POLITECNICO DI MILANO
Facoltà di Ingegneria dell'Informazione

POLO REGIONALE DI COMO

Content-based Search of Model Repositories with Graph Matching Techniques

Supervisor: Prof. Piero Fraternali
Assistant Supervisor: Alessandro Bozzon

Master Graduation Thesis by: Bojana Bislimovska
Student Id. number: 722416

Academic Year 2009/2010

II

Abstract

Project repositories are important sources of knowledge, and their reuse and sharing is essential for improving the process of software design. Repositories contain all the data and documents for a software project, but they also contain business process models and application models. Such models explain in details the data structure, behavior, and component relationships of an application. They have a determined structure specified by the rules of the language. Given the importance of models in a software development and maintenance life-cycle, there is a need of enabling users to query model repositories. Through search engines, models can be reused according to a user's need, helping to achieve reduction of modeling time and costs.

A possible solution for such a need is the adoption of search engines exploiting model similarity match techniques for content-based search, which may allow more expressive user queries, where the information need also considers the model structure. In this thesis, the problem of content-based search for a repository of models is investigated. The scope of the thesis is to provide thorough study of the current techniques and tools for project repository search, and to propose a general framework for content based-search of models. The framework uses querying on graphs, performing similarity search based on graph matching. Furthermore, a prototype implementation for a Domain Specific Language search engine is realized, and its performance is evaluated.

Sommario

Gli archivi di progetti software sono un'importante sorgente di conoscenza; la loro condivisione e il loro riuso sono essenziali per migliorare la fase di progettazione software. Questi archivi contengono tutti i documenti e i dati di un progetto software, i modelli di business e i modelli delle applicazioni. Questi ultimi modelli descrivono nel dettaglio la struttura dati, il comportamento e la relazione fra componenti di una applicazione, hanno inoltre una struttura specifica dipendente dalle regole del linguaggio utilizzato e un vocabolario limitato e controllato. Data l'importanza dei modelli nel ciclo di sviluppo e mantenimento del software, è necessario fornire a progettisti e sviluppatori strumenti per l'esplorazione e la ricerca all'interno di archivi di progetti software.

Una possibile soluzione a questo problema è rappresentato dai motori di ricerca content-based, ovvero motori di ricerca che permettono l'utilizzo di contenuti come query. Questo tipo di interrogazione è in grado di esprimere un bisogno informativo in modo molto più vicino ai bisogni dell'utente, in quanto prende in considerazione la struttura del modello. In questa tesi viene analizzato il problema della ricerca content-based di archivi di modelli. Lo scopo di questa tesi è quello di fornire un approfondito studio dello stato dell'arte delle tecniche e degli strumenti utilizzati per la ricerca sui archivi di modelli e di proporre un framework generico, indipendente dal linguaggio di modellazione, per la ricerca in base al contenuto sui modelli. Il framework è in grado di eseguire interrogazioni sui grafi eseguendo una ricerca basata sulla similarità. È stato inoltre implementato un prototipo di motore di ricerca per un linguaggio di modellazione per applicazioni Web, e ne sono state valutate le prestazioni.

Acknowledgments

It is a great pleasure for me to thank the many people who made this thesis possible.

First, I would like to thank Professor Piero Fraternali , who gave me the opportunity to work with him and his team. I really appreciate all his guidance and strong continiuous support. He always took the time to answer my questions and to show me the right directions for my actual and future scientific work.

I would like to thank Alessandro for introducing me to the world of model repositories and the world of science, and for his great help with his ideas in all the stages of my work. I would like to thank him for the constant motivation, his patience and his understanding.

I would also like to thank Marco Brambilla for his useful ideas, advices and consistent interest in my project.

A big thanks to Chiara, for helping me with the abstract translation, and to Volo, for his original sense of humour, and to both of you guys, for your company during the numerous lunch breaks.

Thanks to the Riciclò team (Fabio, Elisa, Chiara, Anto, Filippo, Giovanni, Riccardo) for showing that a difficult job can become interesting when you turn it into a recycle game.

Thanks to the small, but powerful Macedonian Como team, with whom I shared many moments together. Thank you Magde for being there for me since my first Como days until now. Thank you Elena for keeping the Macedonian spirit and for your willingness to help.

Thanks to the Venini people, good neighbors, for the memorable times spent in cooking, watching movies, and football games, days without heating and hot water, hunting scorpions, making dinners and parties. Thank you Leo for your excellent cooking, and for always helping me with my lugagge. Thank you Rafa for taking care of me while I was ill, and thank you Pedro for borrowing me your computer when mine was broken.

Thanks to my Macedonian friends Aneta, Kate, Viki, Marija for being there for me, melting the distance in kilometres, for our endless

conversations, and great girls nights out.

Thanks to my dear sister Dragana, you are my source of hapiness and joy in all the hard moments. Thank you for always having the patience to listen to me. Thank you for tollerating the books on the table, and the clothes in the closet for all these years. You are the greatest!

Thanks to my uncle Jaroslav for all the beautiful things you gave me in my life.

Thanks to the entire familly Bislimovski for their support.

Thanks to my grandparents Vera i Dragoljub (baba Lepa i deda Buda), who unfortunately are not here anymore. I would like to thank these great people for their devotion, love, support and everything they did for me. I would like to thank them for everything they taught me. I will always admire their optimisim, spirit and passion in living the life.

Last, I want to thank my parents, Jovanka and Vojislav,without you I would have never been able to achieve so much. Thank you for your endless love, care and understanding.Thank you for your sensibility and your wise advices. Thank you for teaching me that the best thing one can have is a freedom of choice. Thank you for giving me that freedom.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As it becomes more common for organizations to store collections of software artifacts, project repositories are growing in size, and are becoming more frequent. Project repositories maintain the consistency of projects, and manage different project versions. They contain knowledge and best practices from past activities, which can be reused and shared for improving and simplifying the design process. This can be achieved by querying project repositories and retrieving projects, through a web search engine. Project retrieval allows better management and software development by reusing the past experience. It can be also used for data integration, enabling compatibility and collaboration across companies.

From the variety of search engines for querying repositories , a large number of them are source code repositories supporting source code search which increases the speed of code implementation. Some source code search engines, like for example Codase, Koders, Google Code Search, are publicly available as a result of the dispersion of the open source software. Research works use Information Retrieval and program analysis techniques for finding relevant source code from repositories, like Sourcerer [5], a framework that offers large scale indexing of open source code, or Exemplar [25]. Other approaches address the problem of model similarity where the query is a model or part of a model, and besides the textual similarity, consider the model structure, its hierarchies and relationships,and in some cases the semantic similarity. This enables results which are closer and more similar to the query. These apporaches apply techniques such as schema matching, graph matching, or specific algorithms for finding similarities between models.

1.1 Model Search Context

Advances in languages and platforms in the last two decades have increased the complexity and the level of software abstraction [52]. In order to allow integration and compatibility between different systems, it is necessary to represent them as models, formalizing the structure, behaviour and requirements through use of a specific notation. This way, improvement and simplification of the design process is achieved.

Model based search engines use models for indexing and querying model repositories.These models can be Domain Specific Language (DSL) models, UML models, providing a notation for object-oriented analysis and design, or BPM (Business Process Modeling), representing business process flows and web service orchestrations. Model search enignes can be designed for a specific model type, or can be model-independent, using metamodel information for the search engine configuration. General feature of the model search engines, is that they use metamodel or model type characteristics to build the index structure. The index can be queried through keywords or models. An example of keyword based model search engine is Moogle [37], which allows use of specific element attributes in the search applied to UML models.

On the other hand, content-based search enables querying model repositories by using models as queries. This way, results that closely resemble the query are obtained. The model (query) must be processed before being matched to the index, considering the model structure and the hierarchies among elements. Furthermore, the query can be expanded by introducing semantic similarity, providing even more precise retrieval of models.

Graphs allow universal representation of models, maintaining the model structure and content. Through their comparison it can be determined how well the models match. Model comparison, where models are represented as graphs, can be applied for content-based search of a model repository. This actually means finding similarities between a query and the models from the repository, and it can be performed using graph matching. Graph matching consists of calculation of the graph edit distance which is NP-complete problem. Therefore, heuristic methods have to be applied, making a trade-off between computational complexity and precision.There are a variety of applications implementing graph matching on workflow processes, business process models and graph databases, using different heuristics as a trade-off between computational complexity and precision. For example, TALE [58] uses subgraph matching for

querying graph databases with large graph queries. The graph edit distance as similarity metric is also used for finding similarities in business process model repositories [33] [50] .

Other apporaches for content-based search rely on schema matching, which finds semantic relationships between elements of two schemas. Furthermore, schemas have uniform internal representation upon which the matching is performed, and can be used for finding mappings betwen different models, as long as they keep the same internal representation. One example is Schemr [13], a schema search engine for querying schema repositories. It uses keyword search and query by example over database schemas, and visualizes a list of schemas as a result.

There are some approaches in the field of graph and semi-structured document query languages dealing with representation of queries as graphs. For example, XML-GL is a graphical query language for XML data [14] with semantics based on graph matching. This way, querying can be made visual as the process of locating a subgraph in a graph. BP-QL[6] is a query language for business processes, that uses the visual representation of business processes as directed labeled graphs.

1.2 Contribution

The goal of this thesis is twofold: on one-side, the thesis aims to provide a thorough overview of the current state-of-the-art in project repository search; on the other hand, the thesis proposes a general-purpose approach using query on graph (by example) for searching repository of models. Most models can be represented as graphs, and querying over a repository actually becomes querying on graphs. This is realized by performing similarity search using graph matching, or calculation of the graph edit distance. At the back-end, models from the repository are transformed from their native language into annotated graphs, considering the model structure. An indexer uses these graphs to build the search index structures offline, by considering the relationships among graph nodes, element names, and types. Queries are processed on these index structures, and matches are ranked for relevance.

To summarize, the contribution of this thesis includes:

- A thorough state-of-the-art analysis of methods and tools for project repository search (Chapter 2);
- A general framework (Chapter 3) for content-based search in

model repositories. The framework provides:

  - Application of a model-independent approach that can be applied to different kind of models. After the models from the repository and the query are being processed, they are transformed into graphs. The graph matching is performed on graphs, independently from the model type.
  - Construction of a structured index reflecting the hierarchies and relationships among model elements. This way, more accurate index is built;
  - Introduction of a different similarity measure, that exploits the semantic relationships between model element types, extracted from the metamodel;

- Implementation and evaluation of the proposed framework using projects and queries encoded in a Domain Specific Language,WebML. The evaluation includes the manual assessment of groundtruth for query-project relevance assessment. The groundtruth has been used as a reference to compare the performance of the system.

1.3 Thesis Organization

The thesis is organized as follows:

- **Chapter 2** presents the state of the art for repository search engines and the WebML (Web Modeling Language) a Domain Specific Language used in the implementation;
- **Chapter 3** describes the content-based search and the algorithm applied in the framework;
- **Chapter 4** illustrates the implementation and experiments performed on a Domain Specific solution, their design and the obtained results;
- **Chapter 5** brings out the conclusions and the future work directions.

Chapter 2

Related work

This chapter describes current academic works for searching artifacts from software repositories. Section 2.1 provides an overview on the current approaches for repository search engines. Section 2.2 focuses on content based search approaches, describing *schema matching* and *graph matching* techniques. To conclude the chapter, Section 2.3 introduces WebML, the Domain Specific Language for Web application modeling that we adopted to conduct the quantitative evaluation provided in Chapter 4.

2.1 Repository Search Engines

As the number of software repositories grows, the problem of searching existing relevant software artifacts contained in the repositories becomes more relevant. The knowledge incorporated inside repositories should be reused to support better software development. Therefore, the search of desired software artifacts using information retrieval techniques has been extensively investigated.

Repository search engines can be classified according to the offered *query type*, the type of *indexing*, the *algorithms* adopted for result matching and ranking, and the way in which results are *presented*. Existing works can also be classified according to the content type addressed by the search system. The evaluation proposed in the following sections identifies three kind of content types: *source code*, *models*, or *business processes*.

2.1.1 Source Code search

Reusing fragments of existing applications brings advantage to programmers by increasing the development speed. There exist open source tools for code search and retrieval. Several of them introduce

information retrieval features for open source code search, by narrowing the information space by indexing code files, e.g., Koders, Codase and Krugle. In the most general case, natural language keyword queries are used to retrieve the results as code snippets. Some of them allow query refinement by means of regular expressions, wild cards, or filters based on syntactic elements such as class, methods, variables; others enable query expansion by expanding the query using concepts with similar meanings (CodeBroker).

Another interesting feature is the adopted ranking function; some of them do not provide ranking; others provide ranking based on the TF/IDF [40] measure, or composite scores that consider project properties, such as number of matches in the source code, recency of the project, number of downloads etc. (SourceForge); some approaches leverage the organization of the underlying code to provide a component-based ranking model, where classes more frequently used by other classes are ranked higher.

Result presentation features include code snippets, modules representation (CodeBroker) and applications (SourceForge), allowing the programmer to better understand the context of a specific result.

We now provide some representative examples of research works in the field of source code search engines that use information retrieval for finding relevant source code.

Sourcerer [5] is an infrastructure for large scale indexing and analysis of open-source code, upon which code search engines and services can be built, such as repository access of all the artifacts, libraries and metadata, and the slicing service (dependency slices of any entity and all its entities on which it depends, from the repository). Sourcerer crawls the internet looking for Java source code from public web sites , open source repositories and version control systems. In Sourcerer, the code is parsed, analyzed and stored in three forms: managed repository, containing versioned copy of the original project content; Code Database stores models of parsed projects, based on the metamodel, and Code Index, stores keywords extracted from the code. The metamodel for structural information is based on the Chen's entity relationship metamodel. A project model element exists for every project (collection of Java source files) in the repository, and every unique jar file. These files are linked to entities and relations between entities. Each entity contains unique id and it is annotated with referring java modifiers, its location, and the project and the file it came from. Most of the structural information is in the relations. A relation is identified by its project, type, and ids of its source and target. All jar files are also uniquely identified, extracted and placed in the database.

Sourcerer uses Lucene for storing indexes.

Maracatu [23] is a search engine for retrieving source code components from (CVS) development repositories. The search engine indexes Java source code components with the Lucene search engine. It combines text mining and facet-based search which gives better solution with respect to the ones obtained when this mechanisms were used alone. First, a filtering precedes the search to exclude components which do not satisfy the constraints, and then the keyword search is performed. A visualization of the retrieved component is allowed before its download.

Exemplar (EXEcutable exaMPLes ARchive) [25] is an approach for finding highly relevant software projects from large archives of applications. It uses information retrieval and program analysis techniques to retrieve applications by linking high-level concepts to the source code of applications via standard third-party Application Programming Interface (API) calls used by the applications. Despite finding the query keyword matches in the descriptions and source code of applications, they are also matched with words in the help documentation for API calls. The matches of the API calls are matched against the name of the functions invoked in the applications. The ranking is obtained by considering the word occurrences, number of relevant API calls and the data flow connections between them.

CodeBroker [61] is a system that uses techniques that allows to autonomously locate components in a repository which are task-relevant and personalized to the background knowledge of the developer (information delivery). The main inspiration comes from the fact that reuse is often unsuccessful because of users' lack of knowledge and their inability to create a well-defined query. CodeBroker utilizes user models to represent their knowledge about the repository. Information delivery reduces the cost of software reuse, makes unanticipated (unknown) components easily accessible, and motivates developers to change the design approach towards reuse.

The work from *Antoniol et al.* [3] proposes a method based on Vector Space Model to find traceability links between source code and free text documentation expressed in natural language. The method has been applied in two case studies and results are assessed by using the most standard IR metrics, precision and recall. The Vector Space Model has been compared to the Probabilisitic Model obtained in a previous study, and it has been proven that both models perform equally well.

SPARS-J is a software component search system which treats source files of Java classes as components [29]. The collection of

software components in the component rank model are represented as weighted directed graphs. The resulting rank (the component rank) allows for highly ranked components to be quickly seen by the user.The results show that a class frequently invoked by other classes has a high rank, with respect to nonstandard classes.

2.1.2 Model search

While source code search engines leverage the grammar of programming language, model based search engines exploit the underlying model structures for indexing and querying repositories. Besides the traditional keyword querying, a query by example (content-based search) can be used for finding results that more closely reflects the user need. The content-based search approaches are given in Section 2.2.

Moogle is a model search engine that uses UML or Domain Specific Language (DSL) metamodels to create indexes for evaluation of complex queries [37]. It also formats the search results in a more readable way by removing irrelevant tags and characters. 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 (advanced search), and by using filters organized into facets, containing values that can be combined in the search (browsing). Moogle uses the SOLR ranking policy of the results. The results are formatted allowing more important info to be highlighted for the user which is more clear with respect to the original XMI format. However, the result preview is not very expressive and requires knowledge of the model content in order to understand the result. An approach for querying UML models using the detailed semantics of the UML and OCL is presented in [2]. The formed queries are as powerful and concise as the queries formed with the help of relational algebra. Therefore, some OCL extensions are added, like for e.g., new operation definitions (project and product,) and concepts, to avoid adding additional model elements.

A model driven information retrieval system that uses classical information retrieval techniques for obtaining information from WebML metamodels in a project repository is proposed in [9]. The realized prototype applies metamodel-aware extraction rules to analyze models. It has a visual interface to perform keyword based queries, performed on whole projects, subprojects, or concepts, and inspect results, presented as a paginated list of matching items with a possibility of snippet visualization. The index is populated with

information extracted from the models. The work of this thesis is built upon this research work, exploring the direction of content-based search of models.

2.1.3 Business Process Models Search

Nowick et al. [47] introduce a model search engine that applies user-centric and dynamic classification scheme to cluster user search terms. User search terms are collected from the log file analysis of a specific website and they are clustered, rather than document keywords, based on their frequency of occurrences and the distance measures (4 types of distance measures are applied). In case of a failed search, when the search engine returns more than 100 or less than 1 result, the advanced smart search is suggested, to narrow/improve the search with the additional terms from the same cluster of the original term. In case of narrow search, the terms from other clusters are suggested.

CORE [22], a tool for Collaborative Ontology Reuse and Evaluation receives an informal description of a semantic domain as a set of terms, and determines which ontologies from an ontology repository most appropriately describe the given domain by automatic use of similarity measures. A set of terms are manually assigned, and the user can expand these terms by using WordNet. The user selects a subset of available comparison techniques, and as a result, a ranked list of ontologies is retrieved for each criterion. A global aggregated measure is used to define a unique ranking , which uses rank fusion techniques.When human judgement is required, Collaborative Filtering Approach is applied allowing manual user evaluation.

WISE [53] is a Workflow Information Search Engine which allows querying a repository of workflow hierarchies using simple keywords. Query results, displayed graphically, are defined as a minimal view of each relevant workflow hierarchy containing matches to all query keywords.

2.2 Content Based Search

Content-based search allows querying over model repositories by using models as queries . Such a query mode allows for more precise information retrieval as the relationships between elements are also taken into account, sometimes also considering the semantic similarity.

This section presents the problem of graph matching and schema

matching, providing some examples that demonstrate their use in the similarity search. Moreover, some visual languages for querying the model content are given. Finally, alternative content-based approaches using specific algorithms are described.

2.2.1 Graph Matching

Graphs provide a convenient way for representing data in many different application domains such as computer vision, data mining and information retrieval. To fully exploit the information encoded in graphs, effective and efficient tools are needed for their querying.

The procedure of finding a mapping between two graphs, the model (query) graph and the data graph, for comparison of their similarity leads to the problem of *graph matching*. Finding a complete node-to-node correspondence without violating both structure and label constraints of a graph, is the problem of *graph isomorphism*. Closely related to graph isomorphism is *subgraph isomorphism*, which can be seen as a concept describing subgraph equality. A subgraph isomorphism is a weaker form of matching in terms of requiring only that an isomorphism holds between a query graph and a subgraph of the data graph. The problem of graph matching is NP-complete, imposing theoretical and practical limitations, and there is no best solution.

In [50], four graph matching algorithms for computation of similarity of business process models represented as graphs are evaluated.
The greedy algorithm marks all possible graph nodes as open pairs, and, in each iteration, selects an open pair that maximizes the similarity induced by the mapping, and adds the pair to the mapping. The algorithm iterates until there is no open pair left. Main disadvantage of this algorithm is the suboptimal mapping as a result of choosing an open pair which can be local maximum, discarding pairs that might increase similarity in later stages.

Exhaustive algorithm with pruning evaluates recursively all possible mappings, and when the recursive tree reaches a size of $pruneAT$ (an algorithm parameter), the algorithm prunes it, keeping only the mappings with highest similarity, whose number depends on the $pruneTO$ parameter. The number of mappings increases exponentially, but it is controlled with the pruning parameters.

Process heuristic algorithm is a variation of the exhaustive algorithm and it is based on the assumption that nodes closer to the start of a model should be mapped to nodes closer to start of the

other model. This leads to higher-quality pruning and the mappings do not increase as rapidly as in the previous algorithm.
A-star algorithm takes the partial mapping map with the maximum graph edit similarity in each iteration. Every node from the first graph is paired with every node from the second graph that does not appear already in the mapping. Additionally, it is created a mapping considering the case when the node from the first graph is deleted. The algorithm finishes when all nodes from the first graph are mapped.The memory requirements of the algorithm can be reduced by considering possibility of node mapping only if the textual similarity between node labels is greater then a determined cut-off value. The A-star algorithm used in this approach is described in [43], as an A-star algorithm for error-correcting subgraph isomorphism detection between two graphs. Here, it has been adapted for business process models. As a result of the evaluation, A-star has slightly better average precision compared to the other algorithms, while greedy algorithm has the fastest execution time, as expected.

This thesis performs content-based search on models, by adopting a graph matching method, using the *A-star algorithm* described in [50], for retrieving similar projects from a repository.

The graph matching problem can also be casted into the *simulated annealing* algorithm [28]. The simulated annealing introduced in [32], starts with an initial value and performs random search in the search space. Optimization is achieved by applying small changes to the current solution. If in this way the solution is improved, the change is accepted. If the solution decreases by some value d there is a probability$P(d, T)$ that it will be accepted. $P(d, T)$ has a high value for small values of d and high values of the temperature T. Temperature decreases with time until it reaches zero. As a consequence, $P(d, T)$ becomes zero as well, and only improvements are accepted.

GRASP (Greedy Randomzied Adaptive Search Procedure) is a meta-heuristic algorithm that develops in two phases [21]. In the first phase, it constructs a Restricted Candidate List (RCL) of elements from a graph, whose cardinality depends on a threshold parameter. Then, from this list, it selects one element randomly, which might not necessarily be the best match, and proceeds with the graph comparison. In the second phase, it performs a local search for improving the current solution by element swapping. After arbitrary number of iterations, the search stops and the best overall solution is accepted as final result.

Extended subgraph isomorphism for data mining is applied in [10]. The proposed approach matches approximate queries over

databases of graphs, where the query is also a graph which may contain *don't care symbols* (match any attribute value in a corresponding position), variables, and they may express constraints on values. Variables may be answer variables indicating which attribute values are to be returned as a query answer, or they can express constraints on the attribute values. The match result must include every edge from the query, and the nodes are mapped only if their types correspond, satisfying all the query constraints.The algorithm terminates as soon as the first match is found.

TALE (Tool for Approximate Subgraph Matching of Large Queries Efficiently) is a general tool for approximate subgraph matching of large graph queries discussed in [58]. It queries graph databases and uses novel indexing method considering the neighbors of each database node, thus, capturing the local structure around each node. A database node matches a query node, only if the two nodes match and their neighborhoods also match. The algorithm consists of determining important nodes in the query and probing them against the index,thus finding the best matching node pair. The degree centrality measure establishes the node importance, such that, nodes with high degrees are more important than low degree nodes. The match is expanded through the neighboring nodes of the matched nodes until no more nodes can be added to the match.

The potential of workflow discovery using graph matching is investigated in [24]. All the workflows, as well as the workflow which is the user input, are parsed into a form the Graph Matcher understands. The Graph Matcher detects which workflows are similar to the input and returns the result formatted by the Formatter into HTML page.Then, the tool is tested within the workflow environment on a real corpus. The tool replicates the average human rankings of workflow fragments. However, a human gold-standard needs to be created for workflow rankings, and the tool needs to be re-evaluated.

Closure-tree [27] is an index structure of graph queries. It organizes graphs hierarchically where each node summarizes its descendants by a Graph closure in order to enable effective pruning, and children of leaf nodes are database graphs. Graph closure has graph characteristics, except that instead of singleton labels on vertices and edges, multiple labels are allowed. Closure-tree supports both subgraph queries and similarity queries. The first type of queries find graph that contains subgraph using pseudo subgraph isomorphism, while with the latter type, a graph edit similarity is measured through graph edit distance using heuristic methods for graph mapping: state search, bipartite method or neighbor biased matching.

The evaluation of this indexing technique is performed on a synthetic dataset generated by synthetic graph generator, as well as on a chemical compounds dataset

An example of graph matching on process models is presented with the framework for process modeling and deployment [36]. It consists of a constraint-based process modeling approach, Business Process Constraint Network and a repository for case specific process models, called a process variant repository (PVR). This framework provides an effective approach for structuring and querying PVR. The query is a (sub)graph, expressing information needs, formulated with respect to the criteria for selection of one or more process variants. The query can be similar to a structural definition of a variant, but may not be identical. Therefore, a selective-reduce method is proposed, which uses graph reduction techniques for finding a match. The result is a collection of variants matching the criteria which can be ranked in case of partial matching.

An inexact process matching approach that enables two workflow processes to be matched with a similarity degree [0,1] is brought out in [62]. The similarity is obtained from the matching degrees of the corresponding sub-processes and activities of the processes, and is characterized through definition of process specialization relationship and activity specialization relationship. The matching degree between two activities depends on the longest activity-distance between them on an activity specialization graph (ASG) defined by the activity-ontology repository. In each ASG, nodes are activities, while arcs represent activity specialization relationships.

An indexing approach for business process models based on metric trees (M-Trees), and a graph edit distance as similarity metric is given in [33]. The index is a hierarchical search structure that partitions the search space by using the distance between characteristic feature values of objects, and saves comparison operations during search with the help of a distance function by excluding the partitions from further exhaustive search. Search in metric spaces takes a query process model and a query radius which describes the acceptable distance of a matched process model compared with the query. The triangle inequality allows pruning subtrees without calculating the distance between the query and the pivots of the subtrees.

Coogle [51] is a search engine that extends the idea of using similarity measures for determining the similarity of Java classes. It is based on tree similarity algorithms: bottom-up and top-down maximum common subtree isomorphism, and tree edit distance. Coogle has been implemented as an Eclipse plug-in. The effectiveness of the

proposed algorithm is evaluated using test-cases, and a Java project, finding similarities between classes of the same project version, as well as when performing comparison of different project versions. The tree edit distance produces the best results.

2.2.2 Schema Matching

Schema matching is the problem of finding semantic correspondences between elements of two schemas[38] [45] [35]. It is a critical operation in many application domains such as semantic web, schema/ontology integration, data integration, E-business, data warehousing [49].

The implementation of schema matching is based on several match algorithms called matchers. The matchers, according to [49][38], are classified as instance and schema matching approaches, element and structure matching approaches, and language and constraint approaches.

Each of the approaches uses different information and has diverse applicability for a specific match. A matcher using multiple match algorithms achieves better match results, than matcher using single approach. Therefore, several match algorithms may be combined in two ways, by using: hybrid matcher and composite matcher. Hybrid matchers use the individual matchers executed simultaneously and in fixed order, improving the effectiveness of matching by early filtering out poor match candidates that satisfy only one matching criteria.On the other hand, composite matchers use several match algorithms ordered to be executed sequentially or simultaneously, making it more flexible then the hybrid matcher.

Many systems and approaches implementing schema matching (semi) automatically have been developed recently, and here are introduced some of them.

Cupid is a hybrid schema matcher applied to XML and relational examples. It is a generic matcher which represents schemas as graphs with nodes that correspond to the schema elements [38], and it is directed towards similarity of atomic elements,where it captures much of the schema semantics. For linguistic matching, a domain-specific thesaurus is used for calculating synonyms, acronyms and short-forms (name similarity), classifying elements into categories and comparing only elements from compatible categories. For calculating the element similarity, the similarity of the neighbors is considered.

TranScm[45] is a hybrid approach for data translation between two schemas represented internally as graphs. First, a schema match-

ing is performed, and the obtained matching is used to translate instances of the first schema to the instances of the second schema. The translation is based on a set of rules (matchers) used for component matching.

Another example of a hybrid match application is *SemInt* [35], a tool that allows identification of corresponding attributes in heterogenous databases with the help of neural networks. For every attribute a signature value is assigned for every matching criterium. These signatures are used for determining similar attributes from the first schema and placing them into clusters, and then, for each cluster,finding the best matching cluster in the second schema.

COMA [16] is a composite schema matching system which combines multiple matchers in a flexible way. It provides a library of individual and hybrid matching algorithms and allows their combination for a match problem. During the matching process the user can interact and provide feedback of the match. It is also possible to reuse previous match results. The quality of the match is evaluated based on the comparison of manually obtained matches with the ones obtained by the automatic match processing.

LSD [17], and its successor, *GLUE* [18] are powerful composite matchers. Learning Source Description (LSD) uses machine learning for individual matcher and combination of match results. A meta-learner unifies the results of individual matchers by weighting them with respect to their accuracy during training. Its successor, *GLUE*, performs semantic mapping,i.e. for each concept in an ontology finds the most similar concept in the second ontology, using the fact that data instances are associated with the ontologies.

sPLMap [46] is a schema matching framework which combines ideas from data integration and data exchange, information retrieval and machine learning. It is based on the ideas of *LSD* which combines several classifiers to find the best mapping based on attribute names and properties of underlying data instances. sPLMap introduces some improvements with respect to LSD in the support of data types in the matching process, the usage of probability theory for estimating the degree of correctness of a set of learned rules, and computation of probabilistic rules for capturing the inherent uncertainty of the matching process.

In [20], another composite approach is presented. It is a framework for multifaceted exploitation of metadata, where potential matches from various facets of metadata are combined to generate confidence values on potential attribute matches. For each facet, a vector of measures for the features of interest is calculated, and machine learning is applied to generate a decision rule and mea-

sure their confidence. Facets of data that were investigated are the ones which usually give humans a clue which attributes to match: meanings of attribute names (WordNet is used), similar value characteristics, presence of expected data values.The goal is to integrate multiple source schemes into a target scheme, where as sources are used Web repositories.

Similarity flooding is a graph matching technique which produces accurate results when applied to the problem of schema matching [42]. The main concept of this algorithm is that two nodes or edges from two graphs that are compared, are similar if their adjacent elements are similar. A matrix for similarity propagation is used to construct mapping between node pairs whose similarity is above a given threshold value. This algorithm operates on directed labelled graphs, and does not produce optimal results on unlabeled edged graphs.

A semantic matching approach where semantic correspondences of two graph structures are obtained by calculating the semantic information contained in the labels of nodes and graphs is presented in [19]. The approach is implemented within the *S-Match* system, based on schema-matching, and computes the strongest semantic similarity for each pair of tree nodes. The key idea is that the labels expressed in a natural language must be translated into an internal representation which expresses concepts. Atomic concepts are represented as single words build after tokenization, lemmatization and with the help of WordNet, while complex concepts are obtained by combining the atomic ones. The concepts of nodes are built in the same way. The relations of concepts of labels and concepts of nodes are computed, and in this way, strongest semantic relation can be determined. The quality measure is obtained by precision, recall, overall, time and F-measure.

Schema Matching Search Engines

Schemr is a schema search engine for querying over database schemas stored in a metadata repository, by using keywords and schema fragments in DDL or XSD format, thus enabling schema share and information integration [13]. The query graph (a keyword is an one item graph) is flattened into a list of keywords and retrieves the candidate schemas from the document index built offline. Then, a schema matching is applied for computing the semantic similarity between the candidate schemas and the query graph. A collection of matchers evaluates the query-graph. The name matcher normalizes terms, while the context matcher finds matches based on

the neighboring-element sets similarity. Each matcher produces a similarity score, weighted and used for the computation of the total similarity score. In order to calculate the overall score, the tightness-of-fit measurement is used by combining total-similarity scores and a penalty. The introduced penalty is based on the distance between relevant elements in the result schema with respect to their entities .The results are ranked and can be visualized in several views allowing users to compare which elements match and how well they match.

An integrated environment for exploring schema similarity across multiple resolutions using a schema repository is given in [54]. Users visualize and interact with clusters of related schemas using a clustering tool named Affinity. Users can drill-down within any cluster, examining the extent and content of schema overlap. The content overlap for a pair of matches, is proportional to the number of existing matches determined by schema matching algorithms (name similarity, synonyms). This approach intends to raise awareness of the relations of organizational data and thus, facilitates data integration.

Case-Oriented Design Assistant for Workflow Modeling (CODAW) [39] is a prototype system, built upon the framework for workflow modeling and design. The framework includes a conceptual model of workflow cases, a similarity flooding algorithm for workflow case retrieval and domain-independent AI planning approach for workflow case composition. The similarity flooding algorithm for workflow (SFW) supports retrieval of procedural workflow models, based on inexact matching of graph queries. It identifies local structural similarities between a query and process graph model from a workflow schema by using iterative fix-point computation. The principal idea is that, elements of two distinct graphs are similar, when their adjacent elements are similar. The algorithm propagates the similarity from a node to its neighbors based on the topology of both graphs.

2.2.3 Visual Query Languages

G-Log is a graph based language for database query evaluation, which combines the expressive power of logic, the modeling power of objects, and the representation power of graphs [48]. It is a declarative language, it does not suffer from the copy-elimination problem properties like some database languages and it is computationally complete.The database schemas, instances and rules are represented by directed labeled graphs. G-Log queries over a database are expressed by programs, consisting of a number of rules. The program

can be written as a single set of rules, in a fully declarative manner, or arranging the rules in sequences, using the procedural style. All sentences in first-order-logic can be written in G-Log. The rule is represented as a graph and made of colored patterns. Rules in G-Log always have two schemas associated to them, the source schema, which is the schema of the instances that can be given as input to the rule, and the target schema, which is the schema of the output instances of the rule.

XML-GL is a graphical query language for querying XML data [14]. XML documents have an intuitive hierarchical structure, and if references between elements are considered, the labeled directed graph with suitable graphic conventions, becomes their most natural representation. XML-GL allows visually expressing different types of queries like, selection,aggregation, grouping, arithmetic calculations, union,difference, cartesian product. An XML-GL query consists of two sets of labeled graphs:

- A query *left-hand side* (LHS) which expresses the information of interest to be retrieved;
- A query *right-hand side* (RHS) which expresses the desired structure and content of the output XML document, and it is connected to the LHS;

The results of an XML-GL query are XML documents and they are presented by using the nested relational data model, representing the containment of elements and properties,as well as element linking. Additional expressive power is obtained by allowing complex queries made of multiple graphs. In this case, the complex query is decomposed into its components, and every component is evaluated separately.

A graphical notation for specifying model queries on UML models based on lexical similarity and structural arrangements (indirect relationships) is described in [56]. Joint Point Designation Diagrams are used to represent queries graphically, simplifying the complex and excessive textual notation. Object Constraint Language (OCL) meta operations,appended to the UML meta model's classes, are deployed in order to retrieve an actual set of model matching elements. Queries should be specified in terms of user model entities and properties for user's comprehensibility.

BP-QL [6] is a language for querying business processes, based on the intuitive model of BPs, an abstraction of the Business Process Execution Language (BPEL) specification. It contains a graphical

user interface allowing simple formulation of queries over a model.It allows retrieving paths and querying over different levels of granularity, as well as, controlling distributed querying. BPs are visually represented as directed labeled graphs.For querying BPs, BP patterns are offered, and flow paths are retrieved as answers.

2.2.4 Other content-based Search Approaches

Business Process Model

An approach for finding similarity between business process models is introduced in [4] It uses the *BPMN-Q* query language expansion, allowing users to make structure related model queries to retrieve models from a repository. For further expansion of the BPMN-Q queries, the enhanced *Topic-based Vector Space Model* is applied (eTVSM). eTVSM is a vector space model that exploits semantic document similarities through the WordNet knowledge. In this way, an ontology is constructed from the repository and the BPMN-Q query is expanded by constructing other queries based on the substitution of the seed query activities with similar ones. Each expanded query is used for retrieving relevant process models from a repository.

[60] illustrates a method where the calculation of the degree of similarity of business process models is constructed with the Event-driven Process Chains (EPCs) language, considering their linguistic and behavioral aspects .The essential behavioral constraints imposed by a process model are captured by the causal footprint, thus avoiding to calculate the model state space. The similarity is computed by using the vector space model. The match between functions from different EPCs are determined by their semantic similarity score as well as the semantic similarity score of surrounding events.

In [41], a framework for querying in the business process modeling phase, the most important phase of the business process engineering chain is presented. It allows querying a library of business process artifacts enabling their reuse, support of the decision making, and querying of the model guidelines. The querying is based on Business Process Ontology (BPO), the same ontological model for process description. The query can be static,with constraints related to the process's static view, and graphical, with requirements from the dynamic view of the process. Flexibility in querying is enabled by allowing users to specify further constraints or eliminate some constraints with respect to their needs.

The problem of discovering business processes by using abstract business processes (ABP), encoded into annotations which semantically describe business processes is described in [7]. Abstract business processes represent class of equivalent business processes, sharing the same set of activities and flow structure. Business processes are semantically annotated by using business process ontology which is created and populated automatically. The query takes the form of an abstract business process, designed by selecting concepts from the task ontology and connecting them using a control flow graph. Ontologies allow more refined way to identify similarities which are semantically close, increasing the recall and allowing reuse of business process models.

Semantic Web

Corese search engine [15] uses ontology-based approach for web querying, using semantic metadata. The query language is based on RDFs and the queries can also be approximate. Queries are transformed into RDF graphs related to the same RDF schema as one of annotations to which it is going to be matched. The Corese approximation evaluates the semantic distance of classes or properties in the ontology hierarchies. In this way, not only web resources whose annotations are specializations of the query are retrieved, but also those whose annotations have a structure upon which the query can be projected and whose concepts and relations are close to those of the query in the ontology hierarchy. Approximate answers can also be retrieved by using specific RDF properties, and by querying for variable length relation paths between concepts.

iSPARQL [31] is an imprecise query engine containing statistical reasoning elements, based on SPARQL. It simplifies the design and implementation of Semantic Web applications. It has been shown through an experimental data set, the MIT Process Handbook, that the combination of different similarity string learning approaches improves the performance of retrieval. Furthermore, a set of machine learning experiments has been performed to learn a set of similarity measures demonstrating that complementary information contained in different notions of similarity strategies enables high retrieval accuracy. As a result, the broader use of statistical reasoning might improve the overall performance of the semantic web.

SOQA-SimPack toolkit is an ontology language independent Java API that allows generic similarity detection and visualization in ontologies. Different data sources often are not related to the same ontologies when describing their semantics. Therefore, it is impor-

tant to obtain the similarity between ontology concepts for ontology alignment and integration. Similarities are calculated between concepts of different ontologies [63]. The toolkit is founded on the SIRUP Ontology Query API for query access to ontological metadata and data, and SimPack, Java library that implements different similarity measures: vector-based, string-based,full-text similarity, distance-based, and information-theory-based measures. Thus, a variety of notion of similarity can be captured.

UML

ReDSeeDS (Requirements-Driven Software Development System) is a web search engine designed to support reuse of software artifacts based on their requirements. In this way, the new artifacts are compared to the ones stored in the repository. The syntax of the artifacts is described by an Essential MOF (EMOF) compliant meta-model which allows storing the abstract syntax of all artifacts as typed, attributed, directed and ordered graphs (TGraph). The requirements are specified by the Requirements Specification Language (RSL), whose components are requirement statements and use cases [8]. 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. The semantic similarity uses a domain vocabulary and a central terminology containing WordNet with some additions.

An approach for querying UML models using the detailed semantics of the UML and OCL is presented in [2]. The formed queries are as powerful and concise as the queries formed with the help of relational algebra. Therefore, some OCL extensions are added, like for e.g., new operation definitions (project and product,) and concepts, to avoid adding additional model elements.

EMF Compare [59] is an approach for comparing models based on complete comparison of their elements.The proposed framework shows the semantic differences in models represented as trees, by comparing the elements' attributes, and computing their edit distance. Each attribute has a weight depending on the volume of information it contains. Models are compared recursively, and when elements are compared, the algorithm proceeds to their children. The tool provides visualization of model differences, their origin and types.

Web Services

Woogle [19] is a search engine for web services that supports finding similar web service operations and finding operations that compose with a given one, in addition to the simple keyword search. A novel clustering algorithm is used that groups names of parameters of web service operations into semantically meaningful concepts, which are then leveraged to determine input similarity.The similarity is determined by considering textual descriptions of operations and web services, and similarity between the parameter names of operations. The clustering is based on the heuristic, that parameters express the same concept if they occur together often. Woogle allows for template search 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.

Use of domain-independent and domain-specific ontologies for retrieving web service from a repository is proposed in [57]. The domain-independent relationships are derived using an English thesaurus after tokenization and part-of-speech tagging. The domain-specific ontological similarity is determined by associating semantic associations with web service descriptions. Domain-independent cues, give a breadth of coverage for common terms, while domain-specific ontological information allow finding deeper relationships based on industry and application specific terms allowing calculation of the the overall similarity score. The semantic and ontological matching are combined with attribute hashing, an indexing method for fast retrieval of services, where candidate attributes are identified for each query attribute, without linear searching of all attributes. The terms in the set of related entities for an entity in the service repository are used as a key to index hash table, meaning that the query entity is a key of the hash function, thus allowing retrieval of ranked relevant services.

2.3 Web Modeling Language

The Web Modeling Language (WebML) is a Domain Specific Language (DSL) for the conceptual design data-intensive Web sites [11]. It allows high-level description of a Web site considering different dimensions. It consists of a *Data Model*, representing the data content, and a *Web Model*, describing the pages that compose it, the topol-

ogy of links between pages, the layout and graphic requirements for page rendering, and customization features for one-to-one content delivery. Every WebML concept can be represented with a graphic notation and a textual XML syntax.

The data content of a Web site is described by the Data Model, in terms of entities, attributes and relationships, exploiting the famous entity-relationship (ER) model. Entities are units of data, while relationships give the associations between entities.

The Web Model specifies the organization of the front-end interfaces of a Web application. The main WebML constructs are pages, units and links, organized into areas and site views. Site view is a coherent hypertext, fulfilling a well-defined set of requirements [12]. Multiple site views can be defined on top of the same data schema. Site views can be decomposed into areas, clusters of pages with a homogenous purpose. Pages are the interface elements that are brought to the users. Each page is constructed by assembling several units. Content units are atomic elements that determine the web page content, combined together to build pages with different complexities. They permit different ways of content organization, extracted from the entities and relationships of the data model, and specify data entry forms for the user input. WebML defines the following standard content units:

- *Data unit* gives information for a single object of a given entity.
- *Index unit* presents a list of properties of instances of an entity, without their detailed information.
- *Scroller unit* allows browsing of an ordered set of instances, enabling access to the first, last and next element in the set.
- *Multichoice unit* shows a list of entity instances with checkboxes, allowing multiple instance selection at a time.
- *Hierarchical index unit* shows nested indexes of different entity instances.
- *Entry unit* builds entry forms and is used to collect the user input.
- *Set unit* stores parameters into the HTTP user session.
- *Get unit* retrieves parameters from the HTTP user session.

Pages and units are linked to each other, forming a hypertext structure. Links are oriented connections between two units or pages. They allow navigation in the hypertext (navigation links),

as well as transfer of parameters between units (transport links). WebML also specifies operation units, introduced to enable operations on data, and they are executed as a result of navigating a link, performing manipulation with data, or execution of an external service. They can be placed outside of the pages, and linked to other operation units, or linked to units in the pages. Every operation unit contains additional outgoing links defined as OK-links and KO-links, corresponding to the operation's success and failure. WebML defines several operation units:

- *Create unit* creates a new entity instance.
- *Delete unit* deletes objects of a given entity.
- *Modify unit* modifies one or more objects of an entity.
- *Connect unit* creates new instances of a relationship.
- *Disconnect unit* deletes instances of a relationship.
- *Login and logout unit* perform the login and logout operations specifying the controlled access to the site.
- *Sendmail* unit allow sending e-mail messages.
- *Selector unit* performs queries and retrieves attributes of entity instances.
- *Switch unit* evaluates a condition that triggers the navigation of one of its multiple outgoing OK-links. It is used for conditional execution of operations or navigation towards different pages.

Operation units can be clustered into transactions, executed automatically, where either the whole sequence of operations is executed successfully, or the sequence is undone.

Figure 2.1 presents an example of WebML Data Model of a web application, where each entity contains a list of attributes and it is connected to other entities by relationships.

In figure 2.2 is given a part of a WebML Web Model of the same application. As it can be noted, the area (Products), part of a site view, contains four pages (Product Page, Images Page, By category, By price). Each page contains content units connected among them with links.

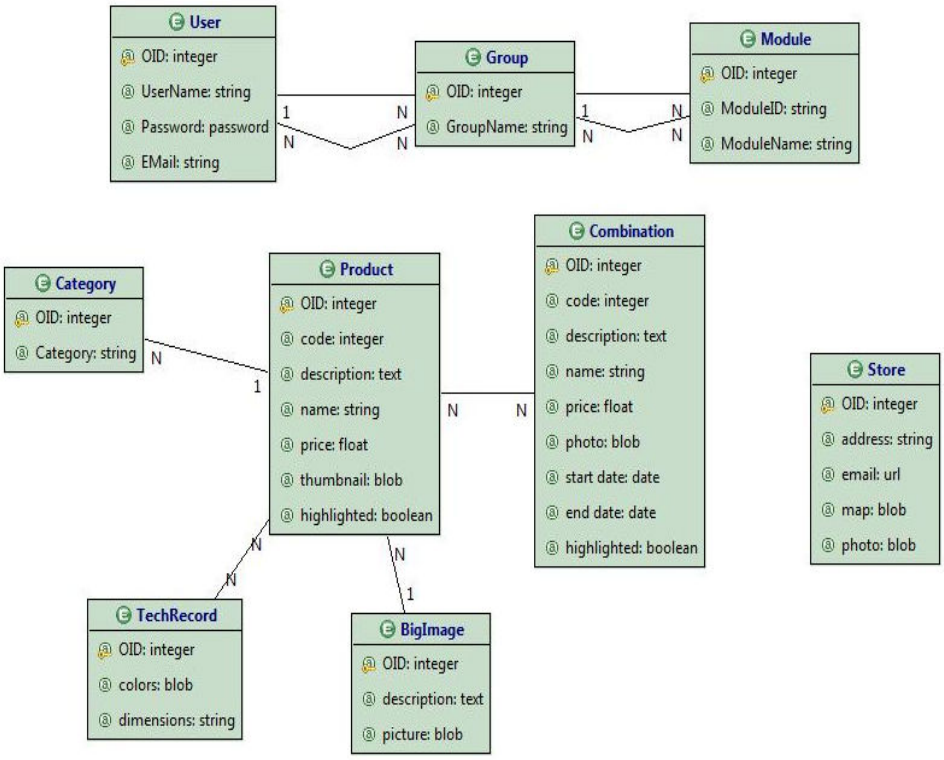

Figure 2.1: WebML Data Model

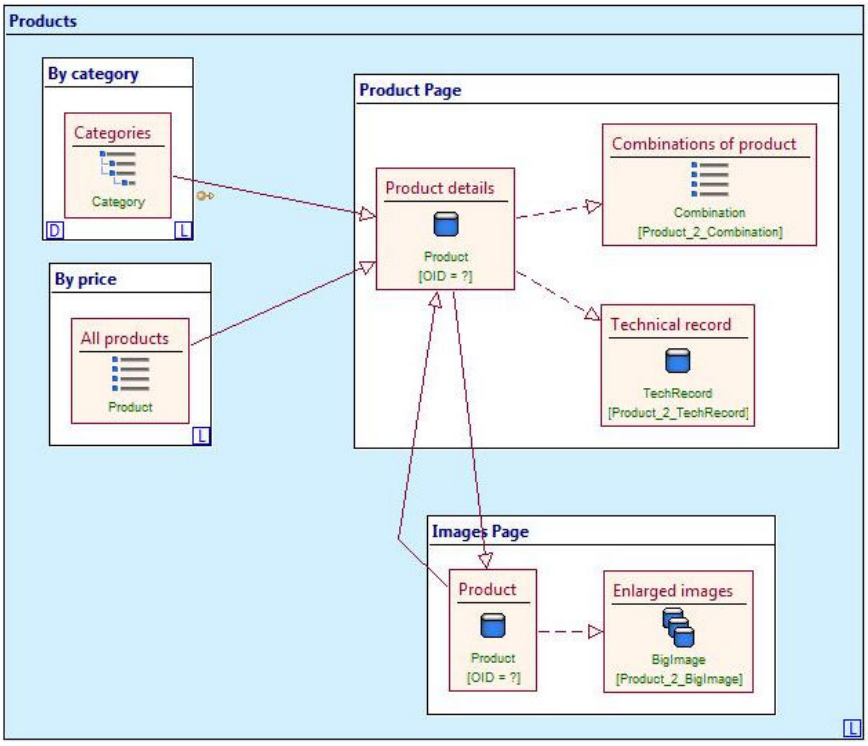

Figure 2.2: A part of a WebML Web Model

Chapter 3

Framework

This chapter describes our framework for content-based search of models in a repository. The proposed framework builds on the work of Bozzon Et. Al [9], that introduced the problem of model repository search, and proposed a solution using textual information retrieval methods and tools. In Section 3.1 we put our work in context by revising [9]; in Section 3.2, instead, we elaborate on the proposed approach for content-based search of model repositories.

3.1 Searching repositories of Web application models

3.1.1 Design Dimensions of a Model Driven IR System

The design space of a model repository information retrieval system is characterized by multiple dimensions: the transformations applied to the models before indexing, the structure of the indexes, and the query and result presentation options.

Segmentation Granularity. An important design dimension is the **granularity** of indexable documents, which determines the atomic unit of retrieval for the user. An indexable document can correspond to:

- A whole design project: in this case, the result set of a query consists of a ranked list of projects.
- A subproject: the result set consists of ranked sub-projects and each subproject should reference the project it belongs to.
- A project concept: each concept should reference its project and the concepts it relates to. The result set consists of ranked

concepts, possibly of different types, from which other related concepts can be accessed.

Our work uses entire projects for building the index. The results are retrieved as a ranked list of projects.

Index structure. The structure of the index constructed from the models represents a crucial design dimension. An index structure may consists of one or more fields, and each field can be associated with an importance score (its weight). The organization of the index into fields allows the matching procedure used in query processing to match in selected field, and the ranking algorithm to give different importance to matches based on the field where they occur.

An index can expose the following organization:

- **Flat:** a simple list of terms is extracted from the models, without taking into account model concepts, relationships, and structure. The index structure is single-fielded, and stores undifferentiated bags of words. This option can be seen as a baseline, extracting the minimal amount of information from the models and disregarding any structure and semantics associated with the employed modeling language.

- **Weighted:** terms are still extracted as flat lists, but model concepts are used in order to modify the weight of terms in the result ranking, so to give a significance boost to terms occurring in more important concepts. The index is single-fielded and stores weighted bags of words.

- **Multi-field:** terms belonging to different model concepts are collected into separate index fields. The index is multi-fielded, and each field can be searched separately. This can be combined with the weighted approach, so as to produce a multi-field index containing weighted terms. The query language can express queries targeted to selected fields (e.g., to selected types of concepts, diagrams, etc).

- **Structured:** the model is translated into a representation that reflects the hierarchies and associations among concepts. The index model can be semi-structured (XML-based) or structured (e.g., the catalog of a relational database). Query processing can use a structured query language (e.g., SQL), coupled with functions for string matching into text data (e.g., indices for text objects).

Moving from flat to structured index structures augments the fidelity at which the model structure is reflected into the index structure, at the price of a more complex extraction and indexing phase and of a more articulated query language.

With respect to this classification, the index model in our work is semi-structured, preserving the relationships among model elements, and incorporating the model features in the index. Thus, the index is more reliable.

Query Language and Result Presentation. An IR system can offer different query and result visualization options. In the context of software model retrieval, the modalities that can be envisioned are:

- **Keyword-based search:** the user provides a set of keywords. The system returns results ranked according to their relevance to the input keywords.
- **Document-based search:** the user provides a document (e.g., a specification of a new project). The system analyzes the document, extracts the most significant words and submits them as a query. Results are returned as before.
- **Search by example:** the user provides a model as a query. The model is analyzed in the same way as the projects in the repository, which produces a document to be used as a query[1]. The match is done between the query and the project document and results are ranked by similarity.
- **Faceted search:** the user can explore the repository using *facets* (i.e., property-value pairs) extracted from the indexed documents, or he can pose a query and then refine its results by applying restrictions based on the facets present in the result set.
- **Snippet visualization:** each item in the result set can be associated with an informative visualization, where the matching points are highlighted in graphical or textual form.

The above mentioned functionalities can compose a complex query process, in which the user applies an initial query and subsequently navigates and/or refines the results in an exploratory fashion.

Our work proposes a content-based search approach (or search by example) for querying the repository of models. Thus, the query is

[1]Here the term *document* means by extension any representation of the model useful for matching, which can be a bag of words, a feature vector, a graph, and so on.

a model, matched against models, and their similarity is calculated using ad-hoc similarity functions. Table 3.1 illustrates the summary of the design options, and positions our work, denoted as F, with respect to them.

Table 3.1: Summary of the design options and their relevance in the proposed framework.

Option	**Description**	**F**
Segmentation Granularity		
Project	entire project	X
Subproject	subproject	
Single Concept	arbitrary model concepts	
Index structure		
Flat	flat lists of words	
Weighted	words weighted by the model concepts they belong to	
Multi-field	words belonging to each model concept in separate fields	
Structured	XML representation reflecting hierarchies and associations	X
Query language and result presentation		
Keyword-based	query by keywords	
Document-based	query through a document	
By example	query through a model (content-based)	X
Faceted	query refined through specific dimensions	
Snippets	visualization and exploration of result previews	

3.1.2 Background

The approach proposed in [9] adopts a model-independent framework that can be customized for any Domain Specific Language (DSL) meta-model. A big advantage of this approach is the automatic extraction of model semantics, without applying the effort of manual metadata annotation. The approach was tested against a project repository of WebML models. Three different scenarios were evaluated:

- Keyword search on whole projects with a flat index structure;
- Retrieval of sub-projects and concepts with a flat index structure;
- Retrieval of sub-projects and concepts with weighted index structure.

The weighted multi-index structure had the following fields:

id|**projectID**|**projectName**|**documentType**|**text**,

where the document Type field can have the values of the fundamental modularization constructs of WebML. Prior to indexing, content processing of the projects was performed. First, a segmentation was applied, which divides the project into smaller units (segments). The information contained in the segment was then used to build

the index. Then, a linguistic analysis was carried out. The segmentation and the linguistic analysis are model-independent, and can be configured by means of model transformation rules encoded in XSLT. The transformation rules correspond to the meta-model, and specify the information to be extracted for building the index.

Moreover, an auxiliary Repository Analysis component makes part of the architecture. It performs offline analysis of the entire project collection by computing statistics used for fine-tuning of the retrieval and ranking performance:

- A list of Stop Domain Concepts, i.e., words very common in the repository
- Weights assigned to each model concept

Concept weights were computed automatically based on the relative frequency of model concepts in the entire repository, but it can be also adjusted manually by the search engine administrator. Content processing was implemented by extending the text processing and analysis components provided by Apache Lucene, an open source search engine.

The relevance of the match was computed differently for diverse scenarios. Keyword search on whole projects used pure TF/IDF measure for textual match relevance [40]. The second scenario, retrieval of sub-projects and concepts with flat index structure also uses the TF/IDF relevance, but due to segmentation, the matching was performed separately on the different model concepts. The third scenario, retrieval of sub-projects and concepts with weighted index structure, considered also the model concepts in the ranking function, calculated as:

$$score(q,d) = \sum_{t \in q} \sqrt{tf(t,d)} \cdot idf(t)^2 \cdot mtw(m,t) \cdot dw(d) \qquad (3.1)$$

The equation above introduces a Model Term Weight *mtw(m,t)*, a meta-model specific weight of a term t in a model concept m, and a Document Weight *dw(d)* a model specific value, conveying the importance of a a given project segment d(project,subproject, concept) in the index. *tf(t,d)* represents the term frequency,the number of times the term appears in a document, while *tf(t,d)* is the inverse document frequency, calculated as

$$1 + \log \frac{|D|}{freq(t,d) + 1} \qquad (3.2)$$

, measuring the informative potential carried by the term in the collection.

The repository was queried by keywords. Furthermore, the users were able to explore the result list in more details by applying faceted navigation. Each facet contained automatically extracted property values of the retrieved results.The selection of a facet value triggered refinement of the current result set, restricted only to the entries associated to that facet value. Faceted search doubles the index size because the original text of facet properties is stored in the index. For visualization and exploration of result previews, snippet visualization was implemented. It shown all the query matches in the current project (subproject or construct), allowing users to determine which model fragments are useful for their information need. Snippet visualization was the most demanding in terms of space requirements, especially in the case of segmented models, increasing the number of indexed documents. Although the knowledge of the meta-model elements increased the response time with the growth of the index size, it gives better results with respect to the flat-index keyword search.

Our work extends the solution described in [9] by proposing content-based approach for model repositories search. The proposed framework shares with [9] principles such as model independence and generality, applying them to the indexing and query processes for model similarity search.

3.2 Content-based search of model repositories with graph matching techniques

3.2.1 The Architecture of a Content-based Model Repository Search System

Applying information retrieval techniques over model repositories requires an architecture for processing content, building up the required search indexes, matching the query to the indexed content, and presenting the results. Figure 3.1 shows the reference architecture adopted in this approach and the two main information flows: *content processing* and *query*.

Content-based search enables finding relevant projects from a project repository by using models or model fragments as queries. Following the architecture of an Information Retrieval system for content-based search, content processing is applied to the projects in the repository in order to extract the relevant information needed for the index creation. This includes general information about the project, like project title, and information for a specific model ele-

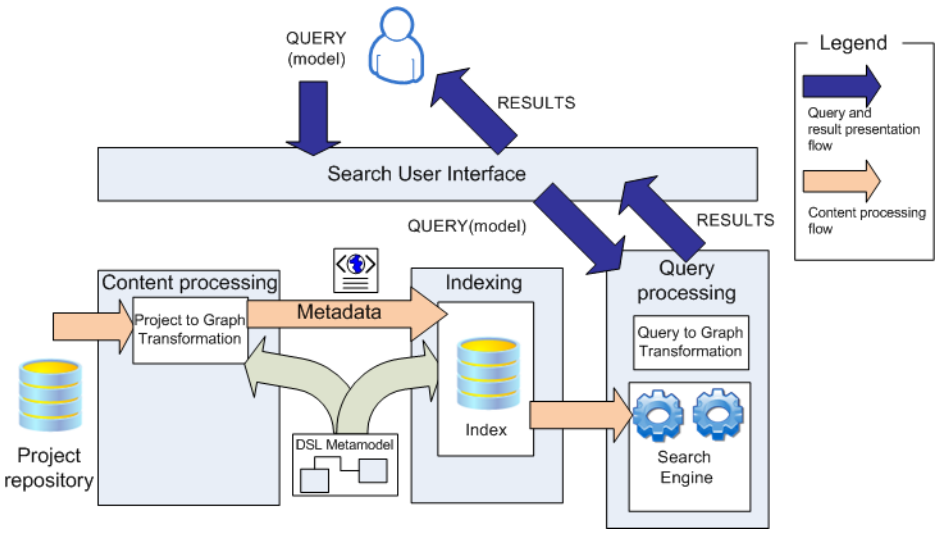

Figure 3.1: Architecture of a content processing and search system

ment, like element name, type, attributes, comments, and relationships with other elements. This is used to build the structured index which besides the basic element information, considers their hierarchical structure and their relationships. A linguistic normalization can also be applied as a plug-in component to the indexing process, e.g. tokenization, stemming, lemmatization etc for optimization of the retrieval performance

The query's content, being also a model or model fragment, needs to be processed in the same way as the projects in the repository. Then, the query is matched against the index using different algorithms based for e.g. on schema matching or graph matching. The results are presented to the user as a ranked list of projects. The query can also be expanded by exploiting semantic similarity, expanding the abbreviations and acronyms , looking up for synonyms, homonyms, hypernyms, or hyponyms in a thesaurus, like WordNet. Word semantics can also be applied to the projects before indexing, producing semantic index [44]. Another possibility is to group model elements into semantic categories, relating them to a known ontology concept. Ontologies offer more elaborated way for detection of semantically close similarities.

In [9] the content processing performs segmentation by splitting the project into smaller units, and then the segment analysis is done, which mines from each segment the information used to build the index (e.g., model elements' names and types, model element relationships, designers's comments). The information extracted from

each project or segment, thereof is physically represented as a document, which is fed to the Indexing component, for constructing the search engine indexes. The meta-model of the DSL used to express the projects is used both in the Content Processing and in the Indexing components: in the former, it drives the model segmentation granularity and the information mining from the model elements; in the latter, it drives the definition of the index structure. The query is keyword based, looking for textual matches in indexes.

One advantage of this architecture is that it supports model-driven processes, that is that the indexing and querying processes are designed to be model-independent. The designer of the system only has to decide which is the information that needs to be extracted, and the meta-model based rules for extraction. After the process, the projects, as well as the query, are transformed into graphs, whose nodes correspond to the model elements, and the edges correspond to the relationships and hierarchies among these elements. Graph representation enables generalization of the approach, since any model type can be mapped to a graph, preserving its structure.

This work implements content-based search by using a graph matching (subgraph isomorphism) technique, to retrieve the most relevant projects from a repository with respect to a query in the form of a model or model fragment.

3.2.2 Graph Matching

One of the most important problems when using graphs is the comparison of graphs with each other. For example, in computer vision, when graphs are used for the representation of 3-D objects, the recognition of known model objects in an unknown scene can be done by trying to find correspondences between the graphs representing the models and the graph representing the scene. Trying to find such correspondences is generally referred to as graph matching. There are three different classes of graph matching, namely *graph isomorphism*, *subgraph isomorphism*, and *error-correcting subgraph isomorphism* [43].

When matching two graphs G_1 and G_2 by means of graph isomorphism, a bijective mapping between the vertices of G_1 and G_2 needs to be found, such that the structure of the edges is preserved by the mapping function. When such a mapping function can be found, then G_1 and G_2 are considered isomorphic. If one of the graphs involved in the matching process is larger than the other, i.e. G_2 contains more vertices than G_1, then a subgraph isomorphism from

G_1 to G_2 should be found. That is, a subgraph S of G_2 should be found, such that G_1 and S are isomorphic. Subgraph isomorphism is a special case of graph isomorphism. Finally, in many applications, the encoding of objects as attributed graphs will not be perfect due to the presence of noise and distortions. Hence, it is necessary to introduce an error model and incorporate the concept of errors into graph matching. The graphs are then compared to each other by means of error-correcting subgraph isomorphism. The definition of an error model is strongly application dependent.

One of the drawbacks of graph matching is its computational complexity. State-of-the-art algorithms for the general graph isomorphism problem require, in the worst case, exponential time [43]. The subgraph isomorphism problem, and also the are NP-complete problems. Consequently, no algorithm could be constructed that guarantees to find (error-correcting) subgraph isomorphisms in polynomial time. However, research has shown that there are methods for graph matching that behave reasonably well on the average in terms of performance and become computationally intractable only in a few cases. Moreover, if the constraints of graph matching are loosened, it is possible to find solutions in polynomial time by using approximate algorithms.

Our work leverages graph matching by finding a mapping between two graphs, a query graph and a project graph. Graphs allow abstract representation of models, encoding the specific model features into structure of nodes and edges. Therefore, the models from the repository, and the queries are represented as directed annotated graphs, preserving model topologies, and encapsulating the model information as annotations.

According to [50], the graph representing a model can be defined as a tuple (N, E, ϕ, λ), where:

- N is the set of nodes
- E is the set of edges
- $\phi : N \to \Phi$ is a function that maps nodes to types
- $\lambda : N \to \Lambda$ is a function that maps nodes to names

Φ and Λ are sets of node types, and node names respectively.

In order to perform comparison of two graphs,the metric based on graph edit distance is used. The graph edit distance between two graphs is the minimal total cost of operations needed to transform one graph into the other. The following operations are considered:

- Node substitution: A node from one graph is substituted with a node from the other graph if they are similar
- Node insertion/deletion: A node is inserted into a graph, and deleted from the second graph or the opposite
- Edge insertion/deletion: An edge is inserted into a graph, and deleted from the second graph, or the opposite

If two nodes are similar, they are substituted. Otherwise, the nodes are inserted or deleted, respectively. An edge is inserted/deleted, if at least one of its incident nodes (that it connects) is inserted or deleted. Each of the above mentioned operations, has a cost associated to it. The cost for node insertion and deletion, and edge insertion and deletion is fixed,usually, a value in the interval [0,1], while the cost for node substitution is one minus the node similarity. Two nodes are similar if their names and types are similar. If node types are similar, then their names similarity is checked.

The similarity of node types depends on the similarity of model elements. The similarity metric used for node names can vary. In [50], for instance, the string edit distance between two strings, also known as the Levenshtein distance [34], has been used.

The Levenshtein distance is defined as the minimal number of atomic operations, needed to transform one string into the other. Atomic operations considered are: inserting a character, deleting a character, and substituting a character. The Levenshtein distance is normalized with the length of the longer string and it is used in calculating similarity between two nodes, obtained by subtracting from 1, the normalized Levenshtein distance. Formally, considering two graphs $G_1 = (N_1, E_1, \phi_1, \lambda_1)$ and $G_2 = (N_2, E_2, \phi_2, \lambda_2)$, the similarity of two nodes $n_1 \in N_1$,$n_2 \in N_2$ can be specified as:

$$sim(n_1, n_2) = \begin{cases} 1 - \frac{ed(\lambda_1(n_1), \lambda_2(n_2))}{max(|\lambda_1(n_1)|, |\lambda_2(n_2)|)} & \text{if } cs(\phi_1(n_1), \phi_2(n_2)), \\ \perp & \text{otherwise} \end{cases}$$

where $ed(\lambda_1(n_1), \lambda_2(n_2))$ is the Levenshtein distance of the node names.

Next, a definition of graph edit distance is introduced. For two graphs $G_1 = (N_1, E_1, \phi_1, \lambda_1)$ and $G_2 = (N_2, E_2, \phi_2, \lambda_2)$, an injective partial mapping $M : N_1 \to N_2$ is introduced that maps nodes from G_1 with nodes from G_2. Domain of M is $dom(M) = \{n_1 \mid (n_1, n_2) \in M\}$, while its codomain is $cod(M) = \{n_2 \mid (n_1, n_2) \in M\}$. A node n can be substituted if and only if $n \in dom(M)$ or $n \in cod(M)$. The set of substituted nodes is denoted as *subn*. As mentioned previously, if nodes are not similar, they cannot be substituted. In this case, they are inserted or deleted. A node $n_1 \in N_1$ is deleted from

G_1,(inserted into G_2) if and only if it is not substituted. Similarly, the same holds for a node $n_2 \in N_2$ being inserted into G_1,(deleted from G_2).The set of all inserted/deleted nodes is denoted as *skipn*. Let $(n_1, m_1) \in E_1$ is an edge in E_1, where n_1, and m_1 are its incident nodes. An edge (n_1, m_1) is deleted from G_1,(inserted into G_2), iff there do not exist node mappings $(n_1, n_2) \in M$), $(m_1, m_2) \in M$), and an edge $(n_2, m_2) \in E_2$ in G_2. Edges inserted into G_1, (deleted from G_2) are defined similarly.The set of all inserted and deleted edges is denoted as *skipe*. An edge is substituted if it is not inserted or deleted. The graph edit distance induced by the mapping can be calculated as:

$$|skipn| + |skipe| + 2 \cdot \sum_{(n_1,n_2)\in m} (1 - sim(n_1, n_2)) \tag{3.3}$$

and it represents the minimal possible distance induced by a mapping. The graph edit similarity is a metric that determines how two graphs are similar in the range [0,1]. It is the maximal possible similarity induced by a mapping between two graphs. Formally it can be defined as:

Let $G_1 = (N_1, E_1, \lambda_1)$ and $G_2 = (N_2, E_2, , \lambda_2)$ be two graphs,$M : N_1 \to N_2$ be a partial injective mapping that maps nodes from G_1 to nodes in G_2, and let *subn, skipn* and *skipe*be the sets of substituted nodes, insereted or deleted nodes and inserted or deleted edges, defined previously. Moreover, three new values are introduced, $0 \leq wsubn \leq 1$, $0 \leq wskipn \leq 1$, and $0 \leq wskipe \leq 1$, representing weights assigned for node substitution, node insertion or deletion, and edge insertion or deletion, respectively.

The fraction of inserted or deleted nodes, *fskipn* is defined as:

$$fskipn = \frac{|skipn|}{|N_1| + |N_2|} \tag{3.4}$$

where | *skipn* | is the cardinality of the set of inserted or deleted nodes, while | N_1 | and | N_2 | are cardinalities of the sets of nodes of G_1 and G_2, respectively. The fraction of inserted or deleted edges *fskipe* is calculated in the following way:

$$fskipe = \frac{|skipe|}{|E_1| + |E_2|} \tag{3.5}$$

, where as previously, | *skipe* | is the cardinality of the set of inserted or deleted edges, while | E_1 | and | E_2 | are cardinalities of the sets of edges of G_1 and G_2, respectively. The average distance of

substituted nodes *fsubn* is defined as follows:

$$fsubn = \frac{2 \cdot \sum_{(n,m)\in M}(1 - sim(n,m))}{|subn|} \tag{3.6}$$

,

where *sim(n,m)* is the similarity of two nodes defined previously, and | *subn* | is the cardinality of the set of substituted nodes. The graph edit similarity induced by the mapping is:

$$1 - \frac{wskipn \cdot fskipn + wskipe \cdot fskipe + wsubn \cdot fsubn}{wskipn + wskipe + wsubn} \tag{3.7}$$

,

where the fraction of inserted or deleted nodes, the fraction of inserted or deleted edges, and the average distance of substituted nodes are weighted with their corresponding weights (costs), normalized by the total sum of weights.

In order to explain the calculation of the graph edit similarity, the example in figure 3.2 representing two graphs is provided. As it can be seen, each node is associated a couple *(name,type)*. In this example,we assume that b and b1 are similar types. Since nodes with same or similar types can form a mapping, then the mappings (Technical Record, Tech Record), and (Product, Product) are created, meaning that 2 nodes are substituted. For the first mapping,the string edit distance is 5, and the length of longer string is 16. Its node similarity can be calculated as:

$$1 - \tfrac{5}{16} = 0.6875$$

. In the second mapping, the nodes are identical, and the string edit distance is 0. Therefore, its node similarity is 1. The average distance of substituted nodes is:

$$fsubn = 1 - \tfrac{2\cdot(0+0.3125)}{2} = 0.3125$$

The number of inserted/deleted nodes is 1. It is (Review, c) and the total number of nodes is 5 (2 in the first graph, 3 in the second graph). The fraction of inserted nodes is:

$$fskipn = \tfrac{1}{5} = 0.2$$

Inserted/deleted edges are: the edge from (Tech Record, a) to (Review, c); and the edge from (Product, b1) to (Review, c). Considering that the total number of edges is 4 (1 in the first, and 3 in the second graph); the fraction of inserted edges is:

$$fskipe = \tfrac{2}{4} = 0.5$$

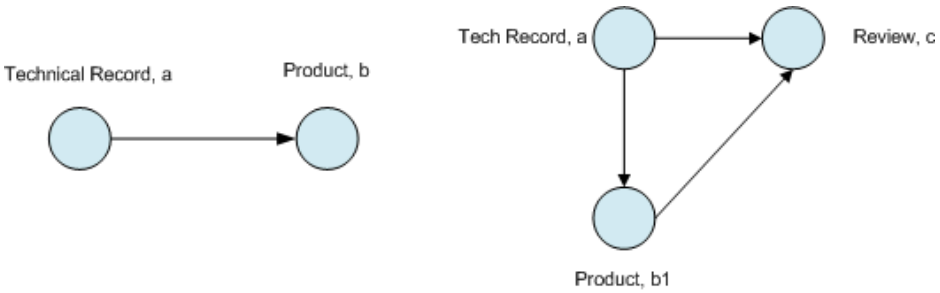

Figure 3.2: Graph example

Using the weights, for example *wskipe=0.4, wskipn=0.2, wsubn=0.7*, the graph edit similarity of these two graphs is:

$$1 - \frac{0.4 \cdot 0.5 + 0.2 \cdot 0.2 + 0.7 \cdot 0.3125}{0.4 + 0.2 + 0.7} = 0.647$$

3.2.3 Algorithm

Calculation of the graph edit similarity requires finding a mapping between graphs that induces their maximum similarity. Many algorithms deal with graph matching, described in the related work chapter. As in our case, the query graph and the project graph will rarely have the same dimensions, an error-correcting subgraph isomorphism should be used. In the past, various approaches to error-correcting subgraph isomorphism detection have been proposed. The most common approach is based on tree search with the A-star algorithm. The search space of the A-star algorithm can be greatly reduced by applying heuristic error estimation functions. By always expanding the node with the least cost, the algorithm is guaranteed to find the optimal mapping. The computational complexity of the algorithm depends on the size of the graphs, the number of labels and number of errors [43]. In [50], A-star provides better precision with respect to the other algorithms in finding business process model similarity. A-star algorithm originally described by [26], is the best-first algorithm that finds the minimum cost path from one node to another node. This algorithm is optimal since it examines the smallest number of nodes necessary to guarantee a minimum cost solution. For this reason, the algorithm must make a decision which node to expand next by using an evaluation function. The A-star algorithm for error-correcting subgraph isomorphism detection used here, has been applied to the problem of graph matching in [43], and then adapted in [50]. The algorithm starts from a node n_1 in the query graph, and creates all the possible partial mappings

from this node to every node in the project graph. Additionally, an extra mapping with a dummy node ϵ is created,(n_1, ϵ), representing the case where n_1 is deleted.The partial mapping with the maximal graph edit similarity is selected, and expanded into a number of larger mappings.The algorithm proceeds with the next node from the query graph, and creates partial mappings with every node from the project graph, excluding the ones already in the mapping.The algorithm always selects the mapping with maximal graph edit similarity, and expands it, by adding a mapping for the next node. The algorithm finishes when all the nodes from the query graph are mapped. Here, an error-correcting function is the graph edit similarity, which selects a mapping added to the existing mapping, that has the maximum graph edit similarity. To reduce the memory requirements, only mappings between similar nodes are allowed. In other words, nodes belonging to similar types and, whose names have string edit similarity greater then a threshold parameter, can form a partial mapping.
The pseudocode of this algorithm for a query graph G_1, and a project graph G_2, defined as $G_1 = (N_1, E_1, \phi_1, \lambda_1)$, $G_2 = (N_2, E_2, \phi_2, \lambda_2)$, respectively, is given here:

Algorithm 1 A-star algorithm

Require: $open \leftarrow (n_1, n_2) \mid n_2 \in N_2 \cup \{\epsilon\}, sim(n_1, n_2) > threshold \vee n_2 = \epsilon$,
for some $n_1 \in N_1$
while $open \neq 0$ **do**
select $map \in open$, such that $s(map)$ is max
$open \leftarrow open - map$
if $dom(map) = N_1$ **then**
return s(map)
else
select $n_1 \in N_1$, such that $n_1 \notin dom(map)$
for all $n_2 \in N_2 \cup \{\epsilon\}$, such that either $n_2 \notin cod(map)$ and $sim(n_1, n_2) > threshold$ or $n_2 = \epsilon$ **do**
$map' \leftarrow map \cup \{(n_1, n_2)\}$
$open \leftarrow open \cup map'$
end for
end if
end while

where *open* is the set of all allowed mappings, and *map* is the partial mapping having the maximal graph edit similarity *s(map)*. The algorithm finishes when all the nodes from the query graph are examined. The returned value is the maximal graph edit similarity for the two graphs.

If the graphs in figure 3.2, and a threshold parameter of 0.5 are

considered, then in the first step, the following partial mappings can be formed: {(Technical Record, Tech Record)}, and{(Technical Record, ϵ)}. In the second step, the mapping with the maximal graph edit similarity is selected, and expanded in two new mappings: {(Technical Record, Tech Record), (Product, ϵ)} and {(Technical Record, Tech Record),(Product, Product)}. The algorithm finishes because all the nodes from the query graph are mapped. The mapping with the maximal graph edit similarity is {(Technical Record, Tech Record),(Product, Product)}, and all the other mappings are discarded. This means that by substituting the two pair of nodes from the mapping, and inserting the node (Review), the maximal similarity value for these two graphs will be obtained.

Considering the algorithm complexity, the best case arises when the vertices of the project and the query graph are uniquely labeled, and the query graph contains an isomorphic, undistorted copy of the model graph [43]. In this case, given a query with n vertices, and a project graph with m vertices, the algorithm first generates $O(m)$ mappings of the first vertex of the query. Out of these mappings, only the one with maximum cost is further extended, resulting in $O(m-1)$ new mappings. This process continues for each of the $O(n)$ vertices of the query, resulting in a total of $O(nm)$ mappings. For each mapping, the algorithm performs $O(n)$ edge tests. Thus, the number of steps for matching a query graph with a project graph, is in the best case bounded by:

$$O(n^2m) \tag{3.8}$$

On the other hand, the worst case arises when the error in the query graph is very large. In this case, the edit costs for correcting the project graph become very high and, consequently, each mapping in the search space will be extended. That is, the first vertex of the query graph can be mapped to $O(m)$ vertices of the project graph. As the edit costs of the optimal mapping are very high, each of the mappings for the first vertex will be extended into $O(m(m-1))$ new mappings. Generally, for the k-th vertex of the query, there will be $O(m^k)$ mappings. Hence, the total number of mappings generated in the worst case is bounded by $O(nm^n)$. For each mapping there are $O(n)$ edge constraints tested,resulting in worst case complexity:

$$O(n^2m^n) \tag{3.9}$$

Chapter 4

Experiments

This chapter contains a detailed description of the implementation of the proposed framework (Section 4.1), and its quantitative evaluation through a set of experiments (Section 4.2). Section 4.3 introduces the adopted evaluation metrics, while an interpretation of the results is given in Section 4.4 .

4.1 Implementation

This section describes the implementation of the framework for content based search on model project repositories proposed in Chapter 3. In details, the implementation activity covered: i) project and query processing, and ii) the execution of model search through the A-star algorithm for error-correcting subgraph isomorphism.

4.1.1 Content processing

Content processing is devoted to the extraction of significant information from models, and to their transformation into a representation suitable for comparison.

In our work, model elements are represented as nodes in a graph, while their relationships are represented as edges.

Considering a WebML model nodes in the graph correspond to an instance of a WebML primitive (e.g. entities, pages, units, etc.), and each node is annotated with the name and the type of such instance.

Relationships in the Data Model describe the associations among entities; relationships among Web Model elements can be classified as *containment* or as *link*. The former reflects the hierarchy of structural elements in the WebML meta-model (e.g.: a site view contains areas, pages and operation units; an area contains area,

pages and operation units; a page contains content units); the latter relates with navigational primitives such as *links* and *link parameters*. Given such relationship classification, the resulting graph representation defines relationship types as :

- *link* representing the hierarchical relations between model elements;
- *edge link*, representing the links that perform transport of parameters between units, the OK-links, and KO-links;
- *edge relationship* representing the relationships among the entities in the Data Model.

The model structure allows existence of multiple links between units, represented as parallel edges in the graph.

An example of a typical WebML model document, which is represented in XML, is given in the following snippet:

```
<WebProject >
  <DataModel>
     <Entity id="ent2" name="Category" duration="persistent"
          gr:x="601" gr:y="439" attributeOrder="att7 att8 att54a att57a"
          db:table="CATEGORY" db:database="db1">
        <Attribute name="OID" id="att7" type="integer" key="true"
            db:column="OID"/>
        <Attribute name="Name" id="att8" type="string" db:column="NAME">
          <Comment xml:space="preserve">
                  &lt;b&gt;Category&lt;/b&gt;'s name
          </Comment>
        </Attribute>
       ...
     </Entity>
     <Entity id="ent10a" name="Team" duration="persistent"
          gr:x="592" gr:y="200" db:table="TEAM" db:database="db1">
        <Attribute name="OID" id="att62a" type="integer" key="true"
            db:column="OID"/>
        <Attribute name="Name" id="att65a" type="string" db:column="NAME"/>
     </Entity>
...
     <Relationship id="rel24a_rel23a" name="Team_Category"
          sourceEntity="ent2" targetEntity="ent10a"
          db:table="TEAM_CATEGORY" db:database="db1">
       <RelationshipRole1 id="rel24a" name="Category_2_Team" maxCard="N">
          <db:JoinColumn attribute="att7" name="CATEGORY_OID"/>
       </RelationshipRole1>
       <RelationshipRole2 id="rel23a" name="Team_2_Category" maxCard="N">
          <db:JoinColumn attribute="att62a" name="TEAM_OID"/>
       </RelationshipRole2>
     </Relationship>
  </DataModel>
```

```
33    <WebModel homeSiteView="sv1" defaultLocale="en_US"
34         layout:linkLayout="">
35  ...
36       <SiteView id="sv1" name="Calendar Manager"
37            layout:style="WebRatioCalendarStyle" protected="true"
38            localized="true" landmarks="page1y area1a page24a page22a scu1y"
39            layout:linkLayout="" homePage="seu3y"
40            layout:pageLayout="WebRatioCalendarStyle/empty">
41  ...
42           <Area id="area1a" name="Administration" gr:x="5354"
43                gr:y="1841" landmarks="page30a page9 page22 page5"
44                landmark="true" protected="true"
45                defaultPage="page26a"
46                layout:gridLayout="WRDefault/Default"
47                linkedResourceOrder="lr3y lr4y lr1y lr2y"
48                layout:pageLayout="WebRatioCalendarStyle/empty">
49   ....
50             <Page gr:x="1487" gr:y="3025" id="page5"
51                  name="Members choice"
52                  linkOrder="" layout:pageLayout="WRDefault/Empty">
53                 <ContentUnits>
54                    <EntryUnit gr:x="8" gr:y="213" id="enu11"
55                         name="Available Users" linkOrder="ln115 ln34">
56                         <MultiSelectionField id="msfld4" name="Members"
57                              type="string" ajaxEventEnabled="false"
58                              ajaxEventLink=""/>
59                         <Link id="ln115" name="Close" to="page21" type="normal"
60                              automaticCoupling="true" validate="true"
61                              preserveForm="true" ajaxEnabled="true"
62                              ajaxCloseWindow="true"/>
63                         <Link id="ln34" name="&gt;" to="seu4" type="normal"
64                              automaticCoupling="false" validate="true"
65                              ajaxEnabled="true">
66                            <LinkParameter id="par82"
67                                 name="Members_KeyCondition4 [OID]"
68                                 source="msfld4" target="kcond2.userOID"/>
69                         </Link>
70                    </EntryUnit>
71  ...
72                 </ContentUnits>
73             </Page>
74   ...
75           </Area>
76  ...
77       </SiteView>
78  ...
79    </WebModel>
80  ...
81  </WebProject>
```

We extract from a WebML model the following information: the *project name*; for each element in the model, the *element id*, the

element name, the *element type*, *links* among elements including their *ids*, the names of the *source* and *target* unit connected with the specific link.

Notice that, in a model, it is possible to have an occurrence of elements having the same node name, even between elements of the same type. We therefore also extract the number of occurrences of an element name for a given element type, and we will use this information to influence the calculation of node similarity as described below.

The graph representation for the previous model will be:

```
<graph edgedefault="directed">
    <node id="ent2">
      <name>Category</name>
      <type>Entity</type>
      <occurence>Category 1</occurence>
    </node>
   ...
    <node id="ent10a">
      <name>Team</name>
      <type>Entity</type>
      <occurence>Team 1</occurence>
    </node>
    <node id="sv1">
      <name>Calendar Managers</name>
      <type>SiteView</type>
      <occurence>Calendar Managers 1</occurence>
    </node>
    <node id="area1a">
      <name>Administration</name>
      <type>Area</type>
      <occurence>Administration 1</occurence>
    </node>
    <node id="page5">
      <name>Members choice</name>
      <type>Page</type>
      <occurence>Members choice 2</occurence>
    </node>
    <node id="enu11">
      <name>Available Users</name>
      <type>EntryUnit</type>
      <occurence>Available Users 1</occurence>
    </node>
...
    <edge id="edge area1a">
      <source>Calendar Manager sv1</source>
      <target>Administration area1a</target>
    </edge>
   ...
    <edge id="edge page5">
      <source>Administration area1a</source>
      <target>Members choice page5</target>
```

```
42      </edge>
43      <edge id="edge enu11">
44        <source>Members choice page5</source>
45        <target>Available Users enu11</target>
46      </edge>
47     ...
48      <edge id="edge link ln115">
49        <source>Available Users enu11</source>
50        <target>Team details page21</target>
51      </edge>
52      <edge id="edge link ln34">
53        <source>Available Users enu11</source>
54        <target>users seu4</target>
55      </edge>
56     ...
57      <edge id="edge relationship rel24a_rel23a">
58        <source>Category ent2</source>
59        <target>Team ent10a</target>
60      </edge>
61  </graph>
```

The node tag incorporates the model element **id** as an attribute of the *node* tag, **name**, **type** and **occurrence** tags (child elements) containing the corresponding data.The edge tag has an **id** as an attribute, consisting of an edge **type**, and its **id**.The edge tag also contains source and target child elements corresponding to the names and ids of the incident nodes of a directed edge.

The whole project repository is transformed into graphs, so to build the offline index required to perform queries. The index is structured, and it contains the graph XML documents. The query is processed in the exact same way as the projects from the repository, thus obtaining graphs with equal representation, suitable for comparison.

Similarity Search

The A-star algorithm for error-correcting subgraph isomorphism detection described in Section 3.2 compares the query graph against each graph in the index, and calculates their graph edit similarity. The result is a ranked list of projects, starting form the project that has maximum similarity with the query.

The model graph representation is implemented in Java with JUNG (Java Universal Network /Graph) Framework [1], a software library that provides common and extendible language for the modeling, analysis, and visualization of data that can be represented as a graph or network. The JUNG architecture has been designed to support a variety of representations of entities and their relations, such as directed and undirected graphs, graphs with parallel edges,

multi-modal graphs and hypergraphs. This implementation uses a *DirectedOrderedSparseMultigraph <V, E>*, a graph that allows parallel edges between same set of nodes, providing best representation of the structure of the model.

When a similarity search is executed, the query and the project are imported into JUNG graphs and the comparison starts.

Each query node is compared with every node from the project to check if they correspond, and then proceeds with the comparison.

Node Similarity The node similarity is calculated as:

$$sim(n_1, n_2) = (1 - levDist(n_1, n_2)) \cdot (1 - \lambda \cdot simType(n_1, n_2)) \quad (4.1)$$

where $levDist$ is the Levenshtein distance (string-edit similarity) of the node names, and $simType(n_1, n_2)$ is the type similarity for the involved nodes. The $sim(n_1, n_2)$ value ranges [0,1]; λ is a parameter that allows a linear combination of both Levenshtein distance and node type distance in the similarity evaluation. When $\lambda = 0$, the similarity is evaluated only as a function of the node name distance; on the contrary, when $\lambda = 1$ the similarity is evaluated as a function of the node name and node type distance.

The next paragraphs will provide additional details about the operands of the $sim(n_1, n_2)$ similarity function.

Levenshtein distance: the *Levenshtein* distance, whose pseudo code is given below, is a string-edit similarity measure, normalized with the length of the longer name (string). If the names are exactly the same, the distance is zero.

In the pseudocode, $d[i, j]$ is a matrix with m+1 rows and n+1 columns, keeping the minimal number of operations that transform one string character into a character from the other string. In

$$d[i, j] \leftarrow minimum(d[i-1, j]+1, d[i, j-1]+1, d[i-1, j-1]+1) \quad (4.2)$$

, $d[i-1, j]+1$ corresponds to deletion of a character, $d[i, j-1]+1$ corresponds to character insertion, while d[i-1,j-1]+1 corresponds to character substitution. The return value $d[m, n]$, gives the total Levenshtein distance. In our implementation, the number of occurences of an element name is used to make further distinction among elements that belong to the same type, and have the same name. Node names together with their number of occurences are used as strings in the calculation of the string-edit distance.

Node Type Similarity: type similarity enables the evaluation, in the similarity function, of semantic relationships between the compared nodes. The calculation of the *Node Type Similarity*

Algorithm 2 Levenshtein distance

Require: $s_1(m), s_2(n)$ where s_1, s_2 are strings with length m, n
  declare $d[0..m, 0..n]$
  for $i = 0$ to m **do**
    $d[i, 0] \leftarrow i$
  end for
  for $j = 0$ to n **do**
    $d[0, j] \leftarrow j$
  end for
  for $j = 1$ to n **do**
    for $i = 1$ to m **do**
      if $s_1[i] = s_2[j]$ **then**
        $d[i, j] \leftarrow d[i - 1, j - 1]$
      else
        $d[i, j] \leftarrow minimum(d[i - 1, j] + 1, d[i, j - 1] + 1, d[i - 1, j - 1] + 1)$
      end if
    end for
  end for
  return $d[m, n]$

exploits the information contained in the meta-model of the indexed projects; it is a value in the range [0,1], where 0 corresponds to non-similar types, and 1 corresponds to equal types. The *Node Type Similarity* is computed as follows:

$$simType(n_1, n_2) = \frac{d_t}{max_t} \tag{4.3}$$

,

where d_t is the distance between two element types in the meta-model, while max_t is the maximum distance between model elements in the meta-model rooted graph. Notice that the *Node Type Similarity* can also be evaluated as a binary function that assigns only {0,1} values respectively to different or equal nodes.

The *Type Similarity coefficient* is then used in the equation 4.1 for calculation of the similarity of nodes n_1, n_2.

The*Node Type Similarity* is expected to give more flexibility in the algorithm, allowing the retrieval of projects whose similarity is also based on the type of the included contents.

Our experiment involved WebML models, thus exploiting the WebML meta-model. Figure 4.1 shows a tree representation of the different types of content units and operation units addressed in our work. The root is the node *Unit*, followed by its two children, *Content Unit* and *Operation Unit*. Their children are all the types of content units, and operation units, respectively. By considering the depth of each unit in the rooted graph, the length of the path

(distance) between each pair of units is calculated.

Matching: once the node similarity has been computed, its value is compared to the **ledcuttoff** parameter (a threshold value), which determines whether these two nodes can form a mapping or not. The **ledcuttoff** parameter is in the range [0,1], and it is introduced to map similar nodes, and to reduce the memory requirements of the algorithm. Otherwise, the performance of the algorithm reduces significantly, and might make it impractical.

The calculated mapping is used in the A-star algorithm, to computed the graph-edit similarity. According to the algorithm (Section 3.2), from all the mappings formed between a query and a project node, only the mapping with the maximal graph edit similarity is expanded, and the process is repeated until all the nodes from the query are processed. The result is the maximal similarity between a query and a project. During the computation of the graph edit similarity, all query-project node pairs that form a mapping are substituted. Otherwise, nodes are deleted (inserted). We also substitute edges between two nodes from a query graph that form mappings with two nodes (also connected by an edge) from the project graph. In all other cases, the edges are deleted (inserted). This information is used to calculate the *average distance of substituted nodes*, the *fraction of inserted/deleted nodes*, and the *fraction of inserted/deleted edges* defined in section 3.2. Then they are weighted with the corresponding weights: weight for node substitution **wsubn**, weight for node insertion **wskipn**, and weight for edge insertion **wskipe** to calculate the graph edit similarity.

4.2 Experiments Design

Our experiments were conducted on the project repository composed of 30 real-world WebML projects [9] from different application domains, all encoded as XML files conforming to the WebML DTD , and then modified. The projects in the repository were in Italian and English, and they had a size varying from 100 to 1700 nodes.

We manually built a query set for the evaluation of the approach. The queries have been defined as follows:

- 1 query is identical with a repository project (1 query);
- the other 14 queries represent a part of a project, with modified element names, types or links.

As a result, 15 queries with different sizes were formed: 5 small queries (1-25 nodes), 5 medium-size queries (60-300 nodes), and 5

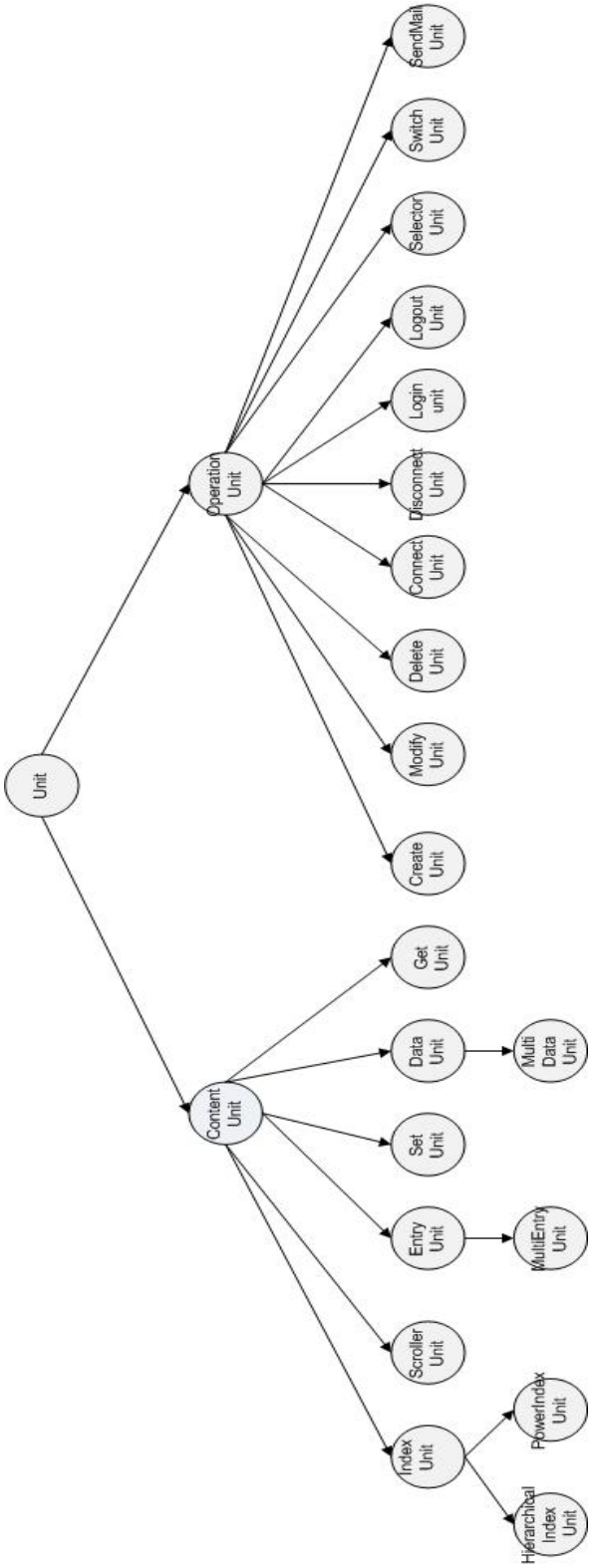

Figure 4.1: Similarity Tree of WebML Units

large queries (300-700 nodes).

4.2.1 Ground-truth evaluation

For evaluation of the implemented approach, a manual assessment was conducted in order to establish a ground truth for query-to-project relevance.

Each query was manually evaluated against the project repository, considering the unit names, unit types, hierarchical positioning of the elements in the query and the project. Then, a relevance for each query against every project has been assigned, ranging from 0-10 (with steps of 0.5), where 0 implies no relevance (i.e. the query has no match in the project), and 10 indicates maximal relevance (the query has at least an exact match in the project).

After this analysis was performed, a ranking of all the repository projects for each query has been produced.

4.2.2 Evaluation setting

As discussed in Chapter 3, the *A-star* algorithm can be configured with the following parameters:

- weight for node insertion *wskipn*, a value in the range [0,1];
- weight for node substitution *wsubn*, a value in the range [0,1];
- weight for edge insertion *wskipe*, a value in the range [0,1].

The **ledcutoff** parameter has been set to 0.6; the parameter is used as a threshold to determine whether two nodes are similar or not, so when set with such a value, the algorithm favors node and edge insertions/deletions, over node substitutions. Therefore, we expect better results when higher weights are assigned to *wskipn* and *wskipe*, while lower weight should be given for *wsubn* [50].

We tested the algorithm on eight specific weight combinations; the goal was 1) to assess the robustness of the algorithm (in terms of produced rankings) w.r.t. changes in the weight parameter values, and 2) to identify the parameter values that could return best results, w.r.t. the defined ground-truth.

- **Experiment1** emphasizes edge insertion (*wskipe*=0.7, *wskipn*=0.2, *wsubn*=0.1);
- **Experiment2** assigns higher weight for node substitution (*wskipe*=0.2, *wskipn*=0.3, *wsubn*=0.8);

- **Experiment3** gives priority to the node insertion, while the other two weights are lower (*wskipe*=0.2, *wskipn*=0.7, *wsubn*=0.2);
- **Experiment4** appoints higher weight for edge insertion, node insertion weight is slightly increased, while node substitution is given a lower value (*wskipe*=0.8, *wskipn*=0.4, *wsubn*=0.1);
- **Experiment5** favours all the weights (*wskipe*=0.9, *wskipn*=0.9, *wsubn*=1.0).
- **Experiment6** proritizes edge insertion and node substitution (*wskipe*=0.8, *wskipn*=0.2, *wsubn*=0.7);
- **Experiment7** gives higher weights for node insertion and node substitution (*wskipe*=0.1, *wskipn*=0.9, *wsubn*=0.8);
- **Experiment8** assigns higher weights for node and edge insertion (*wskipe*=0.8, *wskipn*=0.9, *wsubn*=0.1);

4.3 Evaluation Metrics

Two different metrics were used to perform the ranking analysis:

- Spearman's Rank Correlation Coefficient
- Discounted Cumulated Gain - DCG

4.3.1 Spearman's Rank Correlation Coefficient

The Spearman's rank correlation coefficient, also known as Spearman ρ [55], is a non-parametric correlation coefficient which measures the correlation on ranks. It assesses the relationship between two variables by using a monotonic function. The values of the two variables are converted to ranks. In case of tied scores, the mean rank is assigned. and used to calculate the difference in ranks d_i. This value is then used to calculate the Spearman's correlation coefficient:

$$\rho = 1 - 6 \cdot \frac{\sum_{i=1}^{n} d_i^2}{n \cdot (n^2 - 1)} \tag{4.4}$$

It has values in the range of [-1,1], where -1 corresponds to perfect negative correlation, and 1 to a perfect positive correlation.If ρ is 0, it means that there is no correlation between the variables. The values closer to -1, or +1, show high correlation between the variables.

4.3.2 Discounted Cumulated Gain

The other metric used for ranking analysis is the Discounted Cumulated Gain (DCG) [30], a measure that evaluates the ability of an IR technique for retrieving highly relevant documents. It is based on the following intuitions:

- Highly relevant documents are more valuable than marginally relevant documents;
- The higher the ranked position of a relevant document, the less valuable it is for the user, because there is less probability that the user will examine the document.

DCG is derived from the Cumulated Gain (CG) measure. CG uses the relevance score of each document as a gained value measure for its ranked position in the result for a query, and the gain is summed progressively from ranked position 1 to n. This way, the ranked document list is converted to a gained value list, by replacing the document IDs with their relevance scores. For a given gain vector G, its *ith* position can be denoted as $G[i]$. The cumulated gain (CG) vector is computed as:

$$CG[i] = \begin{cases} G[1], & \text{if } i = 1 \\ CG[i-1] + G[i], & \text{otherwise} \end{cases}$$

The second point stated above, requires computation of the cumulated gain by introducing the rank-based discounted factor. The greater the rank, the smaller share of the cumulative score is added to the cumulated gain.This is achieved by using the log of the document rank as a discounted factor. Therefore, a Discounted Cumulated Gain (DCG) vector is defined recursively as:

$$DCG[i] = \begin{cases} CG[i], & \text{if } i < b \\ CG[i-1] + \frac{G[i]}{\log_b i}, & \text{if } i \geq b \end{cases},$$

where b is the basis of the logarithm, and it is usually 2. Smaller bases model impatient users, while larger bases model persistent ones. Experiments have shown [30], that the results do not vary significantly with the change of the logarithm base. The smaller logarithm base provides stricter test conditions, as in the case of base 2.

Since CG and DCG are defined for a query, by averaging over a set of test queries, the average performance of the IR technique can be analyzed. The obtained CG and DCG vectors can be also compared to the theoretically best possible result. For this best

theoretic result, CG and DCG can be calculated, and they are called ideal CG and ideal DCG.

4.4 Ranking Analysis

Each experiment has been performed on all the 15 queries, and their results have been averaged.

The first analysis has been conducted by considering a **Node Similarity** function where *Levenshtein distance* and *Node Similarity* had equal importance, and where the *Node Type Similarity* evaluates exact similarity.

Table 4.1 reports the Spearman correlation coefficient computed for each query (rows) and each experiment (columns), together with the inter-query mean and standard deviation.The inter-experiment mean and standard deviation are given in table 4.2.

Table 4.1: Spearman's Rank Correlation Coefficient for Equal Types

	1	2	3	4	5	6	7	8
query1	0.5395	0.5575	0.5395	0.5635	0.5575	0.5395	0.5335	0.5275
query2	0.7519	0.6963	0.7853	0.798	0.717	0.6963	0.721	0.8816
query3	0.8268	0.8179	0.8215	0,8268	0.8179	0.8179	0.8179	0.8268
query4	0.9377	0.8661	0.9377	0.8176	0.901	0.9377	0.9377	0.9377
query5	0.5301	0.5891	0.588	0.539	0.5635	0.5034	0.6225	0.5502
query6	0.9656	0.9656	0.9137	0.9656	0.9137	0.9656	0.9656	0.9656
query7	0.6069	0.6069	0.6029	0.6069	0.6069	0.6069	0.6069	0.6069
query8	0.689	0.7014	0.7451	0.7139	0.689	0.6347	0.7139	0.7993
query9	0.7535	0.7749	0.7802	0.7642	0.7695	0.7535	0.7802	0.7749
query10	0.2903	0.3871	0.4323	0.4194	0.3806	0.3161	0.4194	0.5806
query11	0.6952	0.7026	0.7333	0.756	0.7026	0.6799	0.7324	0.7804
query12	0.6097	0.6806	0.6548	0.5774	0.6806	0.5968	0.7129	0.4226
query13	0.7783	0.6973	0.7957	0.8419	0.7089	0.7031	0.7031	0.7783
query14	0.7976	0.7967	0.786	0.7922	0.8189	0.7967	0.8251	0.786
query15	0.8378	0.6545	0.8249	0.8318	0.743	0.6434	0.7533	0.8169
mean	0.7073	0.6996	0.7294	0.7209	0.7047	0.6794	0.723	0.7357
std	0.1736	0.1385	0.14	0.1473	0.1391	0.166	0.1417	0.16

As table 4.1 suggests, the algorithm provides high Spearman's correlation coefficient (i.e., high correlation between the ground truth and the results obtained with the algorithm's execution) for each experiment configuration, thus showing good retrieval performance. On the other hand, the values among the different weight configurations do not vary a lot, demonstrating its robustness with respect to the weight changes, supported with the very low inter-experiment standard deviation (4.2). This also points out the good performance

Table 4.2: Inter-experiment Mean Value and Standard Deviation for Equal Types

	inter-experiment mean	inter- experiment std
query1	0.5448	0.013
query2	0.7559	0.0636
query3	0.8217	0.0044
query4	0.9092	0.0453
query5	0.5607	0.0381
query6	0.9526	0.024
query7	0.6064	0.0014
query8	0.7108	0.0476
query9	0.7689	0.0109
query10	0.4032	0.0878
query11	0.7228	0.0339
query12	0.6169	0.0914
query13	0.7508	0.0548
query14	0.7999	0.0145
query15	0.7632	0.079
mean	0.7125	0.0406
std	0.1445	0.03

of the implemented algorithm.

The inter-query evaluation, though, highlights an higher variety of correlations values (and small standard deviations), thus suggesting that the algorithm performance are a query dependent. To if the performance of the algorithm are affected by the query size, we computed the Spearman's Correlation Coefficient for each group of queries (small, medium, and large queries, defined in Section 4.2). The tables 4.3, 4.4, 4.5 show the results for the Spearman's Correlation Coefficient, as well as the the inter-query and inter-experiment mean and standard deviation. As it can be noted, the algorithm has good retrieval performance for each query group, with slightly worse results for medium-sized queries.

The low inter-query standard deviation suggests high correlation across all queries per experiment and, thus, robustness for all weight combinations.

Figure 4.2 depicts the result obtained calculating the Discounted Cumulated Gain (DCG)[1] measured for the 15 queries and averaging their results. The graph in Figure 4.2 depicts the results for the first 10 projects returned by each query. Ideal DCG represents the DCG calculated on the ground truth. It can be noted that the algorithm performs well, since the values for all the experiment settings are

[1]logarithm base b, is chosen to be 2, because of the stricter test conditions.

Table 4.3: Spearman's Rank Correlation Coefficient for Small Queries

	query3	query4	query5	query6	query7	mean	std
1	0.8268	0.9377	0.5301	0.9656	0.6069	0.7734	0.196
2	0.8179	0.8661	0.5891	0.9656	0.6069	0.7691	0.1652
3	0.8215	0.9377	0.588	0.9137	0.6029	0.7727	0.1677
4	0.8268	0.8176	0.539	0.9656	0.6069	0.7512	0.1746
5	0.8179	0.901	0.5635	0.9137	0.6069	0.7606	0.165
6	0.8179	0.9377	0.5034	0.9656	0.6069	0.7663	0.2039
7	0.8179	0.9377	0.6225	0.9656	0.6069	0.7901	0.1696
8	0.8268	0.9377	0.5502	0.9656	0.6069	0.7774	0.1899
mean	0.8217	0.9091	0.5607	0.9526	0.6064	0.7701	0.1774
std	0.0044	0.0453	0.0381	0.024	0.0014	0.0227	0.0196

Table 4.4: Spearman's Rank Correlation Coefficient for Medium Queries

	query1	query2	query10	query13	query14	mean	std
1	0.5395	0.7519	0.2903	0.7783	0.7976	0.6315	0.2171
2	0.5575	0.6963	0.3871	0.6973	0.7967	0.627	0.1588
3	0.5395	0.7853	0.4323	0.7957	0.786	0.6677	0.1703
4	0.5635	0.798	0.4194	0.8419	0.7922	0.683	0.1832
5	0.5575	0.717	0.3806	0.7089	0.8189	0.6366	0.1708
6	0.5395	0.6963	0.3161	0.7031	0.7967	0.6103	0.1886
7	0.5335	0.721	0.4194	0.7031	0.8251	0.6404	0.1619
8	0.5275	0.8816	0.5806	0.7783	0.786	0.7108	0.15
mean	0.5447	0.756	0.4032	0.7508	0.7999	0.6509	0.1702
std	0.013	0.0636	0.0878	0.0548	0.0145	0.0467	0.0324

Table 4.5: Spearman's Rank Correlation Coefficient for Large Queries

	query8	query9	query11	query12	query15	mean	std
1	0.689	0.7535	0.6952	0.6097	0.8378	0.717	0.0847
2	0.7014	0.7749	0.7026	0.6806	0.6545	0.7028	0.0448
3	0.7451	0.7802	0.7333	0.6548	0.8249	0.7477	0.063
4	0.7139	0.7642	0.756	0.5774	0.8318	0.7287	0.0945
5	0.689	0.7695	0.7026	0.6806	0.743	0.717	0.0379
6	0.6347	0.7535	0.6799	0.5968	0.6434	0.6616	0.0592
7	0.7139	0.7802	0.7324	0.7129	0.7533	0.7385	0.0285
8	0.7993	0.7749	0.7804	0.4226	0.8169	0.7188	0.1664
mean	0.7108	0.7689	0.7228	0.6169	0.7632	0.7165	0.061
std	0.0476	0.0108	0.0339	0.0915	0.079	0.0526	0.0329

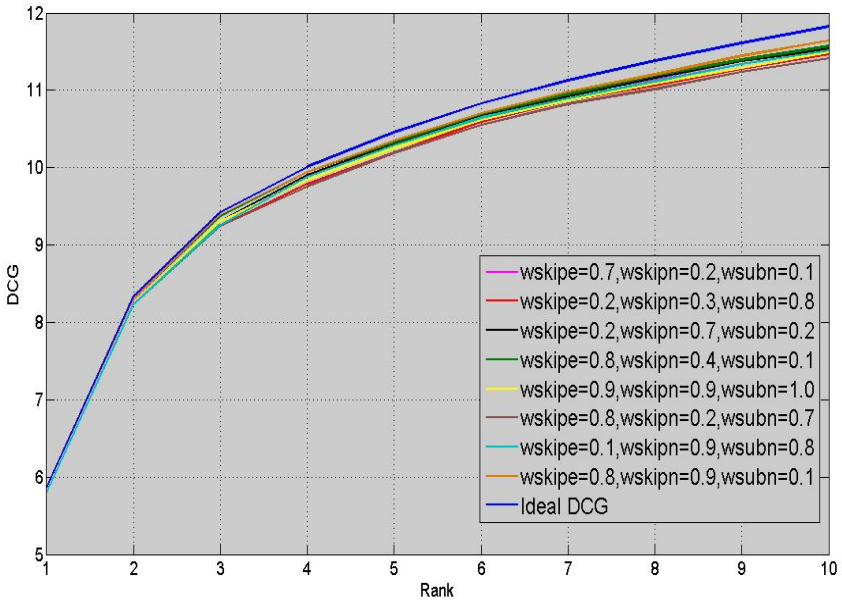

Figure 4.2: DCG for equal types

very close to the ideal DCG. Those results suggest that the algorithm is able to retrieve relevant projects in high positions in the ranking. The algorithm again shows low variability in the results with respect to the change of the weight combinations. From figures 4.3,4.4, and 4.5, it can be observed that the algorithm performs slightly better as the query size increases, a result consistent with the Spearman's correlation coefficient.

To conclude our experiments, we also evaluated the performance of the algorithm when applying a *Node Type Similarity* working on the full WebML meta-model hierarchy. The results from the calculation of the Spearman's Correlation Coefficient, are given in table 4.6, while the inter-experiment mean and standard deviation are presented in table 4.7.

The results show high correlation between the ground truth rankings, and the rankings acquired from the individual experiments, a fact supported by the low inter-query standard deviation. Furthermore, the inter-experiment standard deviation demonstrates very low variability of results with respect to changes in the weight configurations. An analysis was also conducted with DCG, and the results are reported in the graph 4.6. It can be inferred, that the results are very close to the ideal DCG, regardless of the different weight combinations, demonstrating a slight change in the ranking

Table 4.6: Spearman's Rank Correlation Coefficient for Similar Types

	1	2	3	4	5	6	7	8
query1	0.5395	0.5575	0.5395	0.5635	0.5575	0.5395	0.5335	0.5275
query2	0.7519	0.6963	0.7853	0.798	0.717	0.6963	0.721	0.8816
query3	0.7843	0.7754	0.779	0.7843	0.7754	0.7754	0.7754	0.7843
query4	0.9377	0.9377	0.9377	0.9377	0.9377	0.9377	0.9377	0.9377
query5	0.5301	0.5891	0.588	0.539	0.5635	05034	0.6225	0.5502
query6	0.9399	0.9399	0.9399	0.9399	0.9399	0.9399	0.9399	0.9399
query7	0.6069	0.6069	0.6069	0.6069	0.6069	0.6069	0.6069	0.6069
query8	0.689	0.7014	0.7451	0.7139	0.689	0.6347	0.7139	0.7993
query9	0.7535	0.7749	0.7802	0.7642	0.7695	0.7535	0.7802	0.7749
query10	0.2903	0.3871	0.4323	0.4194	0.3806	0.3161	0.4194	0.5742
query11	0.6952	0.7026	0.7333	0.756	0.7026	0.6799	0.7324	0.7742
query12	0.6097	0.6806	0.6548	0.5774	0.6806	0.5968	0.7129	0.4226
query13	0.7783	0.6973	0.7957	0.8419	0.7089	0.7031	0.7031	0.7667
query14	0.7976	0.7967	0.786	0.7922	0.8189	0.7967	0.8251	0.786
query15	0.8378	0.6545	0.8249	0.8318	0.743	0.6434	0.7533	0.812
mean	0.7028	0.6999	0.7286	0.7244	0.7061	0.6749	0.7185	0.7292
std	0.1692	0.1405	0.1408	0.1517	0.144	0.1607	0.1369	0.1559

Table 4.7: Inter-experiment Mean Value and Standard Deviation for Similar Types

	inter-experiment mean	inter-experiment std
query1	0.5448	0.013
query2	0.7559	0.0636
query3	0.7792	0.0044
query4	0.9377	0
query5	0.5607	0.0381
query6	0.9399	0
query7	0.6069	0
query8	0.7108	0.0476
query9	0.7689	0.0109
query10	0.4024	0.0859
query11	0.722	0.0324
query12	0.6169	0.0914
query13	0.7494	0.0541
query14	0.7999	0.0145
query15	0.7626	0.0785
mean	0.7105	0.0356
std	0.1442	0.0329

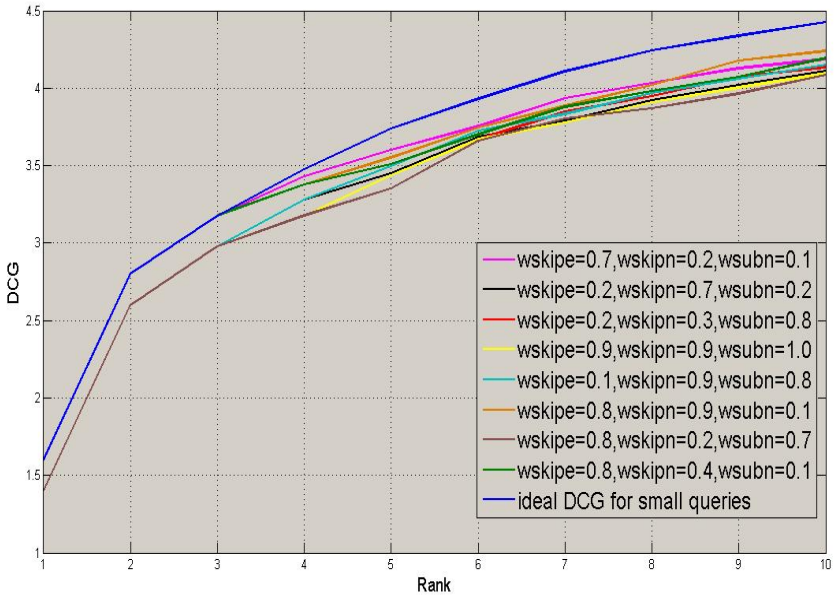

Figure 4.3: DCG for small queries

with the variation of the weight configurations.

The results show no difference with respect to the results obtained considering the exact types case. One possible explanation for this could be that models contain few elements having at the same time, similar names and similar types with the respective query elements from the query model. However, the way of measuring the node distance considering similar types requires further inspection, and it will be studied in a future work.

Final Considerations: the experiments presented in this chapter show high Spearman's Corellation Coefficient values, and DCG curves near to the ideal one, thus indicating its good performance. The very low inter-experiment standard deviation implies its robustness concerning changes in the weight configuration. Results do not fully support an hypothesis of independence of results with respect to the query size, an aspect that will be further analyzed in the future work. Another interesting aspect of this analysis is that results demonstrate small difference between the exact and the graded node type similarity, another result that will be further investigated in the future works.

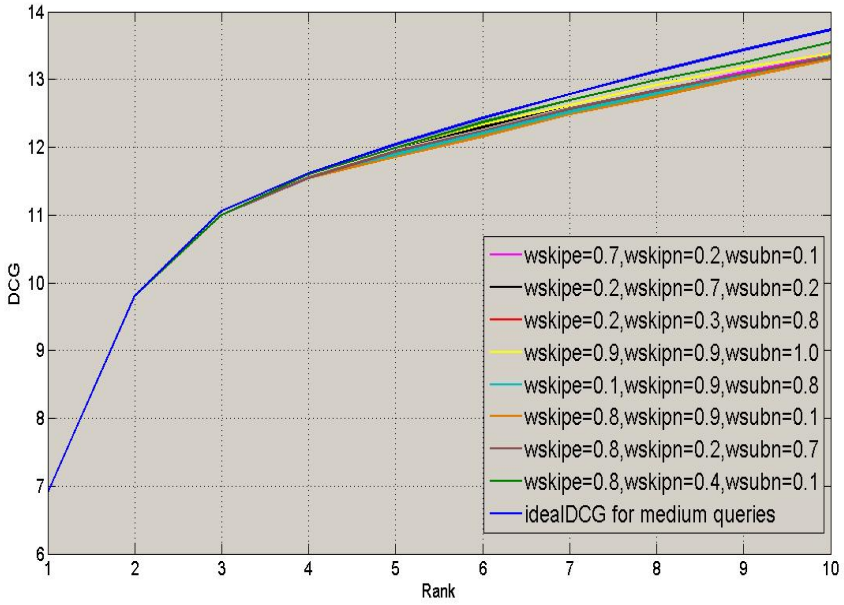

Figure 4.4: DCG for medium queries

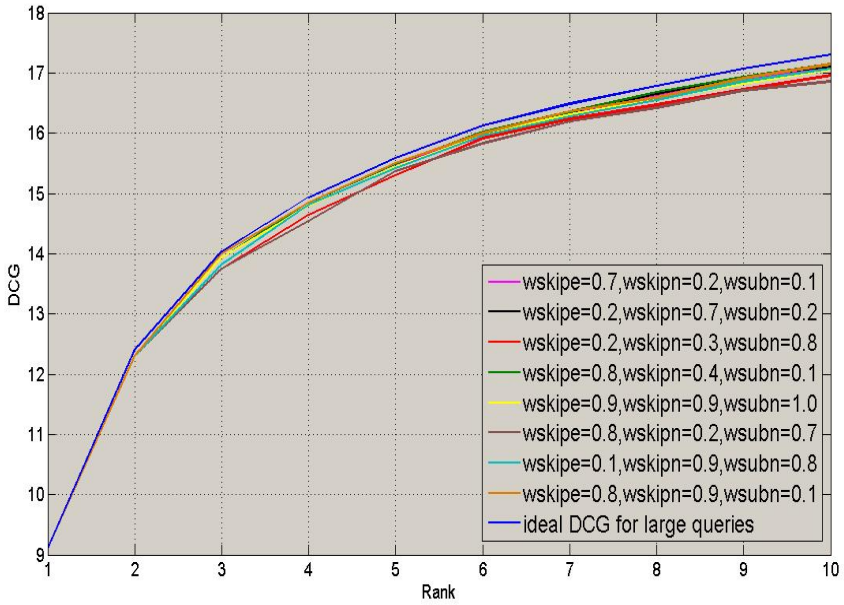

Figure 4.5: DCG for large queries

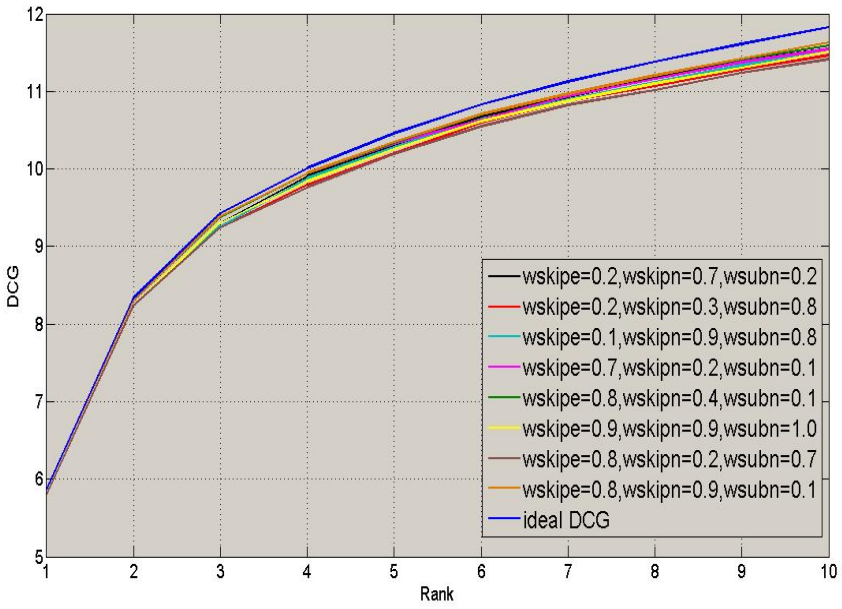

Figure 4.6: DCG for Similar Types

Chapter 5

Conclusions and Future Work

In this thesis, we addressed the problem of content-based search for repository of models. We analyzed the state of the art of different techniques and approaches used for retrieving information from project repositories, and we selected the A-star algorithm for error-correcting subgraph isomorphism detection, which we thought was promising.
Furthermore, we proposed a general framework for content-based search of model repositories based on the selected algorithm. The framework provides: 1) model-independent approach that can be customized for different model types, 2) structured index reflecting the model hierarchies and relationships, and 3) use of a similarity function that exploits the semantic relationships between model element types, contained in the metamodel. The framework was validated through its implementation using WebML models.
Finally, we carried out a quantitative assessment of the performance of the framework. For this reason, we selected 15 queries and manually evaluated them against the projects in the repository. This evaluation was used as a ground truth for analysis of the performance with different configurations. The Spearman's rank correlation coefficient and the Discounted Cumulated Gain (DCG) were used as metrics. The results of the analysis showed that the implemented work performs well, and we believe that it can be successfully used for content-based search in model repositories.

5.1 Future Work

The work on this thesis induced some interesting directions that should be further researched. The future work is planned to pursue the following steps:

- Extensive evaluation of the ground truth with more queries and projects, carried out by independent experts which will provide more objective analysis and it will enhance the existing ground truth. This will be used to fortify the performance analysis;
- Performance analysis with different parameter values, including different threshold values, which will investigate the sensitivity of the different weight configurations with respect to the changes of the threshold value. Another thing that will be studied is the influence of the threshold value when the type similarity is applied in the similarity function;
- Further examination of the type similarity of model elements, and its impact in measuring the similarity, to discover the reason for the performance invariance with respect to exact model element types. This might be inspected by using only the type similarity of model elements as a similarity metric;
- Improvement of the performance and the existing implementation of the framework by using different similarity metrics based on synonym relations, or relating model elements to an ontology concept. Also, more model element attributes can be included in the similarity.

Bibliography

[1] Jung framework. `http://jung.sourceforge.net/`.

[2] D. Akehurst and B. Bordbar. On Querying UML data models with OCL. *UML 2001The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pages 91–103, 2001.

[3] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia. Information retrieval models for recovering traceability links between code and documentation. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 40. IEEE Computer Society, 2000.

[4] A. Awad, A. Polyvyanyy, and M. Weske. Semantic querying of business process models. In *12th International IEEE Enterprise Distributed Object Computing Conference*, pages 85–94. IEEE, 2008.

[5] S. Bajracharya, J. Ossher, et al. Sourcerer: An internet-scale software repository. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 1–4. IEEE Computer Society, 2009.

[6] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying business processes. In *Proceedings of the 32nd international conference on Very large data bases*, page 354. VLDB Endowment, 2006.

[7] K. Belhajjame and M. Brambilla. Ontology-Based Description and Discovery of Business Processes. *Enterprise, Business-Process and Information Systems Modeling*, pages 85–98, 2009.

[8] D. Bildhauer, T. Horn, and J. Ebert. Similarity-driven software reuse. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, pages 31–36. IEEE Computer Society, 2009.

[9] A. Bozzon, M. Brambilla, and P. Fraternali. Searching Repositories of Web Application Models. *Web Engineering*, pages 1–15, 2010.

[10] A. Brügger, H. Bunke, P. Dickinson, and K. Riesen. Generalized Graph Matching for Data Mining and Information Retrieval. *Advances in Data Mining. Medical Applications, E-Commerce, Marketing, and Theoretical Aspects*, pages 298–312.

[11] S. Ceri, P. Fraternali, and A. Bongio. Web Modeling Language (WebML): a modeling language for designing Web sites. *Computer Networks*, 33(1-6):137–157, 2000.

[12] S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, and M. Matera. *Morgan Kaufmann series in data management systems: Designing data-intensive Web applications*. Morgan Kaufmann Pub, 2003.

[13] K. Chen, J. Madhavan, and A. Halevy. Exploring schema repositories with Schemr. In *Proceedings of the 35th SIGMOD international conference on Management of data*, pages 1095–1098. ACM, 2009.

[14] S. Comai, E. Damiani, and P. Fraternali. Computing graphical queries over XML data. *ACM Transactions on Information Systems (TOIS)*, 19(4):371–430, 2001.

[15] O. Corby, R. Dieng-Kuntz, and C. Faron-Zucker. Querying the semantic web with corese search engine. In *ECAI*, volume 16, page 705. Citeseer, 2004.

[16] H. Do and E. Rahm. COMA: a system for flexible combination of schema matching approaches. In *Proceedings of the 28th international conference on Very Large Data Bases*, page 621. VLDB Endowment, 2002.

[17] A. Doan, P. Domingos, and A. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 509–520. ACM, 2001.

[18] A. Doan, J. Madhavan, P. Domingos, and A. Halevy. Learning to map between ontologies on the semantic web. In *Proceedings of the 11th international conference on World Wide Web*, pages 662–673. ACM, 2002.

[19] X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang. Similarity search for web services. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, page 383. VLDB Endowment, 2004.

[20] D. Embley, D. Jackman, and L. Xu. Multifaceted exploitation of metadata for attribute match discovery in information integration. In *Proceedings of the International Workshop on Information Integration on the Web (WIIW01)*, pages 110–117. Citeseer, 2001.

[21] T. Feo and M. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6(2):109–133, 1995.

[22] M. Fernández, I. Cantador, and P. Castells. CORE: A tool for collaborative ontology reuse and evaluation. In *Proceedings of the 4th Int. Workshop on Evaluation of Ontologies for the Web (EON06), at the 15th Int. World Wide Web Conference (WWW06). Edinburgh, UK*. Citeseer, 2006.

[23] V. Garcia, D. Lucrédio, F. Durão, E. Santos, E. de Almeida, R. de Mattos Fortes, and S. de Lemos Meira. From specification to experimentation: A software component search engine architecture. *Component-Based Software Engineering*, pages 82–97, 2006.

[24] A. Goderis, P. Li, and C. Goble. Workflow discovery: the problem, a case study from e-Science and a graph-based solution. 2006.

[25] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby. Exemplar: EXEcutable exaMPLes ARchive. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 259–262. ACM, 2010.

[26] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100 –107, 1968.

[27] H. He and A. Singh. Closure-tree: An index structure for graph queries. In *Proceedings of the 22nd International Conference on Data Engineering, 2006. ICDE'06*, pages 38–38, 2006.

[28] L. Hérault, R. Horaud, F. Veillon, and J. Niez. Symbolic image matching by simulated annealing. In *Proc. British machine vision conference*, volume 90, pages 319–324, 1990.

[29] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto. Ranking significance of software components based on use relations. *IEEE Transactions on Software Engineering*, pages 213–225, 2005.

[30] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems (TOIS)*, 20(4):422–446, 2002.

[31] C. Kiefer, A. Bernstein, H. Lee, M. Klein, and M. Stocker. Semantic process retrieval with iSPARQL. *The Semantic Web: Research and Applications*, pages 609–623, 2007.

[32] S. Kirkpatrick, C. Gelatt Jr, and M. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671, 1983.

[33] M. Kunze and M. Weske. Metric trees for efficient similarity search in large process model repositories. 2010.

[34] V. Levenshteiti. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics-Doklady*, volume 10, 1966.

[35] W. Li and C. Clifton. SEMINT: A tool for identifying attribute correspondences in heterogeneous databases using neural networks. *Data and Knowledge Engineering*, 33(1):49–84, 2000.

[36] R. Lu and S. Sadiq. Managing process variants as an information resource. *Business Process Management*, pages 426–431, 2006.

[37] D. Lucrédio, R. de M. Fortes, and J. Whittle. MOOGLE: A model search engine. *Model Driven Engineering Languages and Systems*, pages 296–310, 2010.

[38] J. Madhavan, P. Bernstein, and E. Rahm. Generic schema matching with cupid. In *Proceedings of the International Conference on Very Large Data Bases*, pages 49–58. Citeseer, 2001.

[39] T. Madhusudan, J. Zhao, and B. Marshall. A case-based reasoning framework for workflow model management. *Data & Knowledge Engineering*, 50(1):87–115, 2004.

[40] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, UK, 2008.

[41] I. Markovic, A. Pereira, and N. Stojanovic. A framework for querying in business process modelling. *In Proceedings of the Multikonferenz Wirtschaftsinformatik (MKWI), M "unchen, Germany*, 2008.

[42] S. Melnik, H. Garcia-Molina, E. Rahm, et al. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Proceedings of the International Conference on Data Engineering*, pages 117–128. Citeseer, 2002.

[43] B. Messmer. *Efficient Graph Matching Algorithms for Preprocessed Model Graphs*. PhD thesis, University of Bern, Switzerland, 1996.

[44] R. Mihalcea and D. Moldovan. Semantic indexing using WordNet senses. In *Proceedings of the ACL-2000 workshop on Recent advances in natural language processing and information retrieval: held in conjunction with the 38th Annual Meeting of the Association for Computational Linguistics-Volume 11*, pages 35–45. Association for Computational Linguistics, 2000.

[45] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *Proceedings of the International Conference on Very Large Data Bases*, pages 122–133. Citeseer, 1998.

[46] H. Nottelmann and U. Straccia. Information retrieval and machine learning for probabilistic schema matching. *Information Processing & Management*, 43(3):552–576, 2007.

[47] E. Nowick, K. Eskridge, D. Travnicek, X. Chen, and J. Li. A model search engine based on cluster analysis of user search terms. *Library Philosophy and Practice*, 7(2), 2005.

[48] J. Paredaens, P. Peelman, and L. Tanca. G-log: A graph-based query language. *IEEE Transactions on Knowledge and Data Engineering*, pages 436–453, 1995.

[49] E. Rahm and P. Bernstein. A survey of approaches to automatic schema matching. *the VLDB Journal*, 10(4):334–350, 2001.

[50] M. RemcoDijkman and L. Garcıa-Banuelos. Graph Matching Algorithms for Business Process Model Similarity Search. In *Business Process Management: 7th International Conference, BPM 2009, Ulm, Germany, September 8-10, 2009, Proceedings*, page 48. Springer-Verlag New York Inc, 2009.

[51] T. Sager, A. Bernstein, M. Pinzger, and C. Kiefer. Detecting similar Java classes using tree algorithms. In *Proceedings of the 2006 international workshop on Mining software repositories*, page 71. ACM, 2006.

[52] D. Schmidt. Model-driven engineering. *IEEE computer*, 39(2):25–31, 2006.

[53] Q. Shao, P. Sun, and Y. Chen. Wise: A workflow information search engine. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 1491–1494. IEEE Computer Society, 2009.

[54] K. Smith, C. Bonaceto, C. Wolf, B. Yost, M. Morse, P. Mork, and D. Burdick. Exploring schema similarity at multiple resolutions. In *Proceedings of the 2010 international conference on Management of data*, pages 1179–1182. ACM, 2010.

[55] C. Spearman. The proof and measurement of association between two things. By C. Spearman, 1904. *The American journal of psychology*, 100(3-4):441.

[56] D. Stein, S. Hanenberg, and R. Unland. A graphical notation to specify model queries for MDA transformations on UML models. *Model Driven Architecture*, pages 77–92, 2005.

[57] T. Syeda-Mahmood, G. Shah, R. Akkiraju, A. Ivan, and R. Goodwin. Searching service repositories by combining semantic and ontological matching.

[58] Y. Tian and J. Patel. Tale: A tool for approximate large graph matching. *Data Engineering*, pages 963–972, 2008.

[59] A. Toulmé and I. Inc. Presentation of EMF compare utility. In *Eclipse Modeling Symposium*, page 2009, 2006.

[60] B. van Dongen, R. Dijkman, and J. Mendling. Measuring similarity between business process models. In *Advanced Information Systems Engineering*, pages 450–464. Springer, 2008.

[61] Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information. In *Proceedings of the 24th international conference on Software engineering*, pages 513–523. ACM, 2002.

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

[63] P. Ziegler, C. Kiefer, C. Sturm, K. Dittrich, and A. Bernstein. Generic similarity detection in ontologies with the soqa-simpack toolkit. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, page 753. ACM, 2006.