POLITECNICO DI MILANO
Scuola di Ingegneria dell'Informazione

POLO TERRITORIALE DI COMO

Master of Science in
Computer Engineering

Model-Driven Retrieval of Model Repositories

Supervisor: Prof. Marco Brambilla
Assistant Supervisor: Prof. Alessandro Bozzon

Master Graduation Thesis by:
Stefano Celentano, Student Id. number 755287
Lorenzo Furrer, Student Id. number 750213

Academic Year 2010-2011

POLITECNICO DI MILANO
Scuola di Ingegneria dell'Informazione

POLO TERRITORIALE DI COMO

Corso di Laurea Specialistica in
Ingegneria Informatica

Model-Driven Retrieval of Model Repositories

Relatore: Prof. Marco Brambilla
Correlatore: Prof. Alessandro Bozzon

Tesi di Laurea di:
Stefano Celentano, matricola 755287
Lorenzo Furrer, matricola 750213

Anno Accademico 2010-2011

Acknowledgements

Stefano

Non ho mai amato troppo leggere i ringraziamenti a conclusione di un lungo lavoro. Si rischia di essere troppo solenni, convenzionali e ripetitivi in ciò che si scrive e quella che dovrebbe essere una pagina piena di parole sentite diventa poi solo un'inutile formalità. Personalmente ritengo queste righe una occasione in cui può essere utile fermarsi un momento a riflettere su ciò che è stato e che dovrà essere di se' stessi e della vita futura. Perciò le scriverò con interesse, provando a uscire un po' fuori dagli schemi.

Pensare agli ultimi cinque anni mi fa scorrere brividi fortissimi lungo tutto il corpo, e qualche "lacrimuccia" incomincia a formarsi. E' un ricordo pieno di esperienze uniche, nuove e (purtroppo) irripetibili, che ho affrontato sempre con il massimo dell'impegno.

Gli ultimi cinque anni di vita sono stati per me una esperienza che va di gran lunga oltre la sola avventura universitaria. Sono cresciuto in tutto: dal modo in cui mi comporto nella vita di tutti i giorni allo studio. I motivi per cui ho lasciato la città, la mia città, quella in cui sono nato e che adoro, la mia famiglia e i miei amici per venire a studiare qui non mi sono ancora del tutto chiari. Ma come spesso accade le cose migliori si fanno anche con una (quasi) totale mancanza di consapevolezza, una specie di salto nel buio. La scelta di cui parlo è stata però certamente indovinata. Ho vissuto in un ambiente nuovo, diverso, stimolante. Ho conosciuto persone interessanti, buone e a volte splendide. Ho imparato un numero incredibile di cose nuove e ho apprezzato studiare e approfondire ciò che rappresenterà il mio lavoro in futuro. E' come se in cinque anni ne fossero concentrati dieci, forse quindici. Ora, guardandomi indietro continuo solo a provare fortissimi brividi, e quasi non riesco a voltarmi. Non sono molto capace di godere dei risultati raggiunti. So anche però che è importante soffermarsi un attimo, pensare a ciò che si è fatto e magari esserne un po' fieri.

So di poter sembrare cinico quando dico che la cosa che ho imparato

a fare meglio nel periodo universitario è stata riuscire a "cavarmela" da solo. Lo dico con il pessimismo che credo abbia sempre fatto parte di me: aver imparato che contare solo sulle proprie forze è fondamentale, è questo l'insegnamento migliore che l'università e la vita mi hanno offerto. In realtà, non sto raccontando tutta la verità (e qui incomincia il pezzo in cui si sorride). Al mio fianco ci sono state molte persone che, in qualche modo, forse inconsapevolmente, mi hanno accompagnato lungo il percorso. Sono le persone che io ingenuamente definisco "buone" e sono quelle che desidero ringraziare immensamente.

Ringrazio il relatore di questa tesi, il prof. Marco Brambilla, e il correlatore, il prof. Alessandro Bozzon, che si sono sempre dimostrati disponibili nel fornirci consigli e osservazioni utili. Ringrazio gli amici che negli ultimi anni mi sono stati più vicini sia nella vita universitaria che in quella quotidiana: Ilio, Eleonora e Lorenzo, amico e collega di tesi con cui ho affrontato quest'ultimo lavoro, e anche Antonio ed Antonella. Ringrazio tutti i miei amici, sia quelli di Como che quelli di Salerno. Tra quelli di Como penso con un profondo sorriso a tutti gli amici dello studentato dia via Pannilani; tra quelli di Salerno penso specialmente agli amici delle scuole superiori con cui ancora adesso adoro passare del tempo quando torno a Casa. Troverei spiacevole fare un lungo elenco di nomi perché è sicuro che dimenticherei qualcuno. Tra di loro ci sono persone molto importanti che ringrazierò con un abbraccio.

Ringrazio (e bacio) la mia ragazza Annalisa (conosciuta anche come Isa, Isetta, Lisa o Annina) che è stata spessissimo fonte dei miei sorrisi.

Più di tutti (e sono sicuro che nessuno me ne vorrà) ringrazio la mia famiglia: mio padre Andrea, mia madre Paola e mio fratello Sergio al quale devo la spinta finale nella scelta di intraprendere gli studi qui a Como. Le parole non possono e non potranno mai dimostrate tutto l'affetto, la devozione, l'ammirazione e la riconoscenza che provo nel mio cuore per loro. Se sono ciò che sono adesso, lo devo a soprattutto a loro, al modo in cui mi hanno cresciuto e a ciò che mi hanno insegnato. Sempre mi hanno lasciato libero di decidere, mai mi hanno fatto mancare consigli preziosi. Se c'è una cosa che mi rende triste in questo momento, è che in futuro, probabilmente, non avrò l'occasione di vederli spesso come vorrei. Quello che è certo è che rimarranno per sempre nel mio cuore come le persone più speciali e preziose della mia vita.

Lorenzo

Siamo alla tanta agognata fine del percorso di studi. Se dovessi fare un paragone navale direi che la nave sta per attraccare al porto. Non si vede mai l'ora di questo momento, ma poi quando accade un pò si vorrebbe tornare indietro. Sarà per l'esperienza, sarà per l'ambiente, sarà per le persone. Ma non c'è niente da fare e soprattutto bisogna fare i conti con le esigenze della vita. Non è nel mio stile dilungarmi troppo, con piacere quindi passo ai ringraziamenti alle varie persone che mi hanno aiutato a non andare alla deriva. Ringrazio i miei genitori, che mi sostengono sempre e che sono insostituibili, si meritano molto di più. Ringrazio Stefano, co-autore di questa tesi, perchè navigare in due è più divertente. Ringrazio il prof. Marco Brambilla e il prof. Alessandro Bozzon per la loro disponibilità a seguirci durante questo progetto. Un particolare grazie va anche a Eleonora, Ilio, Antonio e Antonella: i compagni che mi hanno accompagnato da vicino in questi anni di studio e sacrifici. Le ore in università sono state di sicuro molto più liete. Ringrazio anche Mattia e gli altri compari del milanese, nonostante impegni e distanze non permettano di vederci molto. E infine, un grazie a tutte le altre persone che mi hanno saputo incoraggiare.
Spero di fare in modo che il vostro supporto non sia stato vano.

Abstract

Model-Driven Development (MDD) is a software development methodology that focuses on the creation and maintenance of domain models as the primary form of expression in the development cycle. One of the fundamental characteristics of such approach is the reuse of software artifacts through their model representation. However, software reuse is impaired by the fact that current systems lack an efficient way to search through the model repositories as many of the current solutions don't tackle the relationships between model artifacts. These relationships are instead important to better satisfy the user information need in a model-driven development scenario.

This thesis aims to define a model-driven methodology for creating model search engines. As opposed to many related works, this methodology is metamodel-independent and exploits the metamodel of the searched project models in order to obtain more precise results. A prototype has been implemented to support such methodology. We address two case studies that deal with the indexing and the retrieving of models from two different collections of UML and WebML projects respectively. Each case study involves several experiments adopting different indexing strategies. Finally, after having manually built the ground truth for each repository, we performed various tests using established Information Retrieval measures like DCG, MRR, MAP, Precision and Recall in order to evaluate the results.

Sommario

Il Model-Driven Development (MDD) è una metodologia per lo sviluppo del software che si concentra sulla creazione e sul mantenimento di modelli come forma di espressione primaria nel ciclo di sviluppo. Una delle caratteristiche fondamentali di questo approccio sta nel riuso degli artefatti software attraverso la loro rappresentazione nel modello. Tuttavia, il riuso del software è ostacolato dal fatto che i sistemi attuali soffrono la mancanza di un modo efficiente di cercare attraverso depositi di modelli, siccome molte delle soluzioni attuali non tengono in considerazione le relazioni presenti tra gli artefatti del modello. Queste relazioni sono invece importanti per soddisfare al meglio i bisogni di informazione degli utenti in uno scenario di sviluppo model-driven.

Lo scopo di questa tesi è di definire una metodologia model-driven per creare motori di ricerca di modelli. Al contrario di molti lavori correlati, questa metodologia è indipendente dal metamodello dei modelli considerati e sfrutta questo metamodello per ottenere risultati più precisi. Un prototipo è stato implementato per supportare questa metodologia. Abbiamo investigato due casi di studio che trattano dell'indicizzazione e del recupero di modelli da due collezioni differenti, rispettivamente di progetti UML e WebML. Ogni caso di studio coinvolge diversi esperimenti i quali adottano differenti strategie di indicizzazione. Infine, dopo aver costruito manualmente la ground truth per ciascuno di questi due depositi, abbiamo eseguito vari test usando tecniche di Information Retrieval affermate come DCG, MRR, MAP, Precision e Recall per poter valutare i risultati ritornati dal sistema.

Contents

1 Introduction **1**
1.1 Contribution 2
1.2 Thesis Organization 2

2 Related Works **5**
2.1 Text-Based Approaches 6
2.1.1 Source code search 6
2.1.2 Other text-based approaches 8
2.2 Content-Based Approaches 10

3 Background **13**
3.1 General IR Architecture 13
3.1.1 Content Registration 15
3.1.2 Content Analysis 15
3.1.3 Content Indexation 17
3.1.4 Search 18
3.1.5 Display 19
3.2 Modeling and Model Driven Development 20
3.3 UML 22
3.4 WebML 23

4 Approach **27**
4.1 Abstract Solution 27
4.2 Design Dimensions 30
4.3 Indexing Strategies 33

5 Implementation **37**
5.1 SMILA 37
5.1.1 Architecture 37
5.1.2 Data Model 40
5.2 Solr 42

5.2.1 Design and Index Definition 44
5.2.2 Text Analysis 45
5.2.3 Documents Search 45
5.3 Prototype Architecture 47
5.3.1 Indexing part 48
5.3.2 Query part 48
5.4 Configuration Files 49
5.5 Configurator and user interface 50

6 Case Studies 57
6.1 UML Dataset 58
6.2 UML Case 62
6.3 WebML Dataset 72
6.4 WebML Case 76

7 Tests and Evaluation 81
7.1 Evaluation Metrics 82
7.2 Ground truth 85
7.3 UML Tests and Results 86
7.3.1 Test Queries 86
7.3.2 Test Configurations and Results 90
7.4 WebML Tests and Results 105
7.5 Main findings 114

8 Conclusions and Future Work 117
8.1 Future Work 118

Bibliography 120

List of Figures

3.1 Architecture of a general-purpose Information Retrieval system. 14
3.2 Operations involved in Content Analysis. 15
3.3 Conformance relationship: a model represents a system and conforms to a metamodel. 21
3.4 Traditional OMG's metamodeling infrastructure with four layers. 22
3.5 WebML diagram example and metamodel. 24

4.1 The approach of our abstract solution for a general search engine model repository system. In the upper part there is the *Content Processing Flow* and in the lower part there is the *Query Processing Flow*. 28

5.1 SMILA general architecture 38
5.2 SMILA data model 41
5.3 Diagram summing up all the possible inputs and outputs and the composition of a Solr index. 43
5.4 Hierarchical organization of Solr analyzers. 46
5.5 The diagram of the architecture of the prototype showing the chain of operations of the indexing phase. 52
5.6 Chain of operations of the search phase in the WebML case. . 53
5.7 The pipeline of the Experiment C for WebML displayed by the Eclipse BPEL Designer. 54
5.8 Screenshot depicting the SMILA tab of the configurator. . . . 55
5.9 Screenshot of the help text of the processor.properties Info button. 55

6.1 An example of UML project model from AtlanMod zoos dataset. 59
6.2 Histogram that shows the distribution of the number of classes into the UML projects dataset. 62
6.3 The frequency distribution of terms of the UML dataset. . . . 63

6.4 The pipeline of the UML case Experiment C. 66

6.5 An example of UML project model diagram from the dataset to explain Experiment D purposes. A query string like "entry location" (AND query) without the importation algorithm would produce no results. The algorithm of Experiment D allows to retrieve both classes "/LocatedElement/" and "Entry". 69

6.6 The pipeline of the UML case Experiment D. 70

6.7 Example of an area of a project from the WebML dataset. . . 73

6.8 The frequency distribution of terms of the WebML dataset. . 75

6.9 WebML operations chain. The diagram shows the operations involved in both content-based approach and text-based approaches. The Content Processing phase is showed in the top right, while the Query Processing phase is showed in the bottom left. 76

7.1 Example of UML project model from the dataset of UML class diagrams. 86

7.2 Plot of the DCG and iDCG curves of Experiment A (Project Granularity, Flat Index), Test Configuration 1 (MQ2). 95

7.3 Plot of the 11-points Interpolated Average Precision of the Experiment A (Project Granularity, Flat Index), Test Configuration 1 (MQ2). 96

7.4 Plot of the Precision at k of the Experiment A (Project Granularity, Flat Index), Test Configuration 1 (MQ2). 97

7.5 Comparison between Test Configuration 1 and 2: DCG and iDCG curves of the Expertiments B (Concept Granularity, Multi-Field Index), C (Concept Granularity, Multi-Field Weighted Index) and D (Concept Granularity, Multi-Field Weighted Index, Graph Based) for both the test configurations. 99

7.6 Plot of the Precision at k of Experiments B (Concept Granularity, Multi-Field Index), C (Concept Granularity, Multi-Field Weighted Index) and D (Concept Granularity, Multi-Field Weighted Index, Graph Based), Test Configuration 2 (MQ5). 100

7.7 DCG curves of Experiments C (Concept Granularity, Multi-Field Weighted Index) and D (Concept Granularity, Multi-Field Weighted Index, Graph Based), Test Configuration 3 (MQ5). 101

7.8 DCG curve of Experiment D (Concept Granularity, Multi-Field Weighted Index, Graph Based), Test Configuration 4 (MQ5). 102
7.9 DCG curve of Experiment D (Concept Granularity, Multi-Field Weighted Index, Graph Based), Test Configuration 5 (MQ5). 103
7.10 An example of document-based query used to test the WebML experiments. 106
7.11 An example of WebML area adopted to explain test configuration 4 and 5. 109
7.12 DCG and iDCG curves of the first three WebML test configurations. 110
7.13 Plot of 11-point Interpolated Average Precision of WebML Experiments B (Concept Granularity, Multi-Field Index) and C (Concept Granularity, Multi-Field Weighted Index), Test Configuration 1. 111
7.14 Plot of the Precision at k curve of WebML Experiments B (Concept Granularity, Multi-Field Index) and C (Concept Granularity, Multi-Field Weighted Index), Test Configuration 1. 112
7.15 DCG and iDCG curves of the test configuration 1, 4 and 5. . 113
7.16 DCG and iDCG curves of the first three WebML test configurations. 115

List of Tables

7.1 The meta-queries designed for testing the UML experiments. The "Target" column indicates which type of document the query searches (e.g. a project, or a class); the second column is the identifier of the meta-query; the third column briefly describes the meta-query, namely it describes the information need that can be satisfied by the queries that are instances of the meta-query; the last column shows an example of query. . 88

7.2 The list of the ten instances of the meta-query 2 of the UML case study. 89

7.3 The list of the ten instances of the meta-query 5 of the UML case study. 89

7.4 The first configuration of payload values. This configuration is determined according to simple reasonings on the UML metamodel concepts. 92

7.5 The second configuration of payload values. This configuration is determined by slightly changing the values of the first one. 92

7.6 The first configuration of penalty values. This configuration is determined according to simple reasonings on the UML relationship types. 93

7.7 The second configuration of penalty values. This configuration is determined starting from the first one by slightly changing penalty values. 93

7.8 The third configuration of penalty values. This configuration is determined by multiplying the values of the first one by a factor of 0.1. 94

7.9 MAP results of the test configurations of the UML model-based search engine. MQ2 addresses only Experiment A (Project Granularity, Flat Index); MQ5 addresses Experiments B (Concept Granularity, Multi-Field Index), C (Concept Granularity, Multi-Field Weighted Index) and D (Concept Granularity, Multi-Field Weighted Index, Graph Based). 104
7.10 MRR results of the test configurations of the UML model-based search engine. MQ2 addresses only Experiment A (Project Granularity, Flat Index); MQ5 addresses Experiments B (Concept Granularity, Multi-Field Index), C (Concept Granularity, Multi-Field Weighted Index) and D (Concept Granularity, Multi-Field Weighted Index, Graph Based). 105
7.11 The complete list of the ten text-based queries used to evaluate the WebML case study application. The keywords are extracted from the content-based version of the queries. . . . 106
7.12 MAP results of the test configurations of the WebML model-based search engine. 112
7.13 MRR results of the test configurations of the WebML model-based search engine. 114

Chapter 1

Introduction

Models are becoming more and more important in the software development process. As Model-Driven Development (MDD) practices gain adoption, an increasingly large number of models is being produced and used by software organizations. Some development methodologies strongly rely on the reuse of such models and software artifacts in order to take advantage of previously developed assets. In such an environment, model repositories play an important role, and the ability to automatically search for models inside large repositories may be more critical than searching for code. Searching for software artifacts that are highly relevant to high level concept tasks is difficult because the descriptions of these tasks don't usually match their low-level implementation details. This problem is known as the *concept assignment problem* [1]. General-purpose search engines, such as Google code search, are not designed to search for models. Usually, most of current source code search engines return short code snippets as results to user queries. But, when taken out of context, these brief snippets don't give enough background to help developers determine the correct way to reuse the code, constricting them to spend a significant amount of time and effort to understand how to use these code snippets in larger scopes for their own purposes. Model search also has an important educational value for students and teachers, allowing learners to find good examples that can be used to improve the knowledge acquisition and provide hints at solutions [2]. As a result, the need for efficient model-searching mechanisms arises. Moreover, in many related works, the internal structure of the model (the metamodel) is not taken into account. Thus, it is not possible to search for a specific class of objects or exploit the implicit relations between them.

1.1 Contribution

This thesis aims to define a model-driven methodology for creating model search engines. In the following, we call our system Model-Driven Information Retrieval System. The methodology exploits the metamodel of the searched project models in order to obtain more precise results. The methodology is also metamodel-independent, as changing the metamodel given in input to the system doesn't affect the methodology itself. All the performed operations, as well as their arrangement, are configurable through configuration files. This allows to easily insert new operations or modify the parameters of the existing ones. The system is also suitable to process other artifacts along with models, such as project specifications and user generated annotations. This extra information can help to describe the indexed documents, thus achieving better precision and recall.

A prototype has been implemented to support such methodology. We implemented two case study applications that deal with two collections of two distinct metamodels over which we provide the ground truth. These two metamodels are: UML class diagram, which is the most common diagram found in software engineering representations due to its flexibility and its representation power; WebML, which is a modeling language specifically designed for web applications. In the UML case, we investigated the behavior of the system with various experiments using four different indexing and searching strategies. Namely, we varied the granularity of the returned results according to the concepts of the involved metamodel. Then, we investigated the behavior of the system with and without weights assigned to those concepts. Finally, we propose a new method that uses the graph representation of the analyzed model in order to retain the underlying structure even in a purely text-based search. As in the UML case, the WebML case involves various experiments with different indexing approaches. Then, we compared the results given by different configurations obtained by changing the amount of harvested information, in order to identify the one giving the best results.

In order to assess the quality of the results returned by the system, we used established Information Retrieval measures like DCG, MAP, MRR, Precision and Recall.

1.2 Thesis Organization

The thesis is organized as follows:

- Chapter 2 presents a rundown of the most recent and relevant works related to this thesis.
- Chapter 3 gives some background information on the concepts of the area of work of this thesis, like information retrieval and metamodeling. It also explains some of the basic characteristics of the metamodels addressed in the case studies.
- Chapter 4 describes the approach we adopted to develop our solution. It explains the design dimensions of the methodology and the indexing strategies we used in the experiments.
- Chapter 5 first gives an overview of the framework and of the search platform we adopted, then it illustrates the implementation details of the developed prototype.
- Chapter 6 explains the two case studies we assessed in our experiments, as well as the details of how such experiments were performed.
- Chapter 7 shows the commented results of the performed tests.
- Chapter 8 comes to the conclusions and lists some possible directions which can be taken by future works.

Chapter 2

Related Works

Software reuse is important because it allows for a better and faster development, increasing the quality of the final product and cutting the costs. The number of software repositories grows everyday more and more, however the information stored in them is rarely found in a format already suitable for a human to be searched and understood effectively. There are many reasons why developers would want to search through a software repository. These reasons include: finding reusable components, obtaining code examples and performing impact analysis [3]. As such, many tools and platforms have been and are currently being developed to answer those needs. Most of these tools fall into the category of searching tools, as the amount of information stored in the repositories is huge and there's the need to filter out the non-relevant artifacts. There are many ways to classify those search engines, for example by *query type*, by *indexing type*, by the *algorithms* used for matching and by the way *result presentation* is performed [4]. Another kind of classification is based on the *content type* addressed. These content types could be source code, models, business processes and web services.

Source code search operates at the lowest possible level in the software abstraction chain as it leverages the grammar of the programming languages. These methods are limited by the expressiveness of the programming languages themselves and by the fact that the appropriate solution could be very different from the submitted query, so it might be difficult to match the results to the query according to their semantic meaning. Developers usually search the internal repository for different reasons than when they search the web [5]. So, it is necessary to develop systems specifically designed to exploit the internal repository knowledge base.

Model-based search systems operate at a higher level of abstraction, i.e., the model of the system to be searched. Thus, these methods exploit the

structure and the relationships between artifacts to better capture the semantic meaning of the concepts and achieve a greater level of expressiveness and accuracy. Model-based methods can be applied to many kinds of structured data, like business processes and web services which are targeted by specific search engines due to their popularity. Business Process search is focused on searching business processes. A business process is often represented as a sequence of activities, this implies that searching business processes puts a great emphasis on the pattern in which these activities are arranged. Web services are more and more central in the development of distributed applications over the Web as well as being fundamental components of what is called the semantic web. As such, there are specific search engines and methods designed to address the problem of searching through the offers of the considerable amount of public web services and their relative APIs.

For the purpose of this work, we use the query/indexing type as main classification, distinguishing between text-based and content-based approaches. Works falling into these two categories are found in sections 2.1 and 2.2 respectively.

2.1 Text-Based Approaches

In the text-based approach an artifact is represented as an unstructured text document. The index contains bags of words extracted from the models in the repository. These terms can possibly be weighted based on their concept of belonging, boosting them according to its relative relevance in the used metamodel. The matching is ultimately performed on these textual terms which can also include annotations and comments produced by developers to better describe their meaning, facilitating the retrieval process. Facets are also common since they allow further filtering in case too many results are retrieved. In the following subsections we distinguish between systems specifically designed to search through the source code of applications and systems that target other software representations.

2.1.1 Source code search

Selene [6] is an Eclipse plug-in built around a text-based search engine over a source code repository. It is a code recommendation tool that uses the entire editing code as a query. It searches and displays similar program fragments from a repository of examples programs. The searches are started automatically while the developer is editing the code. Selene is expected to

assist developers in finding usages and idioms of API libraries and frameworks suitable to their operation context, without extensive manual search through tutorials or general search engines.

The work in [7] presents another code recommendation tool that uses the knowledge embodied in the identifiers of variables and functions as its basis for the suggestion. The assumption made is that code fragments using similar terms within the identifiers also reuse similar methods. The system constructs a matrix which associates a method call with the identifiers preceding it. Each row of the matrix is a binary vector stating which terms correspond to a call. These vectors are then matched with a set of terms extracted from the context of the current cursor position as to retrieve possible relevant method calls.

The system in [8] is designed to perform a search for algorithms. Users can enter a free text query which will be used to perform a search through a collection of academic documents, since they generally follow an easier structure for a machine to parse in order to identify the relevant sections containing pseudo-code for the algorithms. This tool is being developed as part of the SeerSuite toolkit, a collection of open source tools for creating academic search engines and digital libraries such as CiteSeerX.

Sourcerer [9] is an infrastructure for large scale indexing and analysis of open-source code, upon which code search engines and services can be built. Sourcerer crawls the Internet looking for Java source code from public web sites, version control systems and open source repositories. In Sourcerer, the code is parsed, analyzed and stored in three forms: Managed Repository contains a versioned copy of the original project content; Code Database stores models of parsed projects; Code Index stores keywords extracted from the code. The metamodel used by Sourcerer to store the structural information is an adapted version of the C++ entity-relationship metamodel. Each library file is uniquely identified across all the projects to maintain cross-project dependencies.

Exemplar (EXEcutable exaMPLes ARchive) [10] combines information retrieval and program analysis techniques to link high-level concepts to the source code of the applications via standard API calls that these applications use. The novelty of this approach is to augment the standard code search to include into indexes also the API documentation of the most widely used libraries (e.g. Java Development Kit). Description and titles of Java applications are indexed independently from the Java API call documentation. Keywords entered by the user are matched separately in these two indexes. As a result two ranked lists are retrieved, the one of the applications and the one of the API calls. Then the system locates the retrieved API calls in the

retrieved applications and the combined score is computed. As final step, the list of applications is sorted using the computed ranks and returned to the user.

Maracatu [11] is a keyword-based search engine for retrieving source code components from development repositories. This search engine combines both text mining and facet-based search, achieving better results with respect to situations where these two techniques were used alone. Before searching, a filtering is performed to exclude components which do not satisfy the constraints. A visualization of the retrieved component is allowed before its download.

2.1.2 Other text-based approaches

CodeBroker [12] is a system that allows to autonomously locate components in a repository by taking into account the background knowledge of the developer (information delivery). This method was inspired by the fact that software reuse is often unsuccessful because of the lack of knowledge and inability of the users to create a well-defined query. CodeBroker utilizes user models in order to represent their knowledge about the repository.

SPARS-J [13] is a Java class retrieval system that uses a graph-represenation model of software libraries called component rank model. This model is based on analyzing actual usage relations of the components and on propagating their significance through usage relations. The resulting rank allows for highly ranked components to be quickly found by uses. Results show that a class frequently invoked by other classes has a high rank, with respect to nonstandard classes.

WISE (Workflow Information Search Engine) [14] is specifically designed to query workflow hierarchies. A workflow hierarchy provides different views of the same workflow. Each view represents the workflow at different depth levels and includes as more tasks as the level is deeper. The user issues keyword queries and the system finds the workflow hierarchies in the repository that contain matches to those keywords. The system then returns search results as the minimal views of the most specific workflow hierarchies that contain tasks matching keywords. Query results defined in this way are then proved to be informative and concise.

After receiving in input an informal description of a semantic domain (represented as a set of terms) and a set of ontologies in an ontology repository, *CORE* [15] (Collaborative Ontology Reuse and Evaluation) automatically determines which ontologies describe most accurately the given domain by using similarity measures. The user selects a subset of available compar-

ison techniques and obtains in output a ranked list of ontologies for each of them. Then a global aggregated measure that uses rank fusion techniques is used to define a unique ranking.

Service-oriented systems (SOS) are search-driven because they are based on using software components usually provided by third parties over the web. Since BPEL documents often contain the invocation and definition of such services, the paper in [16] proposes a way to search for them via BPEL fragments. Retrieved fragments are shown along web services corresponding to the activities within the fragment, as well as the BPEL documents containing such fragment in order to give context on how the retrieved services were used. Fragments are ranked according to their number of occurrences in the documents and on their relevance to the query. The matching is not performed on the documents one by one. Instead, this approach considers all documents at once by merging them in a big graph where the nodes are the basic activities and the edges are the control flow. This way the amount of unnecessary matching is limited since each activity appears only one time in the graph.

Woogle [17] is a search engine that is designed to find web services that offer similar operations. An algorithm clusters the parameter names of the operations into semantically meaningful concepts, which are then exploited to determine input similarity. The similarity is determined by considering textual descriptions of operations and web services as well as the parameter names. The clustering policy is based on the assumption that parameters express the same concept if they occur often together. Woogle allows for both template and composition search. Template search allows the user to specify the functionality, input and output of the web service operation, and returns a list of operations that fulfill the requirements. Composition search on the other hand, returns operations composition that achieve the desired functionality specified in the search.

The work of this thesis is based on the system presented in [18]. The paper describes a model-driven information retrieval system to search through projects expressed in the WebML language. The realized prototype applies metamodel-aware extraction rules to analyze models. It has a visual interface to submit keyword-based queries, performed on whole projects, subprojects, or concepts. Then it inspects the results, presented as a paginated list of matching items with a possibility of snippet visualization. Details of this work will be better described in Section 4.2.

One of the objectives of this thesis is to evaluate the performances of text-based system against content-based ones. So, in the next section we report an overview of most of the current content-based search systems.

2.2 Content-Based Approaches

In a content-based approach the role of the DSL metamodel of the projects in the repository is more prominent, since it takes into account the relationships between the various artifacts. The index structure usually reflects this new amount of information considered. Queries are usually expressed by providing a model fragment, following a Query-By-Example approach. Usually a kind of structural similarity matching algorithm is performed, like graph matching. The query syntax must conform to the same metamodel of the language of the projects in the repository. The DSL metamodel can be used during matching to provide some domain-specific information to tune the quality of the ranked result list.

SECOLD (Source code ECOsystem Linked Data) [19] is a framework that provides source code and facts usable by both humans and machines for browsing and querying. Currently this framework provides line-level and statement-level granularity for the presentation and syntax layer respectively. It adheres to the Linked Data publication standard so that the repository is available online in both HTML and RDF/XML formats. This framework provides a way to uniquely identify resources while they are being analyzed across different tools by agreeing on a common naming format. This way a resource can be identified anytime, independently on the specific naming convention or ID format of each analysis tool. It also provides a set of public services for URL generation and data conversion from source code and version control systems.

Moogle [20] is a model search engine that uses UML or some DSL (Domain Specific Language) meta-model in order to create indexes for evaluation of complex queries. Its key features include searching through different kind of models, as long as their metamodel is provided. The index is built automatically and the system tries to present only the relevant part of the results, for example trying to remove the XML tags or other unreadable characters to improve readability. The model elements type, attributes and hierarchy between model elements can be used as a search criteria. Models are searched by using keywords, by specifying the types of model elements to be returned and by using filters organized into facets. Moogle uses the Apache Solr ranking policy of the results. The most important information of the results are highlighted to make them more clear to the user.

The work described in [21] uses a graph representation able to harness the power of models with the flexibility of adapting to the syntax and semantics of various modeling languages. First the method translates the given models in the repository into directed graphs. Then, a query conforming to the

considered DSL metamodel is submitted to the system. This query is also transformed in a graph in order to reduce the matching problem into a graph matching one. Matches are calculated by finding a mapping between the query graph and the project graphs or sub-graphs, depending on the granularity. The results are ranked using the graph edit distance metric by means of the A-Star algorithm. The prototype considers the case of the domain-specific WebML language.

The paper in [22] presents an inexact matching approach for workflow process reuse. The matching degree between two workflow processes is determined by the matching degrees of their corresponding sub-processes or activities. The matching degree of two activities is determined by the activity-distance between them in an activity-ontology repository. Users are provided with SQL-like commands to specify inexact query conditions to retrieve the required processes from the workflow-ontology repository.

Nowick et al. [23] introduce a model search engine that tries to help users to improve their queries. It uses logs coming from several search engines belonging to three different environments to model users' session characteristics by cluster analysis. An example of session is that of hit and run users who briefly browse results about popular topics. This characteristics are used to improve the search process: in case of a failed search or when the search engine returns more than 100 or less than 1 result, the system suggests the user to try the advanced smart search to narrow down or improve the results adding other terms from the same cluster of the original term. In the case of narrow search, the terms from other clusters are also suggested.

The paper in [24] introduces an approach for finding similarity between business process models. This method assumes an ideal case in which process models are enriched with annotations describing their meaning. This approach uses the BPMN-Q query language expansion which allows users to make structure related model queries. The BPMN-Q expansion applies the enhanced Topic-based Vector Space Model, a vector space model that is able to exploit semantic document similarities via WordNet. The system constructs an ontology from the repository and expands the BPMN-Q query by constructing other queries. Those other queries are based on the substitution of the seed query activities with similar ones.

BP-QL is a visual query language for querying business processes [25]. It allows users to specify a query in the same way they specify the model, hiding the XML details. It permits to query over various granularity levels. Business processes are represented as directed labeled graphs.

The paper in [26] proposes the use of domain-independent and domain-specific ontologies for retrieving a web service from a repository by en-

riching web service descriptions with semantic associations. The domain-independent relationships are derived using an English thesaurus after tokenization and POS (Part Of Speech) tagging. The domain-specific ontological similarity is determined by associating semantic relationships with web service descriptions. Domain-independent terms allow for a wide coverage, while domain-specific ontological information allow for more in-depth finding exploiting industry and application specific terms. Matches due to the two ontologies are combined to determine an overall semantic similarity score.

The work in [27] provides a centralized knowledge base that can be used through case-based reasoning, a paradigm that reuses past knowledge stored in the form of cases. In this context, a case is a UML diagram enriched with some identifiers. It uses WordNet as a common sense ontology to provide classification of software projects described in UML.

ReDSeeDS (Requirements-Driven Software Development System) [28] is a web search engine designed to support reuse of software artifacts based on their requirements. The syntax of the artifacts is described by an Essential MOF (EMOF) while the requirements are specified by the Requirements Specification Language (RSL). The components of the RSL are requirement statements and use cases. The requirements statements are specified by natural language sentences, and the use cases are described by scenarios containing statements in structured English. The similarity of requirements is determined by combining information retrieval methods and similarity measures considering the semantic and word order similarity, as well as the structural similarity.

Chapter 3

Background

This chapter introduces concepts and terms that relate to the field of model-driven information retrieval and are used in the rest of the thesis. In Section 3.1 we briefly explain some of the most important ideas taken from the field of Information Retrieval. In Section 3.2 we give an insight of metamodeling concepts, while in Section 3.3 and in Section 3.4 we discuss about the two metamodels we addressed in our experiments.

3.1 General IR Architecture

An Information Retrieval System (IR) is a system that fetches raw data from some source of information, transforms it into searchable format and provides an interface to allow a user to search and retrieve that information by submitting queries to the system. Starting from this definition, four major processing subsystems can be isolated from the general flow [29]:

- *Content Registration*: this subsystem finds and retrieve from a given data source the items that are analyzed and searched in the following steps. This operation could be done in several ways, for example via crawling networks (on the Internet) as well as receiving new items that are "pushed" to the system (e.g. file system crawling).
- *Content Analysis*: this subsystem is concerned with the analysis and the consequent transformation of the raw data registered by the previous phase. The registered items undergo several elaborations such as tokenization, normalization, format standardization, stemming and other kinds of processing to get a canonical format from the original raw data. This phase can include other content analysis techniques to define and add some metadata to the items that could facilitate the

mapping between the vocabulary of the user and the vocabulary of the author of the original data during the search process.

- *Content Indexation*: this subsystem is concerned with taking the tokens of the normalized items and other normalized metadata to create the searchable index. There are many different approaches to create the index such as Boolean or Weighted. Within the Weighted approach there are the Statistical, Concept and Natural Language indexing approaches.

- *Search*: this subsystem is concerned with mapping the user information need to a processable form and determining which items are to be returned to the user. Within this process lies the calculation of the score of the retrieved documents that is used to order the list of displayed results.

- *Display*: this subsystem is concerned with how the user can locate the items of interest among all the possible results returned. It deals with display options such as highlighting and faceted navigation.

The quality of all these subsystems determines the capabilities in retrieving a higher number of relevant documents needed by the user and in displaying them in a suitable way. Each of these processing phases is addressed in the following sections.

In Figure 3.1 you can see a diagram depicting the general architecture of an Information Retrieval system.

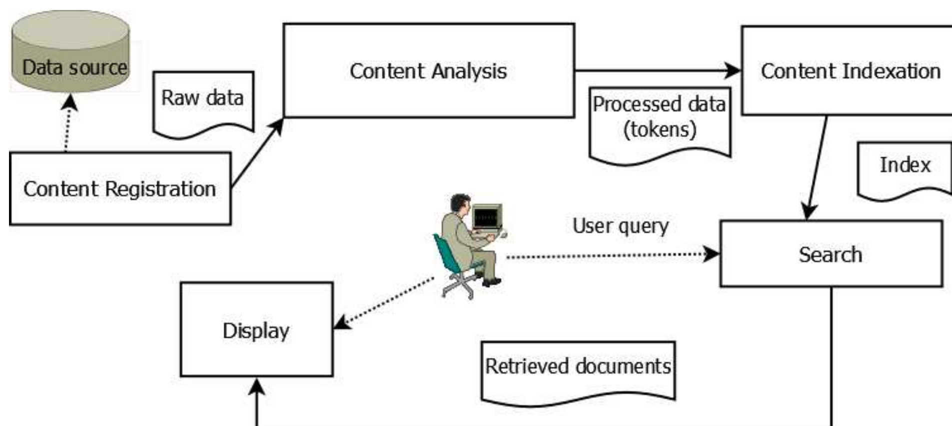


Figure 3.1: Architecture of a general-purpose Information Retrieval system.

3.1.1 Content Registration

Content Registration is the initial process of an information retrieval system. It is the process that receives the items to be stored and indexed and performs their initial processing. The crawling policy can be either a "pull" or "push" process. In the *pull* process the system inspects other locations to retrieve the items (e.g., web crawling). In the *push* process the items are delivered to the IR system. This typically means that one system, different from the IR system itself, writes files to a directory that is monitored and can detect new items. The Content Registration module usually checks whether an item has already been processed by the system. This is accomplished by creating a unique signature key that represents the content of such item. The most common methodology is to create a hash for the complete file.

3.1.2 Content Analysis

The Content Analysis (Figure 3.2) process takes as input the items gathered by the Content Registration.

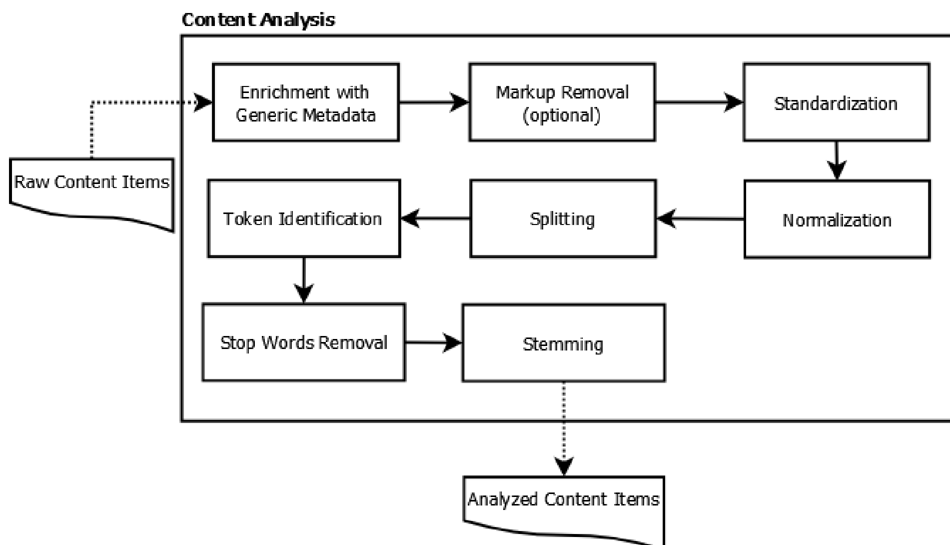


Figure 3.2: Operations involved in Content Analysis.

This subsystem is responsible for the extraction and transformation of information that will actually be part of the index. The Content Analyses produces several metadata that enrich the description of the registered items. Any item information that should be treated as metadata, like for example the date and time of creation, needs to be placed in the appropriate metadata

field. In case of items formatted in structured languages (e.g. HTML/XML), all the markups must be thrown away so that only continuous text is present. The next step could be the *standardization* of text format: this can be done first by inferring the language of the text and then putting it into UNICODE. Once the characters have been standardized to a single format, *normalization* is performed. This activity includes operations such as lower-casing, diacritic removal, and ligature expansion.

Once an item has been selected and normalized, the next step is to "split" the documents and then identify processing tokens for indexing. The *splitting* phase consists of parsing the item and subdividing it into logical sub-parts that have meaning to the user. This process is used to increase the precision of a search and to optimize the display of results. For example, if we want to index books (so in this case items are books), then splitting can be done by dividing the book item into Title, Author and Main Text. These parts will then be inserted in the appropriate index fields. The splitting of the items allows searches to be restricted to a specific part of the item. Another use of splitting and fields is when a user wants to display the results of a search. A major limitation is the size of the display screen which constraints the number of items visible for review. To overcome this problem, the user can decide to display only some splitted part of the original documents in order to browse an higher number of results.

Once the standardization and the splitting have been completed, the information used to create the index needs to be identified. Here, the effort is to analyze and transform the original words contained in the items. The elements that are found are called tokens. The *tokens* are the data that are finally indexed at the end of Content Analysis. Tokens are used instead of words because words are not the most efficient unit on which to base search structures. The first step of token identification consists in distinguishing the words of the items that are suitable for indexing. Generally, systems divide words into three classes: valid word symbols (alphabetic character and numbers), inter-word symbols (blanks, periods and semicolons) and special processing symbols (for example, hyphens). A word is defined as a contiguous set of word symbols bounded by inter-word symbols. In most systems inter-word symbols are non-searchable. Special symbols such as hyphens must be processed in special ways.

Token identification could be followed by word characterization, which includes morphological analysis. Thus, a word such as "plane" is interpreted as an adjective or as a noun according to morphological analysis or even context analysis.

Now that the potential list of processing tokens has been identified, some

can be removed by a Stop List or a Stop Algorithm. The objective of the *Stop function* is to delete from the set of searchable processing tokens those that have little relevance to the user. *Stop lists* are commonly found in most systems and consist of words (processing tokens) whose frequency and semantics use make them of no value. For example, parts of speech such as articles (e.g. "the") have no search value and should be thrown away. The *Stop algorithm* operates according to the Ziph's law, which postulates that, looking at the frequency of occurrence of the unique words across a corpus of items, the majority of unique words are found to occur a few times, so that the product of the frequency and the ranking of a word into the frequency histogram equals a constant.

One of the last transformations often applied to data before placing it in the searchable data structure is stemming. *Stemming* reduces the diversity of representations of a word to a canonical morphological representation called *stem*. The risk with stemming is that the discrimination of concept information may be lost in the process, causing a decrease of retrieving precision and affecting the ability of ranking. The positive aspect of stemming is that it improves recall. A very related operation is called lemmatization. *Lemmatization* is typically accomplished via dictionary look-up which is also one of the possible solution to implement stemming. Lemmatization, besides modifying word endings or dropping them as in stemming ("cats" and possibly "catlike", "catty", etc. are mapped to the root stem "cat"), maps word to another one (for example, it could map "eat" to "ate" and "better" to "good").

3.1.3 Content Indexation

This phase takes as input the processed tokens identified from the registered items. Its goal is to transform the received tokens into the searchable data structure. The index is what really defines an item more than its original content. This is because the primary mechanism to retrieve an item is based upon search of the index. If there are concepts in the items that are not reflected in the index, then a user will not find that item when searching for those concepts. In addition to the mapping of concepts to the searchable data structure, the indexing process may attempt to assign a weight on how much that item discusses a particular concept. This is used in the search phase in order to rank the retrieved documents. The attempt is to get the items more likely to be relevant higher in the list of retrieved documents.

In a weighted index system, each index term receives a weight (a positive scalar) that indicates the degree to which that term represents the related

concept in the original item. The most direct and obvious method to be used in weighting a term is the frequency of occurrence of that term in the item. The query process uses the weights assigned to terms that are present in the query to determine a scalar value for each item in the collection. This value is called *score* and it is used to predict the likelihood that a retrieved item satisfies the user query. There are several approaches to generate the searchable index. Here we discuss the statistical approach, which is then used in the rest of this work and it is the most prevalent in commercial systems. The basics of this approach is the use of the frequency of occurrence of tokens. The possible statistics that are applied to the tokens are probabilistic, Bayesian and vector space. We now illustrate the vector space, which is then adopted in the rest of this work.

The *Vector Space Model* approach is based on a vector model. The semantics of every item are represented as a vector. Each component of the vector represents a term in the vocabulary. A vector has the same dimension of the terms vocabulary. There are two possible domains of values for the vector's components: binary and weighted. Under the binary approach, the domain contains the value of one or zero, so the term is present or not present in the item. In the weighted approach, the domain is typically the set of all real positive numbers. The value for each term represents the relative importance of that term in representing the semantics of the item. A weighted vector acts the same as a binary vector but provides a range of values that accommodates a variance in the value of the relative importance of a term in representing the semantics of the item. Moreover, the use of weights also provides a basis for determining the rank of an item. Weights are determined using the classical Tf-idf weighting, which will be discussed in Section 5.2.3.

3.1.4 Search

The information retrieval processes continues after Content Registration, Content Analysis and Content Indexation with the search against the index. The selection and ranking of the items are accomplished via similarity measures that calculate the similarity between the user's search statement (user query) and the weighted stored representation of the semantics of each item in the index. Relevance feedback can also help a user to enhance the search by selecting items from previous ranked lists. This technique uses information from items judged as relevant or not to determine an expanded ranked list.

The search statements use Boolean Logic and/or Natural Language to

express user needs. The typical search statement consists of few words that the user selects to represent his information need. The user may have the ability to assign different levels of importance to different concepts in the statements (query terms boosting).

Then, the search statement is parsed by the system and used to search against the index. This process is similar to the indexing of an item described before.

The next step is to calculate the similarity between a user's search statement and the indexed items. Thinking about the Vector Space Model, both the user query and the indexed weighted documents can be treated as vectors where each element represents a different term. A variety of different measures can be used to calculate the similarity between the item and the search statement. A common characteristic of all similarity measures is that the result of the formula increases as the items become more similar. An example of similarity measure is the *cosine distance*, that calculates the cosine of the angle between the query vectors and the indexed documents vectors. As the cosine approaches "1", the two vectors approach the same direction, so the item and the query represent the same concept.

3.1.5 Display

Once a search has been completed, the system has identified an ordered list of items relevant with respect to the given user query. The next step is to present this information to the user. This step has a significant impact on the user's ability to find what he's really looking for. There are two stages of information displaying: the first one defines how the list of retrieved documents is presented to the user so he can easily find what it's important for him; the second one is how individual items are presented once the user has selected a specific one. A situation where the user can satisfy his information need without accessing a specific item in the retrieved list is considered exceptional.

What is obvious is that the "hit list" returned by the system contains the most relevant documents according to the system. What is less obvious is how long this list should be. Most systems display the hit list as a sequential list of retrieved documents organized in pages of 10 results per page but the user at most reviews two pages (20-30 results). The list can include the title of the retrieved items, a "snippet" of text coming from the item and a graphic overview of the item. Another interesting feature that can be offered by the system is the highlighting into the search results of the terms that are present in the user query.

3.2 Modeling and Model Driven Development

Our work addresses an issue typical of the *Model-Driven Engineering* practices, which is the use of models as the most important artifact to design and specify solutions. From this arises the problem of searching models stored in repositories, that is also the main topic of this thesis.

One of the currently most active branch of Model Driven Engineering is *Model-Driven Development* (MDD). This approach allows developers to use models to specify what system functionalities are required and what architecture is to be used instead of requiring them to use a programming language to specify how a system is implemented [30]. Code can be generated from the models, ranging from systems skeletons to complete, deployable products. MDD focuses the efforts of software development on the design phase with a greater attention to system architecture.

Model-Driven Architecture (MDA) is a set of standards proposed by *Object Management Group* (OMG) that support the architecture-focused approach of MDD. The standards include a language to write metamodels called the *Meta Object Facility* (MOF).

Metamodeling, a natural consequence of MDD, is the construction of a collection of "concepts" useful for modeling a predefined class of problems. This section introduces the fundamental aspects of metamodeling.

A model is a simplified representation of a certain reality, according to the rules of a certain modeling language. As the map represents a territory and conforms to its legend, the *conformance relationship* says that a model represents a system and conforms to a metamodel [31]. In Figure 3.3 you can see a picture showing this concept.

According to OMG standards, a metamodel is a special kind of model that specifies the abstract syntax of a modeling language. The abstract syntax of a language describes the vocabulary of concepts provided by the language and how they may be combined to create models. It consists of a definition of the concepts, the relationships that exists between concepts and how the concepts may be legally combined.

Figure 3.4 illustrates the traditional four layer infrastructure. This infrastructure consists of a hierarchy of model levels, each (except the top model) being characterized as "an instance" of the level above. Starting from the bottom of the hierarchy, the M0 layer is the real *system*. A *model* represents this system at level M1. This model conforms to its *metamodel* defined at level M2. The metamodel itself conforms to the *meta-metamodel* at level M3. The meta-metamodel conforms to itself. An example of a level M2 is the WebML metamodel, the model that describes WebML itself. M2-

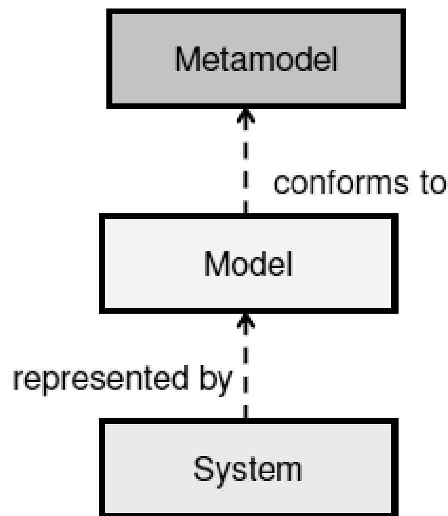


Figure 3.3: Conformance relationship: a model represents a system and conforms to a metamodel.

models describe elements of the M1-layer, and thus M1-models. These would be, for example, models written in WebML. The last layer includes real data and real world objects. UML is a special case because it can be used for describing itself, so it can be used both as a model and as a metamodel.

In language specifications the abstract syntax of the language is specified as a MOF-compliant metamodel. MOF provides the standard modeling and interchange constructs used in MDA. These constructs are a subset of UML modeling constructs, essentially the Class Diagram subset of UML: object attributes, relationships between objects, operations available on objects and simple constraints (e.g., multiplicity). The Eclipse Modeling Framework Project (EMF) includes Ecore, a MOF-like core metamodel.

EMF provides a pluggable framework to store model information, the default uses XMI (XML Metadata Interchange) to persists the model definition. XMI defines an XML-based exchange format for models of M3, M2 and M1 layer and is also the supporting standard of MOF. The typical usage scenario of the metamodeling practices within a system design project and development would be: metamodel specification, model instance generation from the previously created metamodel, generation of Java code for a model, refinement.

As mentioned, the increasing use of metamodeling and MDD brings up the problem of searching an already designed solution. In the following

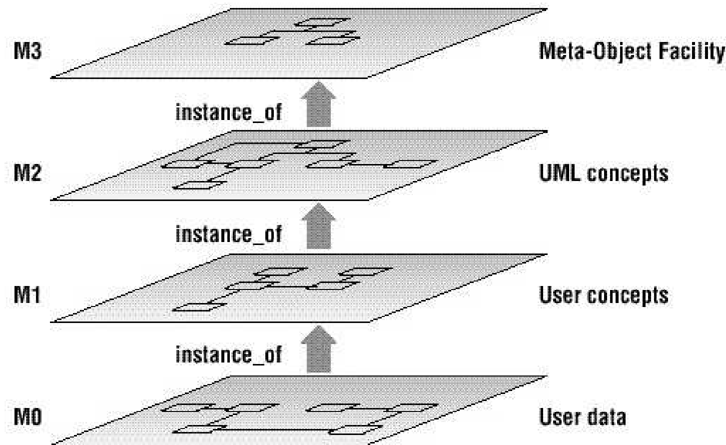


Figure 3.4: Traditional OMG's metamodeling infrastructure with four layers.

chapter we propose an abstract solution for our Model Driven Information Retrieval System and we discuss our implementation experience in which we used two different datasets. The first one is a dataset of models expressed in a general-purpose modeling language, UML, and the second one is expressed in a domain-specific modeling language, WebML.

3.3 UML

Our first implementation case is based on a dataset of UML models. In section 6.1 we describe the UML dataset with further details, while here we briefly describe general characteristics about UML.

UML (Unified Modeling Language) is a semi-formal visual modeling language. It is based on the object-oriented paradigm. UML is a standard used for the specification of software projects. A UML project is composed by a set of diagrams that can be divided in two categories:

- *structural*: they empathize the static structure of a system using objects, attributes, operations and associations. Some examples of structural diagrams are the *class diagram* and the *component diagram*,
- *behavioral*: they describe the dynamic functioning of the components showing the interaction between objects, their internal state and the information flow. Examples of such diagrams are *activity diagrams* and *use case diagrams*.

Our dataset of project models conforming to the UML metamodel consists of a set of class diagrams describing in turn several modeling languages (e.g. HTML, XML, etc.). The details about the metamodel and the dataset are given in Section 6.1.

3.4 WebML

Web Modeling Language (WebML) is a high-level modeling language for designing complex data-intensive Web applications [32]. In essence, WebML consists of simple visual concepts for expressing an hypertext as a set of pages made up of linked content and operation units. It also expresses the binding of such content units and operations to the the data they refer to. Pages are then grouped into wider concepts, such as *areas* and *site views*.

The specification of a Web application through WebML requires four orthogonal perspectives:

- *data model*, defines the data content of the application in terms of entities and relationships. This model takes deep inspiration from the Entity-Relationship database model,
- *hypertext model*, defines one or more hypertexts that can be published. The goal of hypertext modeling is to specify the organization of the front-end interfaces of the Web application. The key ingredients of WebML are pages, units and links, organized into modularized constructs called areas and site views. *Units* are the atomic pieces of publishable content. *Pages* are the actual interface elements delivered to the users and are built by assembling together units of various kinds. Pages and units are linked to form a hypertext structure. *Links* express both the possibility to navigate from one point to another in the hypertext (non-contextual links) and the passage of parameters from one unit to another (contextual-links), which is required for the page computation,
- *presentation model*, defines the layout and graphic appearance of pages, in a way not depending on the output device,
- *personalization model*, defines the user categories and user groups.

In Figure 3.5 [18] you can see a very small example of WebML specification diagram (Figure 3.5(a)) and an excerpt of the WebML metamodel (Figure 3.5(b)). Figure 3.5(a) depicts two pages. The page "Catalogue Home Page" includes an index unit named "List Products" and a data unit named

"Product Details". The two units are connected through a contextual link named "View Details" that transports the identifier of the object selected from the index to the data unit displaying the object details. The two pages are connected through an inter-page link, that is a link crossing the boundaries of pages. Figure 3.5(b) shows a piece of the WebML metamodel describing the site view construct. A site view consists of areas, which in turn may contain pages. Pages include content units for publishing content. Areas may contain operation units whereby users can invoke operations on the site. Units are connected through links that carry parameters (e.g., ids) and trigger the navigation from one page to another.

(a) A small example of WebML specification diagram.

(b) An excerpt of the WebML metamodel.

Figure 3.5: WebML diagram example and metamodel.

All WebML concepts are associated to a graphic representation so that the projects specifications are diagrams. The specification of an application through a diagram hides the full XML representation which encodes the project.

In the code below you can see a small snippet extracted from the XML representation of a WebML Web application:

```
<SiteView name="Product Catalogue">
  <Page id="pag1" name="Catalogue Home Page">
    <IndexUnit id="inu1" name="List Products">
        <Com>List of products in the catalouge</Com>
      <Link name="View Details" dest="dau1" />
    </IndexUnit>
    <DataUnit id="dau1" name="Product Details">
        <Com>Details of a selected product</Com>
    </DataUnit>
    <Link name="Contacts" dest="pag2" />
  </Page>
  <Page id="pag2" name="ContactPage">
   ....
  </Page>
</SiteView>
```

Chapter 4

Approach

This chapter describes our approach for retrieving software artifacts from repositories. In Section 4.1 we describe an abstract solution, that is an ideal Model-Driven Information Retrieval System where, for a given dataset, only a metamodel is needed as input by the indexing and searching process. In Section 4.2 we define the relevant design dimensions for this kind of system. In Section 4.3 we show the indexing strategies we decided to adopt for our experiments.

4.1 Abstract Solution

Our abstract solution for retrieving software artifacts adopts an approach that includes two main information flows as seen in Figure 4.1. In particular, the upper part of the figure depicts the *Content Processing Flow* and the lower part the *Query Flow*.

The information flows are processed by two pipelines: for the content part there is the Content Processing Pipeline, which is triggered by the crawler subsystem, while for the query part there is the Query Processing Pipeline, which is triggered by the user query input. The content processing flow gathers, processes and transforms significant information from the projects in order to make them ready for indexing and searching. The query flow is intended to accept, process and transform the user query in the same way as the content information. Then it performs the searching operations and returns the results to the user interface.

The *Content Processing Pipeline* involves as data source a collection of models that conform to a metamodel. The metamodel can be of any type. The very first phase is the translation of the original models into the XMI format. This translation is not performed by the system but must

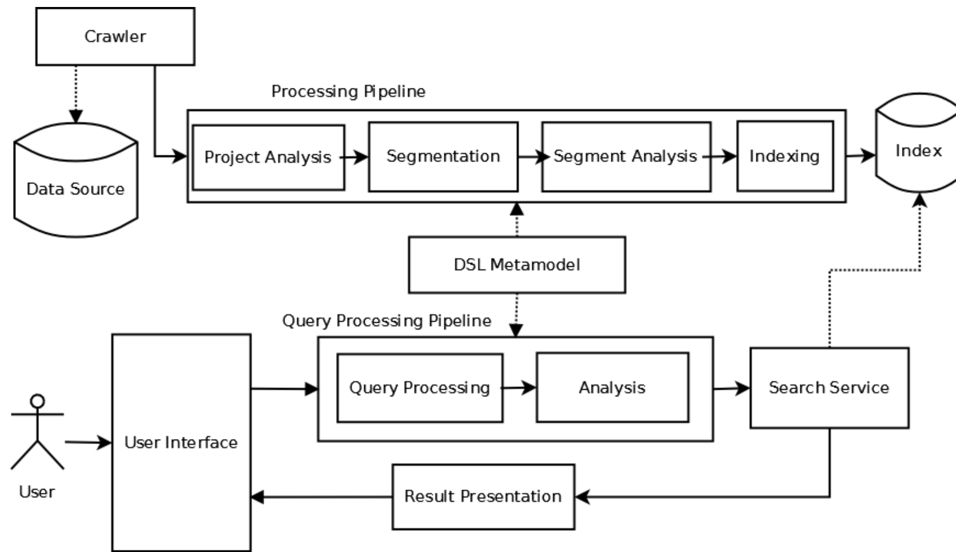


Figure 4.1: The approach of our abstract solution for a general search engine model repository system. In the upper part there is the Content Processing Flow *and in the lower part there is the* Query Processing Flow*.*

be provided by the user. XMI (XML Metadata Interchange) is an Object Management Group (OMG) standard for exchanging metadata information via XML. This format can be used for any metadata whose metamodel can be expressed in Meta-Object Facility (MOF). As XMI is a standard way to represent the project model, their translation eases the automatic data extraction in the following phases.

First, the *crawler* ingests the structure of the entire project. This is important to propagate as much information about the project structure as possible, so that it can be reused in the following operations. This means that the entire XMI code expressing the structure of the whole project is carried forward through the operations chain. This also keeps the indexing and searching approach model-driven.

Next, the Content Processing Pipeline begins. The pipeline contains the following operations: Project Analysis, Segmentation, Segment Analysis and Indexing. Our abstract solution requires that the system has a set of standard routines for each of these operations. The routines are then instantiated according to the metamodel. In general, these routines mine useful elements from the XMI project representation. These elements are tagged by the user in the metamodel with proper tags in order to automatically generate the routines. The routines also save the extracted elements

into suitable data structures, such as records of the SMILA framework. The data structures are passed as input to each successive phase in the pipeline.

The *Project Analysis* extracts generic metadata referring to the entire project, for example the authors and the creation date. The user has already tagged these metadata into the metamodel. The metadata tagged in this way are extracted from the models and saved in the data structures.

Segmentation splits the initial projects into smaller units which are more manageable for the successive operations. As in the previous phase, the user has tagged the metamodel element representing the segmentation unit. A proper routine extracts from the XMI project representation the codes fragment referring to the considered units and saves them in the data structures of the segments.

The *Segment Analysis* mines the elements that will be indexed later from the previously obtained segment. These elements have also been tagged by the user with suitable tags in the metamodel. The Segment Analysis performs text processing on the just extracted elements. The user can configure the analysis type to be performed (e.g. tokenization, normalization, stemming, etc.) for each type of element. The analysis generally consists of a sequence of *analyzers* through which the project model words are transformed.

The last phase is the *Indexing* that stores the data structures obtained up to this point. The storage schema is previously defined taking into account the extracted information.

The *Query Processing Flow* deals with the query ingestion from the user interface, the query processing and analysis, the index searching and the results presentation back to the user interface. The query can be submitted in keyword-based form (textual queries) or content-based form (the query is a model fragment). Here we expect that the query is submitted in content-based form with the same XMI format of project models. This can also be done in a graphical way. If necessary, also the original representation of the query undergoes the same translation performed on the project models.

The *Query Processing Flow* mines the keywords that will be part of the query string from the content-based query expressed in XMI. It basically strips all the markup code and keeps only the names of the project elements given in that particular query instance. The metamodel is needed to extract the keywords from the user query.

The *Analysis* applies the same processing techniques used for the content information. Here, the metamodel is needed to match a given model element to its sequence of analysis.

The *Searching* takes the query string and performs the actual search

against the index in order to find relevant *documents* with respect to the query. The documents are the indexed representation of the projects containing the same information extracted during the content processing flow. The matching between the query string and the indexed documents is performed adopting a specific similarity measure that computes the distance between the query and the documents. The results are returned as a ranked list of relevant documents, which is then presented to the user.

4.2 Design Dimensions

Differently from a generic information retrieval system, a system that exploits information from the documents' domain specific language (DSL) metamodel has some peculiar dimensions to specify in its design [18]:

- *Segmentation granularity*: this is the atomic unit that the IR system retrieves. The granularity can be at different model levels: entire project, subproject or metamodel concept.
- *Elements to extract from the models*: only the most significant elements are extracted from the segments. These elements are the ones that will be processed by the Content Processing Pipeline and searchable through the index where they are stored.
- *Index structure*: the index contains the information extracted from segments in the form of *documents*. Typically each segment extracted from the project models is a different document into the index. The index type can be flat, weighted, multi-field or structured.
- *Query type and result visualization issues*: these parts involve many kinds of design choices, including the query type (keyword-based, document-based, search by example, faceted search) and visualization (snippet visualization, highlighting).

In the following, we explain each design dimension with further details and we specify to which phase of the abstract solution presented in Section 4.1 the design dimension refers.

Segmentation granularity This is a very important dimension which defines the atomic unit that the IR system can process, index and search. The granularity is the level at which the entire project is sliced. This means that the granularity segment corresponds to the size of the documents that user searches through a query and that is present in the ranked list returned

by the IR system. The dimension of segmentation granularity affects the whole Content Processing pipeline (Section 4.1) and, in particular, the Segmentation phase, where the actual splitting is performed. An indexable document can correspond to:

- *Entire project*: in this case, no actual segmentation is performed, an indexable document is equivalent to a project into the repository and the query result is a ranked list of projects.
- *Subproject*: an indexable document corresponds to a smaller piece of the original entire project model. The entire project models have to be split in smaller parts according to some criteria.
- *Metamodel concept*: it's the segment granularity at the lowest level of slicing; in this case an indexable document corresponds to a metamodel concept. A *metamodel concept* is an element of the metamodel of the language used to express the project models. For example, a metamodel concept for WebML is "area", while for UML is "class". Every indexable document corresponds to a concept in the metamodel. Every concept contains a reference to its container element and possibly references to other related concepts. The query result is a ranked list of model concepts, possibly of different types. The user can browse their related concepts.

Elements to extract from the models The segments from the project models can hold different information, thus some of them could be not useful for the purposes of the IR system. Therefore, the designer studies the metamodel elements, their semantics and significance. After that, he chooses the most suitable elements that will be indexed. The actual extraction of the elements that are suitable for indexation is performed in the Segment Analysis phase (Section 4.1).

For example, if we're developing a text-based search engine that searches and retrieves models conforming to a UML class diagram metamodel, the models may include many information, such as the visibility of the class members. In this phase, the designer may notice that this information is not useful for searching purposes and so he may decide to not include it into the index.

Index structure The index structure defines the way the segments and their related information are represented as *documents* into the index. Designing the index structure resembles the design of a database schema. An

index structure consists of one or more fields. The division of the index into fields allows the user to match different parts of the query string to specific fields of the index. These types of queries are called multi-field query. The design dimensions that refer to the index structure affect the Indexing phase of the approach presented in Section 4.1.

The following list shows the options that can be used to design the index:

- *Flat*: this is the baseline for a model-driven IR system. The index structure is single-fielded and stores bags of words in an undifferentiated way. All the elements extracted from the models are put in the only present field without taking into account the metamodel concept of that element, the relationships with other elements or its structure.

- *Weighted*: the index is still single-fielded but this time the terms are weighted according to their concept. The ranking algorithm will give an higher significance to terms occurring in more important concepts. Here we use the words "weight" and "payload" as synonyms. This kind of solution is the same one adopted in some of the experiments we have implemented that are discussed in Section 6.2 and Section 6.4.

- *Multi-field*: the index has several fields which contain terms belonging to different concepts. The index is said to be multi-field and each field is searchable separately (multi-field query). One can also decide to assign a specific weight to a field during the Searching process. The ranking algorithm gives different importance to matches based on the field where they occur according to the specific similarity measure adopted by the system. Notice that this kind of weighting is significantly different from the terms' payload discussed in section 6.2 and 6.4. This solution can also be combined with the weighted approach producing a multi-field weighted index.

- *Structured*: the structured approach represents the model in a way reflecting the hierarchies, associations and relationships among concepts. The index model can be semi-structured (XML-based) or structured (e.g. the catalog of a relational database). In this case the query processing can use a structured query language (e.g., SQL) coupled with functions for string matching.

Query type and result visualization issues In the context of model searching, an IR system can offer different methods for query submission and result visualization options. Regarding the query submission modalities, in the most basic case, the user can submit a *keyword-based query*,

providing a set of keywords that the system matches to the indexed documents. The system then returns a ranked list of documents according to their relevance with respect to the query. In the *document-based search*, the user provides a document (the representation of a project) as query. The system analyzes the document, extracts the relevant keywords and submits them as the actual query. In the *search by example* approach, the user can provide a model as a query. The model is first analyzed in the same way as the project in the data source repository by the Processing Pipeline. This analysis produces a document to be used as a query. Here for document we mean any representation of the model used for matching, which can be a bag of words, a feature vector in the Vector Space Model or a graph in a graph-based searching approach. In the *faceted search* the user explores the repository using *facets*, that typically correspond to the possible values of an indexed field. Another possibility is to submit an initial query first, and then filter the results using faceted navigation. The application of facets refines the results obtained by the query.

Regarding the presentation of query results, there are several ways to improve the user experience when browsing such list. For example, each item in the result set can be associated with an informative visualization of the result (*snippet visualization*). This informative visualization can be a transcription of indexed documents in a flattened textual form or a graphical representation (e.g., for a UML model, one can decide to plot the UML diagram snippet of the returned result). Another way of improving the user experience is by *highlighting* the matched terms into the textual transcription of the returned documents.

4.3 Indexing Strategies

When designing a model search engine, there are several combinations of the design dimensions discussed in Section 4.2 that can be adopted. In this section we present the configurations of the above dimensions that we decided to adopt for our experiments.
We tested the following scenarios:

- *Experiment A*: in this experiment the segmentation granularity is "entire project"; the index structure is almost totally flat (in addition to the field that contains all the elements of the project, there is only a field with the project name for visualization purposes). In the following of this work, the Experiment A will be labeled as *Project Granularity, Flat Index*.

- *Experiment B*: this experiment involves a smaller segmentation granularity that corresponds to a metamodel concept; the index structure is multi-field and the terms are stored without any kind of weighting. This experiment provides a baseline against which possible strategy improvements (like the introduction of weights) will be evaluated. In the following of this work, the Experiment B will be labeled as *Concept Granularity, Multi-Field Index.*

- *Experiment C*: this experiment adopts the same segmentation granularity as of the previous experiment; the index structure is multi-field and, differently from the previous experiment, the index terms are weighted according to the metamodel concept which they belong to. The idea is to assign a different degree of relevance to the different metamodel concepts depending on their importance. In this way, the system would rank at higher positions documents in which the matched terms belong to more relevant concepts. In the following of this work, the Experiment C will be labeled as *Concept Granularity, Multi-Field Weighted Index.*

- *Experiment D*: in this experiment the segmentation granularity is the same as in the previous two experiments; the index structure is still multi-field and the terms are still weighted according to their metamodel concept; the novelty of this experiment lies in the algorithm included before the indexing phase. This algorithm first creates a graph representation of the considered model, using as nodes the elements corresponding to the selected segmentation granularity and as edges their respective relationships. Then, each element is enriched with some information harvested from its neighbours. The idea is to let the system be able to retrieve not only the elements that match a search, but also their neighbours. This could allow some usually overlooked elements to gain importance in the ranking list, thus discovering new solutions. In the following of this work, the Experiment D will be labeled as *Concept Granularity, Multi-Field Weighted Index, Graph Based.*

Case Studies We have implemented two case studies in order to test the above experiments. The first case study deals with a UML class diagram repository. In the second one, the repository consists of WebML projects.

We implemented experiments B and C both for the UML and the WebML case. Experiment A was not developed for the WebML case because it has already been studied here [18]. We tested the Experiment D only on the

UML repository because the chosen segmentation granularity of the WebML case along with the structure of the WebML projects make this experiment unsuitable for that scenario. The details of the implementation of the two case studies are reported in Chapter 6.

Chapter 5

Implementation

In Section 5.1 we present SMILA, the framework we adopted to develop our prototype and in Section 5.2 we provide a brief introduction to Solr, that we adopted as search platform. Section 5.3 describes some implementation details of the architecture of the text-based prototype we developed, while Section 5.5 shows the configurator and its user interface.

5.1 SMILA

SMILA is an extensible framework which allows to build search applications able to access different data sources containing non-structured information. SMILA provides an essential infrastructure comprised of fundamental components and services, which are extendable and exploitable in your own application. We proceed now to illustrate a brief overview of the behavior of these components.

5.1.1 Architecture

Figure 5.1 shows the general SMILA architecture. It is possible to divide this infrastructure in two parts, each corresponding to a different phase of the data processing. The first phase is called pre-processing (left section of the diagram) and the second phase is called information retrieval (right section). In the case of a search application these two phases are commonly referenced as *indexing* and *search* respectively.

Indexing: The indexing phase includes the raw information gathering from the data source. The gathered information generally includes the metadata and the content of the documents as well as possibly other security related information, like access rights.

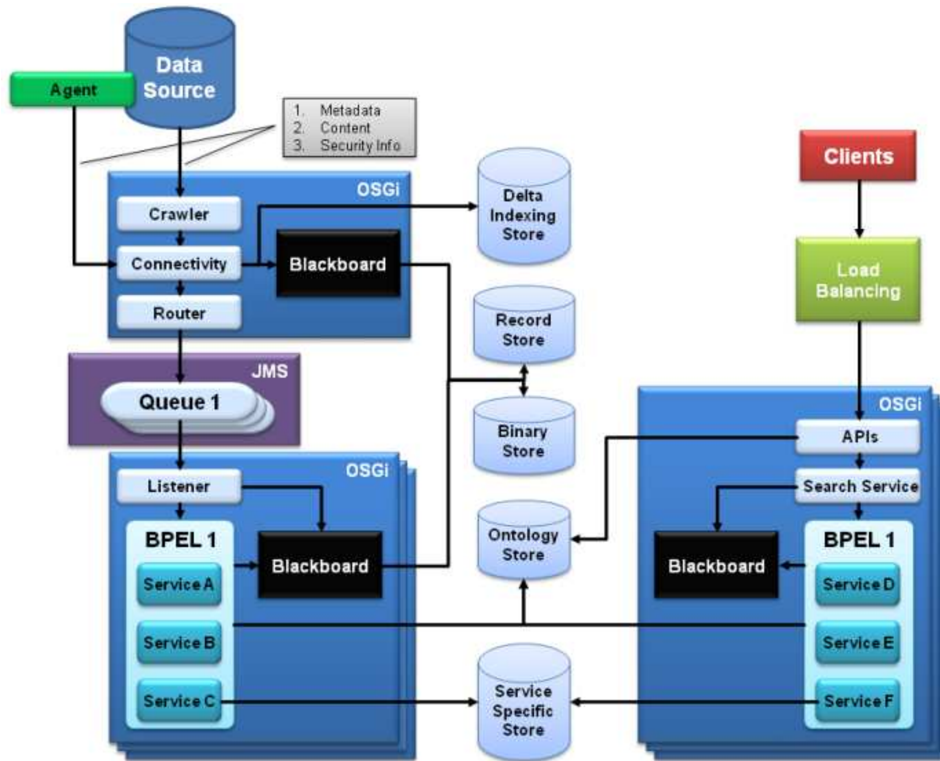


Figure 5.1: SMILA general architecture

This is a summary of the purpose and of the behavior of the main components.

- *Agent/Crawler*: information gathering might be done in two ways, namely push or pull. The agents work in the push way. An agent is always active and constantly monitors the data source. It sends an object of the data source to be elaborated immediately after having found some differences with respect to the same one present in the SMILA storage. This is useful in case of dynamic data sources in which information changes rapidly and/or constantly over time.

  Crawlers gather information in the pull manner. A crawler is started manually and navigates through the data source, sending gathered info to the system to be processed. Once finished, the crawler stops and must be restarted manually if needed. This is good for one-time indexing, when we know that our data will not be subject to changes.

- *Storage*: this is where the information is stored. SMILA provides two kinds of storage: the Record Storage which stores metadata and

access rights of the documents and the Binary Storage which stores the binary content of a document. This division is useful because the binary content is rarely used since indexing operations often use only metadata. Also, copying binary content of the record in the pipeline is usually expensive since it is much bigger than the text content. Thus, this system allows to push in the pipeline only the id and the metadata necessary to identify the record with the possibility to retrieve the binary content only if it is really needed.

- *Delta Indexing*: this module stores information related to the last modification of every document and is able to determine which documents have changed since the last time indexing was performed. This improves the performances during the crawling phase because only the documents that actually need processing are forwarded to the next parts of the workflow chain.

- *Ontology Store*: it's a storage dedicated to the management of ontologies.

- *Blackboard*: the Blackboard is an interface between the system services and the storages. It is an abstraction layer that has the purpose to hide the persistency technology of the records to the services, so that they need not to know what persisting technology is used by the system in order to manipulate the data of the records. Complete record data is stored only on the blackboard which is not pushed through the workflow engine itself. Only the ids of the records are pushed through the workflow. They can then be used to access the record's data via the blackboard API.

- *Router*: After a record has been stored, a message is created. The router sends this message to a queue.

- *Queue*: the queue processes the messages sent by the router and acts as a bridge between the information gathering section and the information processing section. The queue is a fundamental component that increases the scalability of the system. The messages are processed remotely and is easy to spread their elaboration on more clusters. The queue introduces asynchronous execution of the business logic. The technology used is JMS (Java Message Service). SMILA includes ActiveMQ as default provider of JMS services.

- *Listener*: this module draws messages from the queue and starts the right BPEL workflow depending on the configuration rules. Both the

Listener and the Router are configurable using a set of rules which specify the correct way of dispatching of the messages.

- *BPEL Engine*: here the user can define whatever workflow he wants using the BPEL (Business Process Execution Language) format. This workflow will contain the business logic of the application and will perform processing operations on the records. According to SMILA terminology, a workflow is called *pipeline* and the modules of which it is composed are called *pipelets*. A pipelet is a reusable and configurable component and can be orchestrated like any other BPEL service. This means that an instance of a pipelet is not shared across more pipelines. Also, calls to the same pipelet in the same pipeline don't share the same instance. An instance can be accessed by multiple threads, so pipelets need to be developed according to thread-safeness concepts.

Search: The search phase provides access to the previously indexed information. This process is synchronous, so it's necessary to provide an external component in order to allow load balancing to achieve horizontal scalability. It is possible to define a workflow executing business logic managed by a BPEL engine during this phase too.

5.1.2 Data Model

Data in SMILA are represented by a record composed by metadata and attachments. A record can correspond to a document or to whatever resource destined to be indexed and searched.

Metadata contain Value types organized in Maps (key-value associations) and Sequences (lists of anything). The Values can be primitive Java types or Date types. Maps and sequences can be nested arbitrarily. Attachments can contain binary data (byte arrays).

The elements contained in metadata can be interpreted in various ways:

- *Attributes*: this is the most common situation in which the record represents an object of the data source that has to be processed or retrieved by a search. For example, some typical attributes of a web page are its URL, its title, and its textual content. So, the attributes are defined by the specific application domain.

- *Parameters*: there are some record types for which attributes are not adequate. For example, in the searching phase, a record doesn't represent an object of the data source to process but it represents a search

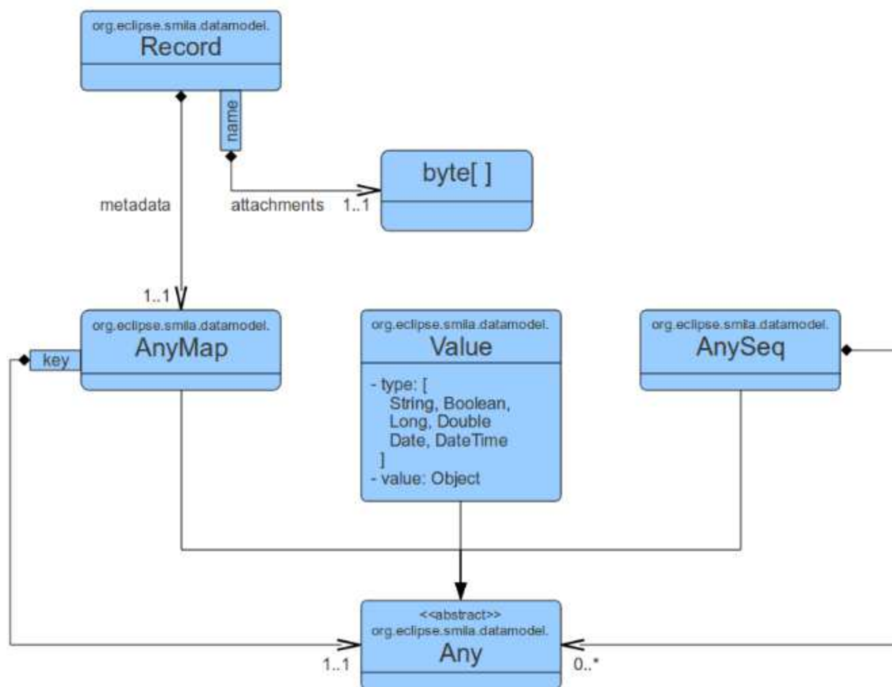


Figure 5.2: SMILA data model

request object. In this case, the record doesn't contain data of the object of the application domain, but it contains the search parameters needed to configure the request.

- *Annotations*: it is possible to enrich the information contained in a record during the processing phase by adding other attributes other than the ones retrieved during the crawling phase. These new attributes are called annotations.

- *System attributes*: these are attributes needed by SMILA to coordinate the processing of a record. The name of these attributes begins with an underscore to discriminate them from the application specific attributes. System attributes include the record ID, unique for every processed record. There isn't a predefined format, so the ID can be built by any string. All the records must contain also another ID that represents the data source from which they have been generated (for example the crawler definition).

5.2 Solr

Solr is an open source enterprise search server developed by the Apache Software Foundation. In addition to the standard ability to return a list of search results for some query, it has numerous other features such as result highlighting, faceted navigation, query spell correction and auto-spell queries. The core technology behind Solr is Apache Lucene, an open source, high-performance full-text search engine library. Differently from Lucene that is just a code library, Solr is a search *server* platform that is easily configurable with XML configuration files. In order to use Lucene directly, one should write code to store and query the index.
The major features of Lucene are the following [33]:

- a text-based inverted index persistent storage for efficient retrieval of documents by indexed terms,
- a rich set of text analyzers to transform a string of text into a series of tokens, which are the fundamental units indexed and searched,
- a query syntax with a parser and a variety of query types, from a simple term lookup to exotic fuzzy matches,
- a good scoring algorithm based on Information Retrieval principles to retrieve the more likely candidate first.

Solr can be described as the "server-ization of Lucene", that is, Solr makes easier the use of Lucene search services by its clients. Solr is executed within a servlet container, such as Apache Tomcat. Clients communicate with Solr by means of HTTP requests. Solr follows the Representational State Transfer (REST) paradigm. The server and schema properties are configured by XML files. Here is the major feature-set in Solr:

- HTTP request processing for indexing and querying documents,
- configuration files for the schema and the server itself through XML,
- Lucene's text analysis library is configurable through XML,
- notion of field type.

The Solr HTTP interface has two main access points: the *update* URL for index management and the *select* URL for query submission. An index is structured in *fields*, each entry in the index is a *document*. Adding new documents to the index is done through a HTTP request using the

POST method. The request body includes the XML representation of the document as index fields. Each document has a unique identifier which is specified in the XML representation using a special field. Documents can potentially be of any type like XML, CSV or "rich documents" such as Word files. It is also possible to define special routines in order to import data with complex structures from relational databases. Moreover, any client able to submit HTTP requests can communicate with the Solr server. As soon as the indexing is completed, it is possible to issue a new HTTP request pointing to the select URL to submit a query to the index.

In Figure 5.3 you can see a diagram summing up all the possible inputs and outputs managed by Solr and its general index composition.

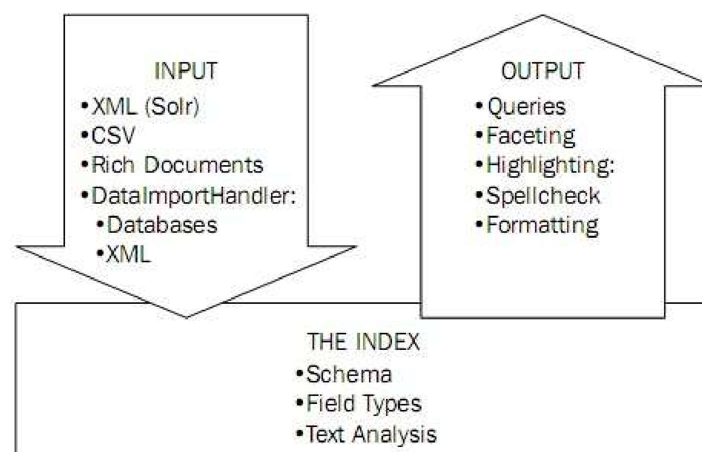


Figure 5.3: Diagram summing up all the possible inputs and outputs and the composition of a Solr index.

The stages to develop a search engine with Solr are essentially three:

- *schema design*: maps the original schema of the considered data into a Solr index, which is necessarily flat (one could face the task of mapping a relational database into a Solr index),
- *schema definition*: configures the *schema.xml* configuration file where the index elements are defined; this file includes the definitions of the fields and of the field types,
- *text analysis configuration*: configures the way the text is analyzed and processed (for example, tokenization and normalization) before indexing; this configuration influences the document retrieval.

The following sections explain in more detail some fundamental features of Solr usage: Section 5.2.1 introduces the concept of field and field type in index design, Section 5.2.2 is about the most important text analysis operations; then Section 5.2.3 provides more details concerning the possible queries for searching the index, the Solr response to queries and factors influencing the score of retrieved documents.

5.2.1 Design and Index Definition

A database and a search index have several conceptual differences. An index is like a very big relational table from a database, but has no support for relational queries (joins). Other differences are:

- in an index the search is done by term and not only by substring matching; this means that it is possible to find different forms of the same words,
- Solr, and more generally every search engine, can retrieve a list of ordered results according to a certain measure of relevance with respect to a given generic query instead of an unordered set of documents obtained by a very specific structured request.

Another important factor to keep in mind when designing a schema index is that every possible data needed to retrieve a certain document must be present in the document representation itself, as it is not possible to use relational queries.

When the index design is done, the next task is defining the actual schema. The first thing to do is defining field types. A *field type* is a data type that can be used in the index. A field type declares its type (boolean, number, date, etc.), has a unique name and it is implemented by a Java class. Next, the fields are defined. A *field* is the atomic cell where data coming from documents is saved. Each field has a unique name, a type chosen among the field types, plus other optional configurations. Field values may be “stored” or “indexed”. A *stored* field can be retrieved during search and then visualized but it is not searchable; an *indexed* field is searchable and its content is not retrievable for displaying. There is the opportunity to have a field that is indexed but not stored or vice versa, or a field that is both indexed and stored.

5.2.2 Text Analysis

Text analysis covers the most important techniques for text processing used on raw data in input: tokenization, case normalization, stemming, synonyms, etc. The goal of this stage is to analyze the text and transform it in a sequence of terms. A *term* is the core atomic unit saved into a field of a Solr index. Terms are what Solr searches at query time.

Thanks to Solr and its configurable infrastructure, the text analysis configuration is straightforward. Each field type has two *analysis chains* attached, each of them defines an ordered sequence of analysis steps that convert the original text in a sequence of terms. There is a sequence of analysis steps for the indexing phase and another one only for queries. Each step has an associated *analyzer*. There are several types of analyzer that perform a lot of different processing tasks: they tokenize the text, filter tokens, add terms and modify terms. The first analyzer of an analysis chain is always a *tokenizer*; its job is to divide the original text in tokens using a simple algorithm (for example, it generates a new token every white space). A *token* is the smallest unit which is matched to a query during search. After the tokenizer, the remaining analyzers are defined as *filters*, and their job is to further transform the tokens. the actual transformation performed depend on the application and are at the designer's discretion. In general, an analyzer is a TokenStream factory, which iterates over tokens. The input is always a character stream.

In Figure 5.4 you can see a diagram depicting some of the analyzers offered by Solr and their hierarchical organization.

5.2.3 Documents Search

After the indexing phase, it is possible to submit queries to the index. Solr has a very useful and easy to use web-based interface. There are several parameters to better define the queries; here we list only some of them:

- *q*: the query string provided in input by the user,
- *q.op*: specifies whether all or just one term in the query should be present in the document so that it can be retrieved,
- *df*: specifies the default search field.

It is possible to use the classical boolean operators AND, OR and NOT, specify sub-expressions, search in a specific field, perform a *phrase query* (a set of terms to be found all together into documents) or using the *score*

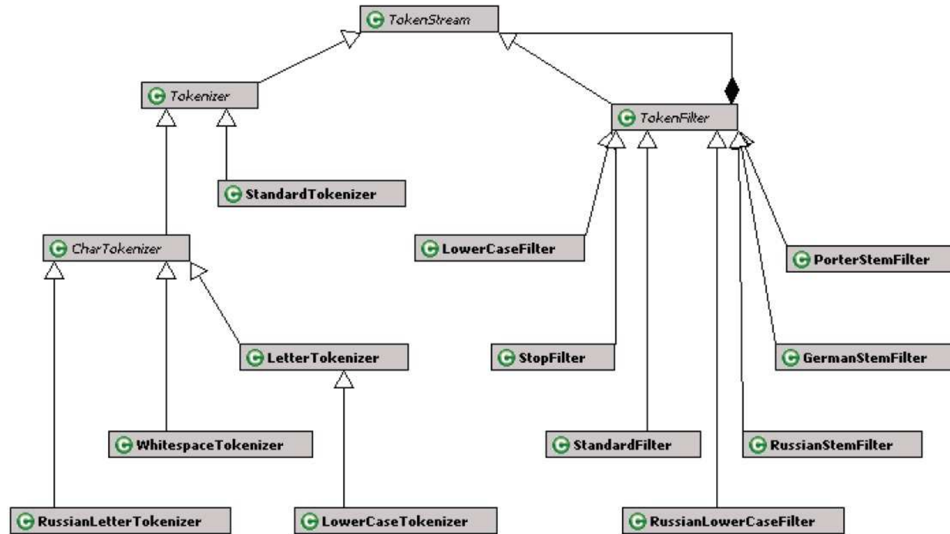


Figure 5.4: Hierarchical organization of Solr analyzers.

boosting that modifies the degree to which a term contributes to the final score of a document. After submitting the query, Solr returns as output a XML document containing the list of retrieved documents and their score. It is also possible to highlight the searched terms among the returned results.

The query processing and parsing in Solr is done through request handlers. A *request handler* performs the search and allows to configure the search parameters and to register some additional components, such as highlighting.

Another important aspect concerns how Lucene and Solr compute the score of a document with respect to a query. Lucene combines the Boolean model (BM) with the Vector Space Model (VSM): the documents "approved" by the BM are scored by the VSM. In the VSM, documents and queries are represented as weighted vectors in a multi-dimensional space, where each term of the whole vocabulary is a dimension (an axis), and weights are Tf-idf values. The VSM score of a document d for a query q is obtained through the *Cosine Similarity* of the weighted query vectors *V(q)* and *V(d)*:

$$CosineSimilarity(q, d) = \frac{V(q) \cdot V(d)}{|V(q)||V(d)|}$$

Where $V(q) \cdot V(d)$ is the dot product of the weighted vectors, and $|V(q)|$ and $|V(d)|$ are their Euclidean norms. Lucene refines this formula in a simplified way

as terms and documents are fielded. The practical scoring function used by Lucene is the following one:

$$score(q,d) = coord(q,d) \times queryNorm(q) \times \sum_{t \in q} (tf(t \in d) \times idf(t)^2 \times norm(t,d))$$

where,

- $coord(q,d)$, is a score factor based on how many terms of the query q are found in the given document d. A document that contains more query terms will receive a higher score than a document containing fewer query terms,
- $queryNorm(q)$, is a normalizing factor used to make scores between queries comparable. This factor does not affect document ranking, since all ranked documents are multiplied by the same factor, but rather just attempts to make scores from different queries comparable,
- $tf(t \in d)$ is the term's frequency, defined as the number of times t appears in the currently scored document d. This means that documents that have more occurrences of a given term receive a higher score. The default computation for $tf(t \in d)$ is $\frac{frequency}{2}$,
- $idf(t)$ stands for Inverse Document Frequency. This value represents the inverse of *docFreq* (the number of documents in which the term t appears). This means that rarer terms give higher contribution to the total score. The default computation is: $1 + \log(\frac{numDocs}{docFreq+1})$
- $norm(t,d)$, the shorter the matching field is (measured in number of indexed terms), the greater the matching document's score will be.

5.3 Prototype Architecture

The architecture of our prototype follows closely the one of the SMILA framework with only a couple of differences. The first difference is that we use Apache Solr as an external service to index and search documents while SMILA uses by default a Lucene distribution already included inside it. In this way, we can exploit the services offered by the more advanced interface of Solr which Lucene lacks. For example, Solr exposes many services and analyzers which are already implemented and easily configurable, while using Lucene would require to implement the routines manually. Calls to Solr are performed via HTTP requests. The other difference is actually a new feature

we added in order to make the configuration of the various components of the operation chain more straightforward to new users. Each component has its own configuration file, but these files are scattered among many directories and locating the one you need is quite disorienting at first. So, we provide a graphical configurator where all the important files are comfortably grouped and accessible by pressing a button. We describe this interface in more detail in Section 5.5.

5.3.1 Indexing part

In Figure 5.5 you can see the chain of operations of the indexing phase including every step followed by the data from crawling to indexing. Every component, with the exception of the crawler and of the pipelets that determine the business logic of the process, is already provided by the SMILA framework. Each of these components is configurable by an XML file or, in the case of the pipeline, by a BPEL file. The BPEL file determines the order in which the various pipelets are called, as well as the configuration of every single one of them. We will explain the details of the internal workings of the content processing pipelines in Chapter 6, as they are different for each experiment.

5.3.2 Query part

The query part of the prototype starts with a servlet. For the UML case, the queries and their relative parameters can be inserted directly by the user through a form which sends them directly to Solr. The Solr response is then parsed and redirected to a JSP presentation page. This page shows the first ten results along with their score for each one of the four experiments. Comparison is aided by a system that highlights the same element among different experiments. For example, if I move the mouse over a class in the Experiment B result list and that same class is present at any location of Experiment C and D lists, all of those classes are highlighted. Clicking on one returned result triggers the visualization of its detailed content as well as its detailed scoring information. That same click also triggers the display of a small graphical snippet of a graph developed in HTML5. The graph portrays the selected element surrounded by its neighbours as they are in the original UML model. This helps to give context to the returned result since a user can see not only which elements are connected to the one of interest, but also which is their relationship.

The WebML case starts with a servlet and a form too, but instead of entering the keywords directly, the user specifies the path of a model to

be used as query. This model has been previously created by the user and must conform to the WebML metamodel. The model then enters a BPEL pipeline where it is subjected to the same WebML-to-XMI translation of the indexing phase. Then, the "name" attributes of the elements of this model are extracted to be sent to Solr as a textual query. The output is redirected to a JSP presentation page with the same behaviour of the one of the UML case except for the graph snippet. The architecture of the search phase with respect to the WebML case can be seen in Figure 5.6. As stated before, in the UML search case the structure is simpler. The keywords are entered directly by the user and there is no need to perform further analysis besides the ones already specified in Solr.

5.4 Configuration Files

BPEL The BPEL file specifies the chain of operations of the various configurations and experiments. We report below a fragment of such file extracted from the pipeline of WebML Experiment C (Area granularity, Multi-Field Weighted Index), specifying an activity to be performed:

```
<extensionActivity>
  <proc:invokePipelet name="invokeAnalyzerSubstitutionPipelet">
    <proc:pipelet class="it.polimi.mdir.webml.pipelet.
        AnalyzerSubstitutionPipelet" />
    <proc:variables input="request" />
    <proc:configuration>
      <rec:Val key="coreName">webml_C</rec:Val>
      <rec:Val key="fieldType">content_analysis</rec:Val>
    </proc:configuration>
  </proc:invokePipelet>
</extensionActivity>
```

This is an example of an *invoke* activity. The *proc:invokePipelet* tag specifies the name of this activity, while the *proc:pipelet* tag specifies the pipelet to be called. *proc:variables* indicates the name of the workflow variable where the SMILA records are located. After that, inside the *proc:configuration* tag it is possible to specify the name and the value of some configuration variables specific to this activity. These variables will take the form of a SMILA record and are read using its conventions.

You can see the graphical representation of the whole pipeline from which this activity was extracted in Figure 5.7 as it is displayed by the Eclipse BPEL Designer, a tool that can be used to graphically design pipelines.

Case-specific configuration files These files configure variables and settings specific to one of the two case studies. It takes the form of a simple Java *properties file*. The example below depicts the one for the WebML case study. Each entry is explained by the commented lines that begin with "#".

```
#Path where to find the WebML projects (parent folder)
WEBML_PATH=C:/WebML_models_012/

#Path where to put the .xmi files in output.
OUTPUT_PATH=../it.polimi.mdir.webml/output/

#Path where the WebML queries are located (parent folder)
WEBML_QUERY_PATH=../webmlQueries/

#---------------------------#
#Weights configuration#
#---------------------------#

#weightmap
siteview=1.3
area=1.2
page=1.1
unit=1.0
link=0.8
```

5.5 Configurator and user interface

The user interface is built to be the central access point for all the major configuration files of both prototypes. As showed in Figure 5.8, the main window is divided in four tabs, each one representing a configurable section of the prototype. Each tab contains some buttons that allow to open the configuration files in a text editor. The sections and the configuration files that can be opened, for each tab, are listed as follows:

- SMILA: contains the main files to configure the SMILA framework. These files include the Listener/Router configuration and the processor properties.
- UML: includes files to configure the UML crawler, the UML experiments properties (like which weights to assign to the various concepts) and the UML pipeline.
- WebML: contains files needed to configure the WebML experiments and the WebML pipelines.

- Solr: contains links to the schema.xml files for each of the experiments as well as a link to the solr.xml file.

The user can access a brief help text by clicking the “Info” button found next to the listed files. A message will popup explaining the purpose of the file (see Figure 5.9). This is just meant as a reminder of the functionalities of such file and doesn’t substitute the complete documentation.

Finally, by clicking the “Open file” button, the corresponding file will be opened using the text editor of choice. Windows Notepad is used by default, but it can be changed by specifying the path to the executable of another text editor in the user interface’s “configuration.properties” file.

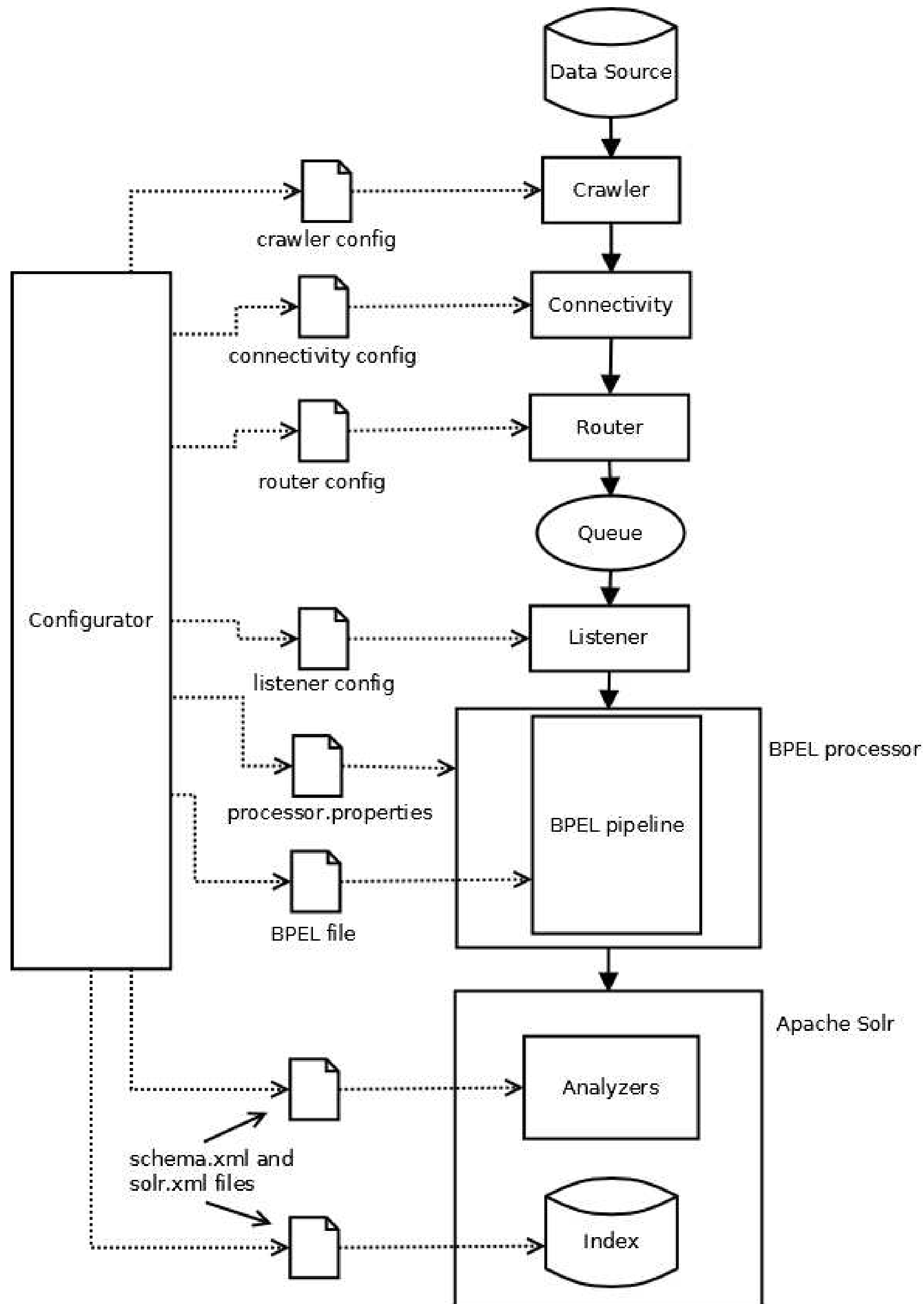


Figure 5.5: The diagram of the architecture of the prototype showing the chain of operations of the indexing phase.

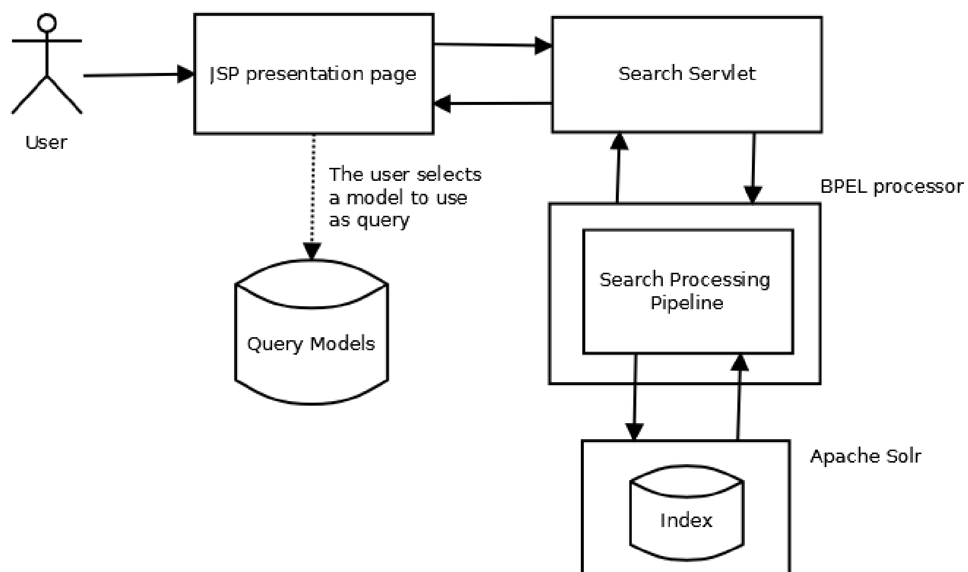


Figure 5.6: Chain of operations of the search phase in the WebML case.

Figure 5.7: The pipeline of the Experiment C for WebML displayed by the Eclipse BPEL Designer.

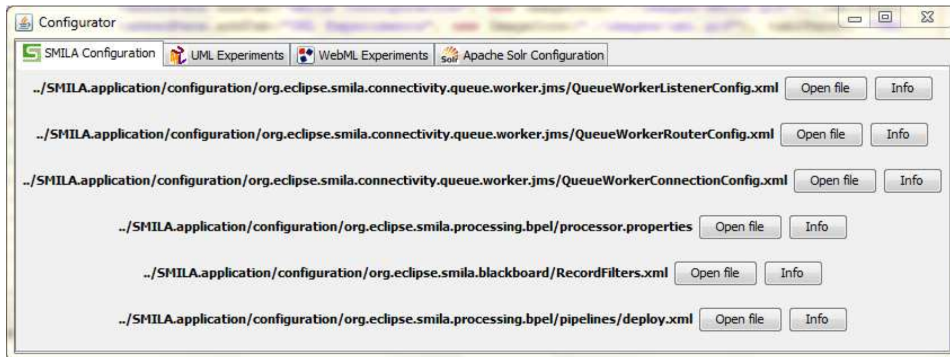


Figure 5.8: Screenshot depicting the SMILA tab of the configurator.

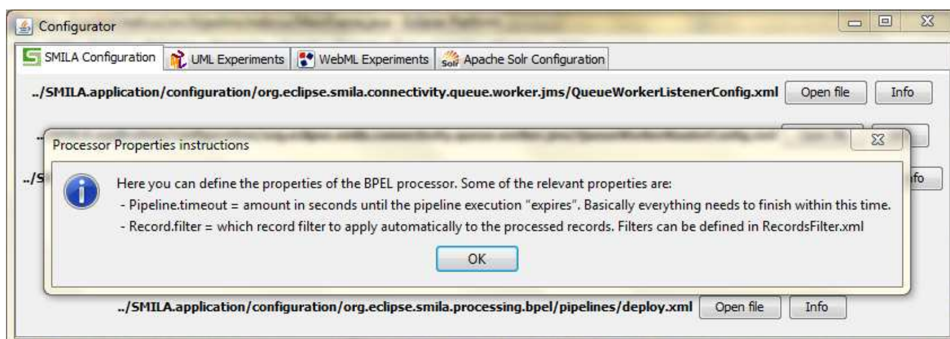


Figure 5.9: Screenshot of the help text of the processor.properties Info button.

Chapter 6

Case Studies

This chapter discusses the case studies of the Model-Driven Information Retrieval System prototype. The case studies we developed consist of two model-based search engines: the first one searches models or model fragments that belong to a repository of UML class diagrams; the second one searches model fragments that belong to a repository of WebML projects. Each case study involves several experiments including different indexing strategies, as explained in Section 4.3.

The outline of this chapter is organized as follows. In Section 6.1 we describe the dataset of UML projects, while in Section 6.2 we discuss the case study of our Model-Driven Information Retrieval System for that specific repository of projects. In Section 6.3 we describe the WebML projects dataset and in Section 6.4 we discuss the case study that involves the search of such projects. Each case study adopts different experiments that in turn include different indexing strategies. The UML Case has four experiments: A, B, C, D. The WebML Case has two experiments: B and C. The WebML experiments B and C have the same labels as the experiments B and C of UML because these experiments are very similar.

As discussed in Section 4.3, each experiments involves various indexing strategies. In the Experiment A, the segmentation granularity is "entire project" and the index structure is almost totally flat. The Experiment B involves a smaller segmentation granularity that corresponds to a metamodel concept, the index structure is multi-field and the terms are stored without any kind of weighting. The Experiment C adopts the same segmentation granularity as of the previous experiment, the index structure is multi-field and, differently from the previous experiment, the index terms are weighted according to the metamodel concept which they belong to. In the Experiment D the segmentation granularity is the same as in the pre-

vious two experiments, the index structure is still multi-field and the terms are still weighted according to their metamodel concept. The novelty of this experiment lies in the algorithm included before the indexing phase. This algorithm first creates a graph representation of the considered model, using as nodes the elements corresponding to the selected segmentation granularity and as edges their respective relationships. Then, each element is enriched with some information harvested from its neighbours.

The WebML case chain of operations consists of two parts. The first one performs a set of operations which is common between text-based search and content-based search. After that, the chain splits splits in two different flows of operation according to the type of search. This work only discusses the text-based search chain. A previous work [21] has already faced the problem of content-based search with the WebML dataset using graph-based techniques and can be an example of operations chain for the content-based part.

6.1 UML Dataset

The UML dataset consists of class diagrams, the most important type of diagrams when designing a new application. These diagrams are very diffused because they can describe both conceptual characteristics of an application and specific modeling details which can be directly translated into programming code.

The dataset is saved in XMI format (XML Metadata Interchange). This format is a standard from Object Management Group (OMG) used for the exchange of metadata through XML. It can be potentially used for every kind of artifact whose metamodel can be fit into Meta-Object Facility (MOF) structure. However, the most usual use case is the exchange of UML models.

The dataset consists of metamodels coming from the AtlanMod zoos, a research team in common between INRIA (Institut National de Recherche en Informatique et en Automatique) and LINA (Laboratoire d'Informatique de Nantes Atlantique). It is composed by a total of 301 models expressed in UML 2.1 conforming to the Ecore metamodel from Eclipse Modeling Framework (EMF). These models are actually metamodels (for example the metamodel of HTML 1.0), but since a metamodel is still a model all the hypotheses made in this document are still valid.

In Figure 6.1 you can see an example of UML project model from the dataset discussed above. The example in Figure 6.1 shows a project model named "MSProject" with two packages. The only significant package is the

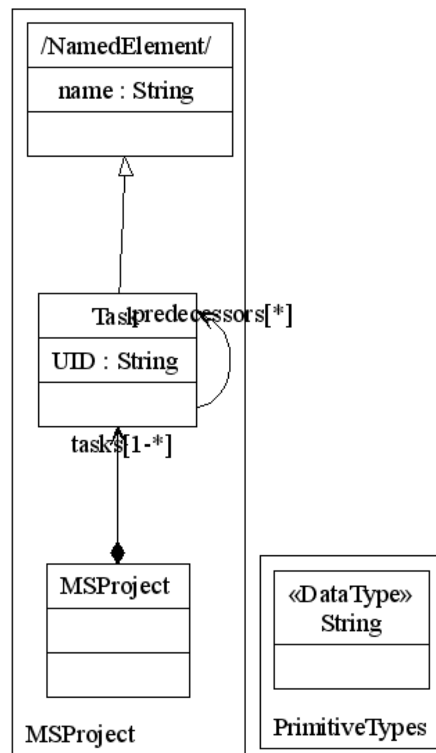


Figure 6.1: An example of UML project model from AtlanMod zoos dataset.

one with the same name of the project. This package contains three classes connected by two relations.

The code below shows the XML representation of the model in Figure 6.1.

```
<...
  <packagedElement xmi:type="uml:Package" xmi:id="
      _e6BjkeiaEd6gMtZRCjS81g" name="MSProject">
    <packagedElement xmi:type="uml:Association" xmi:id="
        _e6BjmeiaEd6gMtZRCjS81g" name="A_MSProject_Task"
        memberEnd="_e6BjmOiaEd6gMtZRCjS81g_
        _e6BjmuiaEd6gMtZRCjS81g">
      <ownedEnd xmi:id="_e6BjmuiaEd6gMtZRCjS81g" name="" type="
          _e6Bjk-iaEd6gMtZRCjS81g" isUnique="false" association="
          _e6BjmeiaEd6gMtZRCjS81g">
        <upperValue xmi:type="uml:LiteralUnlimitedNatural"
            xmi:id="_e6CKpeiaEd6gMtZRCjS81g" value="1"/>
        <lowerValue xmi:type="uml:LiteralInteger" xmi:id="
            _e6CKpuiaEd6gMtZRCjS81g" value="1"/>
```

```
      </ownedEnd>
    </packagedElement>
    <packagedElement xmi:type="uml:Association" xmi:id="
        _e6BjnOiaEd6gMtZRCjS81g" name="A_Task_Task" memberEnd="
        _e6Bjm-iaEd6gMtZRCjS81g _e6CKoOiaEd6gMtZRCjS81g">
      <ownedEnd xmi:id="_e6CKoOiaEd6gMtZRCjS81g" name="" type="
          _e6BjleiaEd6gMtZRCjS81g" isUnique="false" association="
          _e6BjnOiaEd6gMtZRCjS81g">
        <upperValue xmi:type="uml:LiteralUnlimitedNatural"
            xmi:id="_e6CKqeiaEd6gMtZRCjS81g" value="1"/>
        <lowerValue xmi:type="uml:LiteralInteger" xmi:id="
            _e6CKquiaEd6gMtZRCjS81g" value="1"/>
      </ownedEnd>
    </packagedElement>
    <packagedElement xmi:type="uml:Class" xmi:id="_e6Bjk-
        iaEd6gMtZRCjS81g" name="MSProject">
      <ownedAttribute xmi:id="_e6BjmOiaEd6gMtZRCjS81g" name="
          tasks" type="_e6BjleiaEd6gMtZRCjS81g" isUnique="false"
          aggregation="composite" association="
          _e6BjmeiaEd6gMtZRCjS81g">
        <upperValue xmi:type="uml:LiteralUnlimitedNatural"
            xmi:id="_e6CKo-iaEd6gMtZRCjS81g" value="*"/>
        <lowerValue xmi:type="uml:LiteralInteger" xmi:id="
            _e6CKpOiaEd6gMtZRCjS81g" value="1"/>
      </ownedAttribute>
    </packagedElement>
    <packagedElement xmi:type="uml:Class" xmi:id="
        _e6BjlOiaEd6gMtZRCjS81g" name="NamedElement" isAbstract="
        true">
      <ownedAttribute xmi:id="_e6BjluiaEd6gMtZRCjS81g" name="
          name" type="_e6CKoeiaEd6gMtZRCjS81g" isUnique="false"/>
    </packagedElement>
    <packagedElement xmi:type="uml:Class" xmi:id="
        _e6BjleiaEd6gMtZRCjS81g" name="Task">
      <generalization xmi:id="_e6CKouiaEd6gMtZRCjS81g" general="
          _e6BjlOiaEd6gMtZRCjS81g"/>
      <ownedAttribute xmi:id="_e6Bjl-iaEd6gMtZRCjS81g" name="UID
          " type="_e6CKoeiaEd6gMtZRCjS81g" isUnique="false"/>
      <ownedAttribute xmi:id="_e6Bjm-iaEd6gMtZRCjS81g" name="
          predecessors" type="_e6BjleiaEd6gMtZRCjS81g" isUnique="
          false" association="_e6BjnOiaEd6gMtZRCjS81g">
        <upperValue xmi:type="uml:LiteralUnlimitedNatural"
            xmi:id="_e6CKp-iaEd6gMtZRCjS81g" value="*"/>
        <lowerValue xmi:type="uml:LiteralInteger" xmi:id="
            _e6CKqOiaEd6gMtZRCjS81g"/>
      </ownedAttribute>
    </packagedElement>
  </packagedElement>
  <packagedElement xmi:type="uml:Package" xmi:id="
```

```
    _e6BjkuiaEd6gMtZRCjS81g" name="PrimitiveTypes">
  <packagedElement xmi:type="uml:PrimitiveType" xmi:id="
      _e6CKoeiaEd6gMtZRCjS81g" name="String"/>
 </packagedElement>
</uml:Model>
```

As you can see in the code above, each class is represented by a *packagedElement* element with *type* attribute *uml:Class*. The *packagedElement* elements that represent classes are saved inside other *packagedElement* whose *type* attribute is set to *uml:Package*. Class attributes are put in *ownedAttribute* elements saved as children of the class they belong to. This dataset contains three kinds of UML relationships: association, composition and generalization. These three relationships are treated as simple attributes and they are saved as children of the class they belong to: generalizations are saved within a specific element *generalization*, associations and compositions are inside *ownedAttribute* elements (compositions has the *aggregation* attribute set to *composite*). The information about the referenced class is saved in the *type* attribute inside the ownedAttribute element. The cardinalities of the relationships that have the direction specified into the model are saved directly as children of the *ownedAttribute* representing the relationship, while the cardinalities in the opposite direction are saved as children of a different *packagedElement* with *type* attribute set to *uml:Association*. For example the classes *MSProject* and *Task* are connected through the association *tasks* (starting from the class *MSProject* to class *Task*). The association is saved in the class *MSProject* (the starting class). The upper cardinality is "*" (many) and lower cardinality is "1" (a project can have from one to many tasks to be performed). The cardinalities in the opposite direction are saved at the beginning of the XML code within a different packagedElement. The relationship in the opposite direction references the *association* attribute saved into the packagedElement that represents the class owning the relationship through the *xmi:id* attribute. In the example above, the opposite cardinalities (upper value and lower value) are both "1" (a task can belong to only one project).

In Figure 6.2 you can see an histogram showing the distribution of the number of classes into the UML projects dataset.
The histogram points out the following information:

- 272 diagrams have less then 100 classes,
- 22 have a number of classes between 100 and 400,
- 7 diagrams have more than 400 classes (the maximum number of classes for a diagram is 700).

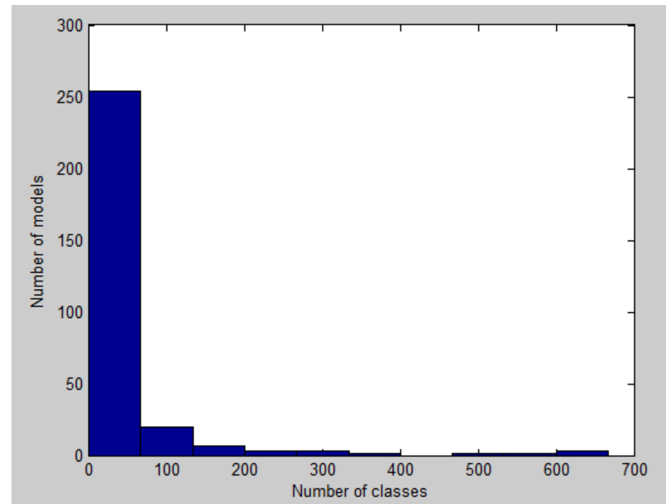


Figure 6.2: Histogram that shows the distribution of the number of classes into the UML projects dataset.

The dataset has an important limitation: the average dimension of the projects is quite small, while only a small number of projects is huge. Another drawback is that, as it often happens in UML modeling, classes are organized in packages, but most of the diagrams from this dataset have only 2 packages, with just 46 diagrams having more than that. Moreover, among the diagrams with 2 packages, most of them contain one package which is not significant as it contains only data types classes.

Figure 6.3 depicts the frequency distribution of terms of the UML dataset (301 models). We show the distribution up to the first two hundred terms.

As the figure suggests, the distribution of terms approximates a power-law function and, therefore, it follows the Zipf's law.

For the purpose of the tests of the UML case study, we used 84 models out of the total of 301 models available in the dataset.

6.2 UML Case

The first implementation experience deals with the indexing and searching of UML projects. As explained before (Section 3.3) the dataset of the UML case consists of UML class diagrams that describe several metamodels. We have designed and implemented four experiments with different types of operations each. The initial crawled item is always a UML project. The list of experiments is presented as follows:

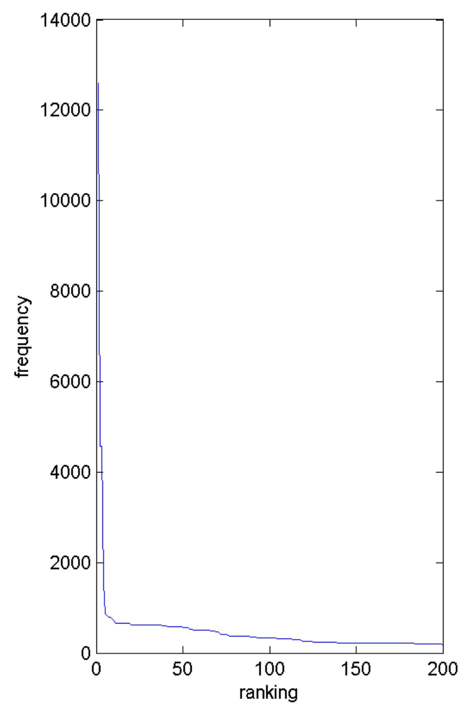


Figure 6.3: The frequency distribution of terms of the UML dataset.

- *Experiment A*: this experiment is the simplest one. The granularity is “project”, which means that the returned documents are projects. The Solr index we obtain after the SMILA pipeline for this experiment is very basic: the project id, the project name and one single “content” field that contains every project element (class names and attributes).

- *Experiment B*: this experiment uses granularity “class”. This means that the retrieved documents are classes belonging to a certain project. The Solr index fields we obtain are the following ones: the id and the name of the project to which the class belongs, the class id, the class name and the attribute names.

- *Experiment C*: this experiment uses the same granularity and the same index structure of Experiment B. The difference in this experiment is that we add payloads to each indexed term so that the searched classes receive different relevance according to the UML concept they refer to (simple attribute, composition, association, class and project).

- *Experiment D*: the granularity of this experiment is still “class”. The experiment involves an algorithm that for each class imports elements of its neighboring classes. This is accomplished by transforming the initial UML class diagram project into a graph where nodes represent classes and edges represent relation between classes (e.g. generalization between a parent class and a child class). The payloads of the imported elements are penalized during the import according to the type of the followed edge (relation) and to the distance in number of hops with respect to the currently processed class.

After the crawling phase performed by our custom implementation of a SMILA crawler, one SMILA record still represents an entire project and contains the following metadata elements that, if not differently specified, are extracted directly from the XML representation of the UML project:

- *Project name*: the file name of the XML representation of the crawled project minus the file extension.

- *Project id*: the id of the project.

- *Class names*: a list of the class names of the current project.

- *Class ids*: a list of the class ids of the current project. These class ids are in pairs with the Class names explained above.

- *Attribute names*: a list of all attribute names of the current project. The attributes are stored in a way that keeps some information about the attributes: in the same string of the attribute are saved the class id to which the attribute belongs and the relation type expressed by that attribute. The relation type can be simple attribute, association and composition. In case of association and composition we also save the information about the cardinalities of the relationships.

Next, the SMILA record enters the *Add Pipeline* which is the BPEL pipeline that processes records. There is a different Add Pipeline for each experiment, but every Add Pipeline ends with an indexing pipelet which analyzes and indexes new documents to Solr. Since the *types of analysis* and the *indexing pipelet* are the same for the four experiments, we discuss them separately at the end of this section, while in the other paragraphs we describe with further details the specific operations of each experiment, providing some examples.

Experiment A This experiment involves two pipelets. The first one simply gets the crawled projects in the form of records from the SMILA crawler and scraps the non-relevant information for this experiment. This means that the information regarding the relation type and the cardinalities is not preserved. After this first pipelet a record just contains three metadata elements: the project id, the project name and the content, which contains class names and attributes in an undifferentiated manner.

The second pipelet is the *Solr Indexer Pipelet* that is described in the last paragraph. In this experiment the index is the most simple one. It is very close to a single-fielded index: each document in the index represents a UML project model and has three fields: project id, project name and content. The last field contains all the attribute names and class names of that project. The project name field is copied into the content field, since we want a single searchable field. In this way, one could also search for a project name. Users have also the possibility to submit a multi-field query specifying different query terms for the project name field and the content field.

Experiment B This experiment involves two pipelets, too. Since this experiment uses “class” granularity, the main task is to segment the initial record representing an entire project into several records representing the classes belonging to that project. Basically this is done by creating a new record for each class of a given project. The id of this record is the id of

the class extracted from the XML representation of the project model. The record also has other fields such as the class name, the attribute names of that class, the project name and the project id to which the class belongs to. This experiment doesn't use the information about the types of relation and the cardinalities, so they are discarded.

The second pipelet, that is also the last one, as usual, is the Solr Indexer Pipelet that is described at the end of this section in the last paragraph. The index for this experiment is more complex than the previous one. Each indexed document represents a class with the following fields: the project id to which the class belongs to, the project name, the class id, the class name, the attribute names. There is also an additional field, the content field, that contains a copy of all the other searchable fields (except the ids, which are not copied there). Also in this case, it is possible to let the user submit a multi-field query, so that he can specify project name, class name or attribute names separately.

Experiment C The pipeline of this experiment, which you can see in Figure 6.4, involves four pipelets.

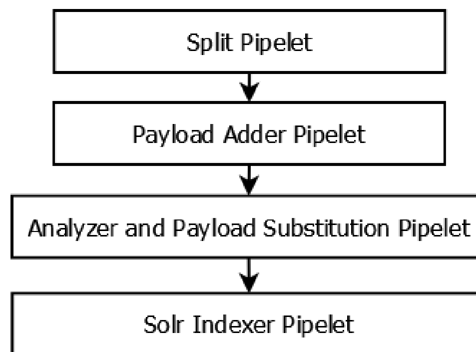


Figure 6.4: The pipeline of the UML case Experiment C.

In this experiment, the first pipelet, *Split Pipelet*, is very similar to the first pipelet of the previous Experiment B. The difference is that the information about relations types and cardinalities is preserved.

The second pipelet, *Payload Adder Pipelet*, uses the information about relation types and cardinalities to assign different payloads to terms according to those information. For example, we can give a higher payload to a term referring to a class name and a lower one to a term referring to a simple attribute. This operation is performed to obtain the following behavior: if there are two classes, let's say class A and class B, where class A contains

the term “java” as class name and class B contains it as a simple attribute, then, in case of a query string containing the term “java”, we want the class A ranked in a higher position than class B. This behavior is achieved by assigning payloads as described above. There are no specific or standard rules to assign payloads to the UML elements and some tuning to adjust the payloads according to the obtained rankings is needed. The payloads we used are determined following simple reasonings: we give a higher payloads to those UML elements that in a sense are more important than others. For example, it makes sense to give an higher payload to a term representing a class name than to a term representing an attribute because the class name is supposed to better describe the general concept of that class while an attribute is too specific. To let Solr understand that a term has a certain payload, one has to add the payload information directly in the token string before analysis and indexing. For example, one can include payload information by marking up the tokens with a special character followed by the payload value: “java|2.0” means that the token “java” has a payload value of “2.0”. Then, a special Solr analyzer, the *DelimitedPayloadTokenFilter*, interprets in the right way the sequence of characters composed by the token string itself and the payload information. In Solr, payloads are byte arrays optionally stored with every term on a field. However, Solr has a particular issue in payloads “propagation”. The payloads are applied only to the original word that is indexed and the WordDelimiterFilter (the analyzer that splits words into subwords according to some user defined rules, like splitting on case transitions) doesn’t apply the payloads to the tokens it generates. For example, let’s say that the string containing the attribute names of a class is the following one: “javaAttribute|1.0”. The DelimitedPayloadTokenFilter applies the payload to the word “javaAttribute”. When the WordDelimiterFilter is called, it doesn’t propagate the payloads to the generated subwords “java” and “attribute”. It is evident that the payload information gets lost for the generated subwords. This problem is solved by the following pipelets.

The third pipelet, *Analyzer and Payload Substitution Pipelet*, does the following operations:

1. Takes from the record the SMILA metadata elements that should be analyzed. These metadata elements, for example the string of attribute names of a class, contain words that already have the payload attached, such as “firstName|1.0 NewEmployee|2.0” and they are not yet analyzed.

2. Extracts and saves temporarily the payload for each word contained

in the metadata element.

3. Calls the Solr analyzers for each metadata elements and performs the analysis (the original words are preserved).

4. Parses the Solr analyzers output (that is expressed in XML) extracting the single subwords derived from the analysis. Then it "propagates" the previously saved payloads by attaching them to the subwords and saves them back to their original SMILA metadata field.

The string at the end of this pipelet looks like this: "first|1.0 name|1.0 new|2.0 employee|2.0".

The fourth pipelet is the last one and it is the same Solr Indexer Pipelet as in the previous two experiments, the difference lies in the type of analysis performed. Since most of the analysis have been already done in the previous pipelet, here the words are processed only by the WhiteSpaceTokenizer to split the words on white space and by the DelimitedPayloadTokenFilter to take into account payloads. This pipelet ends the Experiment C by indexing to Solr the tokens eventually obtained. The index for this experiment is exactly the same of Experiment B index. The difference is that the terms are weighted through payloads.

Experiment D The Experiment D is deeply different from the previous three experiments. The idea is to import, for each class, the elements of the neighboring classes. This kind of processing can lead to a better recall when trying to search for a class. In Figure 6.5 you can see an example of UML project model named BQL from the dataset of UML class diagrams. A user looking for the class "Entry" could also be interested in retrieving the father class "/LocatedElement/" using the query string "entry location" (AND query). An Experiment without elements importation, would have no results. In this experiment, the class "Entry" imports the elements of the neighboring classes, among which there is also the class "/LocatedElement/". The class "/LocatedElement/" imports the class name of the class "Entry" too. This means that, at the end of the algorithm, the document representation of the class "Entry" contains, besides their own elements, also the attribute names of the class "/LocatedElement/" and the class "/LocatedElement/" contains also the class name of "Entry". Therefore, both the classes are part of the ranked list as result of the previously mentioned query.

As already explained before, in this Experiment, we *import* the elements of the neighboring classes for each class. The payload of the imported ele-

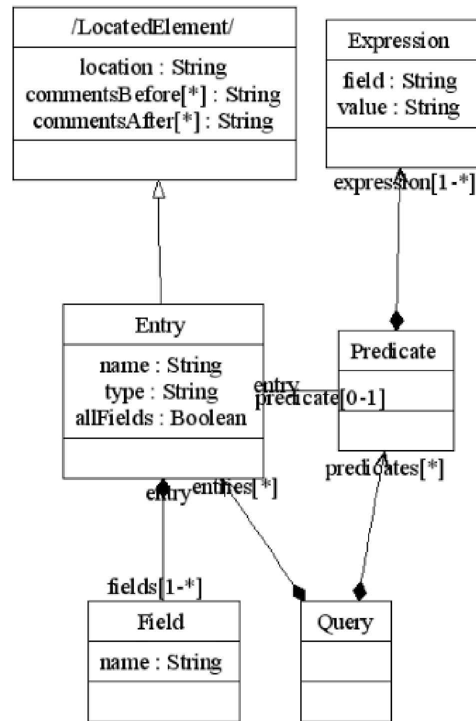


Figure 6.5: An example of UML project model diagram from the dataset to explain Experiment D purposes. A query string like "entry location" (AND query) without the importation algorithm would produce no results. The algorithm of Experiment D allows to retrieve both classes "/LocatedElement/" and "Entry".

ments is properly *penalized* according to the relation type. For *neighboring classes* of a class X, we mean all the classes connected to X through a relation (association, generalization, composition). In Figure 6.6 you can see a diagram showing the pipeline for this experiment. The crawling phase is the same as in the previous experiments B and C, the source consists of the UML project models in XMI format. After this step a SMILA record still represents an entire UML project. Next, the SMILA Processing Pipeline for Experiment D starts.

The *Translate XMI to GraphML* pipelet translates the original UML project model represented as an XMI document into GraphML representation. GraphML is a format to describe graphs. It consists of a language core to describe the structural properties of a graph. Here we use a format to describe graphs to ease the navigation algorithm. The pipelet performs the translation and saves the GraphML files to disk. Our translation from XMI

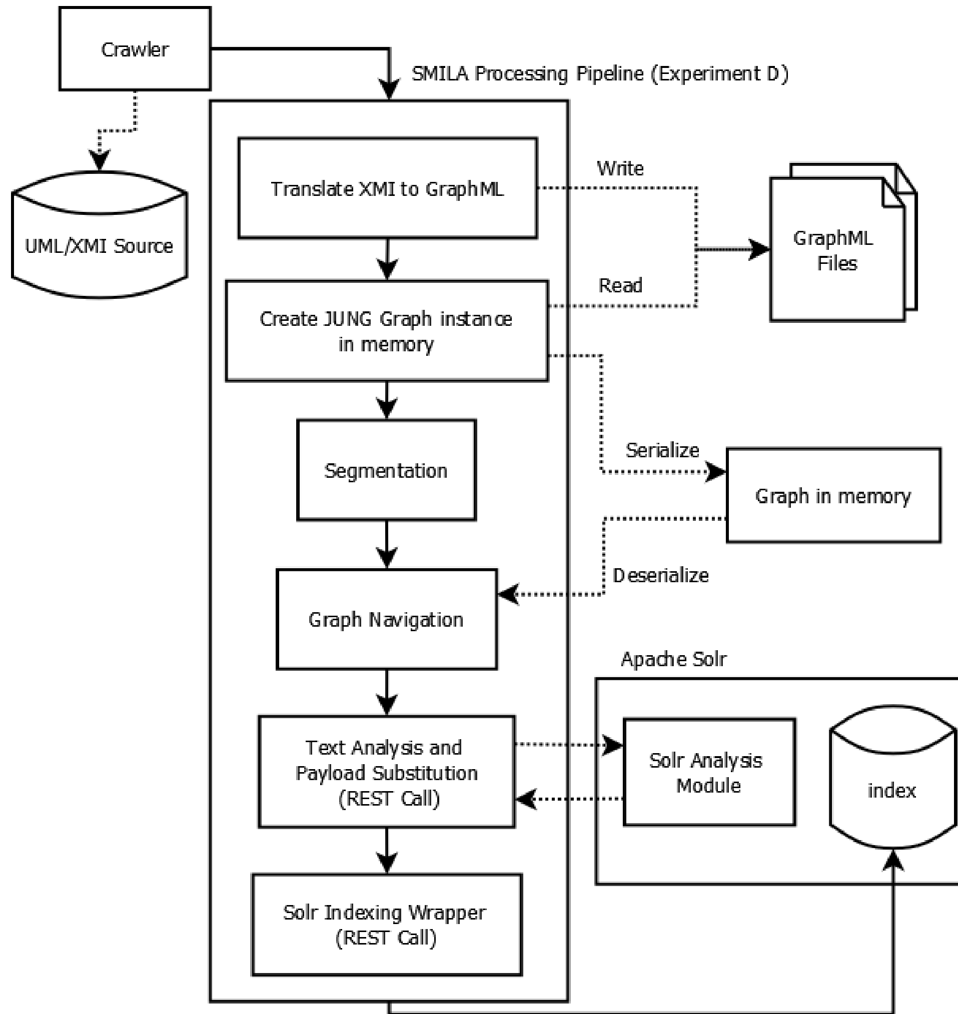


Figure 6.6: The pipeline of the UML case Experiment D.

to GraphML is straightforward. Each class is represented by a node in the graph, and the relation among classes are represented by typed edges. Since the XMI representation only stores the relationships in one way, we added to the GraphML representation the edges in the opposite way. This representation also stores the relation type as an edge attribute and, possibly, relation cardinalities.

The *Create Graph Pipelet* creates an instance of a graph in memory. Given a record representing a project, this pipelet creates its graph with JUNG starting from the GraphML specification. JUNG (the Java Universal Network/Graph Framework) is a software library that provides a common

and extendible language for the modeling, analysis and visualization of data that can be represented as a graph or network. The JUNG graph instance created in this way is serialized to memory.

Since the granularity of the segmentation of this experiment is “class”, the *Segmentation* pipelet splits the original record representing an entire project into several records representing classes, deserializes the JUNG graph instance and starts the navigation and import algorithm. For each class, there is a different run of this algorithm.

The *Graph Navigation* starts the navigation of the current class. Classes navigation and elements import are decoupled so that they can be managed separately. The importation of the elements from neighboring classes is considered as the business logic of the classes navigation. The navigation algorithm visits recursively the nodes and its neighbors. The algorithm takes as input the number of hops that have be to performed. Looking at the previous example in figure 6.5, this means that if the number of hops is two and the current visited class is “Entry”, the list of its imported elements will also include the attributes coming from the “Expression” class. The algorithm looks at the outgoing edges, so that if a node is connected with more than one outgoing edge to other neighboring classes, we process that node more times. The business logic, which is the elements importation, is executed after visiting the last neighbor in a depth-first manner. Since it is very common that the relationships that connect classes form cycles, we needed to implement a solution to avoid importing the same elements more than one time. The solution is to “tag” the edges that have been already followed during the algorithm and visit the nodes connected through edges that have not been processed yet. As said, the importation also involves a penalization of the payloads of the imported attributes. This penalization depends on the relation type, and, possibly, also on the cardinalities of the relationships. Notice that the penalization takes into account the number of hops that the imported elements have done to reach the visited class. With reference to the example in figure 6.5, given the class “Entry” as the current visited node, the attribute “value” coming from the class “Expression” (two hops) is more penalized than the attribute “location” from “/LocatedElement/” (one hop).

The last two pipelets, *Text Analysis and Payload Substitution* and *Solr Indexing*, involve the same operations of the last two pipelets of the previous Experiment C. Also the index of this experiment has the same structure of the Experiment C index.

Analysis and Solr Indexer Pipelet These two pipelets are quite similar in all the four experiments, so we describe them separately in this section.

In Experiment A and Experiment B the analysis and indexing tasks are actually done in the same pipelet, called SolrIndexerPipelet. This pipelet is configured through the BPEL configuration file where we specify a mapping between the SMILA metadata elements and the Solr fields. The Solr configuration file named "schema.xml" contains the declaration of each field that in turn specifies the type of that field. The sequence of analyzers that perform the text processing is specified for each field type in the "schema.xml" file. The SolrIndexerPipelet prepares a new document by adding the specified fields to the XML document that has to be posted to Solr through the *update* URL. This kind of request performs both analysis and indexing. The fields added to the XML document contain the information extracted from the project models.

As explained in the above sections, Experiments C and D perform analysis and indexing in two different pipelets. This is done to avoid the problem of "payloads propagation" that we previously explained and also to better separate the two types of operation. The drawback is that the analysis task is more time consuming since there is a different HTTP request to the Solr server for each SMILA metadata element.

6.3 WebML Dataset

The WebML dataset consists of twelve WebML projects coming from real world WebML applications. In Figure 6.7 you can see an example a fragment of a WebML project showing an area called *Shops* containing three pages and some units. The code below shows the XML representation of the area:

```
<?xml version="1.0" encoding="UTF-8"?>

<Area id="sv3g#area5g" name="Shops" defaultPage="sv3g#area5g#
   page20g" landmark="true">
  <OperationUnits>
    <CreateUnit id="sv3g#area5g#cru11g" name="Save DC" entity="
        ent2">
      <OKLink id="sv3g#area5g#cru11g#oln48g" name="Link OK 44"
          to="sv3g#area5g#page20g" automaticCoupling="true"/>
    </CreateUnit>
    <NoOpOperationUnit id="sv3g#area5g#opu13g"name="nop">
      <OKLink id="sv3g#area5g#opu13g#oln49g" name="Link OK 45"
          to="sv3g#area5g#page20g" automaticCoupling="true"/>
    </NoOpOperationUnit>
     ...
```

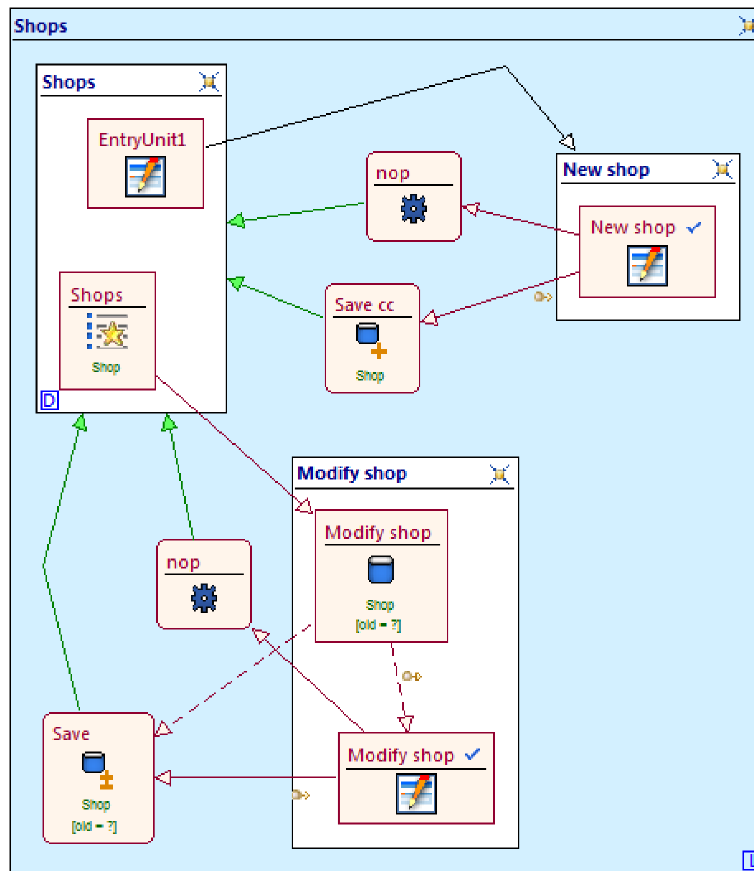


Figure 6.7: Example of an area of a project from the WebML dataset.

```
</OperationUnits>
<Page id="sv3g#area5g#page22g" name="Modify Shop">
  <ContentUnits>
    <DataUnit id="sv3g#area5g#page22g#dau7g" name="Modify Shop
        " entity="ent2">
      <Link id="sv3g#area5g#page22g#dau7g#ln107g" name="Link
          99" to="sv3g#area5g#page22g#enu22g" type="transport"
          automaticCoupling="false" validate="true">
      </Link>
      <Link id="sv3g#area5g#page22g#dau7g#ln31g" name="Link 31
          " to="sv3g#area5g#mfu4g" type="transport"
          automaticCoupling="true" validate="true"/>
    </DataUnit>
    <EntryUnit id="sv3g#area5g#page22g#enu22g" linkOrder="sv3g
        #area5g#page22g#enu22g#ln108g sv3g#area5g#page22g#
        enu22g#ln109g" name="Modify Store">
        <Field id="sv3g#area5g#page22g#enu22g#fld45g" name="
```

```
                Name" type="string" modifiable="true" preloaded="
                true">
          </Field>
          ...
        </EntryUnit>
      </ContentUnits>
      <layout:Grid>
          ...
      </layout:Grid>
    </Page>
    <layout:Grid>
       ...
    </layout:Grid>
</Area>
```

The quantity of concepts expressed by the WebML models is higher (whole projects, siteviews, areas, pages, content units, operation units, links) with respect to UML models (whole projects, packages, classes and attributes). This results in a more complex XML representation for WebML projects, though the information remains well structured. On average the projects are very big: they contain big areas with hundreds of pages and units. Only few projects are small. Because of the size and because of concurrent working issues while editing this models in a real industrial environment, the XML files representing a whole project are splitted up according to the metamodel element (i.e. there are separate XML files for site views, areas, pages, etc.).

Figure 6.8 depicts the frequency distribution of terms of the WebML dataset (10 projects). We show the distribution up to the first two hundred terms.

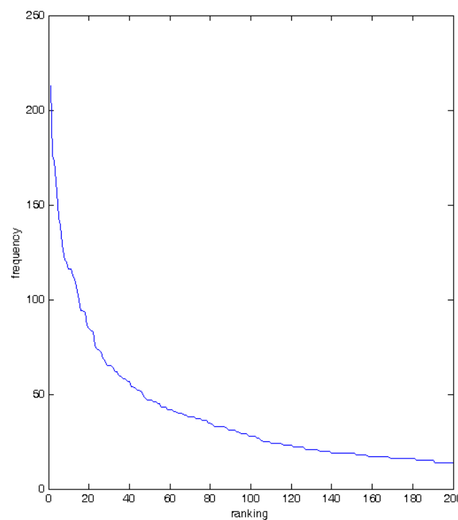


Figure 6.8: The frequency distribution of terms of the WebML dataset.

As shown for the UML dataset (Section 6.1), also the distribution of terms of the WebML dataset approximates a power-law function and, therefore, it follows the Zipf's law. However, in this case, the shape of the curve is less pronounced. This fact is due to the amount of projects, and therefore terms, that are present in the two dataset: the UML dataset contains much more models than the WebML dataset.

6.4 WebML Case

The second case study deals with the indexing and searching of WebML projects. With respect to the previous case explained in Section 6.2, here we adopt a more structured solution, even closer to the abstract solution of Section 4.1. In Figure 6.9 you can see a diagram showing the WebML chain of operations. As usual for a search engine, the whole chain includes two

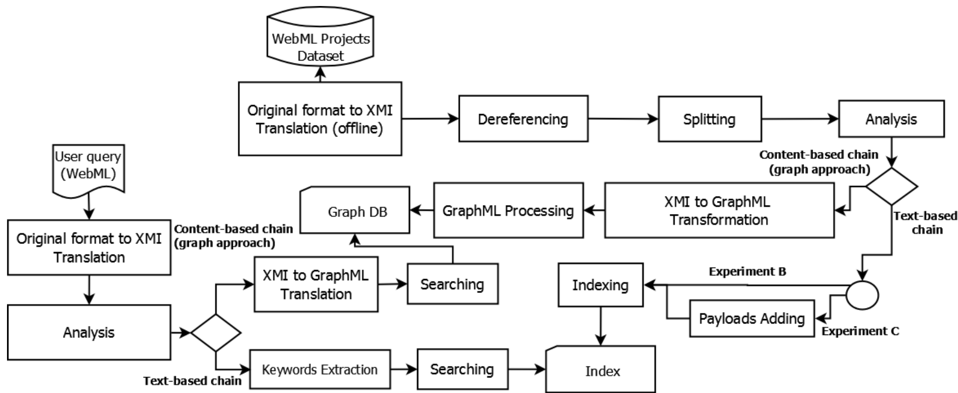


Figure 6.9: WebML operations chain. The diagram shows the operations involved in both content-based approach and text-based approaches. The Content Processing phase is showed in the top right, while the Query Processing phase is showed in the bottom left.

main phases:

- a Content Processing phase that is showed in the top right of Figure 6.9, starting from the WebML Projects Dataset translation,
- a Query Processing phase that is showed in the bottom left of Figure 6.9, starting from the User Query input.

In both these phases, the solution includes a set of operations which is in common between the content-based and text-based approach. The content-based approach involves a comparison between projects and queries that are translated into a graph representation to perform graph matching techniques. In this work we discuss only about the text-based approach, you can find more information about the content-based approach in [21].

In the next paragraphs we explain with further details both the Content Processing phase and the Query Processing phase of the WebML chain, providing information for each operation. Notice that each operation corresponds to a SMILA pipelet and an entire chain is a SMILA pipeline.

Regarding the Content Processing phase, after the common operations are performed, the other operations specific for each experiment start. For the WebML case there are two experiments: B and C. Experiments B and C have similarities with experiments B and C of the UML case so they have the same name. These experiments are explained with more information in the next paragraphs. At the end of this section we also present the Query Processing phase.

WebML Chain - Content Processing The WebML chain starts, as usual, with the crawling phase. The crawler ingests a project representation which is different from the original XML project representation in WebML syntax shown in Section 6.3. This is done because the XML original format doesn't conform to any particular standard model representation language. Therefore, before the actual crawling phase there is an off-line translation from the original format to a target format. The target format is expressed in UML 2.1 conforming to Ecore and it is very similar to the representation format of the projects from the UML dataset.

The code snippet below shows an example of translation from the original WebML representation format discussed in Section 6.3.

```
<packagedElement name="Shops" xmi:id="sv3g#area5g" xmi:type="
    webml:Area">
  <packagedElement entity="ent2" name="Save DC" xmi:id="sv3g#
      area5g#cru11g" xmi:type="webml:CreateUnit">
    <packagedElement name="Link OK 44" to="sv3g#area5g#page20g"
        xmi:id="sv3g#area5g#cru11g#oln48g" xmi:type="webml:OKLink
        "/>
  </packagedElement>
  <packagedElement name="nop" xmi:id="sv3g#area5g#opu13g"
      xmi:type="webml:NoOpOperationUnit">
    <packagedElement name="Link OK 45" to="sv3g#area5g#page20g"
        xmi:id="sv3g#area5g#opu13g#oln49g" xmi:type="webml:OKLink
        "/>
  </packagedElement>
        ...
        ...
  <packagedElement name="Modify Shop" xmi:id="sv3g#area5g#
      page22g" xmi:type="webml:Page">
    <packagedElement entity="ent2" name="Modify Shop" xmi:id="
        sv3g#area5g#page22g#dau7g" xmi:type="webml:DataUnit">
      <packagedElement name="Link 99" to="sv3g#area5g#page22g#
          enu22g" xmi:id="sv3g#area5g#page22g#dau7g#ln107g"
          xmi:type="webml:Link"/>
      <packagedElement name="Link 31" to="sv3g#area5g#mfu4g"
          xmi:id="sv3g#area5g#page22g#dau7g#ln31g" xmi:type="
```

```
        webml:Link"/>
    </packagedElement>
    <packagedElement name="Modify Store" xmi:id="sv3g#area5g#
        page22g#enu22g" xmi:type="webml:EntryUnit">
        ...
    </packagedElement>
  </packagedElement>
</packagedElement>
```

The translation from WebML representation to the target format is not one to one, only the relevant information suitable to be part of the index is kept, therefore some of the elements from the original representation are discarded. For example all the tags referring to the graphical layout specifications are not present into the target representation. Unlike the original files organization, the target representation has all the XML code of a project in one single file.

After the translation the *Crawling* phase starts. The crawler ingests all the projects creating a new record into the SMILA framework for each of them. The record contains the following attributes: the project id and all the structured representation (the XML code conforming to Ecore) of that project model. The project structured representation is carried on to the following stages of the process as much as possible. This is done to keep all the necessary information about the project structure (e.g. an area X is contained into a siteview Y).

The *Dereferencing* is an optional step performed only by some test configurations of the WebML case which are discussed in Chapter 7.

Up to this point of the chain, a SMILA record still represents a whole WebML project (the granularity is still entire project). The *Splitting* (or Segmentation) takes the SMILA record containing information on a whole WebML project and splits it creating several new records that has granularity "area". We chose this level of granularity because WebML areas have a size that is a good compromise between the size of the site views, which is too big, and the one of the pages, which is too small. The XML representation of an area record X contains the following information: the areas and siteviews that are ancestors of X stripped of their content, the area X with all its sub-elements (units and pages) and all the sub-areas of X stripped of their content. For example, let X be an area with father area Y which in turn is children of siteview Z and let A be a sub-area of X. Obviously each of these elements have their content consisting of operation units, pages, content units, etc. The SMILA record representing the area X looks like this:

```
<packagedElement name="siteviewZ" xmi:id="svZ" xmi:type="
    webml:Siteview">
  <packagedElement name="areaY" xmi:id="svZ#areaY" xmi:type="
      webml:Area">
    <packagedElement name="areaX" xmi:id="svZ#areaY#areaX"
        xmi:type="webml:Area">
        ...
        CONTENT OF AREA X (Operation Units, Content Units, Pages
            , etc.)
        ...
    </packagedElement>
  </packagedElement>
</packagedElement>
```

The *Analysis* extracts from the records in input (that now represent areas) the content of the "name" attributes of each area element. Then, the words mined in this way are processed through the Solr analyzers and the original content of the "name" attributes is replaced with the analyzed version. The original content of the "name" attributes is kept inside the field for visualization purposes by separating it from the analyzed content via an escape character like "$". For example, the "name" attribute with content "lazyEmployees" after the analysis becomes "lazyEmployees$lazyemploye lazi employe". The transformations that have to be performed on the text are configurable by modifying the Solr configuration files. This pipelet decouples the Analysis from the Indexing, which is performed at the end of each experiment. The separation between the Analysis and the Indexing allows to share the same analysis operations between the chain of the content-based approach and the chain of the text-based approach. This way, it will be easier in future works to compare the two different indexing approaches since they start from the same word analysis.

After the Analysis there is a branch between the two types of approach. The text-based approach continues with three different experiments which all ends with the indexing step.

Experiment B This experiment doesn't perform any particular operation except for the indexing. The indexing pipelet has still to send HTTP requests to Solr in order to call the WhiteSpaceTokenizer and split the words contained as a single string in the "name" attribute.

Experiment C This experiment calls the *Payloads Adder* pipelet that adds the payloads to the content of the area that has been previously analyzed. After this, the indexing pipelet includes also the DelimitedPay-

loadTokenFilterFactory among the called Solr analyzers for processing the payloads added in the previous pipelet. The payloads are easily configurable through a configuration file. Concepts to which is possible to add them are Siteview, Area, Page, Unit and Link.

WebML Chain - Query Processing The Query Processing deals with all the operations that have to be performed when the user inputs a query to the system. The Query Processing phase is depicted on the bottom left of Figure 6.9. The user can input the query as a WebML model. The query model has its own XML representation in WebML format which is translated in UML 2.1 conform to Ecore format, as in the Content Processing phase. Next, the model is analyzed through the same methods as the projects in the repository. At this point, the Text-based chain for the Query Processing part starts. The issue is to transform a search-by-example query into a keyword-based one. The *Keyword Extraction* extracts the query keywords from the XML document. Only the content of the "name" attributes are extracted and put into the query string. The query is then submitted to the Solr index as an AND query. The user can also submit a keyword-based query without the Keyword Extraction operation.

Chapter 7

Tests and Evaluation

This chapter discusses the results of the tests we conducted on the Model-Driven Information Retrieval System. As explained in Chapter 6 the two case studies of the system retrieve project models that belong to two different datasets. The first dataset consists of a set of UML class diagrams, while the second one consists of a set of WebML real-world applications.

The tests for the UML case study involve different types of keyword-based query. Each type, that in the following is called "meta-query", has different characteristics in terms of the document that is searched by the query (e.g., project, class) and in terms of the information need that is expressed through the query (e.g., the user may want to search a specific project or all the projects related to a topic). We first outlined a set of five meta-queries, then we chose two of them. For each of these, we built a set of ten instances that we used to test the UML case.

The tests for the WebML case involve a set of ten queries. Each query searches WebML models (or fragments of WebML models) that apply the most common patterns, such as the "search pattern" [32]. The queries can be submitted to the system as document-based queries in form of a WebML model. As explained in Section 6.4, the Query Processing phase deals with the translation of the document-based query into the corresponding text-based query that is then sent to the search platform.

For both the UML and the WebML experiments we tested various configurations involving different indexing options. For example, a test configuration of the WebML case involves the dereferentiation of some references of the project models.

To test the effectiveness of the system as a search engine system, in each test configuration we return only the first ten relevant documents. Therefore, the evaluation metrics assess the quality of the system only up to the tenth

rank position.

We conducted a manual assessment in order to build a ground truth for the query-to-project relevance. Each query was manually evaluated against the projects in the repository in order to assign a value of relevance to the project elements with respect to a given query.

This chapter is organized as follows. Section 7.1 presents the theoretical background of the metrics we used to evaluate the experiments. Section 7.2 provides details on the methodology adopted to build the ground truth with which the tests are compared. Section 7.3 shows the types of query we used for testing the UML experiments, then the test configurations and finally it discusses the results. Section 7.4 focuses on the WebML test configurations and their results. Section 7.5 sums up the main findings that emerged from the results of the evaluation.

7.1 Evaluation Metrics

Discounted Cumulative Gain - DCG Discounted Cumulative Gain is a popular measure for evaluating web search engines and related systems. When using DCG there are two assumptions [34]: highly relevant documents are more valuable than marginally relevant documents, and the greater the ranked position of a relevant document, the less valuable it is for the user, because the less likely it is that the user will ever examine the document. DCG is defined as the sum of the "gain" of presenting a particular document times a "discount" of presenting it at a particular rank i, up to some maximum rank l.

$$DCG_l = \sum_{i=1}^{l} gain_i \times discount_i$$

For web search, "gain" is typically a relevance score determined by human judgment , and "discount" is the reciprocal of the logarithm of the rank. Therefore, putting a document with a high relevance score at a low rank results in a much lower value of DCG than putting the same document at a higher rank.

$$DCG_l = rel_1 + \sum_{i=2}^{l} rel_i \times \frac{1}{log_2 i}$$

rel_i are the relevance scores. These typically are scalar values somehow related to the human relevance judgment with respect to a test query. Given the test query, a human provides his judgment about the relevance of each

document in the collection with respect to that test query. An example of relevance score is: $rel_i \in \{0, 1, 3\}$. These values can be interpreted as 0 “not relevant”, 1 “relevant”, 3 “very relevant”.

Obviously, the values obtained by the previous formula can be plotted in a chart with DCG values on vertical axis and rank positions i on horizontal axis. The DCG curve is then compared to the IDCG curve, that is the Ideal Discounted Cumulative Gain curve. The IDCG curve is obtained computing the DCG on the perfect ranking (the ranking where the most relevant documents come first).

Differently from the other measures such as Precision and Recall, DCG also takes into account the positions of the relevant documents among the top l.

The DCG curves can also be averaged over a set of test queries to obtain a more precise assessment.

Precision at k - P@k Precision and Recall are the most popular measures in the Information Retrieval field. They require that a human judge (or another trustworthy system) performs a binary evaluation of each retrieved document as “relevant” or “not relevant“. Moreover, they need to know the complete set of relevant documents within the collection being searched:

$$Recall = \frac{number\,of\,relevant\,documents\,retrieved}{number\,of\,relevant\,documents}$$

$$Precision = \frac{number\,of\,relevant\,documents\,retrieved}{number\,of\,retrieved\,documents}$$

One way of plotting precision is looking at the precision at a fixed critical position of retrieving, that is ”Precision at k“, or P@k. For ranked lists assessment, k can be the number of results we expect users to look at. P@k computes the precision at a certain ranking position after a relevant document is retrieved. If a non-relevant document is retrieved, P@k equals zero.

P@k can be averaged over a set of test queries to obtain a unique curve for the assessment of the system.

11-point Interpolated Average Precision - 11pIAP The 11-point Interpolated Average Precision is the evolution of the Interpolated Precision p_{interp}. The precision-recall curves have a distinctive saw-tooth shape: if the $(k+1)^{th}$ document retrieved is a non-relevant one, then recall is the same as for the previous k documents, but precision drops down. If the $(k+1)^{th}$ document is a relevant one, then both precision and recall increase, and the

curve increases to the top right direction of the precision-recall chart. This implies some jiggles into the curve that it is often useful to remove in order to ease the comparison of different precision-recall curves. This is done by computing the interpolated precision curve. p_{interp} at a certain recall level r is defined as the highest precision found for any recall level $r' \geq r$:

$$p_{interp}(r) = \max_{r' \geq r} p(r')$$

The Interpolated Precision curve is for one ranked list only (i.e. one query). To evaluate the quality of a search engine there is often the need to obtain a single curve from several curves of different test queries. The traditional way of doing this is the 11-point Interpolated Average Precision. For each information need (i.e. each test query), the interpolated precision is measured at 11 recall levels of 0.0, 0.1, 0.2, ..., 1.0 (recall = 0.0:0.1:1.0). Then, for each recall level, the arithmetic mean of the interpolated precision at that recall level for each query is calculated.

Mean Average Precision - MAP Mean Average Precision derives from Average Precision (AP). AP provides a single number instead of a curve. It measures the quality of the system at all recall levels by averaging the precision for a single query:

$$AP = \frac{1}{RDN} \times \sum_{k=1}^{RDN} (Precision\, at\, rank\, of\, k^{th}\, relevant\, document)$$

where RDN is the number of relevant documents in the collection.

Mean Average Precision (MAP) is the mean of Average Precision over all queries. Most frequently, arithmetic mean is used over the query set.

Mean Reciprocal Rank - MRR Mean Reciprocal Rank (MRR) is the reciprocal of the rank of the first relevant result averaged over the number of test queries:

$$MRR = \frac{1}{Q} \times \sum_{q=1}^{Q} \frac{1}{rank(1^{st}\, relevant\, result\, of\, query\, q)}$$

where Q is the number of test queries.

7.2 Ground truth

Besides the document collection and the test suite of information needs expressed as queries, to evaluate the effectiveness of an Information Retrieval system, we need a set of relevance judgments, i.e. an assessment of the relevance degree for each query-document pair [35]. With respect to a user information need, a classification of the level of relevance is given to each document in the test collection. This decision is referred to as *ground truth* judgment of relevance. We performed this assessment manually for both the UML and the WebML model search case studies. We adopted a different type of ground truth depending on the type of evaluation metric. Precision at k, 11-point Interpolated Average Precision, Mean Average Precision, Mean Reciprocal Rank require a binary classification: $relevance \in \{0, 1\}$, which means that a document can be relevant (1) or not relevant (0) with respect to a given query. For Discounted Cumulative Gain we used a ternary decision of relevance: $relevance \in \{0, 1, 3\}$, which means that a document can be not relevant (0), relevant (1), very relevant (3). A ternary relevance is a good compromise between a more nuanced classification and a too sparse set of judgment values, like a binary assessment.

To construct the ground truth, we manually judged all the documents in the test collection with respect to each information need. We considered a document as "relevant", with respect to a given query, according to two main criteria. The first one addresses the similarities between the document and the query in terms of the *topical* characteristics. The document is relevant if it deals with the same topic of the query (e.g. the document and the query include same or similar terms). The second criteria addresses the similarities in terms of the *structural* information and the document is considered relevant if it includes a structure similar to the query. In the case of a WebML application, the structure is represented by the types of units that are used and the way they are connected in order to adopt a particular pattern. With respect to the above graded scale of relevance, we assigned a relevance of 3, if the document is similar to the query both in terms of topical and structural characteristics, while we assigned a relevance of 1, if the document is similar to the query just in terms of one criteria.

7.3 UML Tests and Results

7.3.1 Test Queries

For testing the UML experiments we first designed a set of “meta-queries”. A meta-query is a query type with specific characteristics in terms of the type of document that is searched by the query and in terms of the information need addressed by the query. In the following subsection we discuss about each of these meta-queries.

Figure 7.1 shows an example of UML project model from the dataset of UML class diagrams. We will use this example to present the meta-queries for the rest of this section.

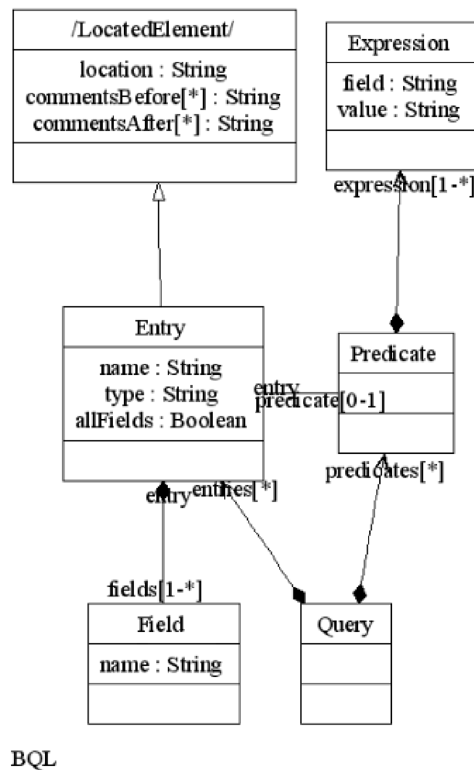


Figure 7.1: Example of UML project model from the dataset of UML class diagrams.

In Table 7.1, we summarize the meta-queries and, for each of them, we show a description and an example referring to the UML model in Figure 7.1. The "Target" column in Table 7.1 indicates the type of document the query is intended to search (e.g. a project or a class); the second column is the identifier of the meta-query; the third column briefly describes the meta-query; the last column shows an example of query instance.

Among those meta-queries, we chose the meta-query 2 and 5 for testing the UML experiments. In the following sections we refer to those meta-queries with the labels "MQ2" (meta-query 2) and "MQ5" (meta-query 5). Each experiment has been performed on ten query instances of the chosen meta-queries, and their results have been averaged. The complete lists of the ten query instances of both meta-queries 2 and 5 are presented respectively in Table 7.2 and in Table 7.3.

Target	Id	Description	Example
Project	1	It searches all the projects related to one specific topic	Query: "BQL"
	2	It searches all the projects related to one general topic	Query: "query language"
"Pattern"	3	It searches a "pattern" by using as query string the terms belonging to different classes connected by some relation	Searched "pattern": *Entry* [is-a] *LocatedElement* Query: "name type location commentsBefore"
Class	4	It searches a class by using as query string all (some) terms belonging to that class	Searched class: *LocatedElement* Query with all terms: "LocatedElement location commentsbBefore commentsAfter" Query with only some terms: "location commentsBefore"
	5	It searches a class by using as query string some terms belonging to that class plus some terms belonging to the project	Searched class: *LocatedElement* Query: "location commentsBefore Predicate Expression"

Table 7.1: The meta-queries designed for testing the UML experiments. The "Target" column indicates which type of document the query searches (e.g. a project, or a class); the second column is the identifier of the meta-query; the third column briefly describes the meta-query, namely it describes the information need that can be satisfied by the queries that are instances of the meta-query; the last column shows an example of query.

1	function source struct char int class member operator variable parameter
2	element node attribute name children
3	table column database
4	business process
5	task activity
6	message operation
7	node attribute
8	formula
9	color print size
10	transition

Table 7.2: The list of the ten instances of the meta-query 2 of the UML case study.

1	jar manifest classpath build
2	node expression
3	tag name
4	event
5	link
6	shape
7	program type source
8	note
9	task
10	process activity status finishmode

Table 7.3: The list of the ten instances of the meta-query 5 of the UML case study.

7.3.2 Test Configurations and Results

In this section we show the results of the tests of the UML experiments (A, B, C, D). We tested different experiment configurations in order to find the one giving the best results. The whole test set is grouped by meta-query. MQ2 has one test configuration and it refers to UML Experiment A (Project Granularity, Flat Index), since both the granularity of this experiment and the target of the meta-query are "project". MQ5 has five test configurations and it refers to UML Experiments B, C and D, since their granularity and the target of the experiments are "class". It is important to notice that the results of Experiment A and the results of Experiments B (Concept Granularity, Multi-Field Index), C (Concept Granularity, Multi-Field Weighted Index) and D (Concept Granularity, Multi-Field Weighted Index, Graph Based) are not comparable with each other because they retrieve different types of documents (Experiment A retrieves projects, the others retrieve classes).

The test configurations differ from each other according to some options: the value of the payloads assigned to the various UML metamodel concepts (only in Experiments C and D), the value of the penalties (only in Experiment D) and the FieldNorm which can be enabled or disabled.

FieldNorm is a factor influencing the score of a document retrieved through a query submitted to Solr. The shorter the matching field is (measured in number of indexed terms), the greater the FieldNorm value will be. Therefore, the score of the matching document will be greater. FieldNorm can be omitted from some fields in the Solr schema configuration. In most of the test configurations the FieldNorm in the Experiment D is disabled in order to solve the following problem: when a relevant class (with respect to a given query) imports many attributes from a neighboring class, the first one gets wrongly penalized by FieldNorm.

We use two sets of payload values. The first set (Table 7.4) is determined according to simple reasonings on the UML metamodel concepts. For example, a term that represents the name of a class should have a greater relevance than a term that represents a simple attribute. In another test, we use a slightly different set of payload values with respect to the previous one (Table 7.5). With this test we want to show that the results do not change significantly if the values of the payloads are slighly changed, thus verifying the stability of the system.

We use two sets of penalties, too. The first set (Table 7.6) results from reasonings on the types of UML relationships. For example, let Mammal and HumanBeing be two classes connected by a generalization relationship,

where Mammal is the parent class and HumanBeing is the child class. Since HumanBeing "is-a" Mammal, we want that the attributes that the child class imports from the parent class are only slightly penalized during the import algorithm, so, in this case, the penalty value is 0.9. As for the payloads, we tested the UML case study application with another set of penalties which shows that, by slightly varying them (Table 7.7), the results do not change substantially. Finally, we conducted a further test in which the values of the penalties are lowered by multiplying them by a factor 0.1 (Table 7.8). This configuration prevents an imported attribute to obtain a payload value greater than the payload value of an attribute originally contained in a class.

The different test configurations listed below show various combinations of the configuration options discussed above:

- MQ2: Experiment A (Project Granularity, Flat Index)

  Test Configuration 1: basic test with FieldNorm enabled.

- MQ5: Experiments B (Concept Granularity, Multi-Field Index), C (Concept Granularity, Multi-Field Weighted Index) and D (Concept Granularity, Multi-Field Weighted Index, Graph Based)

  Test Configuration 1: FieldNorm disabled only in Experiment D; payloads as in Table 7.4; penalties as in Table 7.6.

  Test Configuration 2: FieldNorm enabled in all experiments; payloads as in Table 7.4; penalties as in Table 7.6.

  Test Configuration 3: FieldNorm disabled only in Experiment D; payloads as in Table 7.5; penalties as in Table 7.6.

  Test Configuration 4: FieldNorm disabled only in Experiment D; payloads as in Table 7.4; penalties as in Table 7.7.

  Test Configuration 5: FieldNorm disabled only in Experiment D; payloads as in Table 7.4; penalties as in Table 7.8.

PAYLOAD VALUES (FIRST CONFIGURATION)

Concept	Payload Value
Attribute	1.0
Composition 1-1	1.5
Composition 1-N	1.3
Association 1-1	1.6
Association 1-N	1.3
Class	1.7
Project	1.0

Table 7.4: The first configuration of payload values. This configuration is determined according to simple reasonings on the UML metamodel concepts.

PAYLOAD VALUES (SECOND CONFIGURATION)

Concept	Payload Value
Attribute	0.9
Composition 1-1	1.4
Composition 1-N	1.2
Association 1-1	1.5
Association 1-N	1.2
Class	1.5
Project	0.9

Table 7.5: The second configuration of payload values. This configuration is determined by slightly changing the values of the first one.

PENALTY VALUES (FIRST CONFIGURATION)

Concept	**Penalty Value**
Composition (from composite class to component class) 1-1	0.6
Composition (from composite class to component class) 1-N	0.5
Composition (from component class to composite class) 1-1	0.6
Composition (from component class to composite class) 1-N	0.5
Association 1-1	0.6
Association 1-N	0.5
Generalization (from parent class to child class)	0.75
Generalization (from child class to parent class)	0.9

Table 7.6: The first configuration of penalty values. This configuration is determined according to simple reasonings on the UML relationship types.

PENALTY VALUES (SECOND CONFIGURATION)

Concept	**Penalty Value**
Composition (from composite class to component class) 1-1	0.5
Composition (from composite class to component class) 1-N	0.4
Composition (from component class to composite class) 1-1	0.5
Composition (from component class to composite class) 1-N	0.4
Association 1-1	0.5
Association 1-N	0.4
Generalization (from parent class to child class)	0.5
Generalization (from child class to parent class)	0.7

Table 7.7: The second configuration of penalty values. This configuration is determined starting from the first one by slightly changing penalty values.

PENALTY VALUES (THIRD CONFIGURATION)

Concept	Penalty Value
Composition (from composite class to component class) 1-1	0.06
Composition (from composite class to component class) 1-N	0.05
Composition (from component class to composite class) 1-1	0.06
Composition (from component class to composite class) 1-N	0.05
Association 1-1	0.06
Association 1-N	0.05
Generalization (from parent class to child class)	0.075
Generalization (from child class to parent class)	0.09

Table 7.8: The third configuration of penalty values. This configuration is determined by multiplying the values of the first one by a factor of 0.1.

In the following paragraphs we comment the results of each test configuration.

MQ2: Test Configuration 1 Figure 7.2 shows the plot of the DCG and iDCG curves of Experiment A. It can be noticed that the DCG and the iDCG curves are very close to each other, especially up to the first three positions. In particular, the results show that the Experiment A (Project Granularity, Flat Index) is always able to retrieve the most relevant document at the first position.

Figure 7.3 depicts the 11-points Interpolated Average Precision of the Experiment A (Project Granularity, Flat Index). The results show that the precision, up to the recall level of 0.4, is always 1. After this recall level, there is first a small decrease ($r = 0.5$) and then a more drastic decrease of the curve ($r = 0.82$). This is an expected behavior: as all the relevant documents are retrieved, among these, an increasing number of not relevant documents are retrieved too. The results also show that the Experiment A is able to retrieve all the relevant documents after the first ten retrieved documents.

Figure 7.4 presents the plot of the Precision at k curve of the Experiment A (Project Granularity, Flat Index).

The results suggest that, up to the third position, all the retrieved docu-

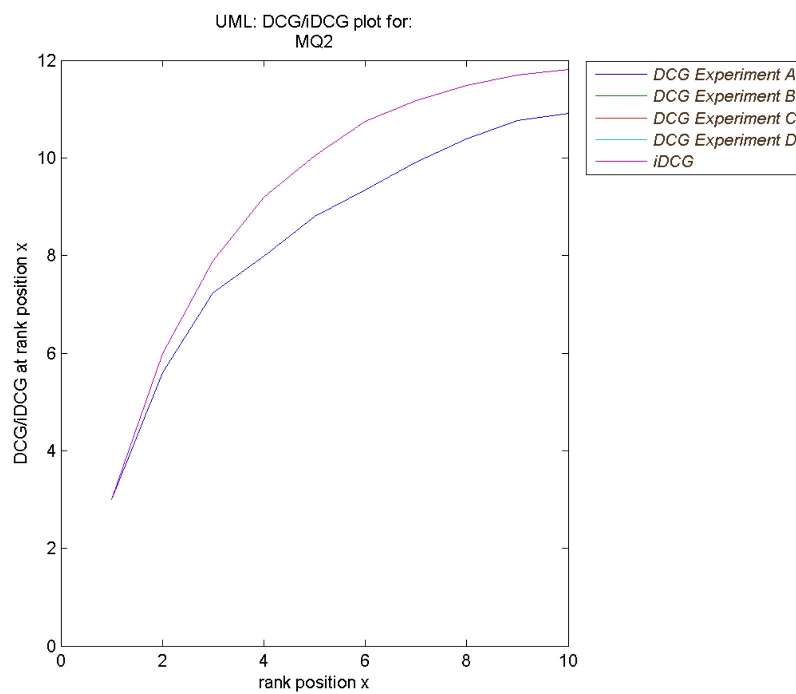


Figure 7.2: Plot of the DCG and iDCG curves of Experiment A (Project Granularity, Flat Index), Test Configuration 1 (MQ2).

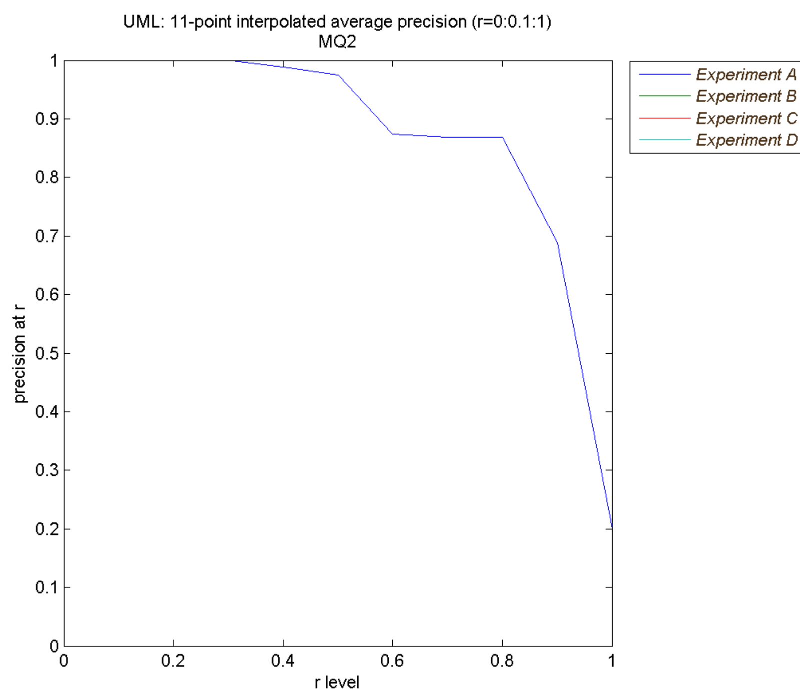


Figure 7.3: Plot of the 11-points Interpolated Average Precision of the Experiment A (Project Granularity, Flat Index), Test Configuration 1 (MQ2).

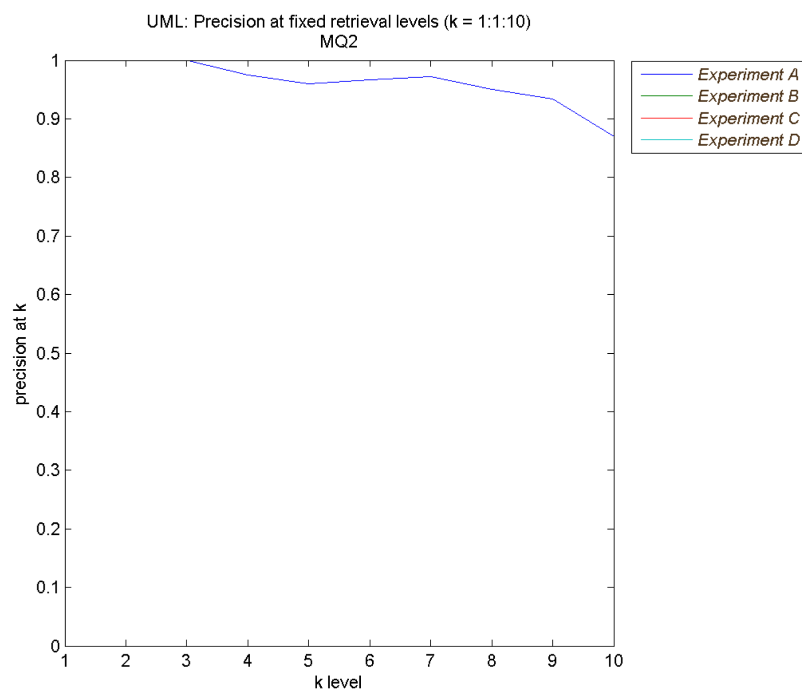


Figure 7.4: Plot of the Precision at k of the Experiment A (Project Granularity, Flat Index), Test Configuration 1 (MQ2).

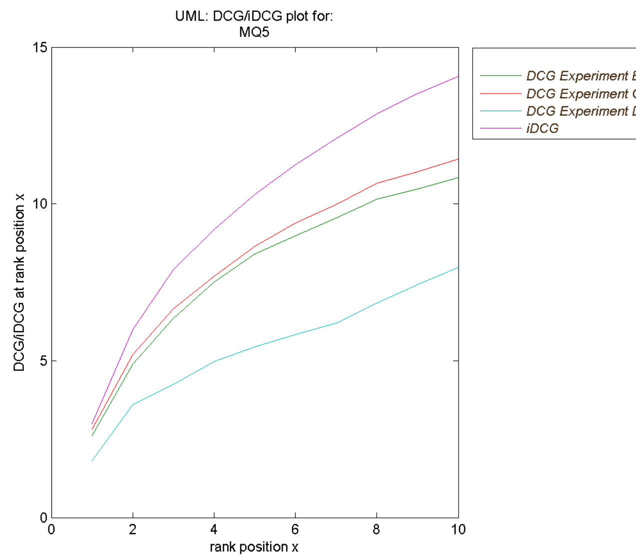
ments are relevant. Between the third and the fifth position there is a small fall in the curve, after which there is a slight improvement. This confirms the results previously shown with the DCG and iDCG curves. However, the results show that the precision level remains high (over 0.8) for all the k levels.

The MAP results for this test configuration are given in Table 7.9, while the MRR results are presented in Table 7.10. MRR suggests that Experiment A is always able to retrieve the first relevant document at the first ranking position.

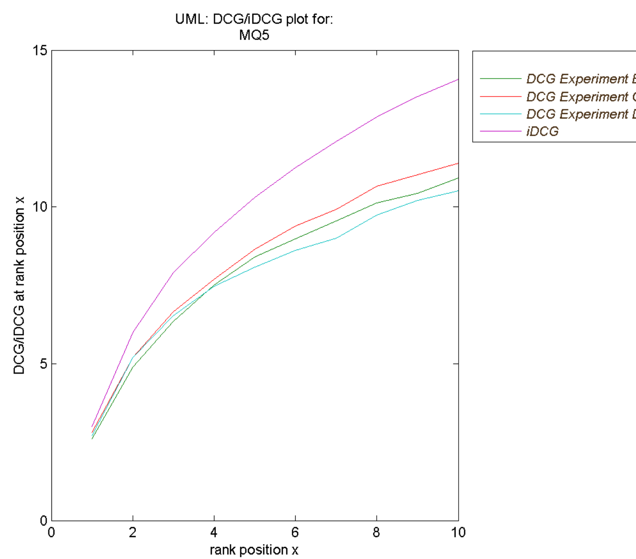
MQ5: Comparison between Test Configuration 1 and Test Configuration 2 In Figure 7.5 you can see the comparison between the DCG and iDCG curves of the Experiments B (Concept Granularity, Multi-Field Index), C (Concept Granularity, Multi-Field Weighted Index) and D (Concept Granularity, Multi-Field Weighted Index, Graph Based) when the experiments use the first test configuration (Figure 7.5(a)) and the DCG and iDCG curves of the Experiments B, C and D where the experiments use the second test configuration (Figure 7.5(b)).

The results for the Experiments B and C are almost the same in Test Configuration 1 and 2. Only the last part of the curves differ. This behavior is due to the way Solr handles ties: in case of retrieved documents that have the same score, a fixed ordering of documents is not respected (e.g., alphabetically). However, this issue occurs only in the lower parts of the ranking. The Figure 7.5 shows that the Experiment C has, for each rank position k, a slightly better value of DCG than Experiment B, showing that the use of payloads improves the results even if not so significantly. Both experiments B and C, up to the second rank position, are close to the ideal curve.

These results presented in Figure 7.5 are intended to show the effects of FieldNorm on the Experiment D. If FieldNorm is enabled (Test Configuration 2), then a class that is relevant with respect to a query and that imports many elements from a neighboring class is unfairly penalized. Therefore, the FieldNorm should always be disabled and Test Configuration 1 should be the best scenario to choose. But, as Figure 7.5 suggests, the results of Experiment D in Test Configuration 2 are much better than Test Configuration 1. The reason is explained as follows. Besides retrieving the relevant classes with respect to a query, Experiment D retrieves their neighboring classes too, which are not necessarily relevant to that query. These neighboring classes are present among the results because they have imported terms that are part of the query string. Since their “content” field is larger due



(a) Test Configuration 1.



(b) Test Configuration 2.

Figure 7.5: Comparison between Test Configuration 1 and 2: DCG and iDCG curves of the Experiments B (Concept Granularity, Multi-Field Index), C (Concept Granularity, Multi-Field Weighted Index) and D (Concept Granularity, Multi-Field Weighted Index, Graph Based) for both the test configurations.

to the imported terms, those neighboring classes are penalized by the Field-Norm and, at the same time, the truly relevant classes are ranked in a higher position, therefore the results are better. To conclude, the FieldNorm helps when it penalizes classes that are retrieved only because they are neighboring of relevant classes, but it provides misleading results when it penalizes the relevant classes due to the larger size of their "content" field after the import algorithm.

Figure 7.6 shows the plot of Precision at k of Test Configuration 2.

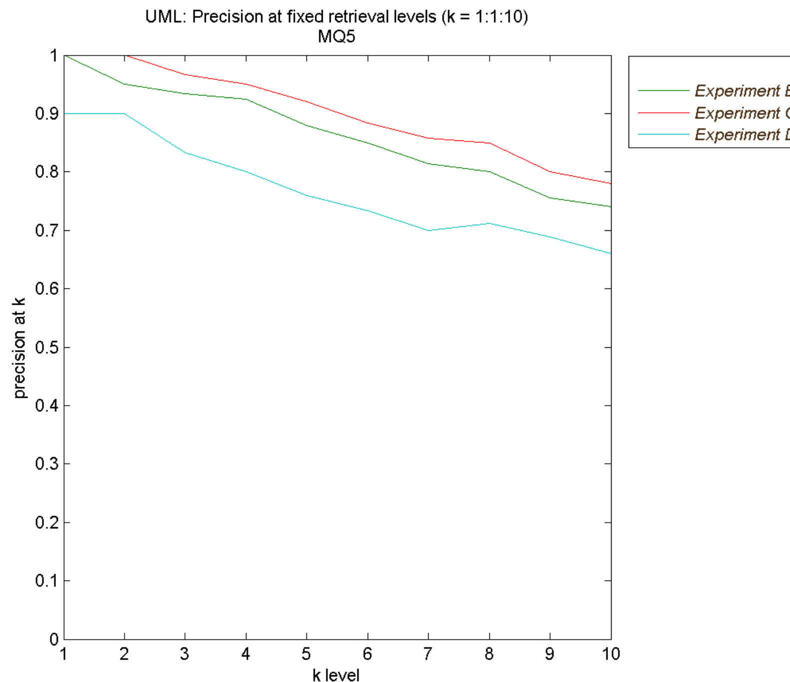


Figure 7.6: Plot of the Precision at k of Experiments B (Concept Granularity, Multi-Field Index), C (Concept Granularity, Multi-Field Weighted Index) and D (Concept Granularity, Multi-Field Weighted Index, Graph Based), Test Configuration 2 (MQ5).

It can be noticed from the results that the curves of Experiments B and C have almost the same shape, while Experiment D presents a peculiar shape. At the first k ranking positions the precision is high and constant, which means that the first part of retrieved classes is relevant. Then the precision decreases for several rank positions due to the neighboring classes of the previously retrieved classes. Approximately at $k = 7$, the precision increases again because other relevant documents are retrieved, then it decreases again. This trend is typical of Experiment D and it would be visible

more times into the curve if more than ten rank positions were showed.

MAP (Table 7.9) confirms that the best performing experiment is Experiment C and that Experiment D performs better in Test Configuration 2. MRR (Table 7.10) suggests that Experiments B and C retrieve as first document always a relevant one.

MQ5: Test Configuration 3 Figure 7.7 shows the DCG curves of Experiments C and D for this test configuration.

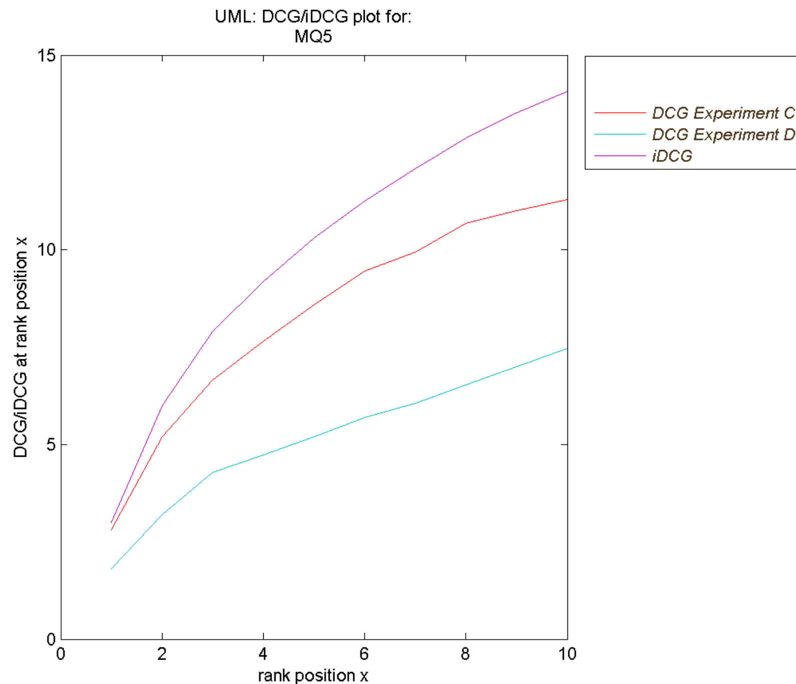


Figure 7.7: DCG curves of Experiments C (Concept Granularity, Multi-Field Weighted Index) and D (Concept Granularity, Multi-Field Weighted Index, Graph Based), Test Configuration 3 (MQ5).

The curves of the Experiments C and D with the payload values configuration slightly changed (Figure 7.7) can be compared with the curves of the Experiments C and D using Test Configuration 1 (Figure 7.5(a)).

These results confirm the low sensitivity of Experiments C and D with respect to a slightly change to the payload values. The slightly changed configuration of the payloads causes only a light improvement to the curve of Experiment D, but that is not the purpose of this test configuration.

We point out that we performed training neither with weights nor with

payload.

MQ5: Test Configuration 4 Figure 7.8 depicts the DCG curve of Experiment D for this test configuration.

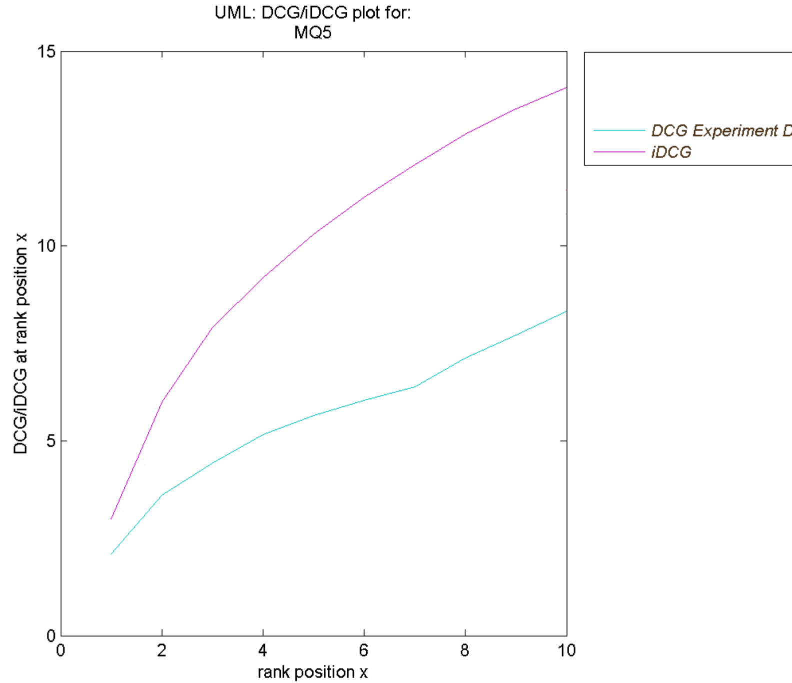


Figure 7.8: DCG curve of Experiment D (Concept Granularity, Multi-Field Weighted Index, Graph Based), Test Configuration 4 (MQ5).

The curve of the Experiment D with the payload values configuration slightly changed (Figure 7.8) can be compared with the curve of the Experiment D using Test Configuration 1 (Figure 7.5(a)).

The Experiment D shows very low variability in the results with respect to a small change of the payload values.

MQ5: Test Configuration 5 As Figure 7.9 suggests, the Experiment D improves its results when using payloads multiplied by a 0.1 factor which decreases all the payload values, and, accordingly, all the payloads of the imported attributes.

This test configuration stresses the point that it is fundamental to use a configuration of both payloads and penalties that prevents the imported

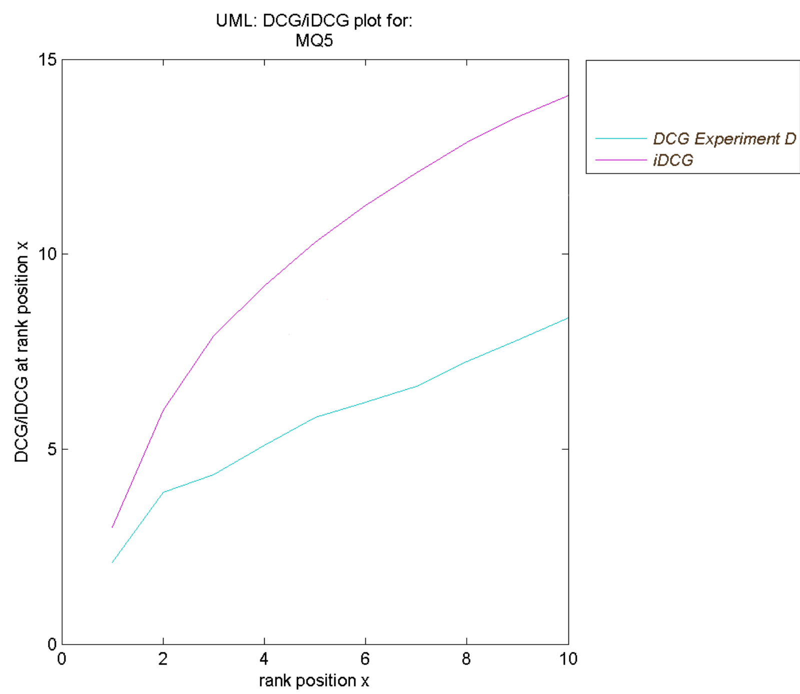


Figure 7.9: DCG curve of Experiment D (Concept Granularity, Multi-Field Weighted Index, Graph Based), Test Configuration 5 (MQ5).

Mean Average Precision (MAP)

	Exp A	Exp B	Exp C	Exp D
MQ2 - Test Configuration 1	0.98	-	-	-
MQ5 - Test Configuration 1	-	0.92	0.95	0.71
MQ5 - Test Configuration 2	-	0.92	0.95	0.84
MQ5 - Test Configuration 3	-	0.92	0.94	0.66
MQ5 - Test Configuration 4	-	0.92	0.95	0.69
MQ5 - Test Configuration 5	-	0.92	0.95	0.69

Table 7.9: MAP results of the test configurations of the UML model-based search engine. MQ2 addresses only Experiment A (Project Granularity, Flat Index); MQ5 addresses Experiments B (Concept Granularity, Multi-Field Index), C (Concept Granularity, Multi-Field Weighted Index) and D (Concept Granularity, Multi-Field Weighted Index, Graph Based).

attributes of a class to have a greater payload value than the one of the attributes already contained in that class.

Mean Reciprocal Rank (MRR)

	Exp A	Exp B	Exp C	Exp D
MQ2 - Test Configuration 1	1.00	-	-	-
MQ5 - Test Configuration 1	-	1.00	1.00	0.76
MQ5 - Test Configuration 2	-	1.00	1.00	0.95
MQ5 - Test Configuration 3	-	1.00	1.00	0.76
MQ5 - Test Configuration 4	-	1.00	1.00	0.81
MQ5 - Test Configuration 5	-	1.00	1.00	0.81

Table 7.10: MRR results of the test configurations of the UML model-based search engine. MQ2 addresses only Experiment A (Project Granularity, Flat Index); MQ5 addresses Experiments B (Concept Granularity, Multi-Field Index), C (Concept Granularity, Multi-Field Weighted Index) and D (Concept Granularity, Multi-Field Weighted Index, Graph Based).

7.4 WebML Tests and Results

For the evaluation of the WebML experiments we manually built a set of ten models with different sizes to be used as queries. The queries reflect the use case in which a user want to search for frequently used WebML patterns. Figure 7.10 depicts an example of query.

The pattern of operations expressed by the piece of project in Figure 7.10 is the following: the user, previously logged into the Web site, browses the “New book request page” and adds a new Book to the collection of book requests through the entry unit called “New book”; the submission of the entry form triggers the creation of a new Book entity and the connection between the newly created Book entity and the User entity; if the tasks executed by the operation units end with success, the user is redirected to the “Book request list” page. Since the WebML case involves content-based queries, the query shown in Figure 7.10 is an entire WebML project that includes all the concepts of the WebML metamodel (site view, area, page, unit, etc.). As discussed in Section 6.4, the Query Processing phase of the WebML case is responsible for transforming the content-based query into a text-based query by extracting the terms contained into the query. In this example the Keyword Extraction task provides the following keywords for the given query: “Book requests Create book ConnectUserToBook New book request New book User Book request list”. In Table 7.11 we show the complete list of the ten queries we used to evaluate the WebML case study application.

As explained in Section 6.3, our experiments were conducted on a project

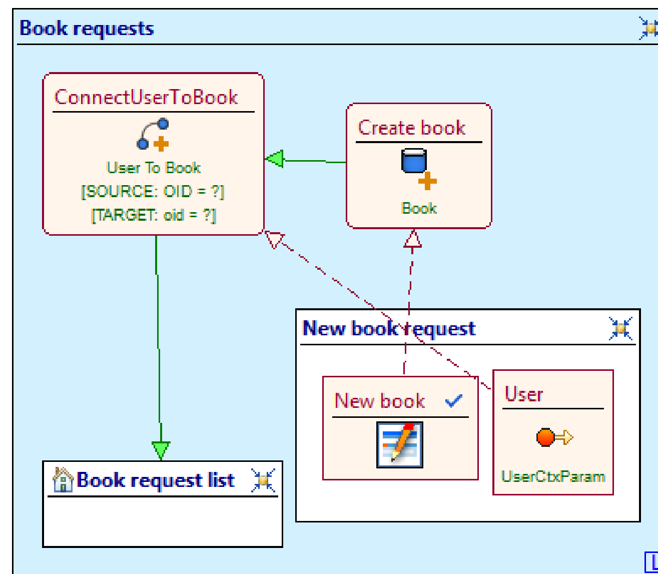


Figure 7.10: An example of document-based query used to test the WebML experiments.

1	Manage Products Manage Products Products List Search Product
2	Publication create new pub Enter New Publication New publication Publication type
3	Modify Modify Modify user Modify user data default group subject List User list
4	Client Details Create Modify Exist Project Details Project details Title
5	Manage clients Make calls Manage clients
6	Responsibles compose mail info Ask for information Mail
7	Manage appointments Manage appointments Appointments List
8	Book requests Create book ConnectUserToBook New book request New book User Book request list
9	Manage documents ModifyUnit1 New Document Modify document Document data Document details Document list Delete document
10	Contract type Delete Contracts Contract types

Table 7.11: The complete list of the ten text-based queries used to evaluate the WebML case study application. The keywords are extracted from the content-based version of the queries.

repository composed of twelve real-world industrial WebML projects from different application domains (e.g., human resource management, Web portals, etc.).

To assess the quality of the WebML experiments we tested them with different configurations:

- *Test Configuration 1*: this is the basic test configuration.
- *Test Configuration 2*: this test configuration includes the dereferentiation of the "to" attributes of Link elements.
- *Test Configuration 3*: this test configuration includes the dereferentiation of the "to" attributes of Link elements and of all the "displayAttributes" and "entity" attributes of the OperationUnit and ContentUnit elements.
- *Test Configuration 4*: this test configuration includes the indexation of the names of the WebML metamodel concepts (e.g., site view, area, page, data unit, etc.).
- *Test Configuration 5*: this test configuration includes the indexation of the names of the WebML metamodel concepts and assigns a payload of 0.1 to those terms.

Test Configurations 2 and 3 enable the dereferentiation of the ids contained in the "entity" and "displayAttributes" attributes of Operation Unit and Content Unit elements, and the "to" attributes of Link elements. This latter references are dereferenced by replacing them with the "name" attribute of the object pointed by the link. The dereferentiation of the ids contained in the "entity" and "displayAttributes" attributes of OperationUnit and ContentUnit elements is done by taking the name referenced by that id from the Data Model of the WebML project in which the Unit element is contained.

The following one is an example of the XML code of an area called "Request Information" containing some ids that will get dereferenced:

```
<packagedElement name="Request_Information" xmi:id="sv1b#area6g"
     xmi:type="webml:Area">
         ...
  <packagedElement name="Request_Info" xmi:id="sv1b#area6g#
       page3g" xmi:type="webml:Page">
    <packagedElement displayAttributes="ent1#att1_ent1#att5b"
        entity="ent1" name="Report" xmi:id="sv1b#area6g#page3g#
        dau2g" xmi:type="webml:DataUnit">
```

```
      <packagedElement name="Link_59" to="sv1b#area6g#opg5g#
          seu9g" xmi:id="sv1b#area6g#page3g#dau2g#ln59g" xmi:type
          ="webml:Link"/>
    </packagedElement>
    <packagedElement name="Info" xmi:id="sv1b#area6g#page3g#
        enu4g" xmi:type="webml:EntryUnit">
      <packagedElement name="Link_57" to="sv1b#area6g#opg5g#
          cru4g" xmi:id="sv1b#area6g#page3g#enu4g#ln57g" xmi:type
          ="webml:Link"/>
      <packagedElement name="Post" to="sv1b#area6g#opg5g#seu9g"
          xmi:id="sv1b#area6g#page3g#enu4g#ln58g" xmi:type="
          webml:Link"/>
    </packagedElement>
  </packagedElement>
        ...
</packagedElement>
```

In the example above, the ids contained in the "displayAttributes" attribute of the Data Unit called "Report" will be dereferenced with the corresponding name of the entities contained in the Data Model. Also the attributes "to" of the Link elements will be dereferenced with the name of the element they point to.

In the Test Configuration 4 and 5 we add to the index the terms that represent the name of the WebML metamodel concepts. Also the names of the metamodel concepts of the query elements are added to the the query string. Figure 7.11 depicts an example of WebML area and the text below shows the field "content" in case of Test Configuration 4 and 5 (the original words have already been analyzed; the field contains the terms generated by the content analysis):

```
web webmodel model web webmodel model administr site siteview
    view compet center area save dc creat createunit unit nop op
    oper noopoperationunit unit nop op oper noopoperationunit
    unit save modifi modifyunit unit compet center page compet
    center power index powerindexunit unit entri unit 1 entri
    entryunit unit new compet center page new compet center entri
     entryunit unit edit code page modifi cc data dataunit unit
    edit compet center entri entryunit unit
```

For example, the entry unit "New competence center" is indexed with the name of its metamodel concept ("entry unit"). The Test Configuration 4 and 5 are intended to increase the recall of the system.

The test configurations are assessed adopting the evaluation metrics discussed in Section 7.1. The results are commented and compared as follows.

Figure 7.12 shows the DCG and iDCG curves of the first three test configurations.

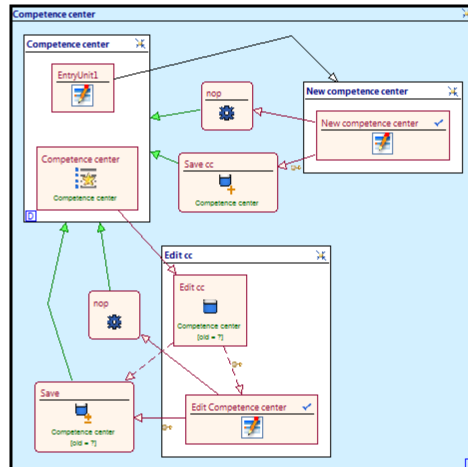
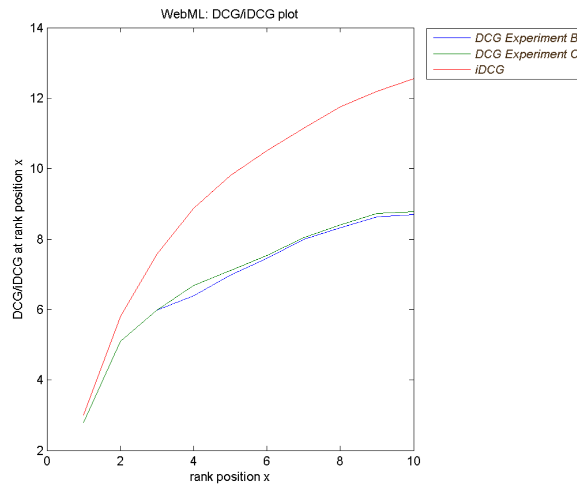


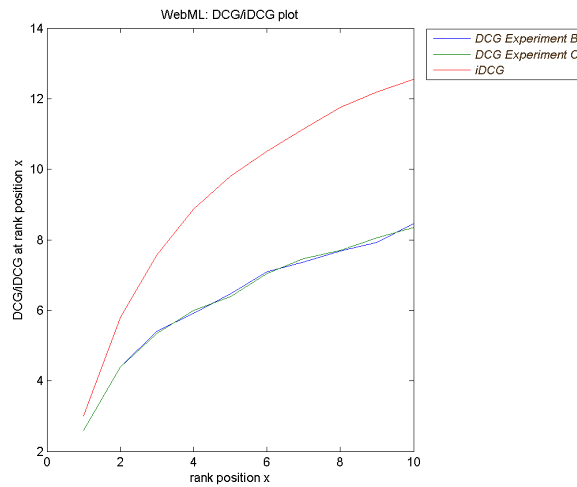
Figure 7.11: An example of WebML area adopted to explain test configuration 4 and 5.

With Test Configuration 1 (7.12(a)), Experiments B (Concept Granularity, Multi-Field Index) and C (Concept Granularity, Multi-Field Weighted Index) perform with the same results up to the second ranking position. Except for Test Configuration 2 (7.12(b)), the Experiment C performs always better than Experiment B: these two facts suggest that the use of payloads slightly improves the quality of the results. The results of Test Configuration 2 (7.12(b)) and of Test Configuration 3 (7.12(c)) compared to the results of Test Configuration 1 (7.12(c)) show that the dereferentiation doesn't improve the quality of the system: in general, the names of the Link elements are not meaningful and their dereferentiation causes the indexing of terms that will never be relevant with respect to any query. At the same time, the "content" field of the relevant documents becomes larger and this penalizes them due to the FieldNorm factor.

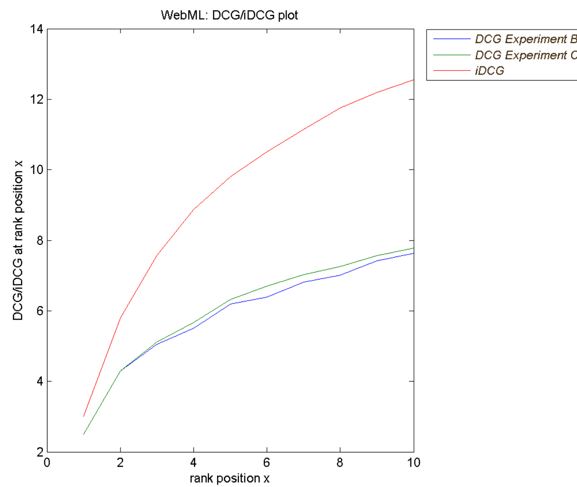
Figure 7.13 depicts the plot of 11-point Interpolated Average Precision of WebML Experiments B (Concept Granularity, Multi-Field Index) and C (Concept Granularity, Multi-Field Weighted Index) with Test Configuration 1. These results show that both Experiment B and C are able to retrieve 40% of the relevant documents after ten documents are retrieved. After the first 20% of relevant retrieved documents, the precision is decreased to quite low values (approximately 0.2). This behavior is due to the way we established the ground truth for the WebML dataset (see Section 7.2). We judged as relevant not only those documents that have terminological or conceptual similarities with respect to the query, but also those documents that employ



(a) DCG and iDCG curves of Test Configuration 1.



(b) DCG and iDCG curves of Test Configuration 2.



(c) DCG and iDCG curves of Test Configuration 3.

Figure 7.12: DCG and iDCG curves of the first three WebML test configurations.

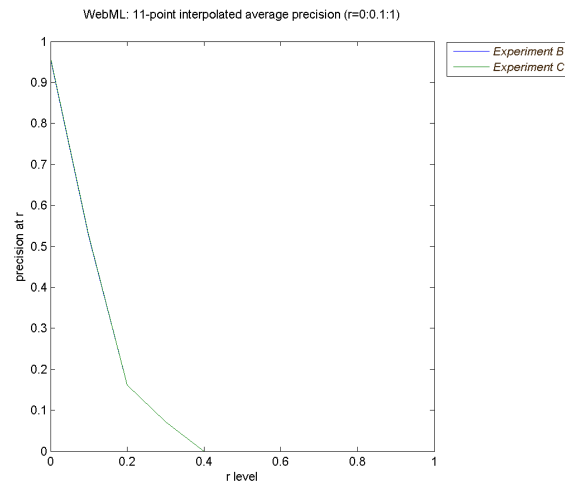


Figure 7.13: Plot of 11-point Interpolated Average Precision of WebML Experiments B (Concept Granularity, Multi-Field Index) and C (Concept Granularity, Multi-Field Weighted Index), Test Configuration 1.

the same kind of WebML pattern adopted in the query and, therefore, have the same structure. Since our system is text-based (although it exploits information of the metamodel), it is not able to retrieve those documents that are relevant because of their structural similarity with respect to the query. This is also a justification to investigate in future works the results that can be obtained by adopting graph-based techniques that takes into account the structural similarities between projects into the repositories and queries.

Figure 7.14 shows the Precision at k curve of WebML Experiments B (Concept Granularity, Multi-Field Index) and C (Concept Granularity, Multi-Field Weighted Index) with Test Configuration 1. These results confirm that the Experiment C is slightly better than Experiment B also in terms of precision. Both Experiments B and C have good results in terms of precision up to the third position (the precision is greater or equal than 0.8).

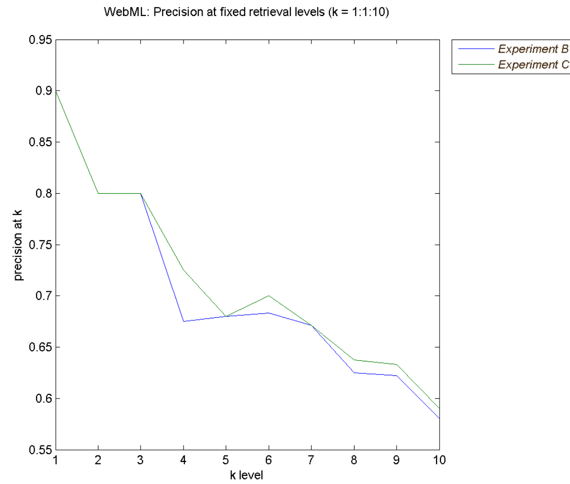


Figure 7.14: Plot of the Precision at k curve of WebML Experiments B (Concept Granularity, Multi-Field Index) and C (Concept Granularity, Multi-Field Weighted Index), Test Configuration 1.

Mean Average Precision (MAP)

	Exp B	**Exp C**
Test Configuration 1	0.80	0.81
Test Configuration 2	0.77	0.78
Test Configuration 3	0.76	0.77

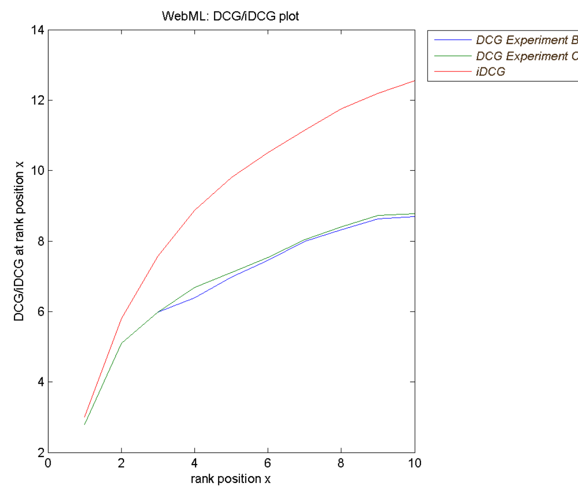
Table 7.12: MAP results of the test configurations of the WebML model-based search engine.

Table 7.12 reports the MAP results of the first three test configurations, while Table 7.13 reports the MRR results.

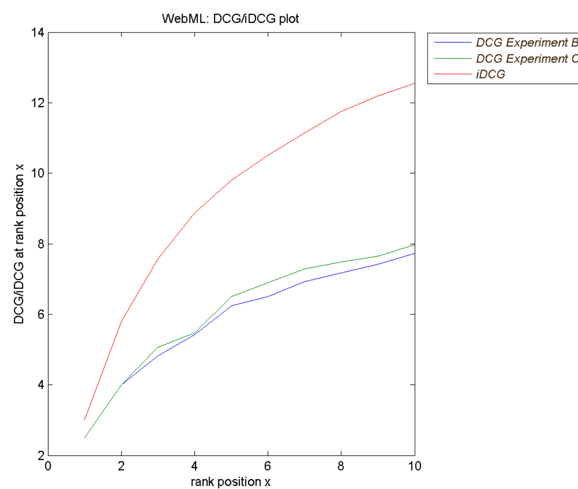
MAP confirms that Experiment C (Concept Granularity, Multi-Field Weighted Index) is the best scenario in all the configurations and that Test Configurations 2 and 3 don't improve the results. MRR suggests that all the experiments, especially with Test Configurations 2 and 3, are able to retrieve the first relevant document at a very high ranking position.

Figure 7.15 shows a comparison between the DCG and iDCG curves of the test configuration 1 (no dereferentiation), 4 (names of the metamodel concepts added to the index and to the query string) and 5 (name of the metamodel concepts added to the index with a payload of 0.1 and to the query string).

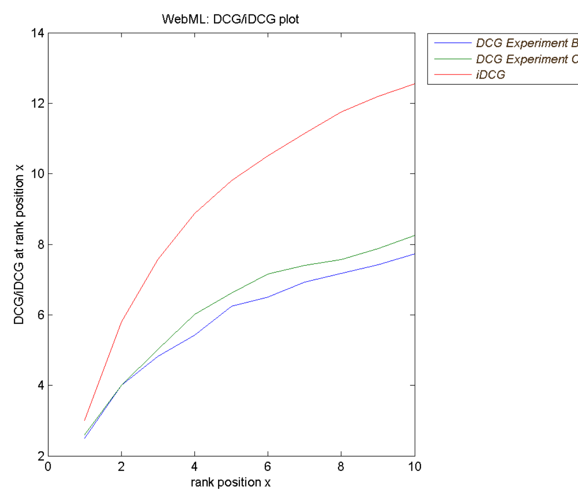
In test configurations 4 and 5 every document becomes a match for a



(a) DCG and iDCG curves of Test Configuration 1.



(b) DCG and iDCG curves of Test Configuration 4.



(c) DCG and iDCG curves of Test Configuration 5.

Figure 7.15: DCG and iDCG curves of the test configuration 1, 4 and 5.

Mean Reciprocal Rank (MRR)

	Exp B	Exp C
Test Configuration 1	0.93	0.93
Test Configuration 2	0.95	0.93
Test Configuration 3	0.90	0.90

Table 7.13: MRR results of the test configurations of the WebML model-based search engine.

query and, as Figure 7.15 suggests, the results decrease with respect to test configuration 1. This is due to the indexation of a lot of terms (i.e., the names of the WebML metamodel concepts) that alter the contribution of the Tf-idf when computing the score of the documents. The prove of this fact is that the results of Test Configuration 5 are better because we decrease the importance of the names of the concepts by assigning to them a payload of 0.1 as opposed to 1.0.

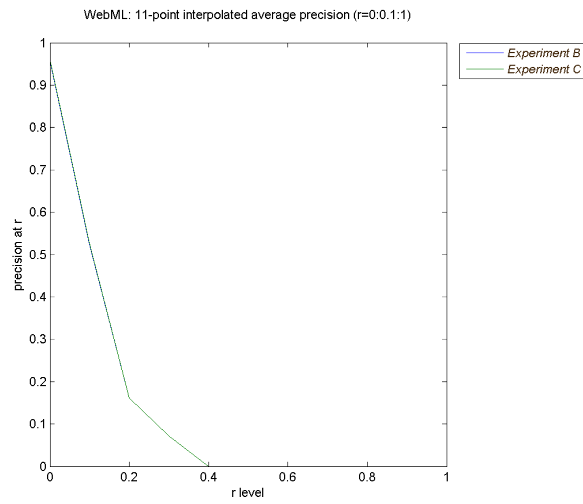
Figure 7.16 shows a comparison between the 11-point Interpolated Average Precision curves of the test configuration 1 (no dereferentiation), 4 (names of the metamodel concepts added to the index and to the query string) and 5 (name of the metamodel concepts added to the index with a payload of 0.1 and to the query string).

Figure 7.16 shows that the recall in the configurations 4 and 5 increases with respect to the Test Configuration 1, at the expense of the precision. These results confirm the conclusions emerged with the DCG curves.

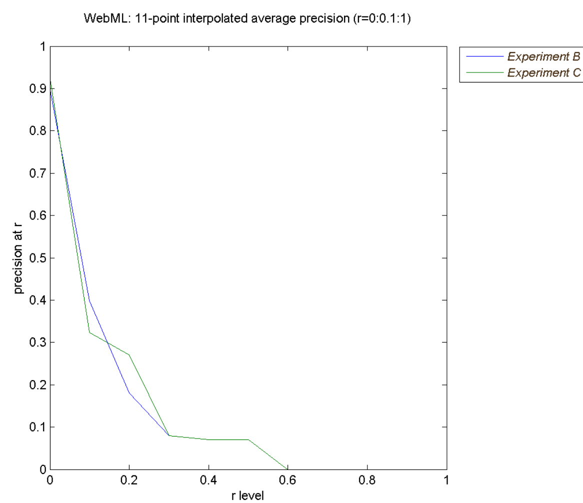
To conclude, the little recall improvements in the configurations 4 and 5 do not justify the clear drop in precision.

7.5 Main findings

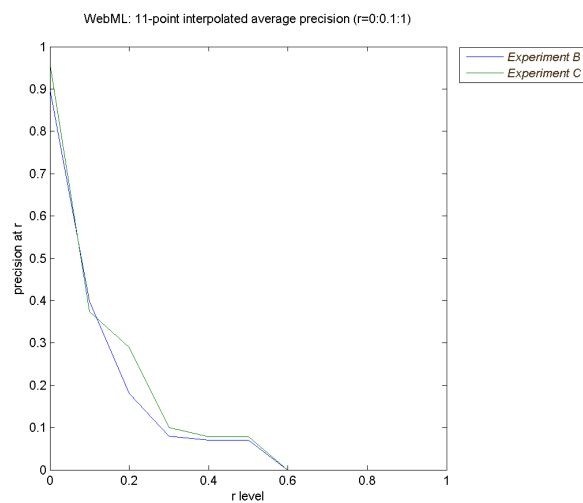
In general, there are no significant differences between the results of the UML and WebML case studies. This means that the behavior of our model search engine is not strictly dependent to the metamodel in use. In all the experiments and in all the test configurations the prototype has been able to retrieve relevant documents in the top positions. Experiment C (Concept Granularity, Multi-Field Weighted Index) takes into account information from the metamodel by properly weighting the index terms according to the concept that those terms represent. The results of this experiment are slightly better than those of Experiment B (Concept Granularity, Multi-Field Index). Experiment D of the UML case study involves a graph-based



(a) DCG and iDCG curves of Test Configuration 1.



(b) DCG and iDCG curves of Test Configuration 4.



(c) DCG and iDCG curves of Test Configuration 5.

Figure 7.16: DCG and iDCG curves of the first three WebML test configurations.

algorithm that creates a graph representation of each model into the repository. The nodes of the graph are the elements corresponding to the selected segmentation granularity and the edges are their respective relationships. Then, each element is enriched with proper information imported from its neighbours. This experiment is an attempt to exploit, the structural information contained into the models even in a text-based approach. Although introducing some structural information of the models into the index, the results points out that the strategy adopted into Experiment D leads to an important issue: in some cases, a class that is relevant with respect to a query becomes non-relevant due to the large amount of imported attributes. To conclude, the prototype has shown good results in retrieving documents that are relevant in terms of conceptual and terminological similarity. Conversely, especially in the WebML case study, the quality of the results decreases when the established ground truth considers as relevant also those documents that are structurally relevant with respect to the query because, by construction, a text-based search engine is not able to capture that type of similarity.

Chapter 8

Conclusions and Future Work

In this thesis we defined a model-driven methodology to search through model repositories. We described the general approach, analyzing the design parameters and some of the possible indexing strategies. The various strategies include: variations in the granularity of the returned result, weighting terms in order to better capture the different level of importance of the concepts of a metamodel and a graph-based method that allows to take into account the relationships between artifacts close to each other even in a purely text-based search. We developed a text-based prototype based on the SMILA framework in order to perform actual tests on the quality of such methodology over two very diverse datasets. The first one consists of models conforming to the UML class diagram metamodel, while the second one consists of models conforming to the WebML metamodel. This choice allowed us to test the system both over a general purpose modeling language (UML) and a domain specific modeling language (WebML). The UML dataset is composed by 84 metamodels, most of which are pretty small in size. The fact that they represent metamodels instead of models could lead to some difficulties in defining queries because of the abstract nature of a metamodel. This also would probably not be the situation of a real-world scenario. But metamodels are still models and we selected the meta-queries with the intention to reduce the impact of this issue, thus we think that our tests are still valid. The WebML dataset is instead composed by 12 real-world industrial projects, all of which vary in size from huge to small. Finally, we performed the evaluation of the results returned by the system with respect to the manually prepared ground truth. In the majority of the tests, the results show that the system is able to retrieve the most rel-

evant documents at the higher rankings. However, there are still margins of improvement especially for the weighted case. In fact, we selected the weights based on our prior knowledge of the metamodel, but we didn't perform any kind of numerical training. To conclude, the prototype has shown good results in retrieving documents that are relevant in terms of conceptual and terminological similarity while the structural similarity is difficult to be considered in a text-based search.

The evaluation of the quality of a search system is totally dependent on the ground truth against which it is assessed. The ground truth itself is subjected to mistakes as it is built manually by humans. Moreover, user information needs can be of many different natures. For example, there's the need to quickly find something specific to which the existence is already known by the user. This particular answer will be the only one able to satisfy the user. Then, there's another kind of need where the user wants to find something more abstract, to which the satisfactory answer can assume multiple forms. It is difficult to reflect in the same ground truth and in the same use cases those different needs. Providing multiple ground truths presents itself with practical problems as well, because the task of creating one is extremely long and prone to errors. So, we assessed just a couple out of all the millions of possible combinations between user needs and system configurations. Those situations are detailed in Chapter 7. It is higly probable that more configurations and use cases need to be investigated to completely assess the quality of a system, because it could perform well in one case and worse in another.

8.1 Future Work

Here is a list of the things that is possible to expand or explore in the future:

- Integrating a model-based solution: since our prototype is text-based, the natural evolution would be to implement a purely model-based system on top of this framework. There is already a started project that is moving in this direction.

- Metamodel integration: there is room for improvement in the way the user, intended as developer or system integrator, can exploit the knowledge of the metamodel. It is possible to implement a "tagging" mechanism that would allow to mark the concepts of interest in the metamodel. The system will then automatically take actions in order to take into account those user-defined directives. The goal is to

make the process of metamodel exploitation more and more automatic resulting in less burden on the user.

- Testing more configurations: we did not investigate all the possible indexing and searching strategies mostly because the number of combinations is huge. But it is surely possible to improve the proposed strategies or propose new ones. This is aided by the fact that the system is easily configurable and designed to easily integrate new operations.

- Weigth training: it is possible to perform a training of the weights for the experiments that make use of them. This way it will be possible to discover the maximum theoretical performance gain between the weighted experiments and the non-weighted ones. The training will be different for each dataset. The objective would be to automatically maximize one of the quality measures, like the DCG.

Bibliography

[1] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. Program understanding and the concept assignment problem. *Commun. ACM*, 37:72–82, May 1994.

[2] Tomás Isakowitz and Robert J. Kauffman. Supporting search for reusable software objects. *IEEE Trans. Softw. Eng.*, 22:407–423, June 1996.

[3] Oleksandr Panchenko, Hasso Plattner, and Alexander Zeier. What do developers search for in source code and why. In *Proceeding of the 3rd international workshop on Search-driven development: users, infrastructure, tools, and evaluation*, SUITE '11, pages 33–36, New York, NY, USA, 2011. ACM.

[4] Bojana Bislimovska, Alessandro Bozzon, Marco Brambilla, and Piero Fraternali. Content-based search of model repositories with graph matching techniques. In *Proceeding of the 3rd international workshop on Search-driven development: users, infrastructure, tools, and evaluation*, SUITE '11, pages 5–8, New York, NY, USA, 2011. ACM.

[5] Rosalva E. Gallardo-Valencia and Susan Elliott Sim. What kinds of development problems can be solved by searching the web?: a field study. In *Proceeding of the 3rd international workshop on Search-driven development: users, infrastructure, tools, and evaluation*, SUITE '11, pages 41–44, New York, NY, USA, 2011. ACM.

[6] Watanabe Takuya and Hidehiko Masuhara. A spontaneous code recommendation tool based on associative search. In *Proceeding of the 3rd international workshop on Search-driven development: users, infrastructure, tools, and evaluation*, SUITE '11, pages 17–20, New York, NY, USA, 2011. ACM.

[7] Lars Heinemann and Benjamin Hummel. Recommending api methods based on identifier contexts. In *Proceeding of the 3rd international*

workshop on Search-driven development: users, infrastructure, tools, and evaluation, SUITE '11, pages 1–4, New York, NY, USA, 2011. ACM.

[8] Sumit Bhatia, Suppawong Tuarob, Prasenjit Mitra, and C. Lee Giles. An algorithm search engine for software developers. In *Proceeding of the 3rd international workshop on Search-driven development: users, infrastructure, tools, and evaluation*, SUITE '11, pages 13–16, New York, NY, USA, 2011. ACM.

[9] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Sourcerer: An internet-scale software repository. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, SUITE '09, pages 1–4, Washington, DC, USA, 2009. IEEE Computer Society.

[10] Mark Grechanik, Chen Fu, Qing Xie, Collin McMillan, Denys Poshyvanyk, and Chad M. Cumby. Exemplar: Executable examples archive. In *ICSE (2)*, pages 259–262, 2010.

[11] Vinicius Cardoso Garcia, Daniel LucrÃ©dio, Frederico Araujo DurÃ£o, Eduardo Cruz Reis Santos, Eduardo Santana de Almeida, Renata Pontin de Mattos Fortes, and Silvio Romero de Lemos Meira. From specification to experimentation: A software component search engine architecture. In Ian Gorton, George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clemens A. Szyperski, and Kurt C. Wallnau, editors, *CBSE*, volume 4063 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2006.

[12] Yunwen Ye and Gerhard Fischer. Supporting reuse by delivering task-relevant and personalized information. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 513–523, New York, NY, USA, 2002. ACM.

[13] Katsuro Inoue, Reishi Yokomori, Tetsuo Yamamoto, Makoto Matsushita, and Shinji Kusumoto. Ranking significance of software components based on use relations. *IEEE Trans. Softw. Eng.*, 31:213–225, March 2005.

[14] Qihong Shao, Peng Sun, and Yi Chen. Wise: A workflow information search engine. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 1491–1494, Washington, DC, USA, 2009. IEEE Computer Society.

[15] M Fernandez, I Cantador, and P Castells. *CORE: A Tool for Collaborative Ontology Reuse and Evaluation*. 2006.

[16] Shingo Takada. Finding web services via bpel fragment search. In *Proceeding of the 3rd international workshop on Search-driven development: users, infrastructure, tools, and evaluation*, SUITE '11, pages 9–12, New York, NY, USA, 2011. ACM.

[17] Xin Dong, Alon Halevy, Jayant Madhavan, Ema Nemes, and Jun Zhang. Similarity search for web services. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 372–383. VLDB Endowment, 2004.

[18] Alessandro Bozzon, Marco Brambilla, and Piero Fraternali. Searching repositories of web application models. In *Proceedings of the 10th international conference on Web engineering*, ICWE'10, pages 1–15, Berlin, Heidelberg, 2010. Springer-Verlag.

[19] Iman Keivanloo, Christopher Forbes, Juergen Rilling, and Philippe Charland. Towards sharing source code facts using linked data. In *Proceeding of the 3rd international workshop on Search-driven development: users, infrastructure, tools, and evaluation*, SUITE '11, pages 25–28, New York, NY, USA, 2011. ACM.

[20] Daniel Lucrédio, Renata P. M. Fortes, and Jon Whittle. Moogle: A model search engine. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, MoDELS '08, pages 296–310, Berlin, Heidelberg, 2008. Springer-Verlag.

[21] Bojana Bislimovska, Alessandro Bozzon, Marco Brambilla, and Piero Fraternali. Graph-based search over web application model repositories. In Soren Auer, Oscar DaÂaz, and George A. Papadopoulos, editors, *ICWE*, volume 6757 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2011.

[22] Hai Zhuge. A process matching approach for flexible workflow process reuse. *Information and Software Technology*, 44(8):445 – 450, 2002.

[23] Elaine Nowick, Kent M. Eskridge, Daryl A. Travnicek, Xingchun Chen, and Jun Li. A Model Search Engine Based on Cluster Analysis of Search Terms. *Library Philosophy and Practice*, 7(2), 2005.

[24] Ahmed Awad, Artem Polyvyanyy, and Mathias Weske. Semantic querying of business process models. In *Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 85–94, Washington, DC, USA, 2008. IEEE Computer Society.

[25] Catriel Beeri, Anat Eyal, Simon Kamenkovich, and Tova Milo. Querying business processes. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 343–354. VLDB Endowment, 2006.

[26] Tanveer Syeda-Mahmood, Gauri Shah, Rama Akkiraju, Anca-Andrea Ivan, and Richard Goodwin. Searching service repositories by combining semantic and ontological matching. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '05, pages 13–20, Washington, DC, USA, 2005. IEEE Computer Society.

[27] Paulo Gomes, Francisco C. Pereira, Paulo Paiva, Nuno Seco, Paulo Carreiro, José L. Ferreira, and Carlos Bento. Using wordnet for case-based retrieval of uml models. *AI Commun.*, 17:13–23, January 2004.

[28] Daniel Bildhauer, Tassilo Horn, and Jurgen Ebert. Similarity-driven software reuse. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, CVSM '09, pages 31–36, Washington, DC, USA, 2009. IEEE Computer Society.

[29] Gerald. Kowalski. *Information retrieval architecture and algorithms*. Springer, New York [u.a.], 2011.

[30] Colin Atkinson and Thomas Kühne. Model-driven development: A metamodeling foundation. *IEEE Softw.*, 20:36–41, September 2003.

[31] Jean Bezivin. On the unification power of models. *Software and System Modeling*, pages 171–188, 2005.

[32] Stefano Ceri, Piero Fraternali, Aldo Bongio, Marco Brambilla, Sara Comai, and Maristella Matera. *Designing Data-Intensive Web Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[33] David Smiley and Eric Pugh. *Solr 1.4 Enterprise Search Server*. Packt Publishing, 2009.

[34] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20:422–446, October 2002.

[35] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schtze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.