International Conference on Software Engineering*.

HENKEL, J. et DIWAN, A. (2003). Discovering algebraic specifications from java classes. *15th European conference on objectoriented programming (ECOOP 2003)*.

HENNINGER, S. (1997). An evolutionary approach to constructing effective software reuse repositories. *ACM Transactions on Software Engineering Methodology*.

HSINCHUN, C. et LINLIN, S. (1994). Inductive query by examples (iqbe): A machine learning approach. *In In Proceedings of the 27th Annual Hawaii International Conference on System Sciences (HICSS-27), Information Sharing and Knowledge Discovery Track*.

HUMPHREYS, B. et LINDBERG, D. (1993). The umls project: making the conceptual connection between users and the information they need. *Bulletin of the Medical Library Association, 81(2)*.

INFORMATICS, S. M. (2001). The protégé project. http://protege.standford.edu.

JANICKI, R. et SEKERINSKI, E. (2001). Foundations of the trace assertion method of module interface specification. *IEEE Trans. Softw. Eng.*, 27(7):577–598.

JEFFORDS, R. et HEITMEYER, C. (1998). Automatic generation of state invariants from requirements specifications. *In SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 56–69, New York, NY, USA. ACM Press.

JIAN ZHANG, Jianfeng Gao, M. Z. et WANG, J. (2001). Improving the effectiveness of information retrieval with clustering and fusion. *Computational Linguistics and Chinese Language Processing*, 6(1):109–125.

JWNL (2003). Java wordnet library-jwnl 1.3. http://sourceforge.net/projects/jwordnet/.

L, J. I. (1998). Human-computer studies. *Special issue on problem-solving methods*, 49(4):305–313.

LASSILA, O. et SWIC, R. R. (1999). Resource description framework (rdf) model and syntax specification. *W3C - World Wide Web Consortium, Cambridge, MA, W3C Recommendation*.

LEE, T. B., HENDLER, J. et LASSILA, O. (2001). The semantic web. *Scientific American*, 284(5).

LEINO, K. et NELSON, G. (1998). An extended static checker for modula-3. *Proc. Compiler Construction: Seventh Int'l Conf. (CC '98)*, pages 302–305.

LI, L. et HORROCKS, I. (2003). A software framework for matching based on semantic web technology. *Proc. 12th Int. World Wide Web Conf., World Wide Web Consortium*, page 48.

LI, Y. (1998). Toward a qualitative search engine. *Internet Computing, IEEE*, *2*:24–29.

LIAO, H.-C., CHEN, M.-F., WANG, F.-J. et DAI, J.-C. (1997). Using a hierarchical thesaurus for classifying and searching software libraries. *In COMPSAC '97: Proceedings of the 21st International Computer Software and Applications Conference*, pages 210–216, Washington, DC, USA. IEEE Computer Society.

LIM, T.-S., LOH, W.-Y. et SHIH, Y.-S. (2000). A comparison of prediction accuracy, complexity, and training time of thirty-three old and new classification algorithms. *Mach. Learn.*, 40(3):203–228.

LLORENS, J., AMESCUA, A. et VELASCO, M. (1996). Software thesaurus: A tool for reusing software objects. *Proceedings of the Fourth International Symposium on Assessment of Software Tools (SAST '96)*, page 99.

MANOLA, F. et MILLER, E. (2004). Rdf primer. W3C Recommendation, W3C.

MANUEL, M., AURELIO, L. et ALEXANDER, G. (2000). Information retrieval with conceptual graph matching. *roc. DEXA-2000, 11th International Conference and Workshop on Database and Expert Systems Applications*, pages 4–8.

MCBRIDE, B. (2002). Jena. *IEEE Internet Computing*.

MELING, R., MONTGOMERY, E., PONNUSAMY, P. S., WONG, E. et MEHANDJISKA, D. (2000). Storing and retrieving software components: a component description manager. *in Proc. of the 2000 Australian Software Engineering Conf., Canberra, Australia*, pages 107 –117.

MICHALEWICZ, Z. (1992). *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, Berlin Heidelberg.

MILI, A. et et AL. (1999). Toward an engineering discipline of software reuse. *IEEE Software*, 16(5):22–31.

MILI, H., MILI, F. et MILI, A. (1995). Reuse software: Issues and research directions. *IEEE Trans. Software Eng*, 21(6):528–562.

MOREL, B. et ALEXANDER, P. (2004). SPARTACAS: automating component reuse and adaptation. *IEEE Transactions on Software Engineering*, *30*(9):587–600.

MORI, A., FUTATSUGI, T. S. K., SEO, A. et ISHIGURO, M. (2001). Software component search based on behavioral specification. *Proc. of International Symposium on Future Software Technology*.

NAKAJIMA, S. et TAMAI, T. (2001). Behavioural analysis of the enterprise javabeans component architecture. *Model Checking Software Proceedings of the 8th International SPIN Workshop, Toronto, Canada, LNCS 2057, Springer*, pages 163–182.

NAKKRASAE, S., SOPHATSATHIT, P. et EDWARDS, W. (2004). Fuzzy subtractive clustering based indexing approach for software components classification. *International Journal of Computer and Information Science*, 5(1):63–72.

NAPOLI, A. (1992). Subsumption and classification-based reasoning in object-based representations. *In ECAI '92: Proceedings of the 10th European conference on Artificial intelligence*, pages 425–429, New York, NY, USA. John Wiley & Sons, Inc.

NATALYA, F. et DEBORAH, L. M. (2001). Ontology development 101:a guide to creating your fisrt ontology. Stanford University.

NAUMOVICH, G., CLARKE, L., OSTERWEIL, L. et DWYER, M. (1997). Verification of concurrent software with flavers. *19th Int'l Conf. Software Eng*, pages 594–595.

NEIL, J. et SCHILDT, H. (1998). *Java Beans Programming from the Ground Up*. Osborne McGraw-Hill.

NOVAK, G. (1997). Software reuse by specialization of generic procedures through views. *IEEE Transactions On Software Engineering*, 23(7).

OLARU, C. et WEHENKEL, L. (2003). A complete fuzzy decision tree technique. *Fuzzy Sets System*, 138(2):221–254.

OSTERTAG, E., HENDLER, J., PRIETO-DIAZ, R. et BRAUN, C. (1992). Computing similarity in a reuse library system, an ai-based approach. *ACM Transactions on Software Engineering and Methodology*, pages 205–228.

PAEZ, M. C. et STRAETEN, R. V. D. (2002). Modelling component libraries for reuse and evolution. *In the Proceedings of the First Workshop on Model-based Reuse*.

PAN, J. et HORROCKS, I. (2001). Metamodeling architecture of web ontology languages. *In International Semantic Web Working Sympsuim (SWWS)*, pages 131–149, California, USA. Standford University.

PAOLUCCI, M., KAWAMURA, T., PAYNE, T. et SYCARA., K. (2002). Semantic matching of web services capabilities. *In 1st Intl. Semantic Web Conference*.

PARNAS, D. L. et WANG, Y. (1989). The trace assertion method of module interface specification. Technical Report 89-261, Queen's University, Kingston, Ontario.

PENIX, J. et ALEXANDER, P. (1999). Efficient specification-based component retrieval. *Automated Software Engineering*, *6(2)*:139–170.

PERKINS, J. H. et ERNST, M. D. (2004). Efficient incremental algorithms for dynamic detection of likely invariants. *SIGSOFT Softw. Eng. Notes*, 29(6):23–32.

PERRY, D. (1989). The logic of propagation in the inscape environment. *In Proceedings of the ACM SIGSOFT 89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*.

PODGURSKI, A. et PIERCE, L. (1992). Behaviour sampling: A technique for automated retrieval of reusable components. *In Proceedings of the 14th International Conference on Software Engineering*, pages 349–360.

POZEWAUNIG, H. et MITTERMEIR, T. (2000). Self classifying reusable components generating decision trees from test cases. *International Conference on Software Eng. and Knowledge Eng.*

QUINLAN, R. (1986). Induction of decision trees. *Machine Learning*, 1:81–106.

ROBERTSON, S. E. et WALKER, S. (1994). Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. *In SIGIR '94: Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 232–241, New York, NY, USA. Springer-Verlag New York, Inc.

ROSA, N. S., ALVES, C. F., CUNHA, P. R. F., CASTRO, J. F. B. et JUSTO, G. R. R. (2001). Using non-functional requirements to select components: A formal approach. *In Fourth Workshop Iberoamerican on Software Engineeringand Software Environment, San Jose, Costa Rica.*

RUGGIERI, S. (2004). Yadt: Yet another decision tree builder. *ICTAI*, pages 260–265.

RUST, H. (1998). A pvs specification of an invoicing system. *Proceedings of an International Workshop on Specification Techniques and Formal Methods*, pages 51–65.

RYL, I., M.CLERBOUT et A.BAILLY (2001). A component oriented notation for behavioral specification and validation. *OOPSLA Workshop on Specification and Verification on Component Based Systems*.

SALIM, F. D., LOKE, S. W., RAKOTONIRAINY, A. et KRISHNASWAMY, S. (2007). Ui aware: A framework using data mining and collision detection to increase awareness for intersection users. *Advanced Information Networking and Applications Workshops, International Conference on*, 2:530–535.

SCHNEIDER, P. P. et FENSEL, D. (2002). Layering the semantic web: Problems and directions. *In ISWC'02: Proceedings of the First International Semantic Web Conference on the Semantic Web*, pages 16–29, London, UK. Standford University, Springer-Verlag.

SEACORD, R., HISSAM, S. et WALLNAU, K. (1998). Agora: A search engine for software components. *IEEE Internet Computing*, pages 62–70.

SESSIONS, R. (1998). *COM and DCOM: Microsoft's vision for distributed objects*. John Wiley and Sons.

SHNEIDERMAN, B. (1997). A framework for search interfaces. *Software, IEEE*, *14*:18–20.

SMITH, M., WELTY, C. et MCGUINESS, D. (2004). Owl ontology web language guide. W3C Recommendation, W3C.

SOFIEN, K., KHALIL, D., HAMADOU, A. B. et MOHAMED, J. (2002). Un environnement de recherche et d'intégration de composant logiciel. *In Seventh Conference On computer Sciences, Annaba*.

SOFIEN, K., KHALIL, D. et MOHAMED, J. (2006). SEC: A search engine for component based software development. *In* SAC *'06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1745–1750, New York, NY, USA. ACM Press.

SOFIEN, K., KHALIL, D. et MOHAMED, J. (2007). Sec+: an enhanced search engine for component-based software development. *SIGSOFT Softw. Eng. Notes*, 32(4):4.

SOFIEN, K., KHALIL, D. et MOHAMED, J. (2010a). An integration ontology for components composition. *In International Journal of Web Portals*, 2(3):35–42.

SOFIEN, K., KHALIL, D. et MOHAMED, J. (2010b). Ontology-based discovery and integration. *In The Third International Conference on the Applications of Digital Information and Web Technologies*, Istanbul, Turkey.

SOFIEN, K., KHALIL, D. et MOHAMED, J. (2011). *Modern Software Engineering Concepts and Practices: Advanced Approaches*, chapitre Description, Classification and Discovery Approaches for Software Components: A Comparative Study, pages 196–219. 9781609602154. IGI Global.

STOTTS, P. D. et PURTILO, J. (1994). Virtual environment architectures: Interoperability through software interconnection technology. *In Proc. of the Third Workshop on Enabling Technologies (WETICE '94): Infrastructure for Collaborative Enterprises. IEEE Computer Society Press*, pages 211–224.

STRUNK, E. A., YIN, X. et KNIGHT, J. C. (2005). Echo: a practical approach to formal verification. *In Proceedings of the 10th international Workshop on Formal Methods For industrial Critical Systems. FMICS '05. ACM Press, New York, NY*, pages 44–53.

SUN, C. (2003). *QOS Composition and Decomposition in UniFrame*. Thèse de doctorat, Department of Computer and Information Science, Indiana University Purdue University.

SYCARA, K., WIDOFF, S., KLUSCH, M. et LU, J. (2002). Larks: Dynamic matchmaking among heterogeneous software agents in cyberspace. *Autonomous Agents and Multi-Agent Systems*, 5:173–203.

THOMPSON, R. et CROFT, W. (1989). Support for browsing in an intelligent text retrieval system. *International Journal of Man-Machine Studies*, 30(6):639–668.

UTGOFF, P. E. (1989). Incremental induction of decision trees. *Machine Learning*, 4:161–186.

VARADARAJAN, S., KUMAR, A., DEEPAK, G. et PANKAJ, J. (2002). Componentxchange: An e-exchange for software components. *ICWI*, pages 62–72.

VASILIU, L., ZAREMBA, M., MORAN, M. et BUSSLER, C. (2004). Web-service semantic enabled implementation of machine vs.machine business negotiation. *Proceedings of the IEEE International Conference on Web Services (ICWS'04), IEEE Computer Society, San Diego, California, USA*.

VITHARANA, P., ZAHEDI, F. M. et JAIN, H. (2003). Knowledge-based repository scheme for storing and retrieving business components: a theoretical design and an empirical analysis. *IEEE Transactions on Software Engineering*, *29*(7):649–664.

WANG, Y. et STROULIA, E. (2003). Flexible interface matching for web-service discovery. *In Proceedings of the Fourth International Conference on Web Information Systems Engineering*, WISE '03, pages 147–, Washington, DC, USA. IEEE Computer Society.

WHALEY, J., MARTIN, M. C. et LAM, M. S. (2002). Automatic extraction of object-oriented component interfaces. *SIGSOFT Softw. Eng. Notes*, 27(4):218–228.

WIKIPEDIA (2007). Tower of hanoi. http://en.wikipedia.org/wiki/Tower_of_Hanoi.

WILLE, R. (1982). Restructing lattice theory : an approach based on hierarchies of concepts. *In I. Rival, editor, Ordered sets*, pages 445–470.

WILLET, P. (1988). Recent trends in hierarchic document clustering: a critical review. *Information Processing and Management*, 24(5):577–597.

XIE, X., TANG, J., LI, J.-Z. et WANG, K. (2004). A component retrieval method based on facet-weight self-learning. *In AWCC*, pages 437–448.

YAO, H. et ETZKORN, L. (2004). Towards a semantic-based approach for software reusable component classification and retrieval. *In ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, pages 110–115, New York, NY, USA. ACM Press.

YELLIN et STROM. (1997). Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, *19*(2).

YELLIN, D. M. et STROM, R. E. (1997). Protocol specifications and components adaptors. *ACM Trans. Prog. Lang. Syst*, *19(2)*:292–333.

YUNWEN, Y. et FISCHER, G. (2001). Context-aware browsing of large component repositories. *Proceedings of 16th International Conference on Automated Software Engineering (ASE'01), Coronado Island, CA*, pages 99–106.

ZAREMSKI, A. M. et WING, J. M. (1995). Specification matching of software components. *In Proceeding. Third Symposium on the Foundations of Software Engineering (FSE3), ACM SIGSOFT*, pages 1–17.

ZHANG, Z., SVENSSON, L., SNIS, U., SRENSEN, C., FGERLIND, H., LINDROTH, T., MAGNUSSON, M. et STLUND, C. (2000). Enhancing component reuse using search techniques. *Proceedings of IRIS 23*.

ZHIYUAN, W. (2000). *Component-Based Software Engineering*. Thèse de doctorat, Faculty of New Jersey Institute of technology.