POLITECNICO DI MILANO
Dipartimento di Elettronica, Informazione e Bioingegneria
DOTTORATO DI RICERCA IN INGEGNERIA DELL'INFORMAZIONE

Textual and Content-Based Search in Software Model Repositories

PhD Thesis by: Bojana Bislimovska

Supervisor: Prof. Piero Fraternali

XXVI Cycle
January 2014

Abstract

Model-Driven Engineering relies on collections of models, which are the primary artifacts for software development. To enable knowledge sharing and reuse, models need to be managed within repositories, which requires the ability of their effective and efficient search for retrieving artifacts that meet the user's need. The search approaches should go beyond simple keyword search, considering the structural and hierarchical nature of models. In this way, an MDE developer should be able not only to search models via keywords, but also to sketch the idea he has in mind in his favorite language and retrieve all models that contain a similar design.

This thesis addresses the problem of designing search systems for repositories of models by examining two major categories: keyword-based and content-based search (also known as query-by-example). From these categories, one keyword-based approach, that employs classical information retrieval techniques, and two content-based approaches, that use representation of models as graphs, have been proposed and implemented. They are contrasted, with respect to the architecture of the system, the processing of models and queries, and the way in which metamodel knowledge can be exploited to improve search. A thorough experimental evaluation is conducted to examine what parameter configurations lead to better accuracy and to offer an insight in what queries are addressed best by each system.

Sommario

Il Model-Driven Engineering si basa su collezioni di modelli, ovvero sugli artefatti primari usati durante lo sviluppo software. Per permettere di condividere e riutilizzare conoscenza già a disposizione, i modelli devono essere gestiti per mezzo di archivi. Ciò richiede l'abilità di recuperare i suddetti modelli in modo efficace ed efficiente, così che i risultati di ogni ricerca incontrino i bisogni dell'utente. Di conseguenza, considerare approcci più complessi della semplice ricerca per keyword diventa essenziale: ogni ricerca deve tenere conto anche della natura strutturale e gerarchica dei modelli. Uno sviluppatore MDE dovrebbe essere in grado non solo di cercare modelli tramite l'utilizzo di keyword, ma anche di disegnare una bozza del modello che ha in mente, nel suo linguaggio preferito, e recuperare tutti i modelli che contengono un design simile.

Questa tesi affronta il problema di progettazione di sistemi di ricerca per archivi di modelli, esaminando due categorie principali: la ricerca keyword-based e la ricerca content-based (conosciuta anche con il termine "query-by-example"). Nel nostro lavoro proponiamo un approccio keyword-based (che impiega tecniche di information retrieval classiche) e due approcci content-based (che usano la rappresentazione di modelli come grafi). Gli approcci proposti vengono confrontati considerando le architetture dei sistemi, l'elaborazione di modelli e di query, ed il modo in cui la conoscenza sui metamodelli può essere sfruttata per migliorare la ricerca. Infine, presentiamo una valutazione sperimentale estesa, volta ad esaminare le configurazioni dei parametri che portano ad un'accuratezza migliore dei risultati e ad identificare le query che meglio vengono trattate da ogni sistema.

Acknowledgements

I would like to thank all the people who made this thesis possible. First, I would like to express my gratitude to Professor Piero Fraternali for giving me the opportunity to work with him and his team. I appreciate his expertise, guidance, motivation and continuous support that contributed to my academic experience.

I would like to thank Marco Brambilla and Alessandro Bozzon for their collaboration,exchange of ideas, motivation and encouragement.

I would like to thank Professor Geert-Jan Houben for his insightful comments.

I would like to thank Professor Tamer Özsu with whom I collaborated during my research stay at the University of Waterloo during 2012 and 2013, which was a great experience that opened an alternative direction for my research and professional development.

I would like to thank Carmen Vaca for being a good colleague and friend. I would also like to thank Eleonora Ciceri for helping me with the translation of the abstract in Italian.

I would like to thank my friend Viki for her friendship, numerous conversations and girls nights out.

I would like to thank the entire family Bislimovski for their support.

I would like to thank my sister Dragana for her love, encouragement, contagious joy and funny little jokes.

I would like to thank my uncle Jaroslav for his support and wise advice.

I would like to thank my boyfriend Güneş for his love, friendship, great support and being always there for me.

I would like to thank my grandparents Vera and Dragoljub, who unfortunately are not here anymore, for their devotion, love, support,for everything they did for me and for everything they taught me.

Last, but not the least, I would like to thank my parents Vojislav and Jovanka, without them all of this would have been impossible. I would like to thank them for their endless love, care, understanding and wise advice. To them I dedicate this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

The increased complexity and pervasiveness of software requires raising the level of abstraction, and automating labor-intensive and error-prone tasks to increase efficiency and effectiveness in software development [94].

An approach that advocates software abstraction through the use of models is Model-Driven Engineering (MDE), widely used in academia and industrial organizations across different domains. MDE promotes the use of models in any engineering activity as abstractions that provide a simplified or partial representation of reality, useful to accomplish a task or to reach an agreement on a topic. Model-Driven Software Engineering specifically considers software models, i.e., abstractions of the static or dynamic properties of a software system. Studies demonstrate that the benefits of MDE in industry are perceived in terms of quickly responding to change of requirements, of streamlining communication among stakeholders [94] thanks to more accessible organizational knowledge [63], and of improving the quality of code design and test case development [5].

The adoption of MDE in academic and business organizations resulted in an increasing number of models collections, stored in *model repositories* [44]. To name but a few: the MIT Process Handbook [93] contains over 5000 business process model entries; the AtlanMod Metamodel Zoos [9] provide a collection of more than three hundred metamodels; the ReMODD repository [106, 45] is collecting case studies, models and metamodels in different modeling languages. In the industry, several MDE tool vendors provide repositories that contain application and component models authored with their tools: examples include the WebRatio Store [123]; the Mendix App Store [90], the CodeCharge Studio marketplace [128], the Genexus marketplace [7], and the Outsystems Agile-Network component repository [98].

Reuse and sharing of software requires the ability of effectively retrieving artifacts that meet the user's need, which is the goal of *software search systems*. Besides the software repositories inside organizations, several on-line tools exemplify the state-of-the-art in sharing and retrieving code, e.g., *Google code*, *Snipplr*, *Koders*, and *Codase*[1]. In the simplest case, the user submits keywords, which are matched to the code, and receives as a response the programs that contain the search terms. Advanced systems offer more powerful functionality: 1) expressive query languages, e.g., regular expressions (in Google Codesearch) or wildcards (in Codase); search over syntactical categories, like class names, method invocations, and variables (e.g., in Jexamples and Codase); result restriction based on metadata (e.g., programming language, license type, file and package names). In the simplest systems the result set is a plain list of unranked hits, but more sophisticated interfaces offer classical IR-style ranking based on term importance and frequency or even composite scores compounding number of matches in the source code, recency of the project, number of downloads, activity rates, and so on; for example, in SourceForge users can receive results ranked by any combination of relevance of match, activity, date of registration, and recency of last update.

Model repositories are not yet as well developed and widespread as source code repositories. The latter enable code-level reuse and thus a reduction of development time and costs, and may improve software quality, as novice programmers can learn from the code produced by more experienced ones. The same advantages could be achieved in MDE, if repositories allowed for efficient retrieval of models relevant to the user's needs. Most industrial repositories offer rather elementary interfaces, where users can only search exact models by matching keywords against the model's description or explore the available content via taxonomical navigation and facets. These repositories do not incorporate the model structure in the search process, and they do not allow users to find models similar in structure or vocabulary to the query. More powerful approaches may foster early stage model reuse and promote the dissemination of modeling best practices across projects and development teams: for example, a developer wishing to implement a given application requirement may find in the company's repository models that solve similar tasks and reuse them entirely or some design pattern embed-

[1]Sites: http://code.google.com, http://www.snipplr.com, http://www.koders.com, http://www.codase.com

ded therein. This type of search can not be achieved using structured query languages since although they utilize model structure, they are limited only to exact search [26, 75]. Furthermore, structured search requires the user to learn a new query language in order to create queries. Model search approaches should exploit in more depth the main difference between source code and models, that is, *the high level, structural, and often visual nature of a model representation.* Ideally, an MDE developer should be able not only to search models via keywords, but also to sketch the idea he has in mind in his favorite language and retrieve all models that contain a similar design, properly ranked according to their relevance to the query. Therefore, similarity search techniques are essential to allow developers' needs be formulated in the same language in which solutions are expressed.

Model search is most useful during the initial phases of application development: the translation of software requirements into design artifacts and the transformation of coarse design models into detailed models. In the former case, requirements can be expressed concisely as keywords and used to find relevant models; in the latter case, coarse design models can be used for retrieving more detailed ones.

With the notable exception of Business Process Models repositories, where research has investigated similarity measures for the specific syntax and semantics of process models [122, 96, 103, 105], content-based search and multimodal search (e.g., keyword *plus* content based search) are still not the state-of-the-practice for model repositories.

1.1.1 Motivating Example

To better motivate the need for search in model repositories, let us consider the scenario depicted in Figure 1.1: *Alice* is a developer in a company adopting MDE for the design and implementation of Web-based information systems. *Alice* is currently working on the development of a novel customer management system and she has to address the requirement of allowing authentication of users through the OAuth[2] protocol. Let's assume her company already had several experiences in the development of system exploiting open authentication techniques; therefore, the model repository contains some project where this specific functionality has been designed already. *Alice* might be or not be aware of such previous work, but the reuse of existing models or the adherence to modeling patterns used in

[2]http://www.oauth.net

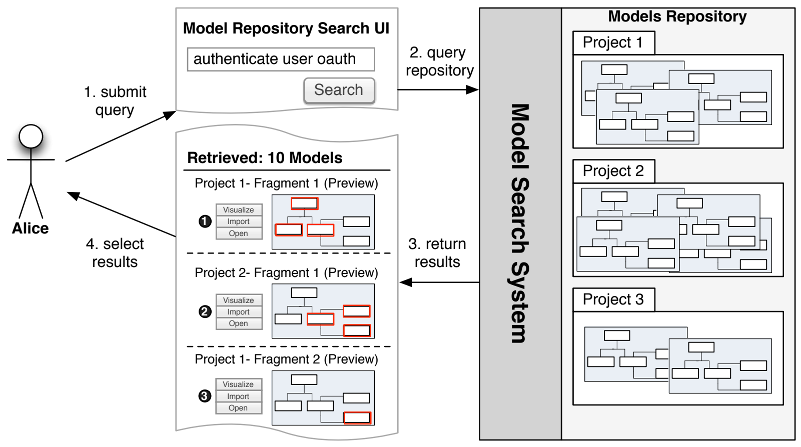


Figure 1.1: Example of interaction between a developer and a search system for model repositories: the user submits her query (1), which is applied upon the repository (2); in turn, the repository returns the matching project fragments (3), among which the user can select the one that fits better the new requirements (4).

previous successful projects will facilitate her job and improve uniformity of modeling style across the company. The goal of a model search system is to assist *Alice* in the retrieval of existing, similar solutions, and thus allow reuse and knowledge sharing.

Alice can express her information need with the textual query "authenticate user oauth", or she can draw a coarse modeling pattern in her preferred modeling language using corresponding model elements and labels that indicate user authentication through OAuth. The search system looks up the repository and returns a list of results at the appropriate granularity, as shown in Figure 1.1. The result set comprises concise previews of the retrieved model fragments; for a better understanding, results are ordered according to their relevance to the query and the parts of the model that match the query are highlighted. *Alice* might want to zoom in and visualize one of the results to better inspect the matching parts, or open the fragment in its original context. If she finds something useful for her current task, she might import the matching parts or the entire model in her workspace.

1.2 Research Objectives

The goal of this research is to study the implications of building search systems for software models expressed according to Domain

Specific Languages, so to *increase the reuse of modeling artifacts and promote the discovery of existing design patterns and the application of modeling best practices from previous projects.* We study two different scenarios of model search: *keyword-based search*, which proceeds in continuity with classical Information Retrieval approaches and source code search techniques; and *content-based search*, which introduces the query-by-example paradigm into model search. The illustrated research aims at addressing the following questions:

Q.1 How can we search model repositories in order to unlock their hidden value and allow efficient reuse of models?

Q.2 How can we adapt text- and content-based search techniques to model repositories, so to exploit metamodel knowledge and improve the quality of results?

Q.3 How do different text- and content-based search techniques behave in terms of performance under different technical configurations of their characteristic parameters?

Q.4 How do users perceive the quality of results retrieved with text- and content-based search?

1.3 Contributions

To address the above questions, the thesis overviews the requirements of model search and investigates keyword-based and content-based techniques for search of model repositories. Keyword-based and content-based search techniques are extended with the injection of metamodel knowledge in the search process, to test its effect on retrieval performance. Three approaches (one text- and two content-based) are implemented, configured and technically evaluated on a real world collection of 341 industrial models, with a panel of 10 queries. Models in the experimental repository are encoded in the Web Modeling Language (WebML) [28], a DSL for Web applications. Furthermore, the same text-based approach was adapted and configured for assessing its performance on a publicly available repository of 84 UML models (class diagrams) with a set of 20 queries.

Performance of the proposed text- and content-based search approaches is assessed by a technical evaluation based on a gold standard defined by experts. Moreover, the text- and one of the content-based approaches (A-star graph-based) applied on the WebML repository, are also evaluated with two user studies, which engaged 25

MDE practitioners in the subjective evaluation of utility and quality of search results. Different variants of the technical configurations of the two systems have been evaluated and compared. The user studies examine the relationship between the performance of the systems and the user-perceived utility of retrieved results.

The contributions of the thesis can be summarized as follows:

- We extend state-of-the-art methods for keyword-based search in order to incorporate metamodel-specific information. We show that augmenting the IR index with metamodel knowledge leads to a performance improvement with respect to conventional, metamodel-agnostic text-based IR techniques. The approach is validated on two different (WebML and UML) model repositories.

- We implement content-based search by means of graph matching; to do so, we extend standard techniques for sub-graph isomorphism (the A-star algorithm) by considering a formulation of the matching score function that takes into account metamodel-specific information. We also investigate how the locality of the match between the query and the project graph affects performance.

- We propose and implement a new content-based search method that employs graph matching. The method uses approximative representation of graph nodes as points in multidimensional space, obtained by multidimensional scaling. These points are used to build an index to allow efficient and scalable search. The method also uses neighborhood information for finding similar subgraphs of a project graph with respect to a query graph. The metamodel information is incorporated in the index, and it is considered while performing search, as well as in the scoring function used to rank the subgraphs.

- We evaluate keyword-based and content-based search systems with respect to retrieval accuracy (precision and recall), ranking accuracy, and stability of the results across different queries, using gold datasets created by experts.

- We report the results of a user study that assesses how model-driven practitioners appreciate keyword-based and content-based search considering the WebML repository.

1.4 Dissertation Organization

The thesis is organized in the following way:

- *Chapter 2* presents the fundamentals of search over model repositories, thus responding to question [*Q.1*], and it gives the characteristics of the WebML and UML modeling languages used as case studies to verify the proposed approaches;

- *Chapter 3* discusses the architecture and configuration of the keyword-based search system for WebML and UML models, addressing the text-based search part of question [*Q.2*];

  Chapter 4 focuses on content-based search, or more precisely graph-based search, describing the architecture and configuration of the A-star graph-based search and the multidimensional scaling graph-based search, replying to the content-based part search of question [*Q.2*];

  Chapter 5 presents the configurations and results of the experimental evaluation conducted on the proposed keyword-based and on the content-based search systems, thus addressing questions [*Q.3*] and [*Q.4*];

- *Chapter 6* describes the state-of-the-art for searching repositories of software artifacts considering source code, software components and models. The chapter also gives focus on the current algorithms for searching graph data;

  Chapter 7 brings out the conclusions and the directions for future work.

Chapter 2

Fundamentals of Search in Software Model Repositories

In MDE, *models* are used to formalize requirements, structure, and behavior of the addressed system; they comply with the syntax of a *modeling language*, which can be formalized as a *metamodel* [74]. Each model *element* has a *type* (i.e., a higher order concept) defined in the metamodel, and is related to other elements by means of typed *relationships*, also defined in the metamodel. One or more concrete syntaxes can be associated to a metamodel. The syntax can be either textual or graphical and defines the way in which the models are represented concretely. Elements and relationships in models are typically enriched with textual labels, provided by the model developer to describe some relevant domain properties or functions of the concept. During the development process, models are typically organized into *projects*, i.e., logical containers that aggregate models and artifacts of the same system or application domain; likewise, projects developed by the same organization are collected in *repositories*.

Model repositories are accessed primarily through *search*, i.e., the retrieval of relevant artifacts upon the expression of a user's need. Search is the main access method for retrieving models since it allows flexibility in the way the user need is expressed (keywords, or coarse model fragments), making the querying process easier and more straightforward. At the same time, model search can also consider the visual and structural nature of the model, offering the possibility to find both exact and similar models with respect to a user's need, which cannot be achieved with the structured query languages that perform only exact search.

In this chapter, we give an overview of the fundamentals of model search (Section 2.1), which addresses research question *[Q.1]*, and we present the features of the WebML and UML modeling languages, used as a case study to verify the proposed approaches (Section 2.2).

2.1 Fundamentals of Model Search

The search process can be schematically represented as a chain of four main steps, as shown in Figure 2.1.

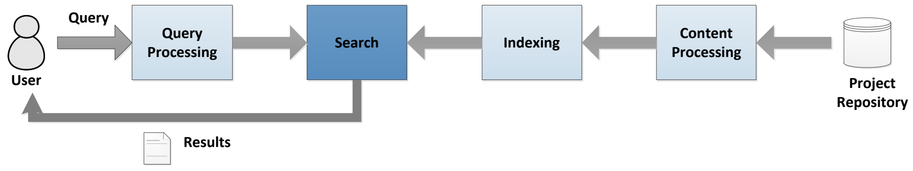


Figure 2.1: Main steps of a model-driven search process.

Starting from a repository of projects, the *Content Processing* transforms each model into a format suitable for efficient indexing and effective search. The *Indexing* step stores the processed models into persistent data structures that contain information amenable to search, typically encoded as *index terms* or as *index data structures*.

Users express their information needs as queries defined in a given format. Among the available query paradigms, we focus on two specific forms: (i) **Keyword-based** queries (also called **text-based queries**) are expressed as bag of words; users translate their information need from their abstract representation (e.g., "find all the projects that model the shopping cart operations of a book e-commerce application") into keywords or simplified phrases (e.g., "book shopping cart e-commerce"). (ii) **Content-based** queries are expressed as model fragments; users ask the system to "find a model like this" and thus formulate their queries in the same language in which the targeted models are expressed. This way of searching is also called **query-by-example**.

Queries are subjected to a *Query Processing* step, in which they undergo a transformation toward an *internal format*, which maps them to the same representation space as the index. For instance, a keyword query could be transformed into a set of stemmed words, or a model fragment could be mapped into a labeled graph.

The *Search* step inspects the index in order to (i) retrieve the models that match with the user query, (ii) rank the matching mod-

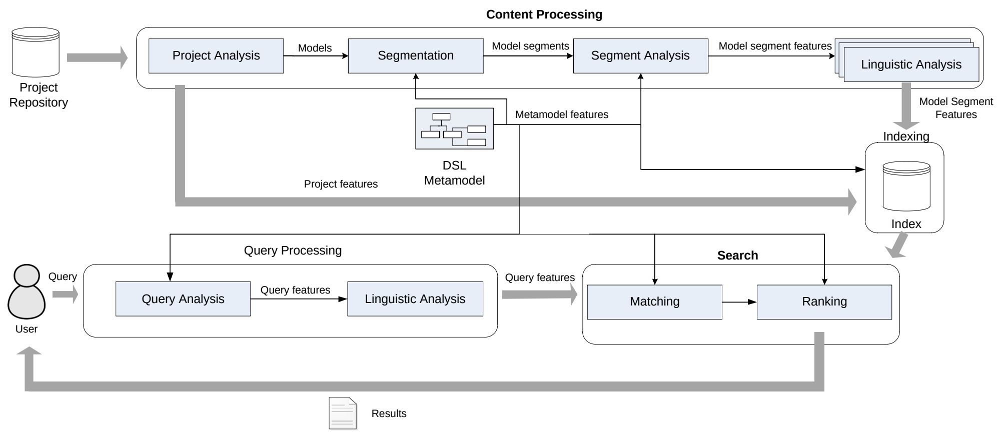


Figure 2.2: Architecture of a model-driven information retrieval system.

els according to their relevance with respect to the query, and (iii) return them to the user as a result set where more relevant hits are displayed in more prominent positions.

Figure 2.2 expands the view of the activities contained in each of the steps summarized in Figure 2.1. Activities can be *metamodel-dependent*, when they exploit knowledge defined in the metamodel (in Figure 2.2 they have an input data flow from the metamodel artifact), or *metamodel-independent*.

2.1.1 Content and Query Processing

The model search process, shown in Figure 2.2, requires both the repository projects and the queries to be properly analyzed to extract information relevant for indexing and searching. The *Query Processing* workflow comprises query analysis techniques that are the same as those for projects; therefore we can limit the explanation to the *Content Processing* tasks.

The *Project Analysis* task starts the analysis workflow by extracting general metadata, such as the project identifier in the collection, its name, authors, etc, useful for result presentation. The *Project Segmentation* activity splits each project into smaller units more suitable for analysis; the segmentation strategy is defined by the system designer, and can occur: (i) manually, by identifying project by project the most meaningful segmentation units; (ii) automatically based on metamodel-driven or collection-specific rules, which may take into account model types, concepts or relationship types, and element frequencies in the collection. For instance, UML class diagrams could be partitioned considering as segments the bottom elements of the package hierarchy.

Each segment is processed by a *Segment Analysis* task, which extracts relevant features for each model element contained in the segment, such as name, type, relationships with other elements, or any other property defined in the metamodel and relevant for search purposes. The extracted textual features might be normalized by applying metamodel-independent *Linguistic Analysis* transformations (e.g., language translation, tokenization, stemming, stop-word removal, etc.).

2.1.2 Indexing

The normalized features extracted from each element are the inputs to the *Indexing* step, as *index documents* [85]. The *index* stores the project metadata, the segment-to-project mapping, and a representation of the extracted model element features, optimized for storage and search purposes. The index can be organized according to one of the following options:

- **Flat index**: the index is structured as a single field which stores all the extracted features of an index document. A flat index does not allow the representation of model relationships, as the model structure cannot be enforced.

- **Multi-field index**: the index is divided into multiple fields, each storing a different subset of the indexable information. Each field can be searched separately, i.e., query matching can be restricted to the selected fields. A multi-field index may be used to encode metamodel information, by associating each field to features (e.g., normalized words) appearing in a distinct model concept or relation. In this way, a query could be restricted only to selected model concepts. Furthermore, each index field can be assigned a *weight* that quantifies its importance according to some a priori knowledge (e.g., the significance of the metamodel concept associated with the field).

- **Structured index**: the index is organized as a (semi) structured document (e.g., mapping each segment to a graph or to an XML document) so as to preserve the relationships among model elements. Structural elements can be assigned a *weight* that quantifies their importance.

Orthogonally to the adopted index structure, terms can be assigned a *term weight* that reflects their significance. Increasing the index complexity, from flat to multi-field, to structured indexes, gives more precise representations of projects and queries, to the price of more complex storage structures, query language, and match algorithms.

2.1.3 Search

The search workflow consists of two tasks. The *Matching* task finds the documents in the index that match the internal representation of the user's query. The matching technique applied depends on the index structure. For flat and multi-field indexes, matching occurs by verifying the presence of query terms in the index; in structured indexes, matching verifies if the query internal representation is at least partially contained in an indexed segment.

The *Ranking* task sorts the found matches with respect to their relevance to the query, calculated as a numerical *matching score*. The ranking techniques also differ according to the index structure. For flat and multi-field indexes, the score can be calculated using text-based similarity measures such as cosine similarity or TF/IDF [85]. For structured indexes, ranking is based on ad hoc structural similarity metrics. More details about the latter case are provided in Chapter 4.

2.2 Software Model Repositories

To make the architecture of Figure 2.2 concrete, it is necessary to instantiate it on a specific set of modeling languages and query paradigms. In this thesis, we focus on both text-based and content-based queries over repositories of models describing a specific class of applications – Web applications, in a single modeling language, i.e., the Web Modeling Language (WebML) [28]. Furthermore, we also applied the text-based approach on a repository of UML models. In the following sections, we give a brief overview of the WebML and UML languages.

2.2.1 WebML

WebML is a visual Domain Specific Language that supports the high-level specification of Web applications, from the perspectives of the composition and navigation of the Web front-end, and of the data accessed by it [27]; the language has a well-established industrial implementation and customer base [2].

The choice of WebML as the target language for experimentation is motivated by several reasons:

- the availability as the base for experimentation of a real-world industrial project repository created by professional developers.
- The generality and interest of the modeling domain (i.e., interactive application front-ends), in which WebML is just a representative of a family of DSLs that comprises several other languages with a similar purpose and structure, both in the academia (e.g., OOH [51], UWE [76], OOHDM [107], WADE [50]) and in the industry (e.g., Rational Web Application Extension [36], Mendix, CodeCharge and Outsystems); the interaction front-end modeling domain recently became an OMG standard [22].
- The visual nature of the language, which makes it well suited to the “query by example” paradigm of content-based search.
- The nature of the WebML metamodel, which comprises different families of containers and modeling elements, with a rich set of relationships.

It consists of a *Data Model*, representing the data content, and a *Web Model*, describing the pages that compose it, the topology of links between pages, the layout and graphic requirements for page rendering, and customization features for one-to-one content delivery. The Data Model describes the Web Site content 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 The main WebML constructs are *pages*, *units* and *links*, organized into *areas* and *site views*. A *site view* is a coherent hypertext, incorporating a well-defined set of requirements for a specific category of users. Site views can contain *Area*s, logical containers that group pages with a homogeneous purpose and can be nested recursively. *Page*s are contained in areas and site views, and represent the interface elements that are actually shown to the users. Site views, areas, and pages form the coarse structure of the front-end, which is then detailed by adding content and business logic components, called *Units*. There are two main types of units: *content units* and *operation units*. Content units are elements that express the content of a Web page, permitting different ways of content organization. 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.

Operation units denote operations on data or arbitrary business actions; they can be activated as a result of a link navigation, 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. WebML defines the following 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 allows 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.

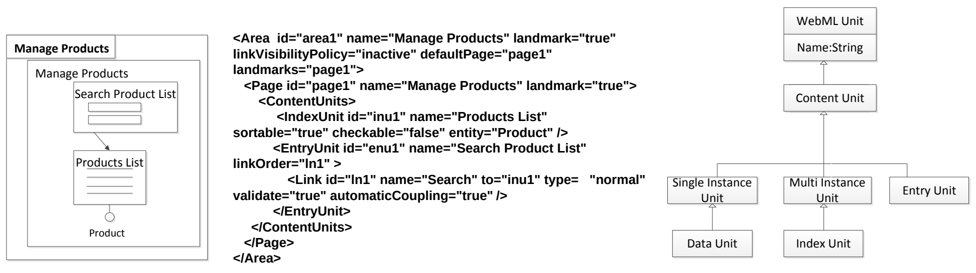
Operation units can be clustered into transactions, executed automatically, where either the whole sequence of operations is executed successfully, or the sequence is undone. Units are connected through *links* forming a hypertext structure. WebML defines the following link types:

- *Navigation link* allows navigating hypertext front-ends.
- *Transport link* passes parameters between units.
- *Normal link* is activated by the user in order to change page content or to move to another page.
- *Automatic link* is activated without user interaction, when the page that contains the source unit of the link is accessed.
- OK and KO links are outgoing links from every operation unit corresponding to the operation's success and failure.

Figure 2.3c contains an excerpt of the WebML metamodel taxonomy for content units: *Data Unit*s retrieve and present information about a single object; *Index Unit*s model the presentation of ordered sets of objects; *Entry Unit*s model Web input for data submission.

WebML models can be represented with a graphic notation or, equivalently, with an XML syntax. Figure 2.3a depicts an excerpt of a WebML model from an e-commerce application: the *Search Products* area contains a *Search Products* page where the user can enter data to search for a product; the search form is denoted by the *Search Product List* entry unit, while the returned product list is denoted by the *Products List* index unit; the link between the *Search Product List* unit and the *Products List* unit represents the navigation action of the user upon form submission, and it also specifies that the parameter required to execute the product search is passed to the index unit. Figure 2.3b contains the XML representation of the model fragment in Figure 2.3a, which comprises also the non displayed metadata of the model elements, e.g., their internal ID.

WebML models are designed by means of the WebRatio tool [2], or by any UML editor, using the WebML MOF metamodel; WebRatio has a basic in-memory project search facility, whereby the developer can execute keyword search within a single project. A repository of WebML models has been recently opened [123], which



```
<Area  id="area1" name="Manage Products" landmark="true"
linkVisibilityPolicy="inactive" defaultPage="page1"
landmarks="page1">
  <Page id="page1" name="Manage Products" landmark="true">
    <ContentUnits>
        <IndexUnit id="inu1" name="Products List"
sortable="true" checkable="false" entity="Product" />
        <EntryUnit id="enu1" name="Search Product List"
linkOrder="ln1" >
            <Link id="ln1" name="Search" to="inu1" type=   "normal"
validate="true" automaticCoupling="true" />
        </EntryUnit>
    </ContentUnits>
  </Page>
</Area>
```

(a) (b) (c)

Figure 2.3: Example of WebML model, (a) its XML representation (b) and an extract of the WebML metamodel (c).

can be browsed with an interface that organizes projects taxonomically and with tag clouds; basic keyword search is supported, with keywords matched in the textual description of projects.

This thesis explores both keyword-based search and content-based search over WebML repository and illustrates the techniques adopted in the indexing, analysis, and querying processes.

2.2.2 UML

Unified Modeling Language (UML) is a general purpose modeling language that allows specification, visualization, and documentation of models of software systems, including their structure, design and meeting application requirements in scalability, robustness, security, extendibility [97]. UML is built to model any type of application, hence, it can run on different hardware, operating systems, programming languages and networks. UML is formulated by using object-oriented concepts, which makes it suitable for modeling object-oriented applications. However, it can be also applied to model non-object oriented applications, as well as, Transactional, Real-Time and Fault-Tolerant systems. UML is an Object Management Group (OMG) standard which contains specifications for providing a way to share UML models between different modeling tools; it defines an infrastructure as a metamodel to provide a foundation for the UML superstructure that defines the UML model elements. The specification also defines OCL, a simple language for writing constraints and expressions for UML model elements [101].

UML was selected as another target language for testing the textual search due to the following reasons:

- General-purpose modeling language accepted as an OMG standard widely used in many domains.

- The visual nature of the language.
- The availability of an open source repository of real-world UML models.

In general, a UML model consists of one or more diagrams, each expressing different aspects of an application design. OMG standard defines thriteen diagrams divided into three groups. Static diagrams describe the static structure of the application; Behavioral diagrams represent the application behavior, and Interaction diagrams consider different interactional aspects. More specifically, structure diagrams include the following types of diagrams:

- Class diagrams use classes and interfaces to capture details about the entities that are constituent part of the system (application) and the static relationships between them.
- Component diagrams show the organization and dependencies involved in the system implementation at different levels of details with respect to a designer's decision.
- Composite structure diagrams are result of the increasing complexity of systems and are used to link class diagrams and component diagrams, demonstrating how different system elements are combined together to form complex patterns.
- Deployment diagrams reflect how different parts of the system are executed and assigned to different pieces of hardware.
- Package diagrams are special type of class diagrams, that focus on how different classes and interfaces are grouped together.
- Object diagrams use class diagrams syntax to show the connection between the instances and classes at a specific time instance.

Behavioral diagrams include the following types of diagrams:

- Use case diagrams capture functional requirements of a system, providing an implementation-independent view allowing focus on the designer needs.
- Activity diagrams capture the flow from one activity to the next, where one activity represents a behavior invoked as a result of the method call.
- State Machine Diagrams capture the internal state transitions of a model element. The element size can vary from a single class to an entire system.

Interaction diagrams, derived from the behavioral diagrams, include the following diagrams:

- Sequence diagrams consider the type and order of messages passed between elements.
- Communication diagrams focus on the objects involved in a particular behavior and the nature of the exchanged messages
- Timing diagrams are used for modeling real-time systems, and they focus on detailed timing specifications considering external interruptions and processing times.
- Interaction Overview Diagrams are a simplified version of activity diagrams, focusing more on the elements involved in performing an activity.

Since in this thesis, we experiment with a dataset of UML class diagrams, we will give their overview here. Class diagrams are a fundamental type of UML diagrams as they allow static modeling of applications through concepts, represented as classes, and the relationships between them. A class denotes a set of objects with common features [15]. Each class that is a constituent part of the UML diagram is characterized with a unique name, a set of attributes, representing a set of class features (details) and a set of operations, associated to a set of class objects. Each attribute has a type, which can be either a primitive one, such as integer, floating point, etc. or it can be a relationship to other complex objects. Operations define the way of invoking a specific behavior. UML specification makes a clear distinction between an operation, and a method which represents an implementation of an operation. The implementation can be provided by the class itself, or it can be inherited from a superclass. The flexibility of the UML specification allows the attributes and operations to be optional.

Since classes by themselves do not give a complete view of the system design, relationships among them need to be considered as well. UML provides several type of relationships [101]:

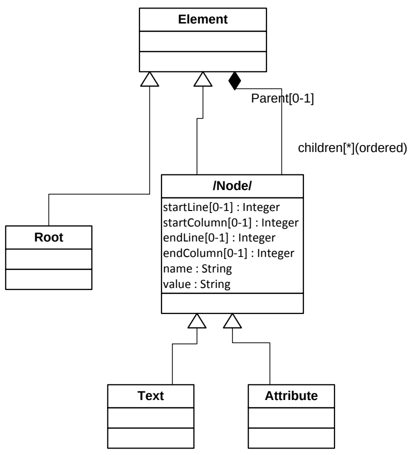
- Dependency is the weakest type of relationship where one class uses another class for some short amount of time.
- Association is a stronger type of relationship where one class preserves the relationship with another class for an extended time period.
- Aggregation is a stronger form of association that implies ownership.

- Composition is a strong type of relationship, capturing a whole-part relationship, i.e. the relationship that one class is part of another class. The part piece of the relationship is involved in only one composition relationship at a given time.
- Generalization expresses that the target part of the relationship is more general than the source part.
- Association class is used to represent complex relationships between classes. The association class has name and attributes, just as any other class.

UML also allows to define interfaces that declare properties and methods, without their implementation. A class is realizing an interface if it provides implementation of the declared operations and properties.

Figure 2.4 is an example of UML class diagram and its corresponding XMI representation, taken from the AtlanMod metamodel zoo, which describes metamodel of a XML file. XML file typically consists of one or more elements. The diagram differentiates between two types of elements: root, a parent of all the other elements, and a node. The node class that has some attributes, uses the composition relationship to express the parent-child relationship among XML elements, i.e., one element may contain multiple nodes. The text class, that gives the textual content of an element, and the attribute class, that provides additional information about elements are also nodes. Therefore, their relationships with the node class are modeled as generalization relationships.

In this thesis, keyword-based search is applied over UML repository.



(a)

```
<uml:Model xmi:version="2.1"
xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
xmlns:uml="http://www.eclipse.org/uml2/2.1.0/UML"
xmi:id="_gylnsOiaEd6gMtZRCjS81g" name="Metamodel">
  <packagedElement xmi:type="uml:Package"
  jxmi:id="_gylnseiaEd6gMtZRCjS81g" name="XML">
    <packagedElement xmi:type="uml:Association"
    xmi:id="_gylnv-iaEd6gMtZRCjS81g" name="A_Element_Node"
    memberEnd="_gylnvuiaEd6gMtZRCjS81g _gylnwOiaEd6gMtZRCjS81g">
      <ownedEnd xmi:id="_gylnwOiaEd6gMtZRCjS81g" name="parent"
      type="_gylntuiaEd6gMtZRCjS81g" isUnique="false"
      association="_gylnv-iaEd6gMtZRCjS81g">
        <upperValue xmi:type="uml:LiteralUnlimitedNatural"
        xmi:id="_gymOxuiaEd6gMtZRCjS81g" value="1"/>
        <lowerValue xmi:type="uml:LiteralInteger"
        xmi:id="_gymOx-iaEd6gMtZRCjS81g"/>
      </ownedEnd>
    </packagedElement>
    <packagedElement xmi:type="uml:Class"
    xmi:id="_gylns-iaEd6gMtZRCjS81g" name="Node"
    isAbstract="true">
      <ownedAttribute xmi:id="_gylnuOiaEd6gMtZRCjS81g"
      name="startLine" type="_gylnwuiaEd6gMtZRCjS81g"
      isUnique="false"/>
      <ownedAttribute xmi:id="_gylnueiaEd6gMtZRCjS81g"
      name="startColumn" type="_gylnwuiaEd6gMtZRCjS81g"
      isUnique="false"/>
      <ownedAttribute xmi:id="_gylnuuiaEd6gMtZRCjS81g"
      name="endLine" type="_gylnwuiaEd6gMtZRCjS81g"
      isUnique="false"/>
      <ownedAttribute xmi:id="_gylnu-iaEd6gMtZRCjS81g"
      name="endColumn" type="_gylnwuiaEd6gMtZRCjS81g"
      isUnique="false"/>
      <ownedAttribute xmi:id="_gylnvOiaEd6gMtZRCjS81g"
      name="name" type="_gylnw-iaEd6gMtZRCjS81g"
      isUnique="false">
      <ownedAttribute xmi:id="_gylnveiaEd6gMtZRCjS81g"
      name="value" type="_gylnw-iaEd6gMtZRCjS81g"
      isUnique="false"/>
    </packagedElement>
    <packagedElement xmi:type="uml:Class"
    xmi:id="_gylntOiaEd6gMtZRCjS81g" name="Attribute">
      <generalization xmi:id="_gymOwOiaEd6gMtZRCjS81g"
      general="_gylns-iaEd6gMtZRCjS81g"/>
    </packagedElement>
...
</uml:Model>
```

(b)

Figure 2.4: An Example of UML Class Diagram (a) and its corresponding XMI representation (b).

Chapter 3

Keyword-Based Model Search

In keyword-based search, the input query consists of a bag of keywords. Each project from the repository is transformed into a set of terms, used to form the index. Keywords are matched against the index, and a TF/IDF based measure is used to compute the rank of the matching elements in the result set. This chapter describes the keyword-based approach on two different model repositories, addressing the text-based search part of question *[Q.2]*. Section 3.1 illustrates the keyword search on WebML models, while Section 3.2 presents the keyword search on repository of UML models. Keyword search evaluation is provided in Chapter 5.

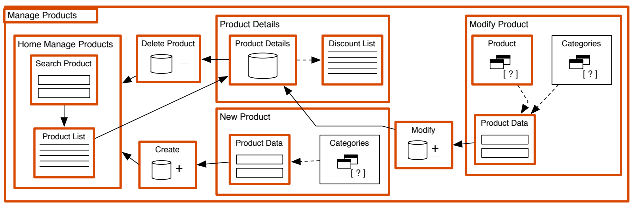
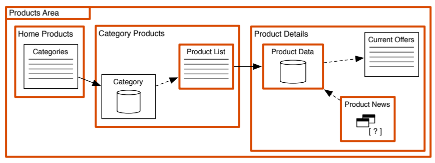
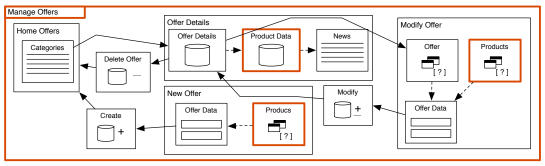
3.1 WebML keyword-based search

Illustrative example

Table 3.1 presents an example of textual query on a WebML model repository. Suppose the user is looking for a model that supports search and management of product lists in a Web-based system. He could formulate his information need as a keyword query like *Manage Search Product List*. Table 3.1 shows the top-3 results returned by our system in response to such query. Each of them consists of a model fragment (a WebML area), with decreasing matching score. The first result is a very relevant match, as the model fragment actually describes all the typical content management operations (creation, deletion, and modification) and contains a form for searching products. The subsequent matches are less precise: the second one misses some features, such as product search and updates; the third one only occasionally mentions products. The result set highlights

Table 3.1: Example of keyword-based query and top-3 results in WebML model repository(with respective score values).

Query	Manage Search Product List

Res. ID	**Model**	**Score**
Result 1		3.9799
Result 2		2.2482
Result 3		2.2896

the model elements that contain at least one of the search keywords and the match score is computed based on the number of matching and non-matching model elements.

Content Processing

The *Project Analysis* activity extracts only the project identifier, used to reference (at retrieval time) segments produced by the same project, and the project name, used for result presentation. *Segmentation* is performed by using a metamodel-driven rule that considers *Area*s as segmentation units. Recall that WebML areas are logical containers of pages with similar purpose, thus guaranteeing a good degree of functional cohesion. For each resulting segment, the *Segment Analysis* task extracts the *name* attribute for each area, page, unit, and link; this attribute has a special role, because it represents an external label defined by the developer and used by the code generator to produce the rendition of the front-end (e.g., a menu item, a link anchor or button text, the heading of a page fragment, or a page name) and thus carries a high relationship to the semantics of the model element it denotes, and to the application domain where the element is applied. A reference between the extracted term and the originating model element type is also kept, to be used later in the indexing step. Finally, the *Linguistic Analysis* task tokenizes the text, removes highly frequent words that bear little information, and stems the remaining words, creating the terms to be stored in the index.

Indexing

To explore how the injection of metamodel information in the index impacts the retrieval performance of keyword-based search, two types of indexing strategies for WebML projects have been exploited, both based on a multi-field index:

- *Metamodel-independent* strategy: for each project, the index comprises two fields: one field (*Area Name*) is reserved for the area name, and the second field (*Area Content*) contains the index terms extracted by the *Content Processing* step. An additional auxiliary field, used only for result presentation, contains the identifier of the project to which the indexed area belongs. Figure 3.1a shows an example of WebML area, and Figure 3.1b the corresponding indexed representation in the metamodel-independent strategy.

- *Metamodel-dependent* strategy: the index field structure is the same as in the previous case, but a weight is added to each term based on the metamodel concept it comes from. Weights are configured manually offline by their fine-tuning. As a result, in the experimental evaluation chapter we show several weight configurations that give the most distinctive results. Figure 3.1c shows the model fragment of Figure 3.1a indexed according to the metamodel-dependent strategy, where the numerical values appended to textual terms are the applied weights.

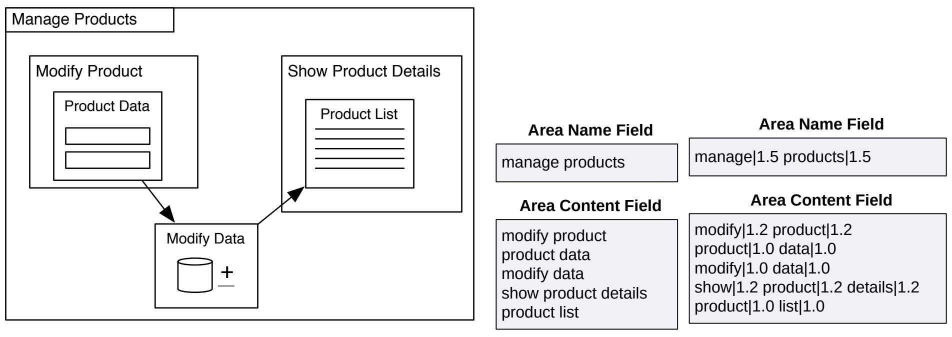
Query Processing

The query is a bag of keywords, subjected to the same linguistic analysis pipeline performed on the projects.

Search

As for indexing, a metamodel-independent and a metamodel-dependent indexing and ranking approaches are employed.

The *metamodel-independent* approach uses the classic TF/IDF measure of IR [85], which combines the frequency of a query term



(a) (b) (c)

Figure 3.1: Example of WebML model(a) and different text indexing techniques: Metamodel-independent indexing (b) and metamodel-dependent indexing (c).

in a document and its inverse frequency in the document corpus, so to penalize terms that occur frequently in a document and boost terms that occur rarely in the entire collection. The total TF/IDF score for a query and a document is computed as a sum of the scores of each query term. The total score is used to produce the ranking of the documents with respect to a given query, with higher score documents ranking higher in the result list.

In this work we propose a *metamodel-dependent* extension of TF/IDF that incorporates metamodel knowledge into a new parametric weighting term mtw, as reported in Equation 3.1:

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

where:

- q is a query, d is the indexed document (a WebML *Area*, in this experimental setting), and t is a term from the query q;
- $tf(t,d)$ is the *term frequency*, i.e., the number of times the query term t appears in the document d;
- $idf(t)$ is the *inverse document frequency* of t, i.e., a value calculated as $1 + \log \frac{|D|}{freq(t,d)+1}$, which measures the informative potential of the term in the entire document collection by calculating the ratio between the number of documents and the frequency of the term in the considered document; as a result, rare terms in the collection are considered more relevant than frequent ones;
- $mtw(m,t)$ is the *Model Term Weight* of a term t, i.e., a metamodel-specific boosting value that depends on the concept m contain-

ing the term t. For instance, in the example of Figure 3.1c, the weight for a term t associated with a *Page* element is set to be higher than the weight given to terms coming from other elements: $mtw(page, t)$ is set to 1.2, whereas mtw for all the others WebML concepts is set to 1.0.

3.2 UML keyword-based search

Illustrative example

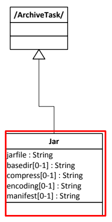
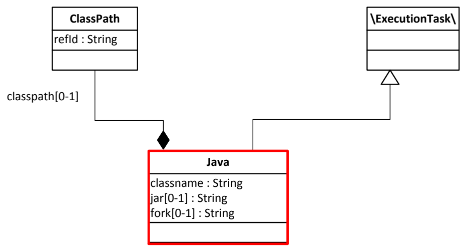
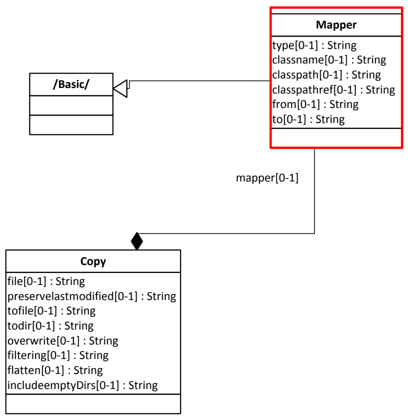
Analogously, we show an example of text query over the UML model repository. The user is searching for class diagrams that model packaging a program into jar file which has a manifest that contains a classpath. This might be expressed in keywords as *Jar Manifest Classpath*. The top-3 results that the system retrieves upon execution of this keyword query is given in table 3.2. The found matches within a project represent a single class, highlighted in table 3.2. The project ranked first has a match with respect to the query in the class name (Jar) and in two of the class's attributes (jarfile and manifest). The project ranked second has matches only in the two attributes' names (classname and jar) with two of the query's keywords. Finally, the project ranked third has matches only in attributes' names (classname, classpath and classpathref) that match only one keyword.

Content Processing

The content processing is performed in the same way as in the case of WebML models, except the *Segmentation* and the *Segment Analysis* task. Two types of *Segmentation* are applied. The first type considers the entire project as a segment, while the second type uses classes as segmentation units since they represent the main building element of the class diagrams. The *Segment Analysis* phase takes from each segment the name of each class, attribute, relationships's opposite end, the project id and name to which the class belongs to. The segment analysis also preserves information about the type of each extracted model element (class, attribute, relationship)and the type and multiplicity of each relationship, which are used in the indexing phase. *Linguistic analysis* is applied on the extracted text like for the WebML models.

Table 3.2: Example of keyword-based query and top-3 results in UML model repository (with respective score values).

Query	Jar Manifest Classpath

Res. ID	**Model**	**Score**
Result 1	 /ArchiveTask/ Jar jarfile : String basedir[0-1] : String compress[0-1] : String encoding[0-1] : String manifest[0-1] : String	1.0348
Result 2	 ClassPath refId : String \ExecutionTask\ classpath[0-1] Java classname : String jar[0-1] : String fork[0-1] : String	0.8063
Result 3	 /Basic/ Mapper type[0-1] : String classname[0-1] : String classpath[0-1] : String classpathref[0-1] : String from[0-1] : String to[0-1] : String mapper[0-1] Copy file[0-1] : String preservelastmodified[0-1] : String tofile[0-1] : String todir[0-1] : String overwrite[0-1] : String filtering[0-1] : String flatten[0-1] : String includeemptyDirs[0-1] : String	0.2251

Indexing

For UML projects, the influence of segmentation and utilization of metamodel knowledge in the index is investigated, which results in four different indexing techniques:

- *Project metamodel-independent* strategy: The segmentation unit is the entire project, and the index is single-field flat index that consists of the project name, names of all the containing classes, names of their attributes, and names of the relationships' opposite ends. This index has no metamodel knowledge. Figure 3.2a represents this index for the model in Figure 2.4.

- *Class metamodel-independent* strategy: The segmentation unit is class. The index is multi-field and it contains one field for the project name and id, another field for the class name and id, and a third field for the names of the class elements, such as names of attributes and relationships. This type of index with no metamodel awareness is given in Figure 3.2b for the class *Node* from the model in Figure 2.4.

- *Class metamodel-dependent* strategy: It uses class as a segmentation unit and a multi-field index, as in the Class metamodel-independent strategy. Metamodel knowledge is injected by adding weights to each name which depends on the concept importance in the metamodel. The weights are assigned manually by fine-tuning as in the case of the WebML metamodel-dependent strategy, and the weight configurations are shown in the experimental chapter. Figure 3.2c represents class metamodel-dependent index for the model in Figure 2.4.

- *Class neighborhood metamodel-dependent* strategy: This index also exploits class as segmentation unit. Besides the information the class metamodel-dependent index has, this strategy considers also the neighboring classes, i.e. those classes connected through a relationship with the class being indexed. A set of penalty weights is assigned to each relationship (through fine-tuning), such that the weight of the imported concepts is obtained by multiplying the metamodel concept weight with the penalty weight depending on the relationship type and its multiplicity. In this way, the imported terms have lower weights with respect to the non-imported terms representing the same metamodel concept. An example index for the model in 2.4 is given in 3.2d.

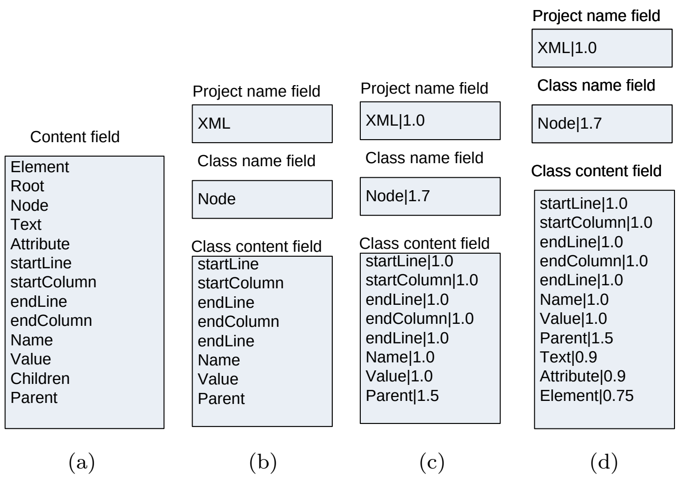


Figure 3.2: Different text indexing techniques for a UML model from Figure 2.4: Project metamodel-independent indexing (a) Class metamodel-independent indexing (b) Class metamodel-dependent indexing (c)and Class neighborhood metamodel-dependent indexing(d).

Query Processing

The query represents a set of keywords, and query processing is performed in the same way as for the WebML projects.

Search

All four indexing strategies are implemented and tested on UML projects. The two metamodel-independent approaches, i.e. Project metamodel-independent and Class metamodel independent techniques, employ the TF/IDF measure just like the metamodel-independent approach for WebML, while the two metamodel-dependent approaches (Class metamodel-dependent and the Class neighborhood metamodel-dependent strategies)use the same extended TF/IDF measure that considers the metamodel information for the case of WebML models. The only difference is that the indexed document is an entire project or a class, depending on the segmentation.

Chapter 4

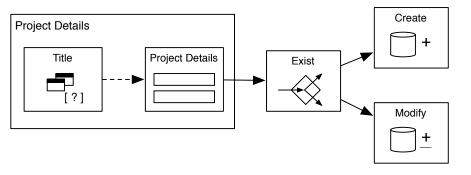
Graph-Based Model Search

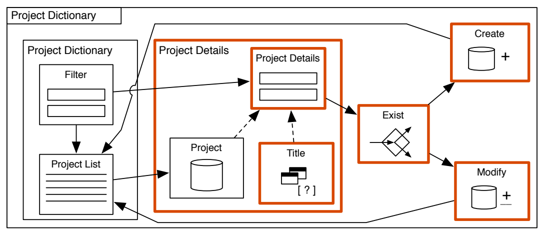
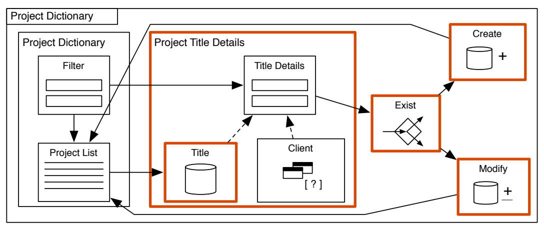
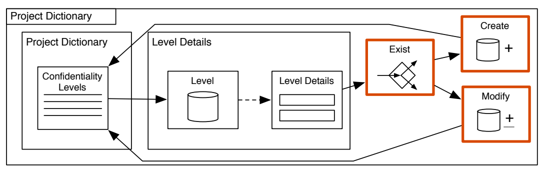
In content-based model search, queries are expressed as model fragments, and projects (or fragments thereof) are processed to preserve relationships among model elements. Models, including WebML models, can be represented conveniently as *graphs* [54, 103], which offer an abstract representation of the model elements and of their relationships. In content-based search, graphs serve as an internal representation of the models, and they are invisible to the user (the user is unaware of them). The query is specified as a model fragment, which is automatically transformed into graph. On the other end, the models from the repository are also transformed into graphs. The end result is represented as a ranked list of models similar to the query. In this chapter we show how project graphs are searched by using query graphs by employing two different graph-based approaches, thus answering the content-based search part of question *[Q.2]*. Section 4.1 proposes a solution for graph-based search using the A-star algorithm, while Section 4.2 proposes an efficient search for finding graph patterns by applying multidimensional scaling, representing graph nodes as points in multidimensional space. The evaluation of the approaches is given in Chapter 5.

Illustrative example

Table 4.1 shows a sample content-based query specified as a coarse WebML model; the query expresses a draft model consisting of a page and some operation units for searching a project by title and creating it (if not existing) or otherwise updating it. The top-3 results are shown; each result consists of a model fragment (a WebML area): the first one is a very precise match, where both structure and textual information fit with the query; the subsequent matches have lower scores because of the decreasing number of matching el-

Table 4.1: Example of content-based query and top-3 results (with respective score values).

Query	

Res. ID	**Model**	**Score**
R1		0.906
R2		0.847
R3		0.828

ements, either due to the imperfect structural overlap of the query and project models or to mismatches in the labels of the model elements. Note a difference with respect to the matches obtained for keyword-based search, exemplified in Table 3.1: in the text-based case, some matches appear only because a label in the model element matches a keyword in the user's query; in the content-based case, matches must adhere both to the textual content and to the structure of the query: for example, in the top result listed in Table 4.1 the Project Dictionary area is not part of the match, even if it contains the word *project* that is part of the query, because it does not correspond to any element appearing in the user's query.

4.1 A-star Graph-Based Search

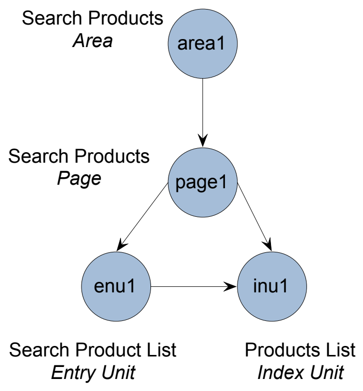
A-star graph-based search uses the A-star algorithm which is a classical solution to the problem of finding graph similarity between a query and a project graph. More specifically, the search verifies if a project graph contains a subgraph similar to a query graph with respect to the similarity of their corresponding nodes and edges. In this thesis, we apply the A-star graph-based search to WebML model graphs considering also information from the WebML metamodel, i.e., the type of each metamodel concept present in the model.

Content and Query Processing

The WebML models from the repository are first split into areas, as in the case of keyword-based search described in Chapter 3. Subsequently, they are translated into directed labeled graphs, according to a mapping proposed in this work. The resulting graphs are used to build the index. Since content-based queries are also WebML models, they can be transformed in the same way as the projects.

A WebML graph is a triple $g = (N, E, L)$, where N is a set of nodes, E is a set of edges, and L is a set of labels representing metadata about the nodes. Each WebML element maps to a graph node, identified with the same XML ID and annotated with its *name* and metamodel *type*. Therefore, each graph node is associated with a pair of labels l_N, l_T, that represent the *name* and the *type* label of the corresponding WebML construct. Two types of relationships between the WebML elements are mapped into graph edges: (i) *containment* relationships, which connect container elements (e.g., site views, areas and pages) with the elements they comprise; for example, a containment relationship exists among an area and all the pages contained in it. And (ii) navigational *Link*s, which model the navigation between pages and the functional dependency (i.e., parameters) between units. For each link, an edge that connects the nodes mapping its source and destination unit is added to the graph.

Figure 4.1a shows the graph representation of the WebML fragment in Figure 2.3a: the *Search Product* area (id = area1), the *Search Products* page (id = page1), the *Search Product List* entry unit (id = enu1) and the *Products list* index unit (id = inu1) are mapped to nodes labeled with the *name* attribute and the *type*, and identified with the same ID of the corresponding WebML elements. The edges connecting *area1* and *page1*, *page1* and *enu1*, and *page1* and *inu1* represent containment relationships in the original



(a)

```
<graphml>
  <graph edgedefault="directed">
    <node id="area1">
      <name>Search Products</name>
      <type>Area</type>
      <occurence>1</occurence>
    </node>
    <node id="page1">
      <name>Search Products</name>
      <type>Page</type>
      <occurence>3</occurence>
    </node>
             ...

    <edge id="edge inu1">
      <source>page1</source>
      <target>inu1</target>
    </edge>
    <edge id="edge link ln1">
      <source>enu1</source>
      <target>inu1</target>
    </edge>
  </graph>
</graphml>
```

(b)

Figure 4.1: Pictorial (a) and XML (b) representations of the graph corresponding to the WebML example of Figure 2.3.

model; the edge that connects *enu1* with *inu1* represents the link connecting the two units. Figure 4.1b shows the equivalent XML representation of the graph in Figure 4.1a.

Differently from the case of keyword-based search, no linguistic analysis is performed on the text extracted from the WebML model. This is justified by the mechanism used for query-to-project matching, described in details in Section 4.1, which exploits a string similarity measure to compare graph nodes.

Search

As both projects and queries are represented as graphs, *search* is performed by verifying whether the query graph is contained in a project graph. In a query-by-example scenario, the query graph will be normally smaller than the project graph. Therefore the goal is to find whether the query graph *is a part* of the project graph; this graph matching problem can be tackled by computing *subgraph isomorphism* [25], i.e., an injective mapping that identifies a subgraph in the project graph that has corresponding nodes and edges in the query graph, preserving the graph structure and label equality constraints.

Subgraph isomorphism is known to be NP-complete [37]; however, query processing does not require finding an *exact* match between the query graph and a subgraph in the project, a case that

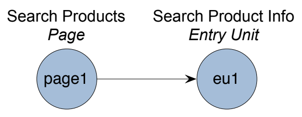


Figure 4.2: An example of a query graph.

would be very rare due to differences in model concept naming and linking. A sufficient objective is to find a subgraph in the project graph that is *equivalent*, or *similar*, to the query graph. Therefore, it is necessary to consider a heuristic algorithm that matches graphs by means of an approximate measure of *similarity* [25]. A classical solution to this problem exploits the A-star algorithm and the graph edit distance similarity measure [109, 53, 105], explained in the rest of this Section.

Graph edit distance A metric that computes the similarity of two graphs is the *graph edit distance*, defined as the minimum number of edit operations that transform one graph into the other [105]. Indeed, the less transformations are applied, the more similar two graphs are. Given a comparison graph G_1 (i.e., the query) and a compared graph G_2 (i.e., the project), the graph edit distance considers the following types of edit operations, namely:

- *Node substitution*: it substitutes (maps) nodes from G_2 that are *similar* to nodes from G_1, under an externally provided notion of node similarity.
- *Node insertion*: it inserts non-similar nodes from G_1 into G_2.
- *Node deletion*: it deletes from G_2 all non-similar nodes. A deletion from one of the two graphs can be treated equivalently as an insertion in the other graph.
- *Edge insertion/deletion*: it inserts into G_2 all edges that do not connect two similar (substituted) nodes of G_1; or, equivalently, it deletes from G_1 all edges that do not connect two similar nodes of G_2.

To exemplify the operations of the graph edit distance, we compare the query graph in Figure 4.2 with the project graph in Figure 4.1a. In this example, we consider that a node from the query graph is similar to a node from the project graph iff they have the same metamodel type and exactly the same name. A more flexible node similarity function will be introduced in a following paragraph. The node *"Search Products"* of type *"Page"* in the query graph is similar to the node having the same name and type in the project graph. The other query node *"Search Product Info"* of type *"Entry Unit"* has no similar nodes in the project; therefore, it is inserted into the project graph. All the other nodes of the project graph are non-similar and thus deleted. Since only one node from the query graph is similar to a corresponding node in the project graph, the edge outgoing from it in the query graph is inserted in the project graph, and all the four edges from the project graph are deleted. In summary, the query graph is obtained from the project graph as a result of 1 node substitution, 1 node insertion, 3 node deletions, 1 edge insertion and 4 edge deletions.

The *graph edit similarity*, defined in Formula 4.1, quantifies, in the $[0, 1]$ range, graph similarity by normalizing the operations considered in the graph edit distance:

$$G_{Sim}(G_1, G_2) = 1 - \frac{w_{nI} \cdot f_{nI}(G_1,G_2) + w_{eI} \cdot f_{eI}(G_1,G_2) + w_{nS} \cdot f_{nS}(G_1,G_2)}{w_{nI} + w_{eI} + w_{nS}}. \tag{4.1}$$

where f_{nI}, f_{eI} are the fractions of inserted nodes and of inserted edges, calculated as the ratio of inserted nodes N_i (edges E_i) *in both graphs* with respect to the total number of nodes (edges) *in both graphs.*

$$f_{nI}(G_1, G_2) = \frac{\mid N_i \mid}{\mid N_1 + N_2 \mid} \qquad f_{eI}(G_1, G_2) = \frac{\mid E_i \mid}{\mid E_1 + E_2 \mid}. \tag{4.2}$$

The values of f_{nI},and f_{eI} increase as the number of non similar nodes or edges grows.

The average distance of substituted nodes f_{nS} is defined as:

$$f_{nS}(G_1, G_2) = \frac{2 \cdot \sum_{(n_1,n_2)}(1 - sim(n_1, n_2))}{\mid N_s \mid}. \tag{4.3}$$

that is the sum of one minus the node similarities of all substituted nodes, normalized with respect to the total number of substituted nodes in both graphs. The average distance increases if the node pairs are less similar, and is 0 iff only identical nodes are substituted.

Formula 4.1 assigns to each edit operation a cost (weight), which gives the corresponding operation more or less influence on the result of the graph edit similarity computation. The constant values w_{nI}, w_{nS}, and w_{eI} range in the [0,1] interval and respectively represent the weights for node insertion, node substitution, and edge insertion. A higher value for an operation increases its contribution in the calculation of the distance between two graphs, i.e., the penalty incurred when one instance of that operation is applied to align the query and the target graphs. Weighting more insertion components of the graph edit distance emphasizes the dissimilarity due to graph topology; increasing the weight of the node substitution augments the penalty for considering equivalent nodes that do not match exactly. Weight values have been fine-tuned and the most distinctive results have been reported in Chapter 5.

As an example, the comparison of the query graph in Figure 4.2 with the project graph in Figure 4.1a results in $f_{nI} = 4/6 = 0.67$, because the total number of nodes *in both graphs* is 6 and the total number of node operations (deletions and insertions) is 4, while $f_{eI} = 5/5 = 1$, because the total number of edges *in both graphs* is 5 and the total number of edge operations (deletions and insertions) is also 5. The average distance of substituted nodes is $f_{nS} = \frac{2\cdot(1-1)}{2} = 0$, because the pair of substituted nodes has similarity 1 (they are identical). If the weights for this example are chosen to be, for example, $w_{nI} = 0.3$, $w_{nS} = 0.8$, and $w_{eI} = 0.5$, then the final graph edit similarity between the two graphs is $G_{Sim} = 1 - \frac{0.3\cdot0.67+0.5\cdot1+0.8\cdot0}{0.3+0.5+0.8} = 0.562$.

Node similarity. A central aspect in the evaluation of the similarity of two graphs is the calculation of the similitude of two nodes in order to determine whether they match, i.e., they can be considered similar, and thus be substituted (since they are interchangeable), instead of inserted. The node similarity can be computed by evaluating a distance function that considers the properties of the evaluated nodes. In our approach, differently from previous work, we adopt a distance function that considers both the metamodel type of the model element associated with the graph node and its textual label, as shown in Equation 4.4:

$$Dist(n_1, n_2) = \lambda\cdot stringDist(name_{n_1}, name_{n_2}) + (1-\lambda)\cdot typeDist(n_1, n_2) \tag{4.4}$$

$Dist(n_1, n_2)$ is calculated as the weighted linear combination of two distances, where:

- *stringDist* is a string distance metric, normalized in the [0,1] range, quantifying the similarity between the nodes' labels; our experiments, detailed in Chapter 5, compared the performance of two state-of-the-art string distance metrics, respectively the Levenshtein distance [80], and the n-gram distance [64].

- *typeDist* is the distance between two concepts in the metamodel, considered as a graph, normalized with respect to the maximum node distance in the metamodel graph.
- The parameter $\lambda \in [0,1]$ determines the relative importance of the name and type distance. $\lambda = 0$ takes into account only type contribution, while $\lambda = 1$ takes into account only the name similarity [1]

To exemplify the computation of the node distance, let us consider the *Search product List* and the *Products List* units of Figure 4.1a. According to the WebML metamodel excerpt of Figure 2.3c, the type distance between an *Entry Unit* and an *Index Unit* is 0.75 (because the distance between the two classes is 3 and the maximum node distance in the graph is 4), while the string distance $stringDist(``SearchProductList", ``ProductsList")$, calculated using the *Levenshtein* distance, is 0.58. Therefore, with $\lambda = 0.5$ the distance between the two nodes is 0.66.

Variations of the λ parameter value allow for different similarity evaluation scenarios. A high value of λ describes the situation in which a user considers two model elements similar only by looking at their names. For instance, the data unit "Product" would be considered "similar" to an index unit "Products", even if the former displays one object, whereas, the latter presents a list of objects. Conversely, a low value of λ would emphasize the semantic similarity, at a metamodel level, of model elements. In this case, an index unit "Product list" would be considered equivalent to an index unit named, e.g., "Offer List".

A-star algorithm The A-star algorithm is a method to compute subgraph isomorphism through graph similarity [113, 109]. Different variations exist; the version used in this work follows the template described in [92], and then modified and applied for searching repositories of business process models in [105]. Our approach is inspired to the latter work, but significantly extends it with metamodel-aware weights, distances, and parameters, which were not considered in the original algorithms.

The algorithm finds the optimal mapping between two graphs, using a best-first search of the solution space (the space of all mappings between the query graph and the project graph); it proceeds iteratively by searching the least-cost extension of a given initial partial graph mapping until a complete graph mapping is found; cost is

[1] As with the weight values, the λ parameter has also been fine-tuned, and the corresponding characteristic values are demonstrated in the experimental chapter.

computed with the graph edit similarity function, which drives the extension of the current partial mapping into the next expanded mapping that yields the maximal graph edit similarity between the query graph and the project graph.

The pseudo-code of ALGORITHM 1 illustrates the A-star procedure. It uses:

- the sets of nodes of the query graph (N_1) and of the project graph (N_2).
- A variable *open*, which is initialized with the set of all *allowed* mappings for an initial arbitrarily selected node n_1 of the query graph; the set of allowed candidate mappings is used to expand the current partial solution; a mapping is allowed if it contains node pairs with similarity above a given *threshold*, or node pairs where the query graph node is mapped to the conventional node deletion symbol ϵ.
- A variable *map*, which contains the current partial mapping solution having the maximal graph edit similarity *s(map)*; *s(map)* is evaluated as in Equation 4.1 by considering all node pairs contained in *map* as substituted, the remaining query nodes as inserted, the unmapped project nodes as deleted, and counting inserted/deleted edges accordingly.

A-star starts from an initially empty current mapping and a node n_q^1 in the query graph, and creates all the possible partial mappings $(n_q^1, n_p^{1_i})$ from this node to every node in the project graph. Additionally, an extra mapping with a dummy node ϵ is created, (n_q^1, ϵ), denoting the case where n_q^1 is deleted. The partial mapping $(n_q^1, n_p^{1_{i^*}})$ or (n_q^1, ϵ) with the maximal graph edit similarity is selected, and added to the current candidate solution mapping. Then, the algorithm proceeds with the next node from the query graph, and creates partial mappings with every other non-mapped node from the project graph. At each round, the current candidate solution mapping is expanded with the mapping of the nodes that produces the maximal graph edit similarity. The algorithm finishes when the current candidate solution mapping contains all the nodes from the query graph. The returned value is the maximal graph edit similarity for the query and the project graphs.

The best case complexity of the algorithm occurs when the nodes of the project and of the query graph have the same labels, and the query graph is an exact copy of the project graph. Therefore, for a query graph with m nodes and a project graph with n nodes, the

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 $open \neq \emptyset$ 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 ($n_2 \notin cod(map)$ and $sim(n_1, n_2) >$
      $threshold$) xor ($n_2 = \epsilon$) do
        $map' \leftarrow map \cup \{(n_1, n_2)\}$
        $open \leftarrow open \cup map'$
      end for
    end if
  end while
```

best case complexity is $O(n^2m)$. The worst case occurs when the query graph is very different from the project graph, both in terms of labels and structure; in such a case, many edit operations are necessary to transform one graph into the other, resulting in exponential complexity. For a query graph with m nodes and a project graph with n nodes, the worst case complexity is $O(nm^n)$. To reduce the search space, and limit space and memory requirements, a pruning rule is used: only nodes with similarity greater than the *threshold* parameter are allowed as candidate mapping pairs.

Let us consider the comparison of the query graph in Figure 4.2 and the project graph in Figure 2.3, assuming a *threshold* for node similarity of 0.7, and the parameter $\lambda = 0.5$, which gives equal importance to name and type similarity. In the first step, if we start with the query node *page1*, this node can form two possible partial mappings:

```
< ("Search Products:Page","Search Products:Page") >
< ("Search Products:Page","ϵ") >
```

The first pair is the mapping with the maximal graph edit similarity and thus is selected and expanded into new partial mappings that include the second query node *enu1*. The following ones are the possible mappings of cardinality 2:

```
< ("Search Products:Page","Search Products:Page"),
                ("Search Product Info:Entry Unit","Search Product List:Entry Unit") >
< ("Search Products:Page","Search Products:Page"),
                ("Search Product Info:Entry Unit","ϵ") >
```

At the second round, the algorithm selects the former complete mapping, which has the maximal graph edit similarity, and terminates, because all query nodes are mapped. The computed mapping specifies that in order to transform the query graph into the project graph, both query nodes are substituted, and the project graph nodes (“`Manage Products: Area`”,“`Products List Index Unit`”) are deleted (or equivalently inserted into the query graph). The edge in the query graph is substituted with the edge between the corresponding nodes in the project graph and the remaining edges from the project graph are deleted (or equivalently inserted into the query graph).

A-star algorithm with local search The original A-star algorithm can map query nodes to graph nodes arbitrarily positioned throughout the project. The cost of including in the match nodes that are far apart in the project graph is proportional to the number of edges that must be inserted to connect such nodes; the relative contribution of edge insertion in large graphs with many edges may be limited, and so A-star tends to accept matches where query nodes are associated with projects nodes far apart in the project graph. The locality of matches may impact the retrieval performance, which raises the issue whether highly connected matches are preferable to more distributed ones. To investigate the effects of locality constraints, we evaluate a variant of A-star, which attempts at boosting more cohesive matches by imposing an additional constraint for adding a node to the current partial mapping: only those nodes that are at the shortest distance with respect to already mapped nodes are used to extend the current mapping.

The pseudocode of the local search variant is listed in Algorithm 2. At the beginning, the set of partial mappings for the initial query node (n_1) contains all the nodes (n_2) from the project graph with similarity to n_1 greater than the threshold, plus the mapping with the node deletion symbol (ϵ). For every node n_2 in the project graph, denoted as $graph_2$, the shortest path to every other node in the graph is computed using the Dijkstra algorithm [40] and saved in the variable *path_set*. In the next step, the algorithm proceeds as the A-star algorithm, by selecting the partial mapping with the maximum graph-edit similarity. Then, the partial mapping is extended: the next query node is mapped to viable candidate project nodes; these are the project nodes with similarity above threshold and *positioned at the shortest distance*, defined as the minimum distance between an unmapped project node with similarity above threshold and an already mapped project node. When multiple paths exist with minimal length, all of them are considered and their source nodes treated as candidates. After identifying all nodes above threshold and within minimum distance, the algorithm expands the current mapping so to maximize the graph edit similarity and proceeds like the normal A-star algorithm.

Notice that, since the local search constrains the candidate matches, it happens that: (1) the number of matches computed by local A-star is smaller or equal than that of the original A-star; and (2) A-star and local A-star differ only when there are multiple matched node pairs between a query graph and the project graph.

A-star graph-based search limitations. One limitiation of the A-star graph-based search is that it does not use any indexing to allow for efficient search. Every node from the query graph is compared to every node from every project graph, which is an acceptable solution for smaller model repositories that do not contain large graphs. Since the complexity of the A-star algorithm itself directly depends on the number of nodes in the graphs, and the number of graphs in the repository, this approach is not efficient and it does not scale to larger repositories. Another limitation of the A-star approach is that it finds only a single match in a graph. Although it represents an optimal match, other potentially interesting matches in the same graph might exist, especially in large graphs, concerning that the user need is defined as a coarse model initially.
A greater difference in size between a query and a project graph, might result in low similarity values, due to the high number of graph edit distance operations that need to be performed, even if the query has all the matches. This raises the need of finding a solution that will find subgraphs in the project graph similar in size to the query graph.
Lastly, the A-star graph-based search considers only the model element type as a metamodel property, and incorporates it in the search part of the approach. Additional metamodel properties can also be taken into account while performing graph search.

4.2 Multidimensional Scaling Graph-Based Search

Multidimensional scaling content-based search allows search of all similar subgraphs in the project graph with respect to a query graph by taking the project graphs' nodes and representing them as points in multidimensional space. In this way, pruning of nodes is permitted during search, thus making the search process more efficient and

Algorithm 2 A-star algorithm with local search

```
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 $n_1 \in N_1$
  for all $n_2 \in N_2$ do
    $path\_set \leftarrow Dijkstra\_Shortest\_Path(graph_2, n_2)$
  end for
  while $open \neq \emptyset$ 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 $n_1 \notin dom(map)$
      $min\_path\_set \leftarrow \emptyset$
      $min\_length \leftarrow \infty$
      for all $path \in path\_set$ do
        if $source(path) \notin cod(map)$ then
          if $sim(n_1, source(path)) > threshold$ then
            if $target(path) \in cod(map)$ then
              if $length(path) == min\_length$ then
                $min\_path\_set \leftarrow min\_path\_set \cup path$
              else
                if $length(path) < min\_length$ then
                  $min\_path\_set \leftarrow \emptyset$
                  $min\_path\_set \leftarrow min\_path\_set \cup path$
                  $min\_length = length(path)$
                end if
              end if
            end if
          end if
        end if
      end for
      $source\_nodes \leftarrow extract\_source\_nodes(min\_path\_set)$
      for all $n' \in source\_nodes \cup \epsilon$ do
        $map \leftarrow map \cup \{(n_1; n')\}$
        $open \leftarrow open \cup map$
      end for
    end if
  end while
```

scalable with respect to the A-star algorithm graph-based search. Furthermore, the search is performed in a more realistic setting without the necessity of splitting the project graph, thus allowing for multiple matches in a graph with sizes comparable to those of the query graph. The approach also includes metamodel information in the graphs incorporating Data Model references and relationship types, which have not been considered in the A-star graph-based search.

Content and Query Processing

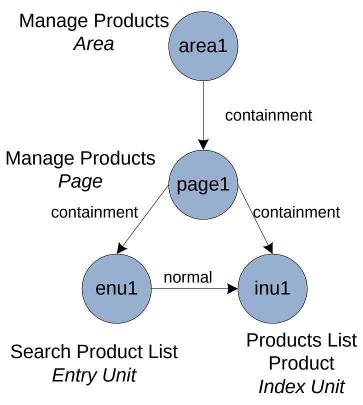
The rich syntax of WebML models can be captured more accurately by representing them as multi-labeled graphs whose nodes contain multiple labels. In this way, with respect to the previous representation, besides the name and the type of the model element, we also consider the WebML Data Model by extracting the entity or relationship associated to a model element. This information represents the entity or relationship to which the model element refers (in case the model element contains this kind of reference). For example, the container elements-site view, area and page do not contain any reference to the Data Model. Furthermore, the type of the links that connect model elements is associated to the edges' labels of the multi-labeled graphs, while the containment edge originating from the containment relationship is assigned a "containment" label. Queries, being model fragments (as in the A-star graph-based search), are also transformed as the project models into graphs.

More formally, a multi-labeled WebML graph G=(N,E) is a directed labeled graph, where:

- $N = \{n_1, , n_m\}$ is a set of nodes, such that each node n_k represents a WebML model element and is associated a list of labels $NL_k = (n_{id}, n_{name}, n_{type}, n_{entity,}, n_{relationship})$, where n_{id} is the element id, n_{name} is the element name, n_{type} is the element type, n_{entity} is the entity used by the model element, and $n_{relationship}$ is the relationship used by the model element. In case the entity or the relationship is not specified, the node will contain only the specified labels.

- $E = \{e_1, , e_n\}$ is a set of directed edges, such that each edge e_i from node n_k to node n_j represents a relationship between WebML model elements, and it has an associated label ELi_{type} that corresponds to a WebML relationship type. Two categories of WebML relationships are considered, described in Section 4.1, namely, containment relationships and links, diversifying further among the different types of WebML links defined in Section 2.2.1. As a result, ELi_{type} can have the following values: $ELi_{type} \in \{containment, normal, automatic, transport, OKlink, KOlink\}$.

Figure 4.3a shows the graph representation of the WebML fragment in Figure 2.3a according to the above-mentioned formal description. WebML model elements are mapped to nodes preserving



(a)

```
<graphml>
  <graph edgedefault="directed">
    <node id="area1">
      <name>Manage Products</name>
      <type>Area</type>
      <occurence>1</occurence> </node>
                ...

    <node id="inu1">
      <name>Products List</name>
      <type>Index Unit</type>
      <entity>Product</entity>
      <occurence>3</occurence> </node>
                ...
       <edge id="edge inu1">
         <source>page1</source>
         <target>inu1</target>
         <type>containment</type></edge>
       <edge id="edge link ln1">
         <source>enu1</source>
         <target>inu1</target>
         <type>normal</type></edge>
     </graph>
   </graphml>
```

(b)

Figure 4.3: Pictorial (a) and XML (b) representations of the graph corresponding to the WebML example of Figure 2.3.

the same id, labeled with *name* and *type*. The *inu1* index unit contains another label *Product* representing the entity to which the index unit refers to. The edges that represent the containment relationships are annotated with the *containment* label, while the edge representing normal link connecting *enu1* with *inu1* is annotated with the *normal* label, corresponding to the link type. The corresponding XML representation of the WebML graph is given in Figure 4.3b.

Considering the formal model-to-graph transformation provided above, we deal with multi-labeled graphs,i.e., graphs whose nodes are annotated with multiple labels. Therefore, we address the problem of graph-based search of models, as a problem of multi-labeled graph search. Single-labeled graph matching algorithms perform inefficiently when applied on multi-labeled graphs [126]. As a result, new approach that considers multi-labeled model graphs is proposed based on multidimensional scaling. Here we provide an overview of the approach which will be explained below in more details. The approach first considers project graphs and applies multidimensional scaling to transform their nodes as points in space considering node labels that represent specific metamodel features, denoted here as feature classes. These points are put in multidimensional grids (one grid per feature class) with respect to points' positions which are used to build the index to provide for efficient search. On the search side, the query nodes are also transformed into points that are used to query the grids. A query point retrieves all the points which are within a specified distance to that query point, searching only grid buckets that are within this distance. The retrieved points are collected for all the feature classes, and their corresponding nodes are tested by checking if their label's real distance is within the specified distance. This constraint is used to further prune results. The neighborhood of the filtered nodes is then further checked in order to keep only nodes that are in the proximity of each other. Finally, all nodes that passed the neighborhood constraint are used to locate project subgraphs in the larger project graphs that are compared to the query graph.

Indexing

In the case of multi-labeled graphs, each node contains multiple labels conforming to classes of features, where each label corresponds to one feature class. Furthermore, each node can by represented by a set of points in multidimensional spaces, where each point corresponds to a feature class label. For each feature class, the transformation of a node label to a point in multidimensional space is based on a node label's distances to the corresponding labels of other nodes. These computed distances help to find for a graph node, its "nearby" graph nodes with respect to a single feature class. However, using brute force to locate these "nearby" nodes is inefficient and consequently, there is a need of building an index which will allow for efficient search by placing the points corresponding to a feature class in a multidimensional grid. This procedure should be performed for every feature class such that the total number of grids corresponds to the number of feature classes.

In many fields, such as social sciences, information retrieval, geology and archaeology, there exist datasets where it is possible to quantify the similarity between their individual objects, called proximity data. Multidimensional scaling (MDS) is defined as a process of mapping high-dimensional objects of a dataset into low-dimensional space while preserving (dis)similarity relationships of objects, and it is typically used for data visualization [95]. However, in this thesis it is used as a basis for performing the indexing because it permits efficient representation of graph nodes as points in multidimensional space, which is justified in more detail in the following discussion. In MDS the (dis)similarity between objects is evaluated as a distance between objects in high dimensional space. The most common distance metric for MDS is the Minkowski distance based on representation of objects as points in multidimensional space, computed as: $d_L(x_i, x_j) = (\sum_{k=1}^{d} ||x_{i,k} - x_{j,k}||^L)^{1/L}$

More specifically, the most used instance of the Minkowski metric is the Euclidian distance for L=2 [95]. Out of many existing MDS algorithms,for e.g. [24, 32, 42], the ones that use the spring model, described below, perform better. The non spring model MDS algorithms have higher complexity ($O(N^3)$ and $O(N^4)$) and are non-iterative which means that they need to re-run enitrely if a small subset of additional data needs to be considered. Furthermore, non spring model MDS algorithms represent the data as a linear combination of the original dimensions which leads to a slower convergence to the optimal solution [29].

In the spring model, attractive and repulsive spring-based forces act between each pair of objects to improve inter-object separation (distances) in space [95]. The system is initialised by placing tha dataset objects in random positions, which makes the springs connecting these objects to be stretched or compressed. When the system is left by itself, the attractive and repulsive spring forces tend to move the system iteratively, until a state of an equilibrium is reached, which authors of [95] argue is a state of minimal energy. In each iteration the position of every object is improved with respect to a previous iteration. When the object positions do not change between consecutive iterations, a state of equilibrium is achieved.

For our dataset we choose the Chalmers algorithm, proposed in [29], since it has better complexity than other existing MDS approaches. The Chalmer's algorithm employs caching and stochastic sampling to perform each iteration of a spring model in linear time. Instead of doing all the possible pairwise $N(N-1)$ computations, the advantage of this algorithm is that it computes force calculations between each object o and the members of two sets, described below, whose size is bounded by a constant [29]. As a result, the computational cost in each iteration is linear with respect to N. The first set is stored as a list of neighbors of o that contains all the objects found so far to have lowest high-dimensional distance with respect to o. The second set is reconstructed in each iteration and contains a random selection of objects not already in the neighbor set. In each iteration, random objects are selected and each of them is tested to determine whether it has a high-dimensional distance lower than one or more of the current neighbors. In this case, the new object is swapped with an object from the neighbor set. Otherwise, the object is added to the second (random) set. In each iteration, the neighbor set becomes more representative of the most similar objects to an object. The forces of the system are computed as being proportional to the difference between the high-dimensional (real) distance and the low-dimensional (current) distance computed by their current positions in the space. The system needs to minimize the difference between these two distances. A loss function, known as *stress*, is defined, to indicate the amount of energy in the system, computed as a measure of the sum of squared errors of distances between objects [29]:

$$Stress = \frac{\sum_{i<j}(d_{ij} - g_{ij})^2}{\sum_{i<j} g_{ij}^2} \tag{4.5}$$

In Equation 4.5, d_{ij} is the real high-dimensional distance between objects i and j, and g_{ij} is the low-dimensional distance. For each object, three properties are maintained: position, velocity and force. In each iteration the force of an object is calculated which is then used to subsequently update the velocity, which is then used to update the object's position.

The number of force calculations required during one iteration of the algorithm is reduced from $n(n-1)$ to $n(V_{max}+S_{max})$, where V_{max} is the maximal size of the neighbor set and S_{max} is the maximal size of the random set and they are constant values, which means that the computational cost of an iteration is linear with respect to the number of objects n. The number of iterations necessary to produce a stable layout is commonly proportional to the dataset size, which makes the overall algorithm complexity $O(n^2)$ [95].

However, the gain in performance comes with a drop in quality because the Chalmers algorithm can only approximate the true distances but can never compute them exactly. While querying, pruning is used to eliminate all irrelevant points; nonetheless, there is no guarantee that all the relevant points will be retrieved.

In the following, we propose an indexing algorithm, with pseudocode given in Algorithm 3, where we applied the Chalmers algorithm on the multi-labeled nodes of the model graphs, denoted as $nodes$. In our model graphs, there are maximum three labels per node, each representing a different feature class: name, type and entity/relationship. These classes of features are forming the feature set, given as $features$ in the pseudocode. Note that the entity/relationship set of labels, representing the entity/relationship feature class, is smaller than the others since, not every model element contains these properties. Since a model element in general might have

Algorithm 3 Indexing algorithm

```
Require: $nodes$,$features$
  for all $f \in features$ do
    $grid[f] \leftarrow grid(f, dim, width)$
    $points \leftarrow Chalmers(nodes, f, dim, iterations, maxCacheSize)$
    for all $p \in points$ do
      $bucket \leftarrow findId(p.coordinates, dim, grid[f])$
      $insertPoint(p, bucket)$
    end for
  end for
```

a reference to entity or relationship of the Data Model, but not both of them, we consider the entity/relationship as one feature class because they are both parts of the Data model. For each feature class f a new grid, *grid*, is constructed with a fixed number of dimensions *dim*. Each grid consists of *dim*-dimensional buckets, and the bucket width parameter *width* determines the total number of buckets in the grid. The Chalmers algorithm is applied for each distinct feature class independently with its own parameters: number of iterations of the algorithm *iterations*, maximal size of the random set (denoted as *maxCacheSize* in our pseudocode) and number of dimensions, *dim*, to compute coordinates of points that represent the nodes. Once the positions of the points for each dataset are found, after the specified number of iterations, each point is used to find the bucket id, *bucket*, in the grid where the point will be inserted. The number of dimensions in the grid is identical to the number of dimensions of the points. All of the above-mentioned parameters are fine-tuned and the most characteristic results are reported in Chapter 5.

The complexity of the indexing algorithm is $O(n^2)$ (where n is the number of nodes in the *nodes* set) and it depends directly on the Chalmers algorithm complexity, since the number of feature classes in the outer for loop in the pseudocode of Algorithm 3 is fixed.

Besides the grid structure, two index structures are built,used as auxiliary structures in the search algorithm:

- Neighborhood index is an inverted index that keeps track of the neighborhood of each node, such that for each node, its 2-hop neighborhood considering both ancestors and descendants can be found.

- Project index is an inverted index on graph nodes based on the project they belong, that allows a node to find its corresponding project graph name during the search.

Search

The search part consists of finding all similar subgraphs in the project graphs with respect to a query graph, such that the dimensions of the similar subgraphs are comparable to the query graph. In a large graph, several subgraphs similar to a query graph might exist. Since a similarity between graphs is required through discovering correspondences between their nodes and edges, allowing node and edge mismatches, the graph-edit distance measure as presented in Section 4.1, is applied for its computation. However, the graph edit distance in this case is modified to consider multi-labeled graphs.

Graph-edit distance Graph-edit distance is computed between a query graph $Q(N_Q, E_Q)$ and a subgraph of the project graph $G_s(N_s, E_s)$ and it is defined as a minimum number of operations that transform the query graph into the project subgraph. The operations considered are:

- *Node substitution*: a node in the project subgraph G_s is substituted with a node in the query graph Q under the notion of node distance, calculated as specified below.

- *Edge substitution*: an edge in the project subgraph, is substituted with an edge in the query graph under the notion of edge distance, as specified below.

- *Node deletion*: it deletes all the nodes from the query graph that do not have corresponding similar nodes in the project subgraph.

- *Node insertion*: it inserts all the nodes from the project subgraph that do not have corresponding similar nodes in the query graph.

- *Edge insertion*: it inserts all the edges from the project subgraph that do not have corresponding similar edges in the query graph.

- *Edge deletion*: it deletes all the edges from the query graph that do not have corresponding similar edges in the project subgraph.

The graph-edit distance is defined, in the simplest way by summing the contributions of each considered operation:

$$dist(Q, G_s) = \sum_{(n_1,n_2)} dist(n_1, n_2) + \sum_{(e_1,e_2)} dist(e_1, e_2) + \mid nodeIns \mid + \\ + \mid nodeDel \mid + \mid EdgeIns \mid + \mid EdgeDel \mid \quad (4.6)$$

where $|nodeIns|$, $|nodeDel|$, $|EdgeIns|$ and $|EdgeDel|$ represent the number of node insertions, node deletions, edge insertions and edge deletions respectively. The contribution of the node substitution is given by the sum of the distances of all substituted node pairs, while the contribution of the edge substitution is given by the sum of the distances of all substituted edge pairs.

With respect to the graph-edit distance presented in our other graph-based approach in Section 4.1, here we consider the edge substitution operation since we introduce edge labels, in order to include more metamodel information in the search. Furthermore, the conditions for node substitution are different due to the additional node label representing the entity/relationship feature class. Also, we use a simplified version of the graph-edit distance formula which is not normalized.

In the following, to better show the difference between the two, we give an example of graph-edit distance computation between a query graph in Figure 4.2 and a project graph in Figure 4.3a, the same graphs used in the graph-edit distance example as in Section 4.1 with an exception that the edge in the query graph should have the label "containment". As in the previous example, we will consider that two nodes are similar iff they have the same metamodel type, exactly the same name and exactly the same entity or relationship, if the reference to the Data model is specified. In our multidimensional scaling approach, a more flexible notion of node similarity is employed. Therefore, as in the previous example, the *"Search Products"* node in the query graph is similar to the corresponding node having the same name and type in the project graph. Since the substituted nodes are identical, the distance of each of the individual labels representing the name and type feature classes is 0, which means that the corresponding node distance is 0. The other query node *"Search Product Info"* has no similar nodes in the project and, it is deleted from the query graph, i.e. $\mid VertexDel \mid = 1$. The remaining three nodes from the project graph are inserted into the query graph, i.e. $\mid VertexIns \mid = 3$. Since there is only one pair of substituted nodes, there are no candidates for similar edges that might be substituted. The only edge from the query graph is deleted,

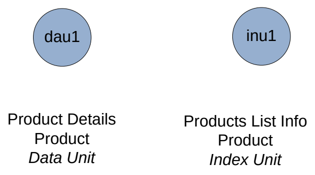


Figure 4.4: An example of two nodes for similarity computation.

i.e. $\mid EdgeDel \mid = 1$, and all the four edges from the project graph are inserted in the query graph, i.e. $\mid EdgeIns \mid = 4$. The final graph edit distance between the two graphs is $Dist = 0+3+1+4+1 = 9$. The bigger the distance is, the less similar are the graphs. If the distance is 0, the graphs are identical.

With respect to the A-star graph-based search that computes graph edit similarity in the range $[0, 1]$ by normalizing the graph-edit operations, here we use graph-edit distance, a metric opposite to the graph-edit similarity metric, without normalizing graph-edit operations. The graph-edit distance range between a query graph $Q(N_Q, E_Q)$ and a project graph $G_s(N_s, E_s)$ is $[0, |N_Q|+|N_s|+|E_Q|+|E_s|]$.

Node distance. Distance between a node n_1 in a query graph Q, and a node n_2 in a project subgraph G_s is defined as a mapping of their label sets, which depends on the distance of each individual label:

- *Name label distance*, $distName(n_1, n_2)$, computed by a string distance metric, as defined in Section 4.1.
- *Type label distance*, $distType(n_1, n_2)$, computed as type distance as stated in Section 4.1.
- *Entity label distance*, $distEntity(n_1, n_2)$, when specified, is determined by a string distance metric.
- *Relationship label distance*, $distRelationship(n_1, n_2)$, when specified, is determined by a string distance metric.

The string distance metric used for this part of the work in the experimental section is the Levenshtein distance. The node distance is determined by summing the distance contributions of each individual label (feature class):

$dist(n_1, n_2) = distName(n_1, n_2) + distType(n_1, n_2) +$
$+ distEntity(n_1, n_2) + distRelationship(n_1, n_2)$

The Figure 4.4 represents two nodes annotated with the labels representing name, type and entity feature classes, that we will use to demonstrate the node distance computation. According to the node distance specification, the distance of each individual label should be considered. The name labels of the two nodes are *"Product Details"* and *"Products List Info"*, and their distance computed by utilizing the normalized Levenshtein distance is 0.44. The type distance between *"Data Unit"* and *"Index Unit"* is 1, according to the WebML metamodel excerpt shown in Figure 2.3c where the distance between the two classes is 4 and the maximum node distance in the graph is 4. The entity label distance is 0 since both nodes have identical entity labels *"Product"*. Finally, the node distance is computed as the sum of the distances of each label: $sim(n_1, n_2) = 0.44 + 1 + 0 = 1.44$.

Edge distance. An edge e_1 in the query graph Q is similar to an edge e_2 in the project graph G_s if the nodes incident to e_1 are similar to the nodes incident to e_2, and these two edges have a similar edge label. In order to better explain edge distance, we introduce the *links* set that contains edge labels corresponding to all WebML link types, as described in Section 2.2.1, where $links = \{normal, transport, automatic, OKlink, KOlink\}$. Two edge labels are similar if they are identical, or if they both belong to the *links* set. Conversely, the distance of two edges $e_1 = (n_{1i}, n_{1j})$ and $e_2 = (n_{2k}, n_{2l})$, can be quantified as:

$$dist(e_1, e_2) = \begin{cases} 0, & \text{iff } n_{1i} \text{ corresponds to } n_{2k}, \\ & n_{1j} \text{ corresponds to } n_{2l},\ EL1_{type} = EL2_{type}; \\ 0.5, & \text{iff } n_{1i} \text{ corresponds to } n_{2k}, \\ & n_{1j} \text{ corresponds to } n_{2l},\ EL1_{type} \neq EL2_{type}, \\ & EL1_{type}, EL2_{type} \in links; \\ 1, & \text{otherwise} \end{cases}$$

This type of distance diversifies between the containment edges and the edges which represent links among model elements. This means that if two edges have the same label, their distance is 0. If both edges do not have an equal type, but their types belong to the *links* set, their distance is 0.5. If one of the edges has a type from the *links* set and the other edge is of type *containment*, then their distance is 1.

Search algorithm The search algorithm provides an efficient search of the index and pruning of candidate nodes based on their real distance and neighborhood constraints. In this way, only project subgraphs with localized matches, whose size is comparable to that of the query graph are considered for graph edit distance computation.

The querying part is carried out node by node, transforming each node object from the query graph into maximum three points corresponding to each of the name, type, and entity/relationship node feature class. The pseudocode of the Search algorithm is given in Algorithm 4. Note that we will use the terms node and its representing point interchangeably. The node transformation for each feature class f is performed using a modified version of the Chalmer's algorithm for querying, denoted as $ChalmersQuery$. This version of the algorithm takes only one query node instead of an entire node array, and it first positions randomly the query node according to the points present in the corresponding grid. In each succesive iteration, the query position is improved by minimizing the stress function with respect to the points indexed in the grid. The complexity of the Chalmer's Query algorithm is O(1), since the new point corresponding to the query node is positioned with respect to a constant number of pivot points as denoted by the neighbor set and the random set. Each point has the same number of dimensions as the points from the project graphs in the corresponding grids. The query point queries the corresponding grid to find buckets that are within the acceptable distance *distance*, a parameter specified by the user. Since the grids are independent, different acceptable distances can be assigned for each feature class. The query point is positioned in a grid bucket, and only buckets that are within the acceptable distance *distance* are searched for candidate points considering the relative position of the query point. In this way, an efficient pruning is achieved, since not all grids are checked, but only those within an acceptable distance. Once all the points are retrieved for each label type (feature class), a union is performed on all points for the three different feature classes, thus merging the results obtained from all the three grids. A final check is done with the *prune* function to verify if the returned nodes have also high-dimensional (real) distance which is within the acceptable distance, which further reduces the number of nodes. For the name and entity/relationship label, the Levenshtein distance, defined in Section 4.1 is used as a real distance, while the type label uses the metamodel type distance defined in Section 4.1. In this way, for each query node we obtain a set of candidate points, denoted as *real*, used in the next phase of the algorithm, thus making it more efficient, since it will be performed only on the set of the filtered nodes, and not the entire node set.

The next phase of the algorithm performs full outer join of the candidate nodes for each query node from the *real* set iteratively, such that in each iteration a new query node is considered, until all query nodes are examined. The join process is handled by the *join* function. In the first iteration, two candidate lists of different query nodes are checked for join. Each node from the first list is checked for join with every node from the second list. A node from the first candidate list joins a node from the second candidate list if they belong to the same project graph and if they are in the 2-hop neighborhood of each other. For each successive iteration, join is performed between a list of tuples of joined nodes, *joinedList*, and a candidate list corresponding to the next query node. Each tuple of already joined nodes is checked for join with each node from the list of candidate nodes associated to the next query node. A join is carried out if the node to be joined to an already existing tuple belongs to the same project graph and it is in the 2-hop neighborhood of at least one already joined node from that tuple. For checking the project graph and 2-hop neighborhood of each node, the corresponding project and neighborhood indexes are employed. Thus, every tuple contains only nodes that belong to the same project and that are in close proximity to each other. Full outer join handles the cases where a candidate node does not join already existing tuple, and in this case it forms a tuple consisting only of the candidate node, or an already existing tuple does not join any candidate node in an iteration, allowing a possibility for joining them in a successive iteration.

Each tuple resulting from the join actually represents a set of substituted nodes in a project graph which are within an acceptable distance with respect to the matching query nodes. The substituted nodes help to identify a subgraph in the larger project graph whose nodes are in the vicinity of each other and whose size is comparable to the size of the query graph. The found subgraph will be compared in terms of graph edit distance to the query graph. The set of nodes to be inserted in the query graph is identified from the intersection of their neighborhoods and it is found with the $findInsertedNodes$ function. The edges connecting corresponding substituted nodes in the project subgraph and query graph are examined for edge distance as defined previously. If the edges have distance less than 1, then they are substituted. The edge substitution is determined by the $findSubstitutedEdges$ function. All other edges in the project subgraph that are not similar or that connect inserted nodes (or connect inserted with a substituted node) are considered inserted edges, a condition checked by the $findInsertedEdges$ function. The substituted nodes and edges, and inserted nodes and edges are used to build the project subgraph using the *buildGraph* function.

Algorithm 4 Search algorithm

```
Require: $N_Q, feature$
  for all $n_q \in N_Q$ do
    $candidates \leftarrow \emptyset$
    for all $f \in feature$ do
      $queryCoordinates \leftarrow ChalmersQuery(n_q, f, dim, iterations, maxCacheSize)$
      $buckets \leftarrow findBuckets(queryCoordinates, grid[f], distance)$
      $points[f] \leftarrow findCandidates(queryCoordinates, buckets, grid[f])$
      $candidates \leftarrow candidates \cup points[f]$
    end for
    $real[n_q] \leftarrow prune(n_q, candidates)$
  end for
  $first \leftarrow true$
  for all $n_q \in N_Q$ do
    if $first$ then
      $joinedList \leftarrow real[n_q]\ join\ real[n_q + 1]$
      $first \leftarrow false$
    else
      $joinedList \leftarrow joinedList\ join\ real[n_q + 1]$
    end if
  end for
  for all $tuple \in joinedList$ do
    $insertedNodes \leftarrow findInsertedNodes(tuple)$
    $substitutedEdges \leftarrow findSubstitutedEdges(tuple)$
    $insertedEdges \leftarrow findInsertedEdges(tuple, insertedNodes)$
    $subgraph \leftarrow buildGraph(tuple, insertedNodes, substitutedEdges, insertedEdges)$
  end for
```

The complexity of the first phase of the search algorithm, which includes finding a set of candidate nodes for each query node, is $O(m)$, where m is the number of nodes in the query graph. In order to determine the complexity of the second phase of the search algorithm, we start from the most complicated *join* function. The *join* function performs join between two lists of candidate nodes corresponding to different query nodes in the first iteration, or between a list of tuples of joined nodes, *joinedList*, and a list of candidate nodes corresponding to a query node for all successive iterations. The *join* function actually checks for join each node from the first list with each node from the second list, or each tuple from the *joinedList* with each node from the candidate list respectively. Therefore, the complexity of the join function is $O(n^2)$ where n is the number of candidates for a query node. Finally, the overall complexity of the search algorithm which finds all subgraph matches for a query graph is $O(mn^2)$, where m is the number of query nodes in the outer for loop. This represents an improvement over the A-star graph-based search, whose complexity for a query graph and only one project graph is $O(mn^2)$ (without considering the entire set of project graphs).

If there exist nodes in the query graph which do not have a match in the tuple, then they represent the deleted nodes. Query graph edges that do not have a matching pair in the project subgraph are deleted. Deleted nodes and edges, besides the inserted nodes and edges, and substituted nodes and edges, serve in the graph edit distance computation where the query graph and the project subgraph are compared. The computed score is used to rank the project subgraphs based on the graph edit distance with respect to a query graph.

Chapter 5

Experimental Evaluation

This chapter presents the experiments on different model search scenarios constructed according to the techniques described in the previous chapters. Section 5.1 describes the experimental settings and the datasets; Section 5.2 specifies the metrics used in the evaluation; Section 5.3 technically evaluates the keyword- and content-based systems, analyzing performance under different system configurations using quality indicators based on gold standards created by a panel of experts, answering to research question *[Q.3]* announced in Chapter 1; Section 5.4 illustrates a user study, where WebML practitioners were asked to assess how much the proposed keyword-based and A-star graph-based methods could help them in the reuse of existing modeling artifacts, thus responding to question *[Q.4]*. Section 5.6 introduces potential threats to validity of the experimental evaluation and the approaches.

5.1 Experimental Settings and Datasets

5.1.1 Test bed

The experiments were performed on two repositories of software models, WebML and UML. The WebML repository is provided by WebRatio[1]; the company that develops the homonymous MDD tool for WebML modeling and automatic generation of Web applications. The repository contains 12 real-world WebML projects from different application domains (e.g., trouble ticketing, human resource management, Web portals, etc.). The projects are encoded as XML files conforming to the WebML DTD, and their domains and size are presented in Table 5.1. For the keyword-based and the A-star graph-based approaches, we segmented projects at the area level,

[1]`www.webratio.com`

which resulted in 341 areas. However, the multidimensional scaling approach uses segmentation at project level, considering projects in their entirety. As a result of the multidimensional scaling appplied on the project graphs, the name and the type grid contain 19012 points each, while the entity/relationship grid contains 8380 points. Fewer number of points in the entity/relationship grid is due to the fact that not every model element contains the entity or relationship feature as already explained in Section 4.2.

Table 5.1: WebML repository. Project ID, domain, and number of contained *areas*.

Project ID	**Domain**	**Number of *areas***
1	Administration	23
2	Human resource management	53
3	Call center web portal	56
4	Calendar management	3
5	Bank account management	58
6	E-commerce	15
7	Rent-a-car	2
8	Adminstration	30
9	Company intranet	58
10	Web portal	5
11	Candidate evaluation	24
12	Trouble ticketing	12

An evaluation set of 10 queries was built as follows: first, to ensure a good coverage of all the system features that we wanted to test in the experiments and the coherence between the evaluation set and the repository content, we defined several exemplary models, satisfying the following properties: (i) they implemented theoretical [28] and real-world WebML modeling patterns; (ii) they used a broad mix of the WebML metamodel concepts; and (iii) they were based on a vocabulary (of labels) consistently used in the experimental project repository. Out of this initial set, a group of three experienced model developers were consulted to select the 10 exemplary models which, in their opinion, better represented the typical user need of MDD developers in their everyday activities. Finally, the exemplary models were transformed into keyword-queries, by selecting as keywords all the significative labels; and into content-based queries, by mapping each WebML model into a graph as explained in Section 4.1.

The keyword-based search was also applied on a repository of UML models, consisting of 84 UML class diagrams taken from the AtlanMod zoos [9], a publicly available repository of metamodels. The class diagrams are actually metamodels which can be also described as models about a model, according to [8]. Main disadvantage of this dataset is that majority of the projects are small in size (not containing many classes) which makes it inappropriate for testing the graph-based approaches. However, to the best of our knowledge, this is the only publicly available repository of UML models that contains bigger collection of UML models. We tested keyword-based search on this dataset by considering the entire project as a segmentation unit and by employing segmentation at the class level.

A set of 20 queries was built for testing the UML repository, adopting a similar principle used in the creation of queries for the WebML repository. A set of exemplary models was constructed, considering the frequency of the terms present in the repository for labeling the model elements, as well as different types of relationships present in the repository. Note that UML class diagrams do not have diversity in terms of modeling patterns (and model elements) as WebML models have. From this set of models, the same model developers selected 10 of them as queries considering their typical user need, as in the WebML case. The queries were then transformed into keyword queries, selecting the labels of the model elements as keywords. This set of queries is defined at project level granularity. An additional set of 10 classes was selected from the exemplary models considering their attributes and relationships to represent queries at class level granularity, to test the case where segmentation is performed at class level. The second set of queries are transformed into keyword queries, considering the class name and its attributes as keywords.

5.1.2 Gold Standard Creation

The gold standard for the comparison of the retrieval methods (except the multidimensional scaling method) for both WebML and UML datasets, was constructed by manually assessing the extent to which each WebML area, or UML class diagram respectively, in the repository contained a model similar to those in the evaluation set; the three experts assigned a relevance score expressed in a tertiary scale where, (i) 0 relevance means no similarity, (ii) 1 means that some textual *xor* structural similarity exists, and (iii) 3 corresponds to a judgment of strong similarity (textual *and* structural). The final relevance was calculated as the average of the three judgments, rounded to the nearest integer. To reduce fatigue and learning bias, the evaluation tasks were spread over multiple days.

Figure 5.1 exemplifies the kind of judgements about query-area matches expressed by evaluators for the WebML dataset. The query (Figure 5.1(a)) looks for an area that implements a creation/modification pattern for new/existing products; the area shown in 5.1(b) contains a pattern that performs the same action, but using slightly different labels. Given that the query's structural pattern is present in the project, and there is also a partial textual similarity (the terms *product* and *title* are present both in the query and in the matching area), a relevance value of 3 qualifies this match. Figures 5.1(c) and (d) show examples of areas where the relevance with respect to the query is 1 and 0, respectively. As it can be noticed, the area with similarity 1 contains a pattern that verifies the existence of a *store* in order to be created or modified. This area has only structural pattern similar to the query, and no textual similarity. Finally, the project with similarity 0 has nothing in common with the query.

The gold standard dataset ranks for each query the areas, according to the average relevance score of the match, breaking ties with a deterministic rule.

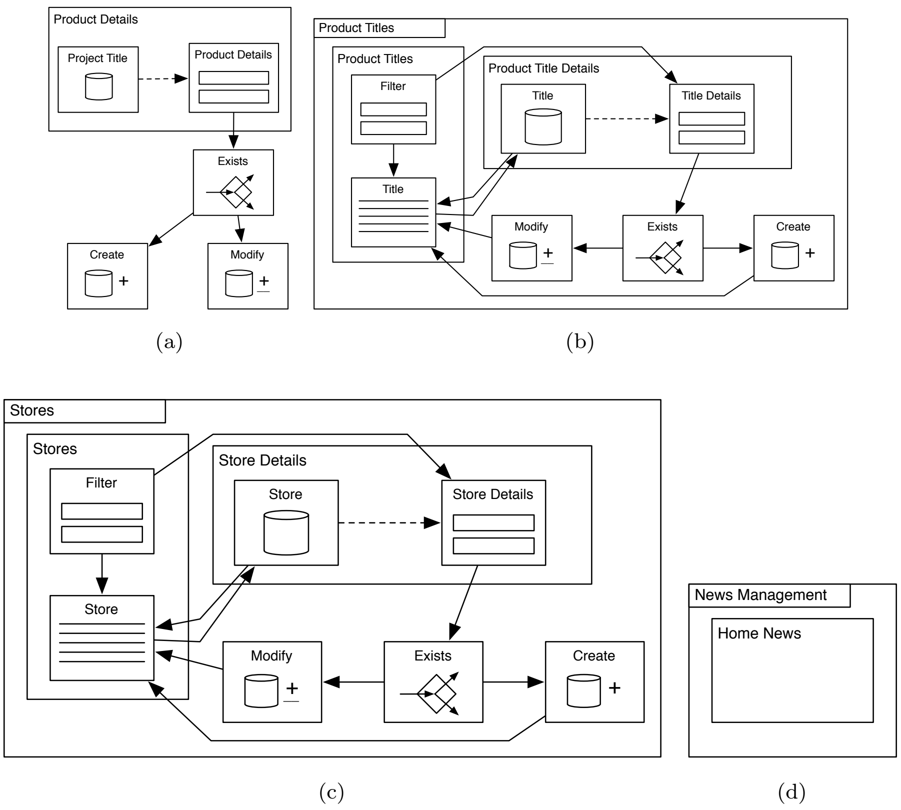


Figure 5.1: Example of a WebML query (a), area with similarity 3 (b), area with similarity 1 (c) and area with similarity 0 (d).

The multidimensional scaling graph-based method uses the same set of WebML queries and projects. However, since the method finds set of similar graphs in the entire set of WebML projects for a query, and it does not use the area as segmentation unit, but an entire project, the existing ground truth can not be used as it is in the evaluation. Moreover, building a new ground truth for the same set of queries considering the entire projects instead of the areas will make all projects equally relevant. Namely, since we are evaluating a similarity of a query with a project, due to the small size of the dataset in terms of number of projects, and large size of individual projects in terms of number of model elements, almost every project will be similar to every query. This means that the query-to-project graded relevance would disappear, which will flatten the results Therefore, we decided to adapt the existing ground truth in the following way: The relevance of each found subgraph from a project graph is evaluated by finding the relevance of each project node with respect to the area to which the corresponding model element belongs in the existing ground truth. Then, these relevances are averaged to obtain the overall relevance of a project subgraph with respect to a query graph. For example, if the found subgraph has three nodes, and the first node belongs to an area that has relevance 1 in the ground truth with respect to the query (meaning that the node's relevance is also 1), the second node has relevance 0, and the third node has relevance 3, the relevance of the project subgraph with respect to a query graph is the average relevance, i.e. 1.33. The relevances are computed for each result set individually and they are used as a ground truth for the result set in its evaluation.

5.1.3 Experimental scenarios

As a baseline for comparing the retrieval accuracy of both the WebML text-based and A-star graph-based approaches, we use randomly generated result sets. For the WebML dataset, a random result set is a sequence of areas randomly extracted from the projects in the repository, ordered randomly. The value of each performance indicator in the baseline case is calculated by averaging, for each query, the results of 10 random area extraction and ranking steps.

In the same way, two random result sets are generated for the UML case corresponding to the considered segmentations at project and class granularity. For the project granularity, projects were randomly extracted from the repository, while for segmentation at class granularity level, classes were randomly extracted from all classes belonging to the projects in the repository. The performance indicator values were calculated identically as in the case of the WebML repository.

We could not compare the performance of our implementation directly with existing works in model search for various reasons: some works are not metamodel-aware at all, and therefore cannot be ported to other modeling languages; some others assess their quality upon non public data sets; and finally, others do not provide publicly available systems to compare with.

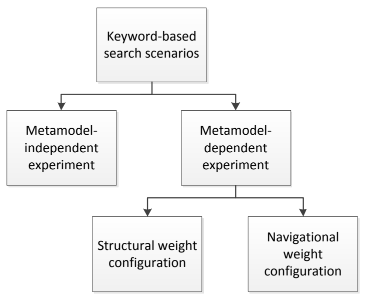
WebML Keyword-based search Figure 5.2a summarizes the evaluation scenarios for keyword based search. Experiments were conducted under a (i) *metamodel-independent*, and a (ii) *metamodel-dependent* configuration. The index of the metamodel-independent experiment contains equally weighted terms, regardless of the metamodel type of the element of origin. Conversely, the indexes of the metamodel-dependent experiments weight terms according to the category of the metamodel type of the element where the term appears.

We categorize the five WebML model primitives relevant for matching the query to the projects into *structural* and *navigational.* Figure 5.2b shows such classification: site views, areas, and pages represent mainly modularization constructs used to group more detailed elements; units and links embody the composition and navigation aspects of the user interface and denote the functions triggered by the user's navigation. The top-down order of elements in Figure 5.2b follows the element containment relations: siteviews contain areas, which in turn contain pages, which in turn contain units, connected through links.

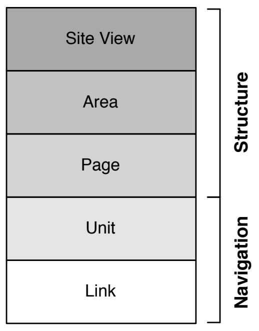
Assuming 1.0 as the minimum term weight, we assigned weights in the [1,2] range[2]. We tested two different weight configurations as illustrated in Figure 5.2a. The first one, named *Structural*, gives more importance to structural metamodel concepts, assigning higher weights to terms associated with site views, areas and pages. The second configuration (named *Navigational*) reverts the weight distribution and assigns more weight to links and units. Table 5.2 shows the two weight assignments used in our experiments.

UML Keyword-based search The UML keyword-based search considers the following four scenarios, given in Figure 5.3, based on the

[2]Higher weight values would introduce too much bias in the evaluation of Equation 3.1, causing the relative weight (with respect to the overall score) of the term to dominate other factors.


Keyword-based search scenarios
Metamodel-independent experiment
Metamodel-dependent experiment
Structural weight configuration
Navigational weight configuration

(a)



(b)

Figure 5.2: (a) Configurations for the WebML keyword-based search scenario; (b) Classification of WebML metamodel concepts.

Table 5.2: WebML keyword-based search: weight configurations for the metamodel-dependent experiment.

Metamodel concept	***Structural*** Configuration	***Navigational*** Configuration
site view	2.0	1.0
area	1.8	1.2
page	1.5	1.5
unit	1.2	1.8
link	1.0	2.0

employed types of indexing as explained in Section 3.2: (i) *project metamodel-independent*, (ii) *class metamodel-independent*, (iii) *class metamodel-dependent*, and (iv) *class neighborhood metamodel-dependent* experiment. The metamodel-independent experiments give equal weights to the index terms, regardless of the metamodel type, as in the WebML case. The difference between the two metamodel-independent indexes is in the number of index fields which depends on the segmentation granularity (project or class). The metamodel-dependent experiments give weights to terms based on the metamodel type in the range $[1 - 2]$, as in the WebML case. We report one weight configuration where higher weight is assigned to a class, than to its relationships that connect it with the other classes and the attributes that the class contains. Other weight configurations with slightly changed weight values have also been tested, but they did not produce any significantly different results. Therefore, only one weight configuration has been reported. The weight assigned to a relationship is lower than the weight assigned to a class, and it depends on the relationship's type and cardinality. The stronger the relationship is, it has a higher weight. Attributes are assigned the lowest weight as the least relevant. Table 5.3 gives the weight con-

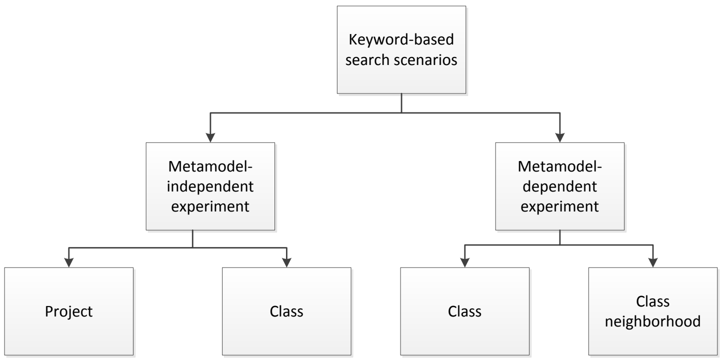

Keyword-based search scenarios
Metamodel-independent experiment
Metamodel-dependent experiment
Project
Class
Class
Class neighborhood

Figure 5.3: Configurations for the UML keyword-based search scenario.

figurations assigned to the metamodel types used in this evaluation. Besides weights given according to the relevance of the metamodel type which represents the class metamodel-dependent experiment, the class neighborhood metamodel-dependent experiment uses a set of penalty weights assigned to each relationship, to penalize the terms imported from the neighboring classes with respect to the terms from the same class. This set of weights is in the range $0 - 1$ thus lowering the weight of the imported terms. The penalty weights are assigned based on the strength of the relationship type and the cardinality, such that stronger relationships have higher weights and they are less penalized. In the case of the generalization relationship the penalty depends on the import direction, i.e. importing from parent class to a child class is considered more relevant than importing in the opposite direction (e.g. in the generalization "monopoly is a game", more importance is given when importing from game to a monopoly than the converse-monopoly is a game; However, not every game is a monopoly). The penalty configuration shown in this evaluation is given in Table 5.4.

Table 5.3: UML keyword-based search: weight configuration for the metamodel-dependent experiments.

Metamodel concept	***Weight*** Configuration
class	1.8
generalization	1.7
composition 1-1	1.6
composition 1-N	1.5
association 1-1	1.4
association 1-N	1.3
attribute	1.0

Table 5.4: UML keyword-based search: penalty configuration for the class neighborhood metamodel-dependent experiment.

Metamodel concept	***Penalty*** Configuration
generalization (import parent class into child class)	0.9
generalization (import child class into parent class)	0.75
composition 1-1	0.6
composition 1-N	0.5
association 1-1	0.4
association 1-N	0.3

A-star graph-based search The A-star graph-based scenario tested four configuration dimensions: (i) the w_{nI} (node insertion), w_{nS} (node substitution), and w_{eI} (edge insertion) weights of the graph edit distance operations, (ii) the parameter λ, which determines the importance of the string distance and type distance in the node similarity function; (iii) the string similarity function used to calculate the node similarity; and (iv) the adoption of locality constraints in the subgraph isomorphism algorithm. In all the reported experiments, after an initial set-up phase with the A-star algorithm, we fixed to 0.6 the node similarity threshold for the pruning rule that discards non-allowed matches, as the value proved best in all the considered settings. Table shows the penalty values used in the experimental evaluation.

The first set of experiments aimed at understanding the impact of the weights assigned to the graph edit distance operations, and we considered three configurations (summarized in Table 5.5):

- *Maximal Substitution* boosts the contribution of the node substitution.
- *Maximal Substitution and Insertion* emphasizes both insertion of nodes/edges and their substitution.
- *Maximal Insertion* stresses only the insertion of nodes/edges.

Table 5.5: A-star graph-based search: different weight configurations.

weights	**Maximal substitution**	**Maximal substitution and insertion**	**Maximal insertion**
w_{eI}	0.1	1.0	1.0
w_{nI}	0.1	1.0	1.0
w_{nS}	1.0	1.0	0.1

The second set of experiments examined the influence of the λ parameter in the node similarity function. We varied the values of λ from $\lambda = 0$ to $\lambda = 1$, with step 0.25. The λ values and corresponding experiment names are reported in Table 5.6: recall that higher values of λ give more importance to name similarity w.r.t. metamodel type similarity.

Table 5.6: A-star graph-based search: different λ values.

Only type contribution	$\lambda = 0$
High type contribution	$\lambda = 0.25$
Intermediate type contribution	$\lambda = 0.5$
Low type contribution	$\lambda = 0.75$
No type contribution	$\lambda = 1.0$

The third set of experiments analyzed the impact of the adopted string distance metrics. String distance metrics are similarity functions that do not consider prior knowledge and thus exhibit performance that is strongly related to the specific application domain [19]. We compared two frequently used functions: the *Levenshtein* distance [80], and the *n-gram* distance [64].

The *Levenshtein* distance is a string-edit distance that, given two strings, finds the minimal number of string edit operations that transform one string into the other, normalized with the length of the longer string. For two identical strings, the *Levenshtein* distance is 0, and the corresponding similarity value is 1.

The *N-gram* distance is a string token distance which finds the common number of n-grams (substrings of the original string with fixed length n) for two strings, normalized with the total number of n-grams. The *N-gram* distance has values in the [0,1] interval, where 0 means no similarity, and 1 is an exact match.

Finally, the fourth set of experiments assessed the effect of applying locality constraints in the selection of candidate mappings in A-star. In particular, we evaluated the original version (Algorithm 1) and the local version (Algorithm 2) of A-star.

Multidimensional scaling graph-based search As for the other algorithms, the evaluation of the multidimensional scaling graph-based search is performed by fine-tuning of its parameters, showing only those values that produce the most distinctive results. As a result, the multidimensional scaling graph-based search evaluation is achieved by firstly tuning and evaluating the Chalmer's algorithm. The Chalmer's algorithm is an iterative algorithm, used for indexing project graph nodes and retrieving candidate nodes which has a set of parameters that need to be tuned: number of iterations, max Cache Size which is the size of points in the random set, and number of dimensions used to represent points in space. The values used for configuring the Chalmer's algorithm, max Cache Size and number of iterations, are presented in Table 5.8 and Table 5.7 respectively, while the number of dimensions is in the range $[1 - 5]$.

Table 5.7: Multidimensional scaling graph-based search: different number of iterations.

Number of iterations	25	50	100	200

Table 5.8: Multidimensional scaling graph-based search: different values of max Cache Size.

Max Cache Size	10	20	40	80	160

Chalmer's algorithm tuning is performed by varying the values of its parameters discussed above. The tuning is done for each feature class individually. The Chalmer's algorithm is applied to transform name node labels from all the nodes in the project graphs into points in space using a configuration of parameter values. A random point is extracted from this set of points, and it is used to query the set of points, by extracting all points which are within a value *delta* which represents a distance. The value of *delta* is varied in the range $[0.05 - 0.3]$ with increments of 0.05. The retrieved points are used to measure the precision and recall of the Chalmer's algorithm by comparing the number of retrieved points, whose Euclidian distance with respect to the query point is within the *delta* value, and the number of nodes whose real distance between the corresponding name labels of the retrieved nodes, represented by the retrieved points, and the query node is within the *delta* value. In our case, we would like to have a good trade-off between precision and recall. However, we consider the recall more significant since the number of relevant points that are retrieved is important for the next phases of the graph-based seach algorithm we propose. The goal is to obtain the best possible performance of the Chalmer's algorithm for each feature class using minimal number of iterations, minimal number of dimensions for point representation, minimal number of points in the random set (maxCacheSize), and minimal delta value.

The points obtained from the Chalmer's algorithm for each feature class are placed in a grid, such that for each feature class there is a corresponding grid. The grid parameter that needs to be tuned is its number of buckets. Therefore, depending on the number of dimensions of each grid, which is identical to the number of dimensions for the points representation selected in the process of Chalmer's algorithm tuning, we tested different bucket widths in the range $[0-1]$

for each feature class, assuming that each grid dimension goes from 0 to 1. The evaluation is performed in the similar way as previously through precision and recall, by extracting a random point from a set of points for a feature class and considering it as a query point used to retrieve points from the corresponding grid within a distance *delta* [3] with respect to the query point.

Final parameter that needs to be tested while a node is querying a grid is the acceptable distance, which retrieves all the points within the specified Euclidian distance with respect to a query node for a specific feature class. Since, its value can be different and can be specified individually for each feature class, we use the following notation: *nameDist* represents acceptable distance for the name feature class, *typeDist* refers to the acceptable distance for the type feature class, and *dataDist* corresponds to the acceptable distance for the entity/relationship feature class (since entity and relationship are constituent elements of the Data Model, hence the name). Table 5.9 represents the configuration values of the acceptable distances for the evaluation of the multidimensional scaling graph-based search scenario.

Table 5.9: Multidimensional scaling graph-based search: different acceptable distances configurations.

Name distance	**Type distance**	**Data distance**
0.4	0.4	0.2
0.4	0.4	0.4
0.6	0.4	0.2
0.6	0.4	0.4

5.2 Evaluation Metrics

Performance is evaluated using three standard information retrieval measures: (i) 11-point interpolated average precision; (ii) Mean Average Precision (MAP); and (iii) Discounted Cumulative Gain (DCG).

Precision and recall are the two most used IR evaluation measures. Precision considers the fraction of retrieved documents that are relevant, regardless of the ranking, while recall measures the fraction of relevant documents that are retrieved.

The *11-point interpolated average precision* combines precision and recall by measuring the best precision that can be obtained at

[3] *delta* distance values are the ones obtained from the Chalmer's algorithm tuning phase

11 standard levels of recall (0.0, 0.1,...1.0) [85]. At each each recall level r_i, the interpolated precision is obtained as an average over the sample queries and represents the highest precision that can be obtained for recall values $r_j \geq r_i$. The 11-point precision value decreases for increasing recall, as for a growing number of retrieved results, the likelihood of irrelevant matches typically increases.

Mean Average Precision (MAP) [85] is a single figure quantification of the average precision across recall levels and queries: for each query, the average precision is computed as the average of the precision value obtained in the set of top-k documents that are retrieved to get to the j-th relevant document. More precisely, if the set of relevant documents for a query $q_j \in Q$ is $\{d_1, \dots d_{m_j}\}$, where m_j is the number of relevant documents, and R_{jk} is the ordered set of the first k ranked results, then:

$$\mathrm{MAP}(Q) = \frac{1}{|Q|}\sum_{j=1}^{|Q|}\frac{1}{m_j}\sum_{k=1}^{m_j}\mathrm{Precision}(R_{jk}) \tag{5.1}$$

When the first k positions of the result set contain no relevant documents, the precision value in Equation 5.1 is 0. In our case, MAP is calculated up to the top 10 matching projects.

Finally, the *Discounted Cumulative Gain* (DCG) [66] is a graded relevance measure that evaluates the ability of an IR system to retrieve highly relevant documents at high positions in the result set. DCG considers the fact that the lower a document is ranked in a result set, the less likely it is for such a document to be examined by a user. DCG is computed as:

$$DCG_p = \sum_{i=1}^{p}\frac{2^{rel_i} - 1}{\log_2(1+i)} \tag{5.2}$$

where rel_i is the relevance of the document at the i-th rank position obtained from the gold standard dataset evaluation.

5.3 Quantitative Evaluation

5.3.1 WebML keyword-based search

Table 5.10 shows the values of MAP for different indexing structures. All index structures achieve good performance (with peak MAP value of 81%), significantly better than the random baseline. Adding metamodel-dependent weights to the index slightly increases the performance for the tested queries (4% MAP increase in the best case).

Figure 5.4a and Figure 5.4b show the results of DCG and 11-point precision. Also these measures support the conclusion that the different configurations of the index for the textual search exhibit a comparable *average* behavior. Boosting the weight of more specific elements (units and links) over high-level ones (site views, areas) provides slightly improved performance: the average performance of the navigational configuration increases by 2% for DCG and 5% for 11-point precision with respect to the structural configuration, and 7% for DCG and 3% for the 11-point precision with respect to the metamodel-independent configuration.

Table 5.10: WebML keyword-based search: values of MAP

Experiment	*MAP*
Random	0.19
Metamodel-independent	0.77
Metamodel-dependent *Structural* Configuration	0.78
Metamodel-dependent *Navigational* Configuration	**0.81**

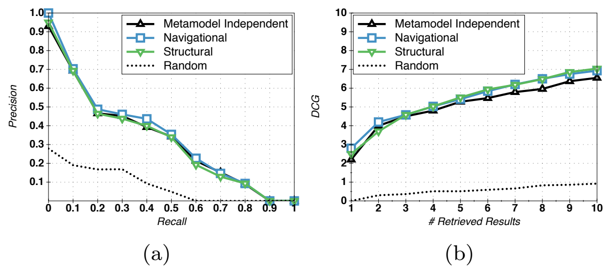
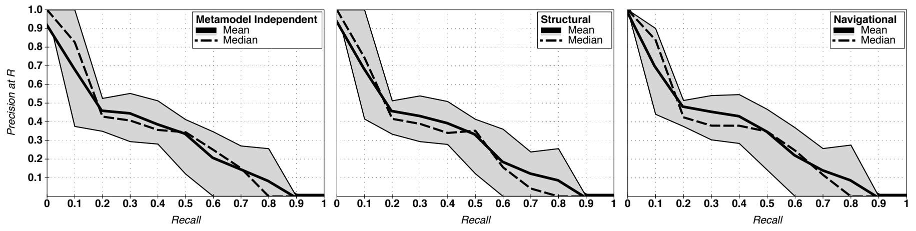


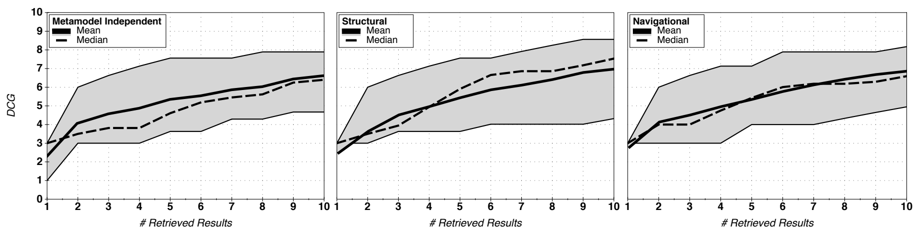
Figure 5.4: WebML keyword-based search: 11-point precision (a) and DCG (b).

Figure 5.5a and 5.5b explode Figure 5.4 to examine the average and median values over the sample queries, and the upper and lower quartiles (gray area). The growth of the DCG values slows down at higher rank positions. Since DCG depends not only on the precision and recall but also on the rank order of the retrieved documents, the slow down at higher ranks shows that even if relevant documents are retrieved they are not ranked optimally, w.r.t the gold standard, when one looks at larger result sets.

The comparison of the metamodel-dependent and the metamodel-independent index structures in Figure 5.5b shows that the latter exhibits an average DCG value consistently higher than the median, thus indicating the presence of several outliers and, therefore, a less uniform ranking behavior across sample queries. The distribution of differences in the 11-point average precision graph, i.e., the gray area between the lower and upper quartile in Figure 5.5a, at lower levels of recall shows that the structural setting has more performance fluctuation in finding the top matches than the navigational one.



(a)



(b)

Figure 5.5: WebML keyword-based search: average, median, lower and upper quartile of: 11-point precision (a) and DCG (b) for Metamodel-independent, Structural, and Navigational index configurations.

5.3.2 UML keyword-based search

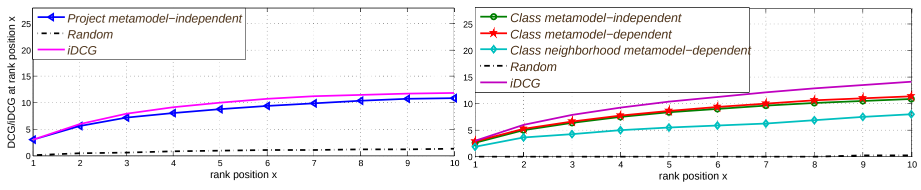
We first present the result of MAP in Table 5.11. Note that the project index structure cannot be directly compared to the class index structures, since they are queried with different sets of queries which express different user needs. It can be observed that all indexing structures have high performance with project metamodel-independent index performing the best with MAP of 98%. Regarding the class granularity indexes, it can be noticed that adding metamodel information to the index slightly increases MAP (by 3%), while adding neighboring information decreases performance (by 25%). All of the experiments perform much better with respect to the random baseline algorithms corresponding to the project and class granularity.

Figures 5.6a and 5.6b show the results of the DCG for project and class index structures, respectively, while the figure 5.7 illus-

Table 5.11: UML keyword-based search: values of MAP

Experiment	*MAP*
Project metamodel-independent	**0.98**
Class metamodel-independent	0.93
Class metamodel-dependent	0.96
Class neighborhood metamodel-dependent	0.71
Random project	0.22
Random class	0.02

trates the 11-point precision results. The results are consistent with respect to the MAP results, verifying that adding metamodel information slightly increases the performance, while adding information from neighbor classes decreases performance. Although including neighborhood information in the index incorporates more structural information in it, the drop in performance is due to the content of the retrieved result set (average 38% of drop for DCG and 32% of drop for 11-point precision with respect to the class metamodel-independent experiment, and 45% of drop for DCG and 35% for 11-point precision, with respect to the class metamodel-dependent experiment). The reason for the worse performance is that while querying the index, besides retrieving a class, the method returns the neighbor classes to this class, which evidently are not deemed relevant to the query by the evaluators when building the ground truth. The best performing index structure is the project metamodel-independent index, which always retrieves the most relevant document at the first position as it can be noticed from the DCG figure (Figure 5.6a). The better performance of project metamodel-independent index over class index structures might be explained with the fact that users are more likely to search for entire projects than a single class inside a class diagram.



(a) (b)

Figure 5.6: UML keyword-based search: DCG for project metamodel-independent index(a) and DCG for class indexes (b).

Figure 5.8a shows that the median, the first and the third quartile are identical with the curve expressing the average which reaches

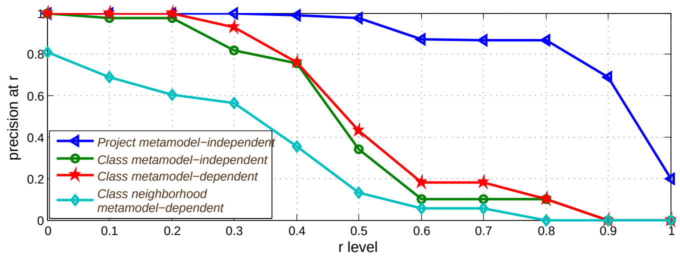


Figure 5.7: UML keyword-based search: 11-point precision.

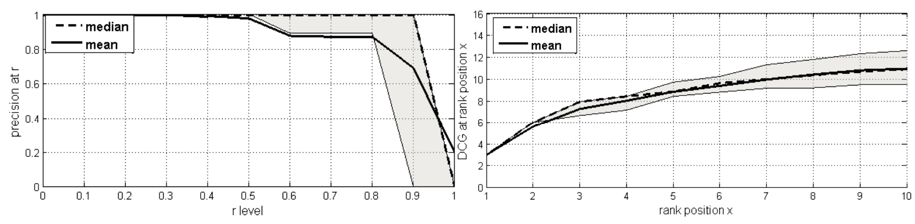
the maximum precision value of 1 for lower levels of recall (up to 40% of recall). This means that all the retrieved models are relevant across the queries for lower recall levels. A slight fluctuation across queries' performance can be noticed for higher recall levels, demonstrating the stability of the project metamodel-independent experiment. A similar behavior is observed for the DCG, given in Figure 5.8b, confirming the equally good performance for all queries at high ranks.

Class metamodel-independent and the class metamodel-dependent experiments, whose performance is given in Figures 5.9 and 5.10 respectively, perform similarly. Moreover, the metamodel-dependent experiment performs slightly better, with less variability in the query performance due to the smaller area between the first and the third quartile. An interesting comparison can be made between the class neighborhood metamodel-dependent experiment (quartiles shown in Figure 5.11) and the other above-mentioned class granularity experiments. The class neighborhood metamodel-dependent experiment has bigger area between the first and the third quartile with respect to the class granularity experiments, indicating that there is more variability in the queries' performance even at lower recall levels. The average DCG value is higher than the median, with the decrease of the rank, and the average 11-point precision is also higher than the median as the recall levels increase, revealing existence of outliers.

5.3.3 A-star graph-based search

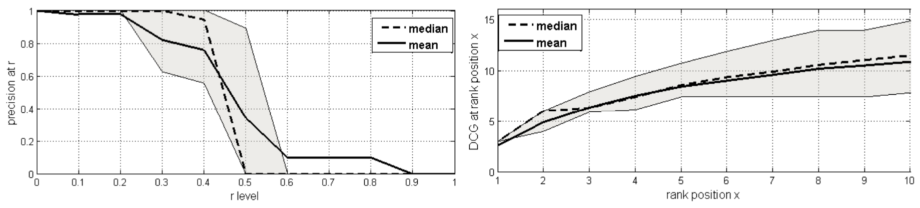
The evaluation of the A-star graph-based search first examined the influence of the λ parameter with respect to each weight configuration in the graph edit distance, adopting the *Levenshtein* string distance metric.

Table 5.12 summarizes the MAP values for different λ values and graph edit distance weight configurations. Figure 5.12a and Figure



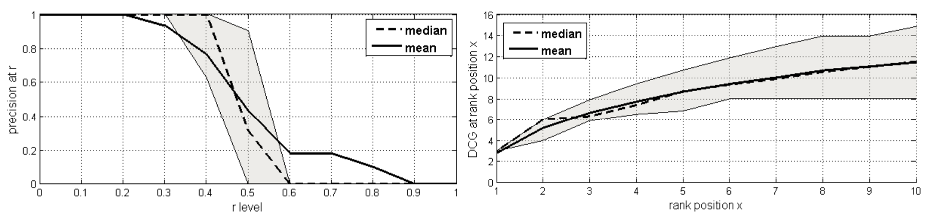
(a) (b)

Figure 5.8: UML keyword-based search: average, median, lower and upper quartile of: 11-point precision (a) and DCG(b) for Project metamodel-independent experiment.



(a) (b)

Figure 5.9: UML keyword-based search: average, median, lower and upper quartile of: 11-point precision (a) and DCG(b) for Class metamodel-independent experiment.

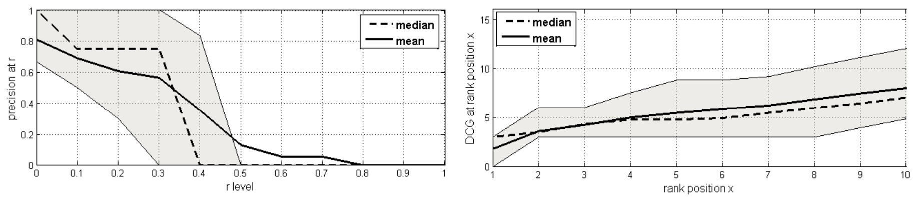


(a) (b)

Figure 5.10: UML keyword-based search: average, median, lower and upper quartile of: 11-point precision (a) and DCG(b) for Class metamodel-dependent experiment.

5.12b respectively show the 11-point interpolated average precision and DCG results for the various weight configurations; each curve in one diagram corresponds to a specific value of λ.

From Table 5.12 and Figure 5.12a and 5.12b, it emerges that neither the metamodel type alone nor the element label alone are the best options for node matching. When the *Only type contribution* configuration is used, the 11-point precision graphs show that, regardless of the adopted weights configurations, very few relevant



(a) (b)

Figure 5.11: UML keyword-based search: average, median, lower and upper quartile of: 11-point precision (a) and DCG(b) for Class neighborhood metamodel-independent experiment.

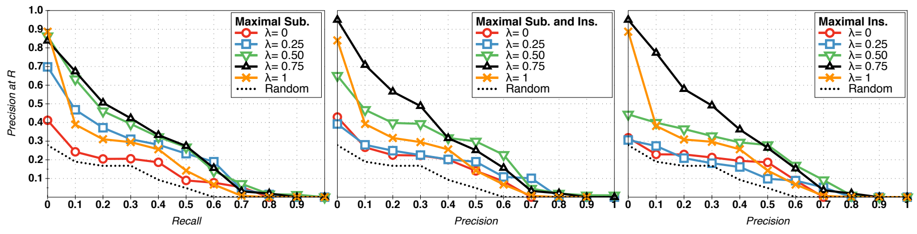
Table 5.12: A-star graph-based search: values of MAP for different values of λ and weight configurations in the graph edit distance.

Experiment	*Maximal substitution*	*Maximal subst. & insertion*	*Maximal insertion*
Random	0.19		
Only type contribution (λ=0)	0.34	0.32	0.29
High type contribution (λ=0.25)	0.56	0.34	0.24
Intermediate type contribution (λ=0.5)	0.74	0.55	0.38
Low type contribution (λ=0.75)	0.72	**0.83**	**0.86**
No type contribution (λ=1)	0.74	0.7	0.73

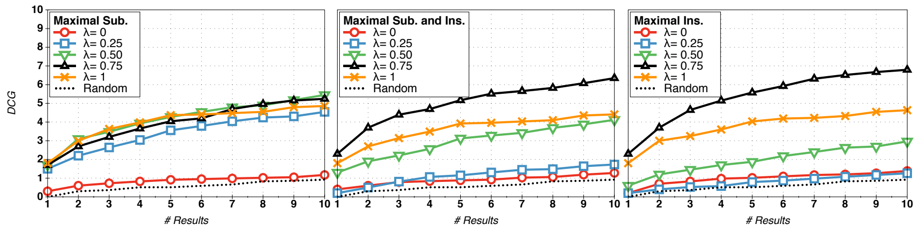
documents are retrieved (curves show low precision values), which is confirmed by the DCG graphs and MAP values. Adding a "touch" of metamodel type knowledge to the node similarity function leads to better performance: the *Low Type Contribution* configuration (λ=0.75) emerges in most cases as the most viable trade-off between label and metamodel type information (up to 13% better than *No type contribution* and up to 57% better than *Only type contribution* in the MAP table).

The greater relative importance of element names over types in the best performing case is explained by the occurrence of false positive matches: overemphasizing metamodel types quickly leads to cases in which some project graph nodes representing a modeling concept present in the query (e.g., a given type of operation on data) are considered similar and thus matched to project nodes that operate on content unrelated to the query.

The DCG graphs (Figure 5.12b) suggest a correlation between the value of λ and the graph edit distance weight configuration policy. The spread among the curves at different values of λ is very limited for the *Maximal Node Substitution* configuration, and more sensible for the other two configurations. This shows that *Maximal Node Substitution*, which gives importance only to node substitution operations (i.e., similarity depends on finding as many "right" model elements as possible, and not, or less, on how the model elements are arranged or on missing model elements), makes the rank



(a)

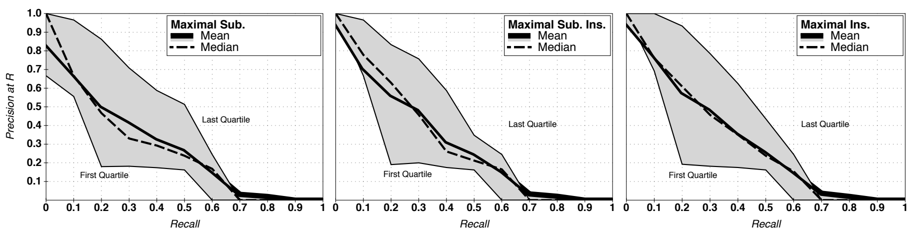


(b)

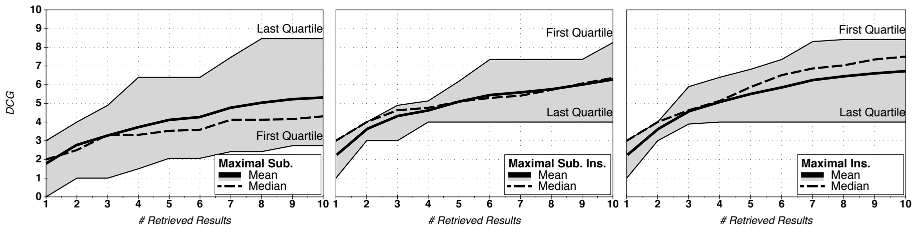
Figure 5.12: A-star graph-based search: 11-point interpolated average precision (a) and DCG (b) for different λ values and weight configurations (maximal substitution, maximal substitution and insertion, and maximal insertion).

order of results less sensitive to the name-type tradeoff in the node similarity metrics, but for the case of λ=0 which remains dominated in all weight configuration policies. Symmetrically, the policies that emphasize node/edge insertions (*Maximal substitution and insertion* and *Maximal insertion*) achieve better MAP figures, but the rankings they produce are more sensitive to the tuning of λ. A possible interpretation of this phenomenon is that the *Maximal substitution and insertion* and the *Maximal insertion* policies, which penalize node and edge insertions in graph similarity, require the “right” node similarity function, to compensate the fact that even slight topological differences (e.g., differences in containment and linking, or missing model elements) in the query and the project model can push a relevant match down in the result list (and hence, the DCG curves for “wrong” λ values are more separated from the curve at the “right” value λ=0.75).

Figure 5.13 shows the average, median, and lower and upper quartile for 11-point precision and DCG curves. It confirms the performance improvement obtained when considering insertion operations, because in both the 11-point precision and DCG the distribution of differences shows less variations with respect to the *Maximal Node Substitution* configuration. However, Figure 5.13 also shows that the distribution of results is wider than the one shown in Figure 5.4 for WebML keyword-based search; this means that the



(a)



(b)

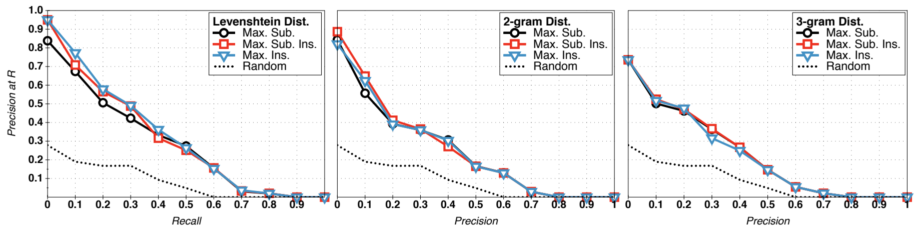
Figure 5.13: A-star graph-based search: average, median, and lower and upper quartile of 11-point precision (a) and DCG (b), with $\lambda = 0.75$

performance of the A-star graph-based scenario varies more across the sample queries.

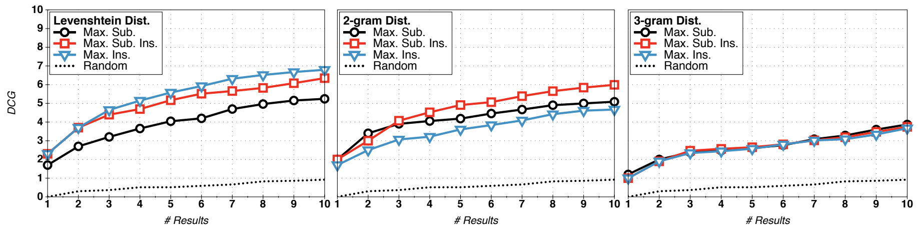
Table 5.13: A-star graph-based search: values of MAP for different string distance metrics.

Experiment	*Maximal substitution*	*Maximal substitution and insertion*	*Maximal insertion*
Levenshtein distance	**0.72**	**0.83**	**0.86**
2-gram distance	0.72	0.75	0.78
3-gram distance	0.60	0.63	0.59

String similarity function comparison Figure 5.14 shows the third experiment with A-star graph-based search, which evaluates the adoption of different string similarity functions. We set λ to 0.75 (*Low type contribution*) and evaluated the *Levenshtein* distance and the *N-gram* distance under the three graph edit distance configurations. The best results are obtained when using the *Levenshtein* distance. N-gram distance was tested for 2-grams and 3-grams. With respect to the *Levenshtein* distance, 2-grams respectively decrease the 11-point precision and DCG, for an average of 22% and 13%, while 3-grams decrease, on average, the 11-point precision by 43% and the DCG by 32%. Noteworthy, the three graph edit configurations perform consistently with both the *Levenshtein* and the *n-gram* distances, as the *Maximal Insertion* configuration outperforms the



(a)



(b)

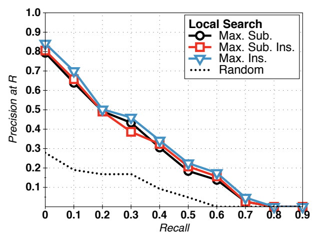
Figure 5.14: A-star graph-based search: 11-point interpolated average precision (a) and DCG (b) for of *Levensthein*, *2-gram*, and *3-gram* string distances ($\lambda = 0.75$)

others. The performance behavior of each string distance metric is further confirmed by the MAP results reported in Table 5.13.

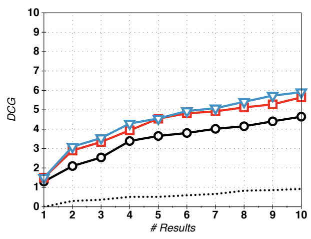
In summary, the best performance in both precision and ranking is obtained for a moderate metamodel type contribution in node similarity evaluation (*Low type contribution* a.k.a λ=0.75), *Levenshtein* distance for name similarity, and weight assignment configurations that appraise *both* node similarity and model topology. Therefore, textual similarity remains fundamental to achieve good results also in A-star graph-based search, but metamodel-dependent information *and* the topology of the query must be exploited to retrieve more relevant results and sort them in a more proper order.

A-star graph-based search with locality constraints

As a last experiment, we compared A-star graph-based search with and without locality constraints for candidate mapping nodes. For both the original A-star and A-star with locality constraints we set λ to 0.75 (*Low type contribution*) and used *Levenshtein* distance in the node similarity function. Table 5.14 reports the MAP values for A-star and A-star with locality constraints. Figures 5.15 and 5.16 chart the 11-point precision and DCG curves. As can be noted in the above mentioned results, the application of locality constraints slightly worsens on average the performance of graph-based search.

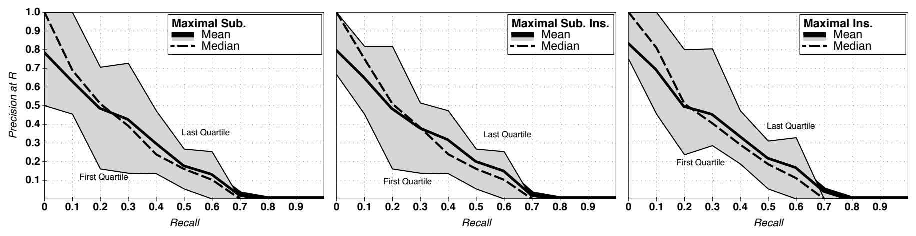


(a)

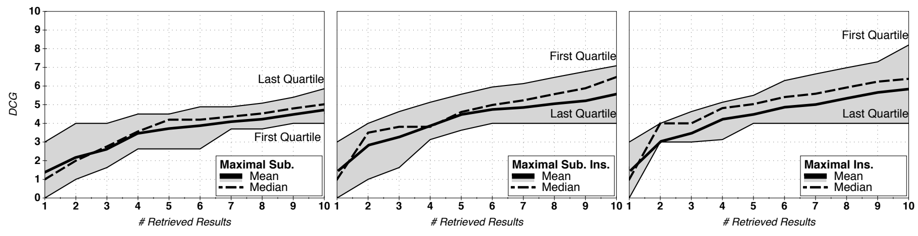


(b)

Figure 5.15: A-star graph-based search: 11-point interpolated average precision (a) and DCG (b) with locality constraints ($\lambda = 0.75$, *Levensthein* distance)



(a)



(b)

Figure 5.16: A-star graph-based search: average, median, and lower and upper quartile of 11-point precision (a) and DCG (b), with $\lambda = 0.75$, *Levenshtein* distance, and locality constraints.

Inspection of results reveals the following behavior:

- Locality constraints prevent the selection of disconnected matching nodes. This promotes in the result set matches with patterns that conform to the majority of the elements in the query and penalizes matches with projects that, although topically relevant, contain only partial reusable patterns scattered in different places of the model. This effect, in our experimental query panel and project repository, tends to favor local A-star.
- Some queries that perform well with A-star worsen their performance when locality of matching is applied, because the relevant results end up having less matching nodes, which lowers

their rank score and thus diminishes their separation from not so relevant results; then it may happen that a less relevant result overcomes a more relevant one in the result list. This behavior tends to favor the original A-star.

- The two abovementioned effects compensate each other, with a slight predominance of the cases where locality worsens the performance.

Table 5.14: A-star graph-based search: MAP values for A-star algorithm with local search.

Experiment	*Maximal substitution*	*Maximal substitution and insertion*	*Maximal insertion*
A-star	**0.72**	**0.83**	**0.86**
A-star + locality constraint	**0.70**	**0.72**	**0.77**

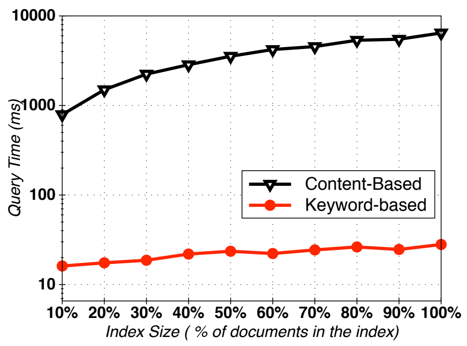


Figure 5.17: Comparison of response time at varying number of indexed projects for WebML keyword-based search and A-star graph-based search.

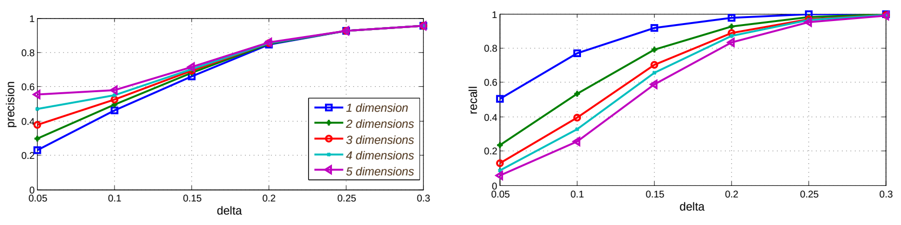
Query Execution Time The last quantitative experiment compares the performance of the *WebML keyword-based* and *A-star graph-based* search approaches with respect to *response time* required for query execution. All the experiments have been conducted on a machine equipped with Intel dual Core Processor 2.4GHz, 6GB RAM, and Windows 7 (64-bit) operating system; the reported values are averaged over 10 executions.

Figure 5.17 shows the query execution time for all the 10 queries considered in the experiments with respect to the index size. As expected, A-star graph-based search is considerably slower than WebML keyword-based search, which executes in quasi-constant time. Despite the exponential complexity of graph matching, the A-star graph-based approach shows a quasi-linear correlation with respect to the index size for the considered repository. Notice that no query execution optimization (including optimized indexing of the repository) has been adopted during experiments and therefore we expect a wide range of possibilities for improving the performance of the content-based system.

5.3.4 Multidimensional scaling graph-based search

Chalmer's algorithm tuning The evaluation of the multidimensional scaling graph-based search starts with fine-tuning of the Chalmer's algorithm used as a base for indexing. Since Chalmer's algorithm has three parameters to be tuned, precision and recall are evaluated by varying one of its parameters at a time, while the other two remain fixed. Chalmer's algorithm is tuned for each feature class individually, to select the best parameter settings that are used in the proposed multidimensional scaling graph-based search method.

Figure 5.18 shows the precision and recall of the Chalmer's algorithm by varying the number of dimensions for the name feature class, with 25 iterations and maxCacheSize of 10. It can be noticed that while precision increases, with the increase of the number of dimensions, recall decreases. Considering the delta value, both precision and recall curves converge after *delta* = 0.25. This happens because after some delta, the number of retrieved points becomes the same regardless of the dimensions number. One observation that can be made is that, for *delta* = 0.15 there is a good trade-off between precision and recall when the number of dimensions is 3. Therefore, these two values will be fixed while changing the other two parameters. Note that we also tested further increase of the number of dimensions(up to 12), which does not further improve performance and recall, and therefore those results are not shown.



(a) (b)

Figure 5.18: Chalmer's algorithm: Precision (a) and recall (b)for name feature class considering different number of dimensions.

Figures 5.19a and 5.19b represent precision and recall considering the maximal cache size and the number of iterations for the name feature class. Varying the maximal cache size does not significantly change performance, and therefore, a small value for the cache size should be chosen. We choose the value of 14, at the intersection of the precision and recall curves. It can be also noticed that the variation of the number of iterations does not influence the performance, so the number of iterations can be low for achieving a good precision/recall balance. As a result, we select 25 iterations.

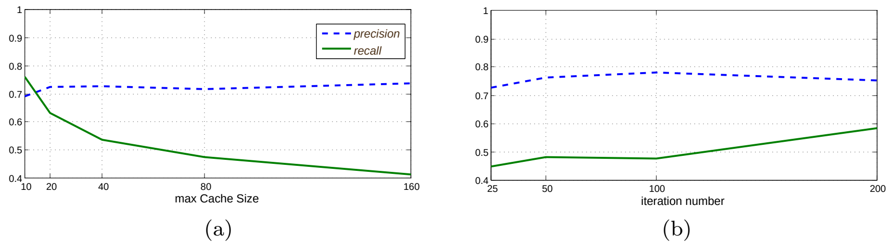


Figure 5.19: Chalmer's algorithm: Precision and recall for name feature considering maxCacheSize (a) and number of iterations (b).

Finally, for the name feature class the selected parameters for constructing the index with the Chalmer's algorithm are: 3 dimensions, 25 iterations and 14 points in the random set ($maxCacheSize$), with $delta = 0.15$.

In the following, we are showing the parameter configuration for the type feature class. Figure 5.20 illustrates the precision and recall with respect to the number of dimensions with 25 iterations and 80 points in the random set. Increase of the number of dimensions increases precision for small values of delta, while recall remains constantly high and close to one, regardless of the number of dimensions or delta values. Therefore, since the choice of the dimensions number depends entirely on precision, we select 5 dimensions and $delta = 0.10$.

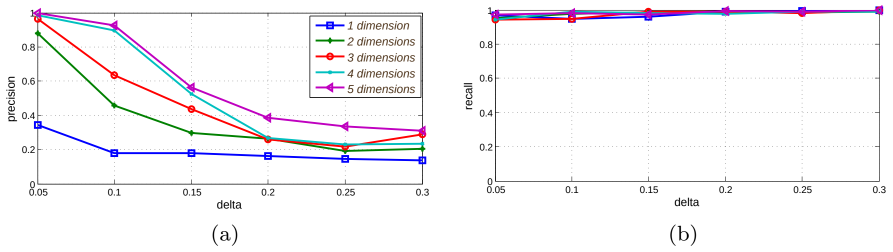


Figure 5.20: Chalmer's algorithm: Precision (a) and recall (b)for type feature class considering different number of dimensions.

Figures 5.21a and 5.21b show the precision and recall with respect to the cache size and number of iterations, for *delta* = 0.10 and 5 dimensions for representing points in space. Although precision and recall do not vary much considering the cache size, the peak values are achieved when the value for the maxCacheSize is 40, and we choose it as a value that will be applied in our algorithm. Precision and recall are both close to one regardless of the number of iterations. We choose 25 iterations for the Chalmer's algorithm for this feature class in the indexing phase.

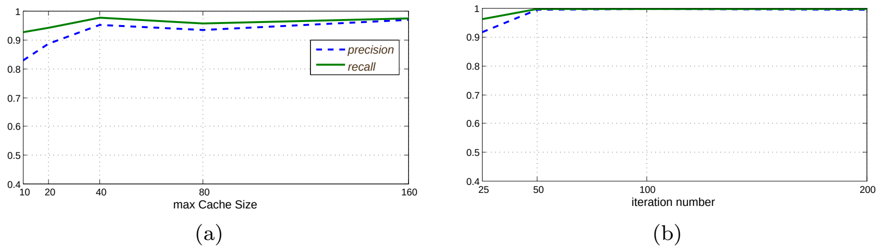


Figure 5.21: Chalmer's algorithm: Precision and recall for type feature class considering maxCacheSize (a) and number of iterations (b).

For the type feature class, the value of the parameters selected for the Chalmer's algorithm used to build the index are: 5 dimensions, 25 iterations and 40 points in the random set, with *delta* = 0.10.

Figure 5.22 presents the precision and recall with respect to the number of the dimensions for the entity/relationship feature class. Precision and recall values do not change much as the number of dimensions increases. However, precision increases as the delta value increases. The recall, on the other hand, does not change its value across the different delta values. In fact, it is always close to 1. Based on the precision curves, we choose to represent the points corresponding to the entity/relationship feature class in 2 dimensions (choosing 1 dimension is trivial). Regarding the delta values, for *delta* = 0.2 the precision has a high value. It is also a value where the precision is almost identical for all the dimensions.

Figures 5.23a and 5.23b illustrate the precision and recall with respect to the cache size and number of iterations, while *delta* = 0.20 and number of dimensions for representing points in space is 2, as chosen previously. Considering the cache size, since the recall is almost 1, and the precision is almost constant (a small drop in the value when maxCacheSize=80 is due to the randomness of the algorithm) across different cache sizes, cache size of 10 is chosen as a maximumCacheSize for the entity/relationship feature class.

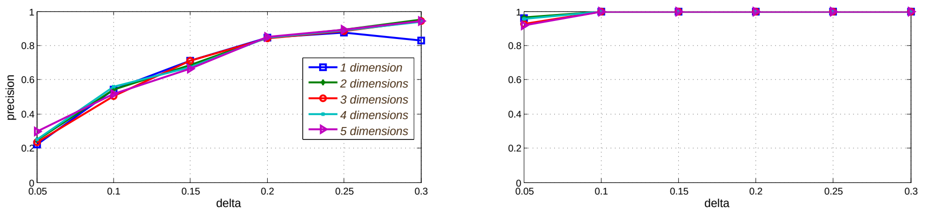


Figure 5.22: Chalmer's algorithm: Precision (a) and recall (b)for entity/relationship feature class considering different number of dimensions.

The same behavior can be noticed for the precision and recall with respect to the number of iterations, which results in selecting the lowest number of iterations for this feature class, i.e. 25, for the indexing algorithm.

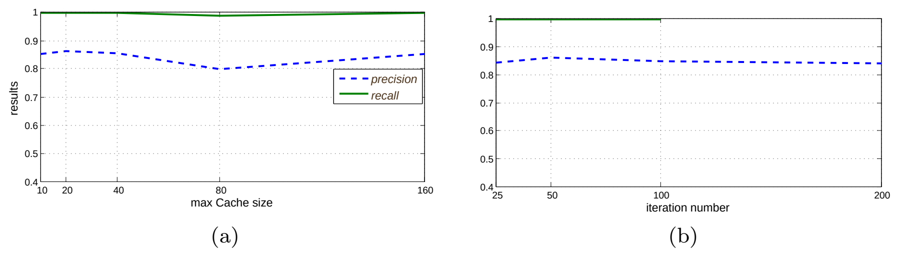


Figure 5.23: Chalmer's algorithm: Precision and recall for entity/relationship feature class considering maxCacheSize (a)and number of iterations (b).

The parameter values chosen for the Chalmer's algorithm for indexing the entity/relationship feature class are: 2 dimensions, 25 iterations and 10 points in the random set with $delta = 0.20$.

The chosen parameter values for every feature class show that Chalmer's algorithm performs well when performed on the WebML dataset, achieving high values of precision and recall. This gives a good motivation to use it in the indexing phase of our algorithm.

Bucket Width Evaluation Once the Chalmer's algorithm parameters for each feature class have been determined, and the nodes are transformed into three sets of points, they need to be placed in grids that construct the index. Each grid consists of buckets, whose number depends on the bucket width and number of dimensions of the points to be placed in the grid. Bucket width selection should be done carefully, since choosing a small bucket width increases exponentially the number of buckets which increases exponentially the space complexity. In our experiments, as it can be noticed in Figure

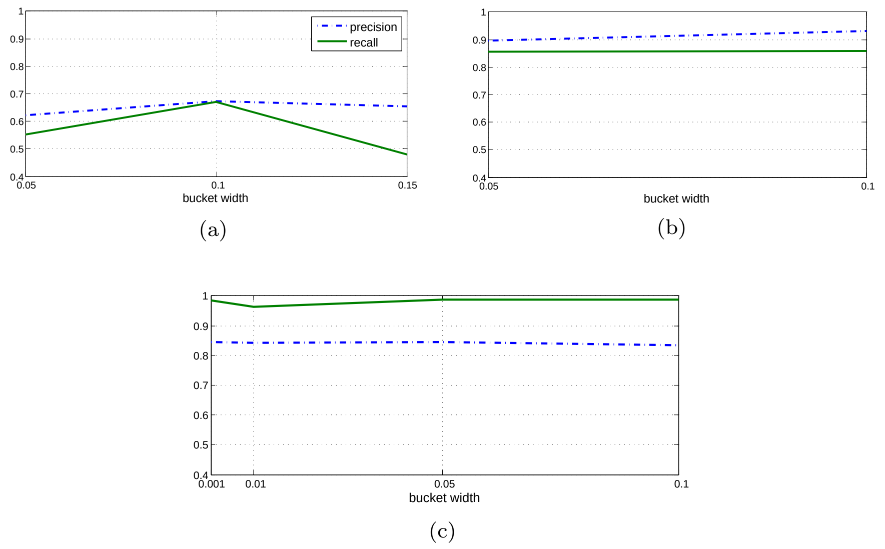


Figure 5.24: Precision and recall considering bucket width for name feature class (a), type feature class (b) and entity/relationship feature class (c).

5.24, we found that the number of buckets in the grid does not significantly change the precision and recall of the Chalmer's algorithm when querying the grid, since the same set of points is just placed in the grid's buckets. Small variations in the recall values are due to the randomness of the algorithm. Therefore, a small bucket width would not bring any advantage. The experiments were performed for each feature class for different bucket widths, considering that the number of dimensions is already fixed by the Chalmer's algorithm, and assuming that each grid dimension goes from 0 to 1. Considering the experimental results from Figure 5.24 and the above-mentioned recommendations for bucket width selection, we choose the following bucket widths: for the name feature class, we select a bucket width of 0.1. Since each name point has 3 dimensions, we obtain 10 buckets per dimension, and the resulting name grid contains 1000 buckets; for the type feature class, we choose 0.1 as a bucket width which results in a type grid with 100000 buckets since the corresponding points have 5 dimensions; for the entity/relationship feature class we choose bucket width of 0.05 and considering that the corresponding points are two dimensional, the total number of buckets in the grid is 400.

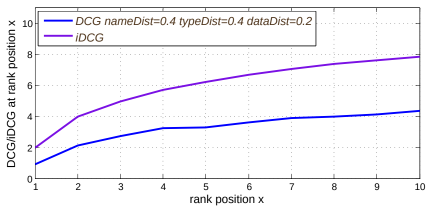
The selected bucket widths are used in the grids of the multidimensional scaling graph-based search algorithm.

Multidimensional scaling graph-based search evaluation The evaluation of the algorithm is performed by using the distance configurations from Table 5.9. Using smaller distance values (close to the delta values fixed by the Chalmer's algorithm) for the name and type feature classes does not retrieve a sufficient number of candidate points, since in the later stages of the alghorithm further pruning is applied (based on the neighborhood of the corresponding nodes and nodes belonging to a same project),their number will decrease even more. Conversely, further increase of the distance values (above the values shown in the table 5.9) will retrieve increased number of candidate points, including more points which are less similar with respect to the query node, which will decrease performance respectively.

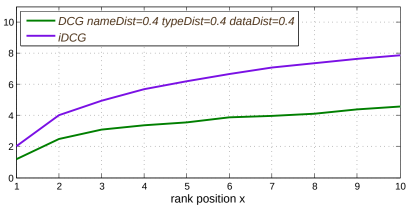
The results for the MAP considering the top 150 generated results are given in Table 5.15. The MAP values show that increasing the data distance does not significantly change results, while increasing the name distance changes more significantly the MAP by decreasing it.

Table 5.15: Multidimensional scaling graph-based search: values of MAP

name distance	*type distance*	*data distance*	*MAP*
0.4	0.4	0.2	0.43
0.4	0.4	0.4	0.44
0.6	0.4	0.2	0.31
0.6	0.4	0.4	0.29



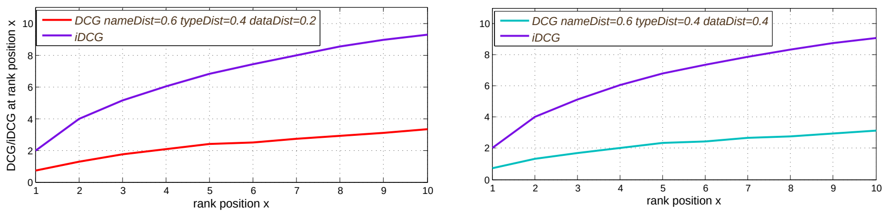
(a)



(b)

Figure 5.25: Multidimensional scaling graph-based search: DCG where $nameDist = 0.4$, $typeDist = 0.4$ and $dataDist = 0.2$ (a) and $nameDist = 0.4$, $typeDist = 0.4$ and $dataDist = 0.4$ (b).

Figures 5.25 and 5.26 present the results for DCG for the considered distance configurations. Each experiment has its own ideal DCG, since the ground truth is formed from the retrieved subgraphs as explained in details in Section 5.1.2. The DCG results are consistent with the MAP results, as increasing the name distance worsens



(a) (b)

Figure 5.26: Multidimensional scaling graph-based search: DCG where $nameDist = 0.6$, $typeDist = 0.4$ and $dataDist = 0.2$ (a) and $nameDist = 0.6$, $typeDist = 0.4$ and $dataDist = 0.4$ (b).

the results. The best performing configuration $nameDist = 0.4$, $typeDist = 0.4$ and $dataDist = 0.4$ has 52% and 56% better DCG than $nameDist = 0.6$, $typeDist = 0.4$ and $dataDist = 0.2$ and $nameDist = 0.6$, $typeDist = 0.4$ and $dataDist = 0.4$ respectively, and 9% better DCG than $nameDist = 0.4$, $typeDist = 0.4$ and $dataDist = 0.2$. Figure 5.27 shows the 11-point precision results for the considered distance configurations which confirm the results obtained with the other metrics. It can be noticed that the best performance is obtained for $namedistance = 0.4$ and $typedistance = 0.4$, while changing the $typedistance$ from 0.2 to 0.4 insignificantly influences performance (by 3% worse). Increasing the name distance value to 0.6 decreases significantly 11-point precision (by 102% per average).

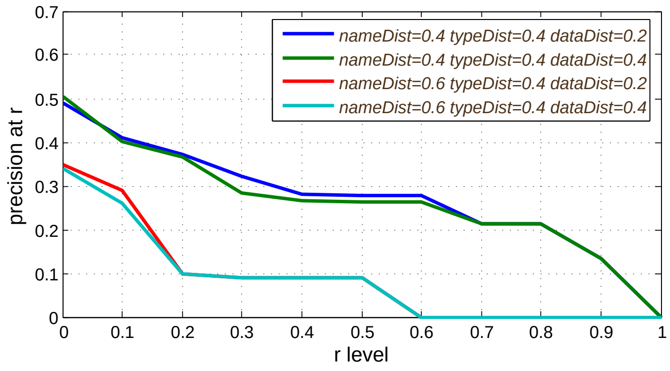


Figure 5.27: Multidimensional scaling graph-based search: 11-point precision.

The ground truth shows that although the algorithm retrieves relevent subgraphs, which can be seen from the ideal DCG curves, it does not put them at high ranking positions. This is also confirmed by the 11-point precision curves, considering the relevance of the first

150 ranked results, which remain constant at medium recall values 0.3-0.6 when $nameDist = 0.4$, and 0.2-0.5 when $nameDist = 0.6$ respectively, showing that the retrieved instances are still relevant at those recall values. We also experimented by using the similar scoring function for graph edit distance as for the A-star algorithm, by computing it as a weighted average distance in the $[0, 1]$ range, but there was no any performance improvement. Other types of scoring functions will be tested in future.

The performance of the algorithm can be explained by the loose conditions of the algorithm regarding the concept of similarity. Namely, the selection of candidate nodes is performed based on three appropriate distances (name, type and data distance). A node is chosen to be a candidate if at least one of the distances with respect to the query node is within the given distance values. These candidate nodes are joined with other candidate nodes considering the neighborhood information and the project, to form local subgraph patterns of the big project graph. However, the algorithm allows, to form not only subgraph patterns where each node satisfies all the three distance values, but also subgraph patterns where each node satisfies only one type of distance. For example, if all the candidate nodes of a pattern are within the type distance criteria, the corresponding pattern matches the query structurally, since their nodes are of similar types; It is also possible to have matches, where each node is within a different distance criteria. This kind of alternative matches are still reusable, although they have not been assumed relevant by the ground truth which is reflected in the results.

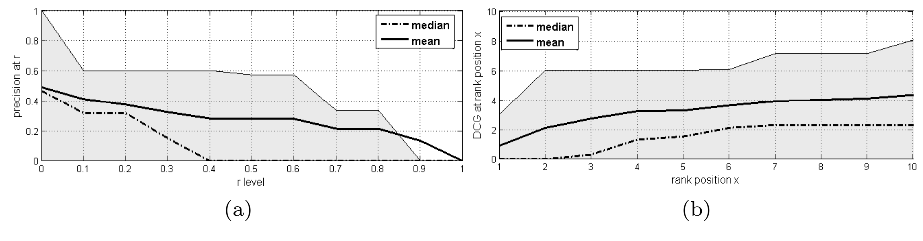


Figure 5.28: Multidimensional scaling graph-based search: average, median, lower and upper quartile of: 11-point precision (a) and DCG(b) for $nameDist = 0.4$, $typeDist = 0.4$ and $dataDist = 0.2$ configuration.

Figures 5.28, 5.29, 5.30 and 5.31 show the fluctuation of results across queries for DCG and 11-point interpolated precision for the examined distance configurations. The big area between the first and third quartile for both DCG and 11-point precision in Figures 5.28 and 5.29 for the configurations $nameDist = 0.4$, $typeDist = 0.4$

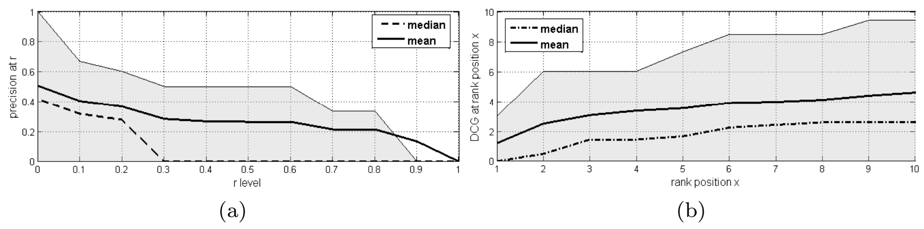


Figure 5.29: Multidimensional scaling graph-based search: average, median, lower and upper quartile of: 11-point precision (a) and DCG(b) for $nameDist = 0.4$, $typeDist = 0.4$ and $dataDist = 0.4$ configuration.

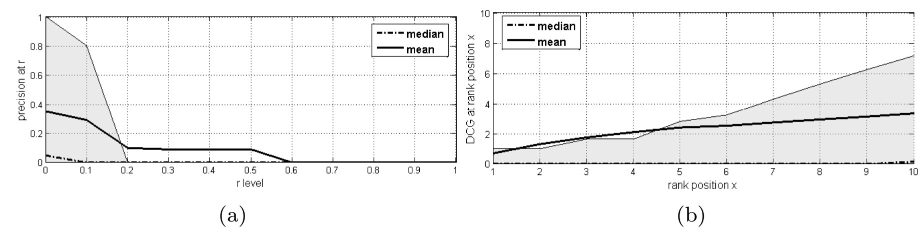


Figure 5.30: Multidimensional scaling graph-based search: average, median, lower and upper quartile of: 11-point precision (a) and DCG(b) for $nameDist = 0.6$, $typeDist = 0.4$ and $dataDist = 0.2$ configuration.

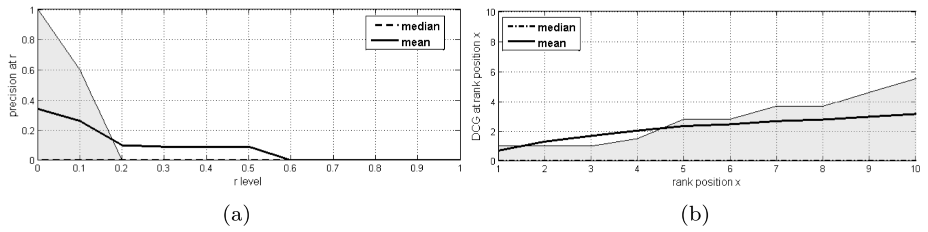


Figure 5.31: Multidimensional scaling graph-based search: average, median, lower and upper quartile of: 11-point precision (a) and DCG(b) for $nameDist = 0.6$, $typeDist = 0.4$ and $dataDist = 0.4$ configuration.

and $dataDist = 0.2$, and $nameDist = 0.4$, $typeDist = 0.4$ and $dataDist = 0.4$, show that there is a great variability in performance of the individual queries,i.e, there are some queries that perform well, while others perform bad. This is further confirmed with the values of the first and third quartile, which correspond to the minimum and maximum value at $recall = 0$ for 11-point precision, and the first rank position for the DCG. The size of the area does not decrease with the increase of the ranking position for DCG, while it decreases slightly for 11-point precision as recall increases, showing that the performance of the queries continues to vary until high levels of recall and at least until the 10th rank position.

The other two worse performing configurations in Figures 5.30 and 5.31 have less fluctuation in results across the queries which means that all of them do not perform well, also demonstrated with the low value of the median. For the 11-point interpolated precision, the third quartile has still high values for lower levels of recall (0.1), explained with the fact that there still exist some queries that return relevant results at those recall levels. Ragarding the DCG, the variability of the results increases with the increase of the ranking position.

We performed further analysis of the 11-point interpolated average precision for the individual queries considering the best distance configuration $nameDist = 0.4, typeDist = 0.4$ and $dataDist = 0.4$, and in Figure 5.32 we show the two best performing queries . As it can be noticed, the algorithm reaches a maximum precison of 1 at lower levels of recall (up to 0.3) for Query 6, and until recall of 0.8 for Query 2. For the other queries, the algorithm performs worse which influences the overall algorithm performance averaged across all the queries.

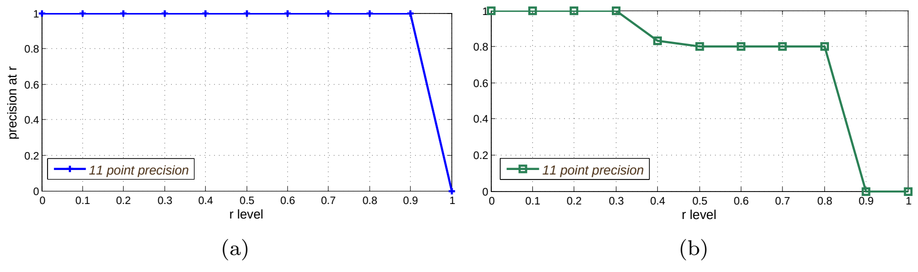
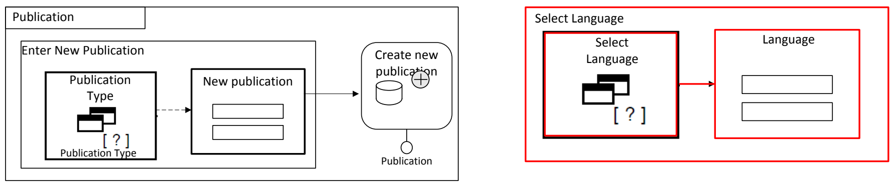
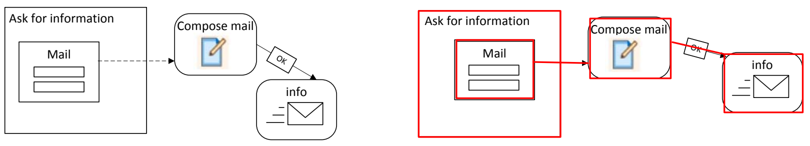


Figure 5.32: Multidimensional scaling graph-based search:11-point interpolated average precision for the best performing queries for $nameDist = 0.4$, $typeDist = 0.4$ $dataDist = 0.4$ configuration: Query 2 (a)and Query 6(b).

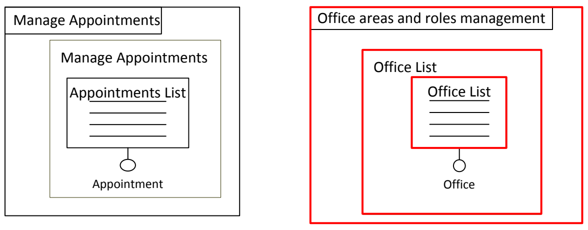
Figure 5.33 shows examples of good and bad performing queries, and their highly ranked matches, highlighted by red color, considered relevant or irrelevant by the ground truth, demonstrating that regardless of the ground truth relevance, the algorithm generates potentially reusable matches. Query 2 in Figure 5.33(a) that expresses how to enter and create a new publication record considering its type, is the best performing query, that has a partial match with the sample pattern. The query's page and two of its units match with the corresponding elements of the pattern based only on the type equality of the model elements. Figure 5.33(b) shows Query 6, the second best performing query, which expresses how to ask for information by sending a mail, and a sample of its corresponding relevant match which is its exact copy. Finally, Query 7 that expresses how to manage appointments by showing their list in Figure 5.33(c), represents a bad performing query whose model elements completely match with the corresponding structural pattern. Although considered irrelevant by the ground truth, this pattern completely corresponds to the query and can be reused.



(a)



(b)



(c)

Figure 5.33: Multidimensional scaling graph-based search: Example of a) good performing query and its corresponding relevant result, b) good performing query and its corresponding irrelevant result, c)bad performing query and its irrelevant result.

Query Execution Time The last experiment measures the performance of the multidimensional algorithm considering the query execution time, varying the size of each of the three node grids representing different feature classes (name, type and entity/relationship). The values are averaged over ten executions and over the queries and they are represented in Figure 5.34. The experiments have been performed on the same machine used to perform the same type of experiment on the WebML keyword-based search and A-star graph-based search (Intel dual Core Processor 2.4GHz, 6GB RAM, Windows 7 (64-bit) operating system).

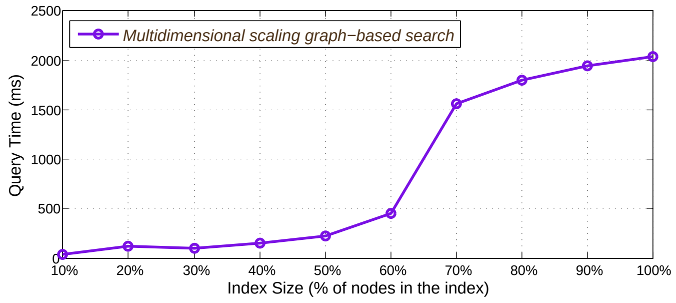


Figure 5.34: Multidimensional scaling graph-based search:Query Execution Time at varying number of indexed nodes for multidimensional scaling graph-based search.

As it can be noticed from Figure 5.34, multidimensional scaling graph-based search increases its execution time exponentially with the growth of the index size until 70% of the index size is reached; Further increase of the index size, causes the execution time to grow linearly. Another important observation is that the multidimensional scaling performs more efficiently than the A-star graph-based search whose execution time is given in Figure 5.17, which shows that using an index optimizes the performance and makes the multidimensional scaling algorithm scalable for larger data sets. Optimizations can be applied to the multidimensional scaling in order to further improve its efficiency.

5.4 User Study

The evaluation reported in Sections 5.3.1 and 5.3.3 compared the results of the WebML keyword- and A-star graph-based search systems with a gold data set constructed manually by experts and aimed at assessing the ability of each system to extract models similar to the user need, under the notion of structural and topical similarity provided by the experts. To evaluate the user-perceived utility of both systems during a development task, we conducted a controlled study organized in two distinct sessions, with the help of 25 industrial software developers (7 females and 18 males). Participants were volunteers with at least one project developed using WebML, with limited experience in exact model search and database structured search, engaged as follows:

- First, users had to fill-in a pre-experiment questionnaire, to provide demographic information and self-assess their experience with WebML on a 3-point Likert scale, ranging from 1 (novice) to 3 (expert). Of the 25 participants, 9 evaluated themselves as *expert*, 9 as *practitioner*, and 7 as *novice.*
- Before the start of the study, participants watched a video tutorial showing how to perform two different evaluations, described next.
- Next, users accessed an ad hoc Web application and performed the actual evaluation
- Finally, users filled-in a post-experiment questionnaire, where they could provide feedback in free text format.

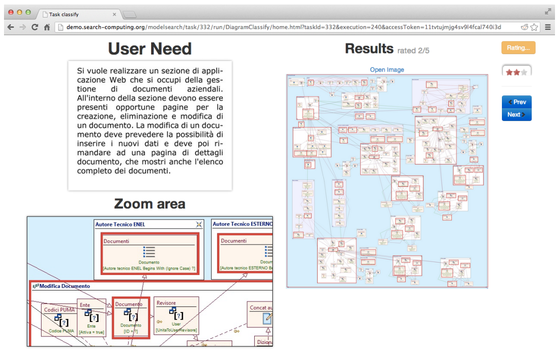
A pool of 10 tasks, defined in collaboration with the WebML experts and inspired to the development of the exemplary models used for the gold standard creation, was exploited in the user study. The following is an example of such tasks:

> *Assume you have to design a new Web application for the management of an e-commerce system. One of the requirements is the management of the sales operation; specifically, the site should contain a Web page devoted to the search of products in the catalogue; upon submission of the search conditions, the same page should show the list of products matching the user query. You want to identify existing projects (or fragments thereof) that can be reused to fulfill this requirement.*

Given a task description, the queries representing it in textual and WebML format were defined and respectively submitted to the WebML keyword-based and to the A-star graph-based system, set-up in their best configuration (the *Metamodel-dependent Navigational* configuration for WebML keyword search and the *Maximal Insertion* with $\lambda = 0.75$ and Levenshtein string similarity for A-star graph-based search, as discussed in Section 5.3). Results of query processing were collected and used for building the user evaluations described in the following sections.

User Study 1: single system evaluation

The first session elicited the users's judgement on the utility for reuse of each result computed by one of the two systems. Given a task such as the one exemplified above, users were presented the top-5 results, without disclosing which system they originated from. Users had to assess each result using a tertiary scale, where: (i) 0 meant not useful for reuse, (ii) 1 meant partially useful, and (iii) 3 meant very useful. Figure 5.35 shows the interface created for performing the User Study 1; it contains the task description and one result at a time, with commands for zooming the model, evaluating it, and scrolling to the other results of the top-5 result set. Each user evaluated the result sets of 10 tasks, assigned by mixing an equal number of responses to keyword- and content-based queries. To reduce learning bias and fatigue, the experiment was designed using a *graeco-latin square* scheme [116, 69], with system type (keyword-based and content-based) and task as dependent variables. To minimize the impact of prior experience in WebML projects, tasks were assigned to participants randomly. To reduce bias due to the rank position, the order of presentation of results in the interface was random.


Task classify
demo.search-computing.org/modelsearch/task/332/run/DiagramClassify/home.html?taskId=332&execution=240&accessToken=11tvtujmjg4sv9l4fcal740i3d
User Need
Results rated 2/5
Rating...
Open Image
Prev
Next
Zoom area
Autore Tecnico ENEL
Autore Tecnico ESTERNO
Documenti
Documento
[Autore tecnico ENEL Begins With (Ignore Case) ?]
Modifica Documento
Ente
Revisore

Si vuole realizzare un sezione di applicazione Web che si occupi della gestione di documenti aziendali. All'interno della sezione devono essere presenti opportune pagine per la creazione, eliminazione e modifica di un documento. La modifica di un documento deve prevedere la possibilità di inserire i nuovi dati e deve poi rimandare ad una pagina di dettagli documento, che mostri anche l'elenco completo dei documenti.

Figure 5.35: User Study 1: Interface of the evaluation system.

For each system, task, and result position, votes were averaged to calculate a global DCG curve for the keyword- and content-based systems, reported in Figure 5.36. Figure 5.37 shows the DCG curves, broken down task-by-task. Note that the DCG curves determined with the User Study 1 compare the result sets produced by the search systems with the best ordering of results emerging from the user's votes based on the perceived reusability of the project fragments with respect to the task description; conversely, the DCG curves previously shown in Sections 5.3.1 and 5.3.3 compare the results calculated by the system under multiple configurations with the gold standard created by the experts, who evaluated the technical quality of matches based on the degree of textual and/or structural relevance of the WebML area.

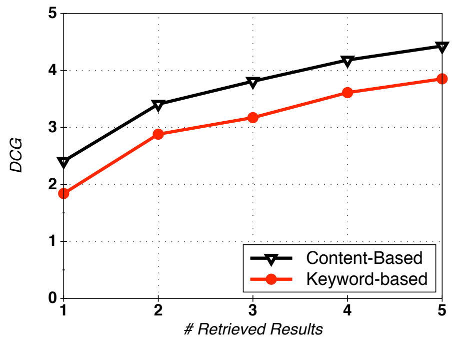


Figure 5.36: User Study 1: DCG curves averaging the user evaluations (top-5 results).

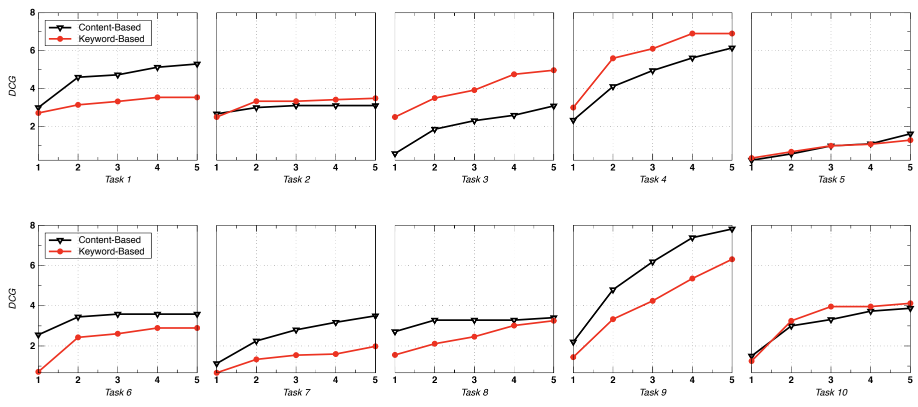
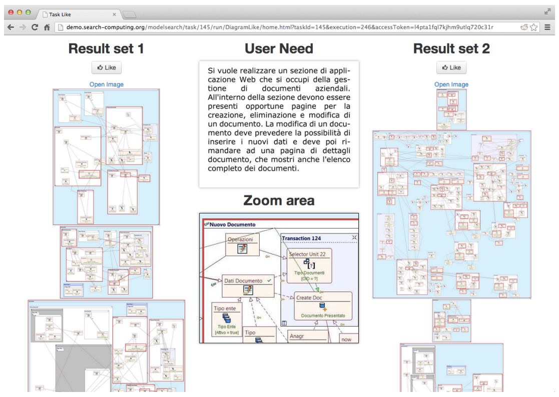


Figure 5.37: User study 1: Task-by-task DCG curves (top-5 results).

User Study 2: system to system comparison

The second user study focused on the direct comparison of the top-5 result sets produced by the WebML keyword- and A-star graph-based search systems. The experiment complements the first user study by including in the evaluation also the ranking performance of the two systems. To this end, we designed a pairwise comparison task, with the intent of reducing the cognitive effort that otherwise would be required for the separate evaluation of two ranked sets of models; the face-to-face appraisal of whole result sets supports not only the judgement about the relevance of the retrieved models, but also the direct comparison of the order in which these are presented. Given a task, users reviewed two result sets and indicated the one that in their opinion was globally more useful in terms of reuse, considering both the utility of the returned results, and their ranking positions. Figure 5.38 shows the evaluation interface developed for the second experiment: the description of the task is shown in the middle of the page, with the two result sets to be compared placed at its left and right. Figure 5.39 reports the direct comparison of the preferences for one system or the other, task by task.



Si vuole realizzare un sezione di applicazione Web che si occupi della gestione di documenti aziendali. All'interno della sezione devono essere presenti opportune pagine per la creazione, eliminazione e modifica di un documento. La modifica di un documento deve prevedere la possibilità di inserire i nuovi dati e deve poi rimandare ad una pagina di dettagli documento, che mostri anche l'elenco completo dei documenti.

Figure 5.38: User Study 2: Interface of the evaluation system.

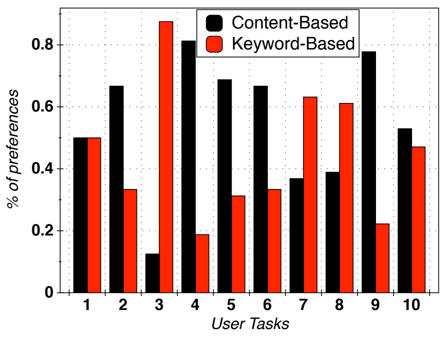


Figure 5.39: User Study 2: Task-by-task preferences for the two systems.

Analysis of Results

Coherently with the gold standard evaluation, both user studies show that the A-star graph-based search system provides, on average, better results than the keyword-based system, which suggests a correlation between the performance of a retrieval system and the user-perceived utility for reuse.

The DCG curves of Figure 5.36 show values similar to the ones described in Section 5.3.3, but with higher values for the A-star graph-based system; the histogram of Figure 5.39 show that the result lists produced by the content-based system have been preferred 60% of the times. Further analysis can be done by considering the task-by-task performance in Figure 5.37 and 5.39. The former shows how well the ordering of the result set *of a single system* adheres to the preferences expressed by the users; the latter shows, task-by-task, which system the users preferred, when confronted simultaneously with the result sets produced by both ones. Four situations emerge:

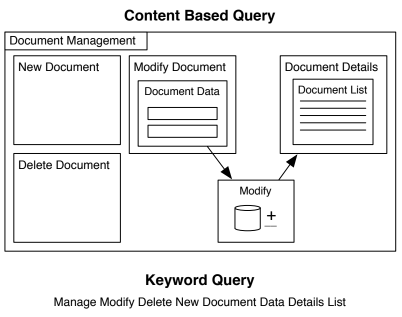
- *Content-based search is better* for `Task 1`, `Task 2`, `Task 6` and `Task 9`. Note that in Task 1 keyword- and content-based search get an equal share of preference in the direct comparison of result sets, but the DCG curve shows that the ordering of results is closer to the user's judgement for content-based search. Figure 5.40a reports an example of content-based query in this class: it expresses an object management pattern (distinct pages for the creation, modification, and deletion of instances) over the `document` entity. The better performance of content search is due to the nature of the query, which exploits a very characteristic design pattern and thus benefits from the match computed using graph similarity. Conversely, the corresponding keyword-based query contains rather frequent words ("document" occurs 81 times in the repository, "create" occurs 112 times, "modify" occurs 127 times, and "delete" occurs 118 times), which do not produce selective matches in the text retrieval system.

- *Keyword search is better* for `Task 3` (depicted in Figure 5.40b) and `Task 4`. In this case the selectivity of textual terms dominates the characteristics of the structural pattern. For instance, for `Task 3` the total number of occurrences of the term "default" in the repository is 5, while the term "subject" occurs 9 times. The specificity of these terms, which are rare in the repository, makes the keyword-based search more selective than the content-based counterpart, even if the content-based query exhibits a fairly articulated model. The greater number of preferences obtained by `Task 4` in the second user study is justified by visual bias in the comparison of result sets (see point (2) below), which diminishes the perceived utility of the retrieved set of results.

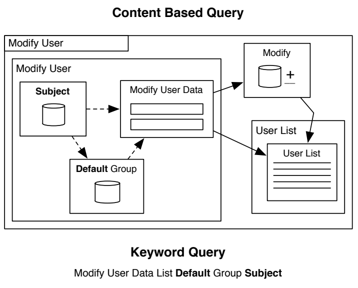
- *Comparable results* for `Task 7`, `Task 8` and `Task 10`. In this case both systems exhibit a comparable performance, with no clear winner or discordance between the direct comparison of results sets and the appreciation of each result in isolation. As an example, Figure 5.40c shows `Task 10`, which features a fairly complex structural model and good keyword selectivity (terms such as "dictionary", 44 occurrences, and "contract", 5 occurrences).
- *No satisfactory results* are retrieved for `Task 5` (shown in Figure 5.40d), which expresses a need formulated either as a model fragment with rather general structure and labels or as a bag of keywords having low selectivity. In such a case, both A-star graph matching and *TF-IDF* text matching do not perform well, as no distinctive feature of the query allows for high-confidence retrieval.

Further analysis of the results of the two user studies, also confirmed by the feedback provided by the users, show that:

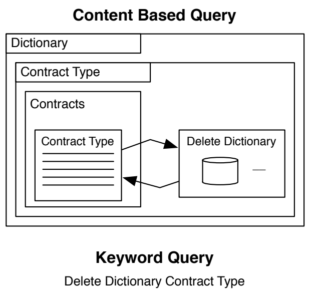
1. In some cases (e.g., `Task 1`, `Task 2`, `Task 6`, and `Task 8)` the main contribution to the utility of the result set is due only to a very relevant top-1 result, as shown by the DCG curve starting at the highest value for x=1 (i.e., 3) and then flattening out. In this case, the system retrieves a very good match to either the keyword-based or the content-based query, but then the other results are judged much less useful.
2. Some other tasks (`Task 4` and `Task 9`) instead retrieve results that are perceived as good all over the result set, as shown by a steadily increasing DCG curve both in keyword-based and content-based search. In the direct comparison of the result sets, users tend to assign higher preference to content-based results though, even when the precision and order of the result set is judged better for keyword-based search (this is the case of Task 4). Post-experiment comments from the users suggest that the favorable perception for content-based search is influenced not by the relevance per se of the result, but by a visual bias induced from the highlight of the matching elements. Content-based search results match mostly elements that appear visually also in the content-based query and in its neighborhood, whereas keyword-based search matches all elements



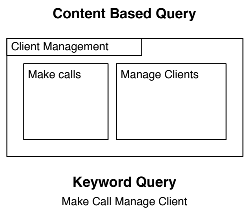
(a) `Task 9`: good content-based search



(b) `Task 3`: good keyword search



(c) `Task 10`: equivalent keyword and content search



(d) `Task 5`: no satisfactory results

Figure 5.40: Example of task with a) good performance in content-based search, b) good performance in keyword-based search, c) equivalent performance, and d) unsatisfactory overall performance.

that contain at least one keyword. In the abovementioned tasks, it was easier for the users to appreciate the reusability of the content-based result than of the keyword-based one, which had many highlighted elements and resulted confusing.

3. Another factor that blurs the perceived differentiation between keyword- and content-based search is the size of the returned model element. Tasks like `Task 7`, `Task 8`, and `Task 10` happen to match well with rather large WebML areas, making it more difficult for the users to perceive the utility for reuse.

5.5 Discussion

Relevance of Metamodel information

Overall, the results of the experimental evaluation show that the inclusion of metamodel-dependent information in the model-search process is beneficial for performance; this is demonstrated both in the WebML keyword-based search system, where the evaluated *Metamodel-dependent* strategies outperform the *Metamodel-independent* one, and in the A-star graph-based search system, where the injection of metamodel information in the node similarity function provided a considerable performance boost. However, in keyword-based search, the very simple approach of extracting the text content from projects and indexing it with off-the-shelf IR tools still yields acceptable results (MAP = 0.77). The same can be observed with the multidimensional scaling graph-based search algorithm which introduces additional metamodel knowledge with respect to the other algorithms in the indexing and search process. The evaluation of the multidimensional scaling algorithm demonstrates that the performance improves if all the available metamodel knowledge is used (i.e. both the type of the model element and the Data Model information).

For UML keyword-based search, the inclusion of metamodel information slightly improves the performance for the class granularity indexing strategies, while including neighborhood information slightly worsens the results. However, metamodel-independent indexing techniques have achieved very high performance, especially for the project granularity case (MAP = 0.98).

WebML Keyword- Vs. A-Star Graph-Based Search

We compare WebML keyword-based search and A-star graph-based search in their most performing settings, respectively the *metamodel-dependent navigational configuration* and the *Low type contribution, maximal insertion* configuration that uses *Levenshtein* distance.

The MAP values suggest that A-star graph-based search ($MAP = 0.86$) is overall more precise than WebML keyword-based search ($MAP = 0.81$); however, results from 11-point interpolated average precision show that the best WebML keyword-based experiment at *recall* $= 0$ slightly outperforms A-star graph-based search, as the former features a precision of 1. A-star graph-based search provides better precision for $(0.1, 0.2, 0.3)$ recall levels (up to 30% of relevant projects); for greater recall levels the WebML keyword-based search consistently outperforms A-star graph-based search. A similar performance profile resulted from the first experiment of the user study, where the DCG curves resulting from the evaluation of the top-5 results show that, on average, the A-star graph-based system is perceived as performing slightly better for reuse purposes.

Therefore, we might conclude that, in the evaluated setting, content-based search is suitable for applications where precision matters the most. On the other hand, keyword-based search can prove suitable in applications where recall is important (e.g., recommendation systems). Obviously, these considerations must be taken with care, because comparing information retrieval results across diverse systems and query paradigms can only give a coarse indication of the respective capabilities.

Multidimensional Scaling Vs. A-Star Graph-Based Search

Although the multidimensional scaling graph-based and A-star graph-based search are configured for the WebML dataset they have a different purpose; The multidimensional scaling graph search efficiently finds all the small localized patterns in a project graph considering additional metamodel information from the WebML Data Model, while the A-star graph search finds the most similar subgraph in an area. The evaluation demonstrated that A-star graph based search performs better than the multidimensional scaling graph based search for all the applied evaluation metrics. Regarding the execution time, as expected, multidimensional scaling search is more efficient than the A-star graph search, thus overcoming the problem of scalability at the price of performance loss. However, multidimensional scaling still finds some well matching structural patterns that can be reused.

Therefore, if precision matters, and the size of the dataset is not a concern, A-star graph-based search is more suitable. In case where efficiency matters, the multidimensional scaling graph-based search should be preferred. As previously, these considerations should be taken with a caution, because both of these algorithms are searching for different types of matches.

Search system design guidelines

The user study revealed possible sources of cognitive bias that may alter the perception of the utility of the retrieved results, even if they are relevant from a technical standpoint (i.e., they do contain the queried keywords or model fragment). These results suggest two recommendations for the designers of model search systems:

- *Project segmentation*: project segments should be semantically meaningful as potential units of reuse and have comparable size. Good example are WebML areas, while classes, used as project segments for the UML dataset, might be small and do not capture the model structure.

- *Matching results highlight*: keyword-based search may be an interesting tool to recall more potentially relevant matches than content-based search, but it suffers from the visual overhead induced by the matches of many model elements of different types. As a possible countermeasure, the interface should support commands for toggling the highlight of selected metamodel types. In this way, the user could selectively turn on the highlight for the type of model element he is looking for (e.g., only for pages, or units of a given kind), exploiting metamodel information also for the visualization of results.

Project design guidelines

As a final remark, the findings about the performance of model-based search systems can be read also as recommendations for project developers and DSL designers. In general, using selective and precise textual labels for model elements is the first best practice to consider; given the importance of the text match component in both WebML and UML keyword- and A-star graph-based search, using scarcely descriptive labels for model elements obviously degrades search. For the WebML dataset, most of the reviewed projects contained no comments associated with model areas, pages, and meaningful patterns, even if this feature is supported by WebML and WebRatio. Another best practice for WebML models is the adherence to standardized design patterns: many functions (e.g., the interfaces for performing CRUD operations on data, composition and sending of messages, and so on) can be modeled in standard ways, but the projects exhibited a lot of semantically equivalent but slightly different variants for doing the same thing. This (not so necessary) variability impacts the calculation of the graph edit distance in the A-star and the multidimensional-scaling graph-based search, which is sensitive to link and containment topology. Last, DSLs that are designed to be extensible and incorporate third-party components, like WebML, should care for preserving the precision of the metamodel: a good classification taxonomy of custom components can help the metamodel type part in finding candidate nodes for content-based search.

5.6 Threats to Validity

The thesis proposes a set of approaches for applying metamodel-based search to model repositories, verified with concrete experiments over repositories of models. All of the experiments were performed on models conforming to a specific DSL, namely WebML. Furthermore, keyword search was also tested on a repository of a general-purpose modeling language, i.e., UML class diagrams. However, the quantitative results from the experiments, are relevant for the WebML and UML case, and cannot be directly generalized to other languages. Still, the discussed method for studying the configurations of the search systems and for tuning their parameters can be reused, as it was already demonstrated by applying keyword search on a UML repository. Also, as discussed in Sections 2.2.1 and 2.2.2, WebML is a representative of a family of languages for interactive application modeling, while UML is a general purpose modeling language, widely adopted and used across different domains. This makes the experiments described in the thesis, although not directly portable to other languages, potentially useful for supporting the evaluation of the search system in other languages for model-driven interactive application development. While it would have been good to evaluate the system on other larger repositories, finding realistic and sufficiently rich datasets has been challenging. Indeed, models are the core asset of MDE companies, which are therefore reticent to share them. Anyway, the WebML repository we have been able to collect contains a considerable amount of model artifacts (a total of 19,246 searchable elements have been counted) and covers a wide spectrum of application domains. Based on the user feedback and on our empirical assessment of the repository, we think that the obtained results are accurately describing the system behavior for the WebML modeling language. The UML repository contains 84 class diagrams where the majority of projects is small, which makes them inappropriate for content-based search. However, the richness of the terms present in the UML repository's vocabulary makes it suitable for keyword-based search, thus configuring and testing the keyword search on two different repositories. Although classes might be small units for project segmentation, they served to investigate how small segments influence the performance.

Another potential threat comes from the quality of the testbed and of the gold standard. We applied all the known techniques for reducing the bias of evaluators and we were not included in the set of experts evaluating the data. The same care has been applied to the definition and execution of the user study. While the projects in the WebML repository could not be chosen (they were provided by WebRatio), the selection of the queries for the experiments was performed based on various language and dataset objective characteristics, to minimize the introduction of bias from our side, in consultation with experienced model developers. The selected number of tasks (10) has been deemed a reasonable compromise between the effort required for constructing the gold standard and the coverage of several aspects of the DSL and of typical design patterns that we observed in the provided repository. When designing queries, it is important to consider the repository content (in terms of vocabulary and design patterns) in order to build the ground truth and thus, enable the evaluation of our approaches. In general, for purposes of using the repository, a user can create a query in a specific modeling language without having any knowledge of the repository.

For the UML repository, the projects were selected randomly out of a publicly available dataset, in order not to influence the choice of projects, and to be more similar to the WebML case. The same criteria applied to the WebML dataset were considered when creating queries.

One might argue that the ground truth in the multidimensional scaling graph-based approach evaluation is altered. However, the ground truth is not modified, it is just used to find the relevance of subgraphs generated in the result set, as in the other methods. The same subgraph will always have the same relevance.

The result of the user study could have been influenced by the number and expertise of the involved participants. However we believe that number of involved users (25) suffices for a meaningful evaluation, while the levels of expertise were fairly distributed.

Finally, a last factor that could have influenced the user study is the user interface we built for the purpose. This interface may have introduced exogenous complexity to the evaluated variables, e.g., due to factors such as the limitations of a browser-based interface, the system response time, the cognitive load associated with a new interface, time pressure, and the kind of interaction commands allowed. To minimize the impact of such factors, we provided equal training to all the participants, and did not pose a time limit for the execution of the evaluations. However, as we have commented in Section 5.5, model-driven search surely poses challenges in the system interface design, related to model complexity and size and highlight of matches, which we consider interesting future research directions to explore.

Chapter 6

Related Work

This chapter describes the state-of-the-art techniques for searching artifacts in software repositories and graph data. Section 6.1 focuses on searching artifacts from software repositories and it includes source code search, software components search and model search. The model search section (Section 6.1.3) is divided into approaches that perform keyword-based search and content-based search on models. The model content-based search (in Section 6.1.3) incorporates graph search techniques, structural query languages and content-based approaches that use specific algorithms for searching models. Finally, section 6.2 provides an overview on the current approaches for searching graph data.

6.1 Search of Artifacts in Software Repositories

6.1.1 Source Code Search

The need for searching source code for improving the process of software development and supporting software reuse resulted in emergence of several on-line tools and research works that implement and explore this problem. Some examples of existing on-line tools for sharing and retrieving source code are *Google code*, *Snipplr*, *Koders*, and *Codase*[1]. As explained in [21],the most basic solution is the case where queries in form of keyword(s) are simply matched to the code and the results are the exact locations where the keyword(s) appear in the matched code snippets. However, online tools allow advanced search by using regular expressions (Google Codesearch), wildcards (Codase); supporting search of specific syntactical categories, like class names, method invocations, variable declarations

[1]Sites: http://code.google.com, http://www.snipplr.com, http://www.koders.com, http://www.codase.com

(Jexamples and Codase); making the search more specific by indicating fixed set of metadata (e.g., programming language, license type, file and package names). Source code online tools also have to consider a way to compute a relevance score between the query and the matched source code, and present the corresponding results to the user [21]. Regarding this aspect, some approaches retrieve a list of matches without providing ranking, while others implement IR-style ranking using the standard TF/IDF measure, or ranking which besides the matches with the source code takes into account the project properties such as recency of the project, number of downloads, activity rates etc.

Research works for source code search are based on IR techniques [52] and techniques which employ the source code structure in the search [11, 61].

Sourcerer [11] is an infrastructure for large scale indexing and analysis of open-source code, upon which code search engines and services can be built. Sourcerer crawls the internet looking for Java source code from public web sites, open source repositories and version control systems. The code is parsed, analyzed and stored in three forms: managed repository, containing versioned copy of the original project content; Code Database that stores models of parsed projects based on a metamodel, and Code Index which stores keywords extracted from the code. The metamodel for structural information is based on the Chen's entity relationship metamodel.

Maracatu [46] 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, combining text mining and facet-based search. 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 provided before its download.

Exemplar (EXEcutable exaMPLes ARchive) [87] 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. It does not only find query keyword matches in the descriptions and source code of applications, but keywords are also matched with words in the help documentation for API calls. The matches of the API calls are compared against the functions' names invoked in the applications. The ranking is obtained by considering the word occurences, number of relevant API calls and the dataflow connections between them.

CodeBroker [127] is a system that uses techniques 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 for the information delivery concept comes from the fact that reuse is often unsuccesful because of users' lack of knowledge and their inability to create a well-defined query. CodeBroker utilizes doc comments and program's signatures, discourse models that partially describe the developer's tasks, and user models that represent users' backgound knowledge about the repository thus personalizing information delivery with respect to the developer's unique needs. The component repository contains indexes from a standard Java documentation and links to the Java documentation, and reuse queries are extracted autonomously by monitoring development activities. Information delivery reduces the cost of software reuse, making unanticipated (unknown) components easily accessible, and motivates developers to change the design approach towards reuse.

A method for finding traceability links between source code and free text documentation expressed in natural language, by using probabilistic and vector space information retrieval model is proposed in [6]. The query consists of identifiers extracted from the source code component to retrieve documents relevant to the component. In the probabilistic model, free-text documents are ranked according to the probability of being relevant to a query based on a stochastic model that assigns a probability to every string of words taken from a prescribed vocabulary. The vector space model treats documents and queries as vectors in an n-dimensional space, where n represents the number of indexing features (words in the vocabulary). Documents are ranked against queries by computing a distance function between the corresponding vectors.

The work in [104] proposes an approach for code search that uses an open set of program transformations to map retrieved code into a user specification. The user specifies its query through keywords, class or method signatures, test cases, contracts and security constraints. The keywords are used to select an initial set of candidate matches which is then transformed into a more appropriate resultant set using Abstract Syntax Trees (AST). This set is restricted by checking the solutions against the static specifications. Each solution is then augmented with an additional dependent code from the original file. The final solutions are checked against dynamic specifications such as test cases. Depending on the test result, additional transformation might be applied. The solutions are formatted using adaptive formatting and presented to the user.

CodeGenie [79] is a tool that utilizes test-driven approach to search and reuse code available in large-scale source code repositories. Developers design test cases for a desired feature first, which CodeGenie uses to automatically search for an existing implementation. The suitability of the candidate results is checked by incorporating the result into the developer's project and tested using the original tests. CodeGenie is built on top of Sourcerer [11] which handles code searching and wrapping. Test cases improve the quality of the found results since they test code results retrieved from many different sources.

The work in [71] proposes a source code search using ontologies to model and connect source code fragments extracted from repositories found on the Internet. The approach allows to search and reason across project boundaries while dealing with incomplete knowledge and ambiguities. Source code is modelled using Description Logic (DL) and Semantic Web reasoners supporting complex queries and handling missing knowledge. The knowledge base is built incrementally without the need to re-visit fragments or compile the source code, and it can be visited by web crawlers.

Assieme [60], is a web search interface that allows for common programming search tasks by combining information from web-accessible Java Archive (JAR) files, API documentation and pages that include explanatory text and source code. The approach finds (and resolves) implicit references to JAVA packages, types and members within the sample code on the Web, using relevant data collected from different sources on the Web. The interface provided by Assieme allows programmers to quickly examine different APIs that might be appropriate for a problem, obtain more information for a particular API and see API's usage samples. This way programmers can obtain solutons faster and using fewer queries with respect to a general Web Search interface. Assieme uses information regarding the extracted code to form a ranking score, which depends on the ratio between the text length that surrounds the code samples, and the code sample length,favouring pages that contain more text.

Portfolio [88] is a source code search system that provides retrieval and visualization of functions,and supports the analysis of chains of dependencies of the retrieved functions with the help of navigation and association models.It supports programmers in finding relevant functions that implement high-level requirements reflected in query terms, determining how these functions are used in a highly relevant way with respect to a query, and visualizing dependencies among retrieved functions to show their usage. This is achieved by combining Natural Language Processing (NLP) and indexing techniques with PageRank and Spreading Activation Network (SAN) algorithms. NLP and indexing techniques help to find initial focus points with respect to the query's keywords, PageRank models the behavior of the programmers, and SAN elevates highly relevant function calls chains to the top of the search results.

Strathcona tool, described in [61], locates relevant code in an example repository based on heuristically matching the structure of the code under development to the example code. The approach extracts automatically necessary information to query the repository, freeing the developer of learning any query language, or writing the code in a particular style. The repository is generated easily from existing applications, and consists of a relational database that stores the code structure through classes,methods, fields, inheritance relations, types instantiated by the code and the calls between the types. The examples returned by Strahcona are subsets of the code provided to the repository. The tool employs three types of heuristics: inheritance heuristics,that matches on the parents and types of fields of a class, calls heuristics based on the targets of the method calls, and uses heuristics based on the types a developer declares and uses within a method.

SNIFF [31] is a novel code search technique which allows flexible searching for code while using plain English to obtain a small set of code snippets to perform a desired task. The method uses the API documentation provided in the library methods to annotate undocumented publicly available Java user code with plain English meaning. The annotated code is then indexed for performing search with free-form queries. The approach takes a type-based intersection of the candidate code snippets obtained from the query search, removing irrelevant lines of code and keeping the code lines which co-occur (through presence of keywords or issues related to correct implementation), to generate a set of small and highly relevant code snippets. SNIFF eliminates the requirement of previous knowledge about API, thus achieving a more effective search.

Coogle [108] is a search engine that extends the idea of using similarity measures for determining the similarity of Java classes. It transforms abstract syntax tree representations of the Java classes' source code into intermediary FAMIX tree representations, a programming language-independent model representations of object-oriented source code. As a result, the similarity of these trees is determined by 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 evaluation of the effectiveness of the proposed algorithm showed that the tree edit distance produces the best results.

6.1.2 Component Search

Software components represent a cohesive and compact unit of software functionality with a well defined user interface, ranging from simple programming classes to more complex web services [62]. Software component search supports their reuse by assembling retrieved software components into applications which improves the quality and decreases costs of software development. In the following part of this subsection are given examples of component search works that search software components in general [48], or search for specific type of software components [65, 102].

Agora [110] is a specialized search engine that automatically indexes and generates a worldwide database of software products classified by component model. The users can search for components by describing specific properties of a component's interface. The system combines Web Search Engines with an introspection process, which corresponds to a component's capability to provide information about its own interface. Agora supports data collection which includes finding components on the Internet, collecting their interface information and record it in a local database, to allow data search and retrieval. Special functions let users narrow the search criteria which depends on the characteristics of the component type. Agora also uses lexicons to facilitate search in a specific domain, appending it to the query.

The work in [48] proposes an approach for searching software components by associating algebraic specification to each software component. It allows user queries to be represented with syntactic declarations and results for sample execution. Standard programming notation can be used for the queries which is subsequently translated in algebraic notation. The search process exploits a multilevel filtering strategy, implemented incrementally in order to allow backtracking to lower ranked components. The result of each level gives partial matches: early stages of filtering narrow the search space by using simple procedures, middle levels find partial signature matches, and the final filter uses semantic information with term rewriting. Indexes may also be used to speed up early and middle filtering.

SPARS-J is a software component search system which treats source files of Java classes as components [65]. For this purpose, it uses a component rank model that represents a software component library as weighted directed graph, such that graph nodes correspond to components, and graph edges represent cross-component usage. Graph nodes are ranked by their weights , defined as eigen vector elements of an adjacent matrix for a directed graph. In this way, 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.

The approach [102] is used to discover web services using a vector space search engine to index descriptions of already composed services. The approach joins detached document repositories to a single one and execute queries upon the vector search space. A service description in form of WSDL file or UDDI registry contains different types of data: plain-text user comments, endpoint URLs, types and their attribute names, messages, and XML comments. Keywords are extracted from this data and a vector is generated for the document based on the keyword frequencies. The query processor takes a query string and splits it into a list of keywords used to build a corresponding query vector. The query vector is projected into a local term space and a cosine similarity is used to evaluate the query vector with respect to the document vectors.

Merobase [2] is an open-source search engine for software components: source code, compiled software components,web services, physical and logical containers, that references over 10 million components. Merobase specializes in finding components based on their interface, rather than the strings they contain in the source code. It supports searches using the Merobase Query Language (MQL) which allows to look for components with a particular name or contain a given string in the source code. Components can also be searched based on a particular logical (function-oriented or logical-oriented) abstraction. Merobase also supports constraints to narrow down the search. The queries can also be formulated in a programming language syntax or in UML-like operating systems.

The technique introduced in [17] uses an ant colony algorithm for generating rules to store and search components in a software repository. The user query is interpreted as a set of keywords and WordNet is used to further expand the query for synonyms and identification of relations among these keywords. Generating rules for component classification is an important step in identifying software components for a given context. An ant colony algorithm selects terms and it derives their inter relationships. If a term contributes

[2]http://www.merobase.com

to a given context, a rule is generated. Rule quality is evaluated on the basis of various parameters. If quality of two consecutive rules converges, it is included in the list of rules for a corresponding context.

Automatic Tags Extraction (ATE) retrieval proposed in [130], is an approach for retrieving components from large-scale component repositories by automatically extracting component tags from application domain terms, high-frequency, high-weight and facet terms in the application's document description. Then, ATE uses improved Vector Space Mode(VSM) similarity algorithm to retrieve from the index constructed from the extracted tags. Queries are submitted in a natural language which is semantically expanded to improve the retrieval keywords used to extract functional terms. An index is used to find intersection of functional terms in order to identify appropriate components (which have all the functional terms) for the candidate set. Similarity is calculated for each candidate component, reading tag information from the tag index and comparing it to the retrieval keywords allowing to sort the results.

The tool in [55] proposes a software component repository that uses components' aspects to index and query components according to their high-level systemic characteristics. The aspects describe the required and provided services, and related and non-functional constraints for capabilities like their user interface, persistence, distribution, security and collaborative work. Index on component's capabilities is generated through use of high-level component characterisations. The tool considers user queries about component capabilities and it considers the context in which queries are performed. A partial automatic query reconstruction is also enabled, based on the reuse context of the component. New components can be added to the repository's index automatically, generated from their high-level aspect characterizations.

Woogle [41] 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 algortihm 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. 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 lets the user 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 composition of operations that achieve the desired functionality specified in the search.

Use of domain-independent and domain-specific ontologies for retrieveing web service from a repository is proposed in [117]. 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 overall similarity score. 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.

6.1.3 Model Search

Model search approaches utilize the underlying model structure for retrieving models from model repositories. Regarding the model type, a majority of model search approaches are for Business Process Models, although few works exist for searching UML models too. Besides traditional keyword querying [84, 43], query by example (content based search) is used to incorporate model structure in the query [121, 13].

Keyword Model Search

Keyword search uses a set of keywords to query model repositories, and it is inspired by traditional information retrieval based on frequency of occurence of search terms. However, there exist works which include the model structure in the matching process [131, 84].

The next couple of works focus on keyword search of UML models.

Moogle is a model search engine that uses UML or Domain Specific Language (DSL) metamodels to create indexes for evaluation of complex queries [84]. It also formats the search results in a more readable way by removing irrelevant tags and characters. Model element's type, attributes and hierarchy with respect to other 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 higlighted 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. Unlike our approach, Moogle supports only text queries which are refined by specifying the type of the desired model element to be returned.

The approach in [131] proposes an UML model querying method based on structure pattern matching. The system requires a target model pattern and UML system model for querying. The target model, described in a textual query language, stored in a textual file, is parsed and matched with the system UML model. The structure matching considers the model structure, which includes the relations among classes, and features for each model element. More specifically, the matching algorithm checks for class structure matching, based on the match of the classes general information, their attributes and operations, and relation structure matching which defines the match in the model structure. Different model elements and feature values have different discrimination property of models depending on their frequency in the model, such that rare model elements and features allow pruning of unmatched elements. This approach considers the UML model structure, but it also requires specifying a model pattern in a query language.

CORE [43], 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 is 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. Although CORE is used for retrieving ontologies only, the idea for using semantic similarity for search might be used in our future work.

The following two methods use Information Retrieval techniques for finding workflows. The work [112] is more restrictive since all query keywords need to match a workflow, while [34] composes workflows out of existing repository which differs form a user goal we set,i.e., to find existing models or model fragments in the repository.

WISE [112] 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. The result set is constructed dynamically by categorizing workflow hierarchy nodes based on the matches to the query keywords.

Auspice [34] is a system for automatic composition of workflows, given a keyword query. The approach indexes data sets and Web Services on their tagged keywords and metadata, generates ontology based on concepts relationships, and uses them to construct the workflows. An IR-based retrieval model evaluates the index relevance considering the query keywords. The concepts are mapped onto query keywords to construct sets of unsubstantiated concepts,containing targeted concept elements, and a set of value-substantiated concepts, that have been assigned a value from the query. The two sets together with the ontology are used to find and compose workflows that derive some user-targeted concept with the support of the given attributes specified in the query parameters set. The resulting workflows are ranked according to a score computed as a function of the number of concepts relevant to the query. The system not only returns previously defined workflows, but it also identifies and composes web services and data sets to respond to the information received by the query.

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 [21]. 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 keyword search part of the thesis extends this research work, by implementing the architecture and evaluating different configurations using ground truth for two datasets: WebML and UML.

Content-Based Search

Some content-based search works use graph representation of models thus proposing graph search solutions [105, 135] which might include indexing [78, 125], while others define querying languages for better expressing the user need [26, 12]. There are also content-based approaches that apply specific search algorithms [86] that do not belong to any of the above-mentioned categories. Based on that, we divided the content-based search state-of-the-art into graph model search works, works that describe structured query languages and other content-based approaches.

Graph Model Search

In [105], four graph mathing 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' (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 the 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. 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. Three similarity metrics for querying business process models are presented in [39]: label matching similarity, structural similarity, which considers the topology of models, and behavioral similarity, which focuses on the causal relations between models. Label matching metric computes optimal matching between process models' nodes by comparing their labels. The structural similarity metric uses the representation of process models as graphs and existing techniques for comparison based on graph-edit distance. The behavioral similarity metric represents the causal relations between tasks in models as causal footprints which provide abstract representation of the models' behavior. All three metrics outperform text-based search engines, while the structural similarity metric outperforms the other two proposed metrics. Although the A-star algorithm of the graph-based part of our work is inspired by the abovementioned approaches, the existing works are restricted to queries over BPM models, which have a simpler syntax and semantics than a DSL for interactive application front-ends modeling.

The following are examples of graph-based search over business process models. Although they use different search approaches, all of them search for similar non-exact models, allowing mimatches in nodes and edges between a query graph and a model graph.

An example of graph matching on process models is presented with the framework for process modeling and deployment [83]. It consists of a constraint-based process modelling 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.

The work [122] presents a framework for comparing complex process models combining metrics for comparing whole or partial process models, graph partitioning to compare the component parts of process models at different levels of abstraction, visualization techniques inspired by relief maps for intuitive model presentation, and statistical significance tests to give meaning to metrics and evaluate the confidence. Graph partitioning based on Edge Betweenness and Spectral Partitioning is used to recursively split process models into logical subgraphs and calculate the similarity metrics between them as they are complete process models, which are then used to match pairs with the lowest distance. The respective metrics are based on the frequency distribution of the data attributes and on the weighted average of the difference between the weighted edges and weighted nodes. Statistical tests were performed considering the frequencies of nodes and edges in the models.

The work in [54] introduces a BPEL ranking platform for service discovery employing graph matching, which finds a set of service candidates satisfying user requirements and ranks them using a behavioral similarity measure. A graph error-correcting matching algorithm is used for approximate matching, starting from the subgraph-edit distance, and then extending it by adding new graph-edit operations that consider the difference of granularity levels that could appear in the two models: decomposing a vertex into two vertices, and joining two vertices into a single vertex. The approach considers two types of vertices: activity nodes, representing business functions, and connector nodes, expressing control flow constraints. When comparing process graphs, the approach checks how mapped activities are connected and defines rules for comparing different types of activities.

Another behavioral similarity measure for artifact-oriented business processes, using the Petri Net notation, is proposed in [82]. It computes the similarity between business core data by measuring the similarity between key artifacts, the similarity between task dependency relation sets of the task paths according to the lifecycle features of the core artifact, and the similarity between attribute assignment sequence sets in the task execution path.

The work [122] presents a framework for comparing complex process models by (i) combining metrics for comparing whole or partial process models,(ii) performing graph partitioning to compare the component parts of process models at different levels of abstraction, (iii) incorporating visualization techniques inspired by relief maps for intuitive model presentation, and (iv) applying statistical significance tests to give meaning to metrics and evaluate their confidence. Graph partitioning based on Edge Betweenness and Spectral Partitioning is used to recursively split process models into logical subgraphs, calculate the similarity metrics between them as they were complete process models, which is then used to match pairs with the lowest distance. The respective metrics are based on the frequency distribution of the data attributes and on the weighted average of the difference between the weighted edges and weighted nodes. Statistical tests were performed considering the frequencies of nodes and edges in the models.

In [96] a technique for process models retrieval based on clustering of related pairs is discussed, which combines semantic, string-based, and an hybrid metric for comparing process models. The proposed approach finds node label based differences between two process models by disaggregating them into similarity related subgraphs. The approach considers string-based similarity, semantic similarity, as well as hybrid-similarity approaches for computing label-based similarity. Related cluster pairs concept combines node similarity and structural characteristics used to compute the overall process similarity based on a dynamic wieghted average that depends on a weighting function which specifies the intensity of the weighting. The main differences with respect to our work is the focus on business processes and the use of comparison mainly based on node labels rather than on structural information.

The approach in [30] proposes a technique for assisting business process design by considering the context of the neighborhood activities, by recommending to the designer activities which are close to the process under design, based on an existing collection of business process models represented as graphs. A context is defined as a business process fragment around an activity, including the associated activity and connection flows associating the activity with its neighbors. The neighborood context of a selected activity finds similar neighborhood contexts of other activities considering how many connection flows separate the activity from a neighbor. The matching between neighborhood context graphs also incorporates the behavioral similarity of the associated activities. Although this work restricted to Business Process Models acknowledges the neighborhood context, it does not consider graphs whose nodes have multiple labels.

The following works use indexing for efficient search of business process models, pruning the quantity of models on which graph match will be performed. The work [68], and its extension [67], use Petri Net representation of models instead of graphs. In comparison to the graph-based part of our work, these approaches are limited to business process models, and although they introduce filtering as another step in the processing to achieve more efficient search, we consider multiple labels, and map the graph search problem into multidimensional spaces using multidimensional scaling.

An indexing approach for business process models based on metric trees (M-Trees), and a graph edit distance as similarity metric is given in [78]. 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.

A technique for efficient querying of bussiness process models is [125]. It applies model features used to efficiently estimate model similarity between them and classify them according to their relevance with respect to the query based on the ratio of common features. Features are selected as representative abstractions of Bussines Process Models and they include structural features, that depend on the structural role of the feature in the graph, as well as labels. The approach uses M-tree index on node labels for finding similar items with respect to an item to a given degree, an inverted index that maps node labels to nodes, and a parent-child index that stores the relation between smaller child features and larger parent features, and matches two child features if their parent features match. A greedy algorithm besed on graph-edit distance is used to evaluate the candidates filtered by the indexes.

The work in [103] presents an approach for clustering business processes based on their underlying topic, semantic and structural similarities. The approach considers high-level topic information collected from process description documents and keywords as well as detailed structural features such as process control flows in finding similarities among business process models. The proposed clustering method has two levels: the first level is based on the topic similarities and it uses a topic language modeling approach, while the second level is based on pairwise structure similarities between all the processes. For a given query process, the approach returns the top-k most similar processes based on the clustering.

An approach for efficient querying of bussiness process model repositories modeled as Petri nets is proposed in [68]. It uses B+-tree indexes based on transition paths, such that every tree contains paths with length n and all the models that contain it. The querying consists of two stages: the first stage utilizes the indexes to find candidate matches, while the second stage uses an adaptation of the Ullman's subgraph isomorphism algorithm to Petri nets.

The work in [67] introduces a structural technique for efficient retrieval of BPM models represented as Petri nets, with the help of an edge-based index which filters promising candidates. During query processing, a number of edges that needs to be contained according to a given similarity threshold is evaluated and used to obtain the candidate models from the index. A similarity based on Maximum Common Edge Subgraph is computed between a query condition model and all the candidates that passed the filter. Since the size of the candidate set is much smaller than the size of the entire repository, query efficiency is improved.

The following three approaches are examples of graph-based search on workflows, used to find similar workflows. Work [135] uses specfic algorithm, while [16] and [47], employ subgraph-isomorphism techniques.

An inexact process matching approach that enables two workflow processes to be matched with a similarity degree [0,1] is brought out in [135]. The similarity is obtained from the matching degrees of the coresponding 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 in an Activity Specialization Graph (ASG) defined by the activity-ontology repository. In each ASG, nodes are activities, while arcs represent activity specialization relationships.

The work in [16] proposes a model for representing semantic workflows, focusing on business and scientific workflows, as semantically labeled graphs with a related model for knowledge intensive similarity measures (similarity measure model). Workflow similarity computation is performed based on the A-star search, considering similarity computation and case selection. Semantic workflows are enriched by using metadata and constraints to individual workflow elements, formalized by ontologies. Each graph's node and edge is semantically labeled with metadata in a semantic description. The similarity measure of workflows depends on the node similarity expressed through their similarity in semantic descriptions and the edge similarity of the workflow graphs which considers not only semantic descriptions, but also the similarity of nodes they link. Aggregation function is used to combine the individual similarity to obtain the overall workflow similarity.

The potential of workflow discovery using subgraph isomorphism matching is investigated in [47]. The approach uses a Graph Matcher which has the ability to find workflows similar to the input and returns the result formatted into HTML page. The workflows, as well as the workflow which is the user input, are parsed into a form the Graph Matcher understands. The tool is tested within the workflow environment on a real corpus.

Structured Query Languages

The work in [75] defines extensions of OCL (Object Constraint Language) for allowing queries over complex model repositories. It proposes MoScript, a textual language for model querying and management. Users can write scripts with MoScript that contain queries and manipulation instructions (e.g., transformations on sets of models) upon models and store them back in the repository. The MoScript scripts can perform description and automation of complex modelling tasks, allowing several consecutive manipulations on a set of models. MoScript can use rich metadata to validate model manipulations. While the approach is metamodel-independent, the user is left in charge of writing complex OCL-like queries that only retrieve exact, non-ranked models.

The work in [26] studies the problem of answering queries over UML Class Diagrams (UCD) and restricted class of OCL constraints by relating it to the problem of query answering under guarded Datalog±, a powerful Datalog-based language for ontological modeling, in order to verify whether an instance of a system modeled by the UML class diagram satisfies a specific property. An expressive fragment of the UML class diagrams with a limited form of OCL constraints, named Lean UCD is identified to enable controllable query answering with respect to the time complexity.

An approach for querying UML models using the detailed semantics of the UML and OCL is presented in [4]. 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 definitons (project and product,) and concepts, to avoid adding additional UML model elements and modifying the UML model.

Another approach for specifying model queries on UML models based on lexical similarity and structural arrangements (indirect relationships)through graphical notation is described in [115]. 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 [12] 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.

As in the case of graph query languages, our approach departs quite radically from the mentioned ones, as none of these systems considers query-by-example scenarios, and they require knowledge of the query languages used to specify a query, which is not suitable for end users.

Other Content-Based Model Search Approaches

Other works use specific algorithms for model search.

For example, an approach for finding similarity between business process models is introduced in [10]. 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. This work also requires knowledge of a specific lanugage for performning queries.

The following methods use behavioral porperties of business process models for their search. Our work instead, compares retrieval techniques based on purely textual representations and on graph representations upon which graph similarity is computed.

Calculation of the degree of similarity of business process models, constructed with the Event-driven Process Chains (EPCs) language, considering their linguistic and behavioral aspects is illustrated in [121]. 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 similiarity score of surrounding events.

The work in [114] addresses the problem of querying business process models represented as Petri nets, allowing for queries to express behavioral requirements. The approach proposes using a temporal-order preserving complete finite prefix (TPCFP) that considers all temporal relations between tasks and reduces the state space for the processes. TPCFP is generated for every process model in a repository and it is used to determine whether the process models are compatible to a given temporal query. A linear temporal logic formulae are applied as a behavioral query language whose formal semantics are defined over TPCFP. Query evaluation over TPCFP is performed using depth-first search strategy. This work requires users to be knowledgeable about temporal logic in order to retrieve business process models.

The work [91] analyzes similarity between process model behaviors, defined in terms of causal footprint. This raises the level of abstraction of the models and thus allows comparison of models specified in different business process modeling notations. Similarity is calculated with a vector model that considers nodes, look back links and look ahead links of the causal footprints as features.

With respect to our work, the following techniques leverage on semantic descriptions of the models. This means that models need to be enriched with annotations from ontologies for improving the retrieval performance.

In [86], 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 [13]. 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.

The work [73] proposed the use of semantic business processes and offer an approximate query engine based on iSPARQL to perform the process retrieval task and to find inter-organizational matching between business partners. 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. As a result, the broader use of statistical reasoning might improve the overall performance of the semantic web.

The following works propose content-based search over UML models. While [18] requires knowledge of a specific language to perform search, [49] and [14] require the query to be specified as a partial or complete UML model.

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 metamodel 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 [18]. 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 structural similarity. The semantic similarity uses a domain vocabulary and a central terminology containing WordNet with some additions.

Rebuilder [49] is an environment for retrieving UML class models based on the WordNet ontology used to group UML model elements in categories. The approach applies indexing of models by assigning to each model a context synset, a set of cognitive synonyms each

expressing a distinct concept [3]. The query is content-based, consisting of partial UML packages classes or interfaces. The query also specifies the number of models to be retrieved.
The search process starts by finding a synset for the query model. Then, a spreading activation algorithm is initiated to visit nearby synset nodes,incorporating all the models retrieved during spreading. The algorithm finishes when the number of models specified in the query is found, or all nodes have been alredy visited. This way, all models found during the algorithm execution are ranked with respect to their similarity to the query, and they are included in the result. With respect to our approach, this work is limited to UML model queries only.

The work in [14] proposes structural and behavioural retrieval technique that takes into account the heterogeneity of the components repository (abstraction level, technology, domain). The approach combines formal and semi-formal specifications used to describe components and to make the retrieval process more efficient. The components are not indexed directly, but through indexing their UML representation: class diagram, use case diagram, sequence diagram and communication diagram. Class diagram focuses on the structural aspects of the component, while the use case diagram, sequence diagram and communication diagram describe the behavioural property of the component. This representation is independent with respect to the components' abstraction levels. Indexing is based on two types of models: UML and relational model. The query is expressed in UML format, which represents a semi-formal language, so the user does not need to be familiar with formal methods.

EMF Compare [120] 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. EMF Compare can be applied for any modeling language, provided it is supplied with its metamodel, but it can compare only two models at the same time, and it is not being adapted for querying entire repositories.

[3] http://www.http://wordnet.princeton.edu

6.2 Graph Search Techniques

Graphs allow universal representation of data from various application domains, such as social networks, computational biology, software models, computer vision, software bug localization, to name just a few. They are capable of storing heterogenous data and modeling complex structures and ineractions within them. Based on these different applications, graphs can vary in size (from several to several million nodes) and structure, which explains the fact why different applications have different requirements for managing graph data. The emergence of massive and complex structural data modeled as graphs, raises the need for existence of efficient tools for their search using graph indexes [33, 77, 100] to enable quickly pruning of graphs that violate the query requirement [3]. Besides indexing, another challenge in managing graph data is processing queries, typically expressed as graphs, especially because increased data complexity leads to increased complexity in query structure. Some works formalize the concept of graph querying by proposing graph query languages [57, 99], but there also exist approaches that simplify the querying process by using simple keywords [133, 70]. A classical formulation of the problem of searching graphs is through finding an exact or approximate correspondence between a query graph and a data graph, known as graph matching. The problem of graph matching is related to the problem of (sub)graph isomorphism which finds a mapping that preserves structure and labels between a query graph and a subgraph of the data graph, or more generally, it finds a maximal common subgraph with maximum number of common nodes belonging to the graphs. Since in real datasets there is a presence of noise and incompletness, a more practical solution would be to use approximate matching [119], and determine mapping similarity. Nevertheless, most graph matching variants are NP-complete, and many approaches use pruning techniques in order to decrease the number of necessary NP-complete operations. In the following, we represent current state-of-the-art methods for searching graph data.

The work in [23] extends the subraph isomorphism concept, to match graph queries approximately over databases of graphs. The queries may contain *don't care symbols* (which match any attribute value in a corresponding position) and variables. Variables may indicate which attribute values are desired to be returned as a query answer, or they can express constraints on the attribute values. Node matching condition is based on node type correspondence, satisfying all the constraints imposed by the query, while each query edge is included in the final result. The algorithm terminates when the first match is found. Although this work considers approximate matching of graphs, it does not provide efficient search solutions.

TALE (Tool for Approximate Subgraph Matching of Large Queries Efficiently) is a general tool for approximate subgraph matching of large graph queries studied in [119]. It queries graph databases and uses novel indexing method considering the neighbours 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. With respect to the graph-based part of our work, this approach performs subgraph matching for large graph queries.

Closure-tree [59] is an index structure for efficient graph quering using subgraph and similarity queries. Subgraph queries find all the graphs that contain subgraph through pseudo subgraph isomorphism, while similarity queries use graph edit similarity, measured through graph edit distance using heuristic methods for graph mapping: state search, bipartite method or neighbor biased matching. Graphs are organized hierarchically in the index, such that each node summarizes its descendants by a structure called Graph Closure in order to enable effective pruning. Children of the leaf nodes are database graphs. Graph Closure has characteristics of a graph, except that instead of singleton labels on vertices and edges, multiple labels are allowed for each node descendant. However, this approach is not suitable for graphs whose nodes originally contain multiple labels, since this might complicate the index structure.

MuGram [77] is a multi-labeled graph matching approach that handles graphs with multiple labels for both vertices and edges. It uses an indexing technique which besides the labels' information contains neighborhood information for each vertex which allows pruning incompatible candidates at early stages of matching. The matching process is based on neighborhood connectivity check that ensures that the graph invariant property for each query vertex is captured by the matching reference vertex. An index on the query graph is maintained to avoid repeatable processing of the query graph for each query vertex. The approach has been tested on different types of graphs representing protein networks, scientists' network and an email network.

DELTA [126] studies the problem of subgraph indexing and matching in multi-labeled graphs by storing the label set of each vertex into high-dimensional box stored in an R-tree, such that each dimension in the R-tree corresponds to a label category. R-tree calls for efficient vertex matching procedure, transforming the graph query into a spatial range query. For each vertex, the work uses two indexes: an index on the vertex labels, and a neighborhood index. The matching is performed such that for each query vertex, both indexes are used to find matching candidates. The approach considers two types of queries: existence query that checks whether the query is subgraph isomorphic to the database graph, and location queries, which find all query matches in the database graph. The matching is expanded through Breadth First Search for location queries that find all query graph matches in a database graph, and through Depth First Search for existence queries that evaluate subgraph isomorphism between a query and a database graph. The multidimensional scaling graph-based model search algorithm is inspired by the works on multi-labeled graphs, but with respect to our method, they are tested on graph databases and they do not consider similarity, but only exact matching.

The following approaches are based on indexing graph features which are not suitable for our graphs representing WebML models. Namely, due to the hierarchical nature of WebML models, some metamodel concepts will be completely disregarded in forming matches, thus loosing the model structure.

FG-Index [33] is a nested-inverted indexing technique based on a set of frequent subgraphs from a graph database. If a query is a frequent graph in the database, the results returned by the index represent the exact answer set. For an infrequent query, the FG-index returns an exact set of candidate answers, close to the exact answer set. The obtained answer set is a subject to subraph isomorphism, but since the query is infrequent, the number of tests is small. For compressing the set of frequent graphs, a notion of δ-Tolerance Close Frequent Subgraphs is introduced for tuning the index size in a parametrized way.

SAGA [118] is an approximate subgraph matching tool for biological graphs that computes graph similarity by allowing node gaps, node mismatches and graph structural differences. SAGA uses index based on small fragments of graphs in the database. The query graph is broken into small fragments that search the index to find most similar fragments in the database. The found matched fragments represent candidate matches that are assembled together to build a larger match. Each candidate match is examined to form a set of real matches calculating the real subgraph matching distance. In[81], a subgraph query processing is presented that generalizes exact edge matches to path matches constrained by a path length. The order of the matching vertices is optimized by choosing the next vertex to be matched to minimize the search space. The work proposes three different type of indexing: distance index, that considers the distance among all pair of vertices on the graph; Frequent Pattern Index based on the Frequent Generalized Subgraph (FGG), a frequent subgraph pattern whose frequency is greater than a threshold; Star index based on a star structure in the graph where one vertex is chosen as central and all its incident edges to the other vertices are considered.

The paper [132] studies the graph similarity join problem that returns pairs of graphs such that their distances are not larger than a threshold. The indexing is based on q-grams extracted as paths from graphs with length q, inspired by the n-gram string distance, used to generate a set of candidates with respect to a query. Mismatched q-grams are analysed and used to build filtering techniques to improve the graph similarity join. A verification algorithm is performed that employs multiple filters to quickly prune unpromising candidates and graph-edit distance computation based on the A-star algorithm.

Lindex [129] is a lattice-structure index for efficient and fast answering of subgraph queries, reducing the subgraph-isomorphism comparisons. Each node in a lattice represents a graph, where any pair of graphs has at least upper bound and a greatest lower bound. Nodes in the index represent key-value pairs, where the key represents a subgraph in the database, and the value is a list of database graphs containing the key. An edge between two index nodes indicates that the key in the parent node is a subgraph of the key in the child node. The query answering algorithm identifies a set of maximal subgraphs in the index and obtains a candidate set of answers by intersecting direct value sets of these subgraphs. The candidate set is pruned by identifying supergraphs of the query and eliminating graphs in the database that contain these supergraphs (from the candidate set).

The work in [111] proposes a connected substructure similarity search that retrieves graphs that approximately contain the query graph. The approach applies graph indexing technique, callled GrafD-Index, based on graphs' distances to the features, which prunes unpromising graphs and includes graphs that have a similarity greater than a threshold. The problem of connected substrucure similarity search is defined through finding the maximum connected common subgraph(MCCS) between the query and the database graphs, and the measure of similarity depends on the distance between the query and the MCCS.

An approach for approximate search of medical images by content that represents images as attributed relational graphs is given in [100]. It also considers the fact that images besides known objects also contain unknown(unlabeled) objects which are used to build an index. The index is obtaned by splitting the graphs into smaller subgraphs that contain a fixed number of unlabeled objects. All subgraphs (subimages) are mapped into points in multidimensional feature space, stored in an R-tree, thus transforming the image search into spatial search. The query is an image that is matched against the R-tree to retrieve all the images that contain subimages that contain the query within a fixed tolerance. This method is restricted to images.

The approach [124] introduces an indexing technique for graph databases in order to facilitate the subgraph isomorphism and processing similarity queries. The index contains two data structures: Directed Acyclic Graph (DAG) where each node represents a unique induced subgraph from the database graphs, and a hash table that enables a lookup function to quickly locate a node in the DAG isomorphic to a query graph. The approach considers different query types: Subgraph isomorphism query which finds a node in the DAG isomorphic to the query and report all its descendants; Similarity query which finds all graphs for which the subgraph mismatch score is less or equal to the query's range; Near-neighbor query that finds all graphs that share a subgraph common to the query whose size is at least the difference between the size and query range; Query for greater ranges that uses branch-and-bound technique to find a mapping which meets the range requirement; Query for far neighbors that finds all graphs which share no labels common to the query graph. However, this method works on small database graphs whose size is around 20 nodes.

The following works focus on efficient answering of graph queries.

The work in [136] addresses the problem of answering pattern match queries over a large data graph reducing the search space significantly. Main feature of a pattern match query is that it specifies the vertex labels and the connection constraints between the vertices. A set of candidate matches in the data graph is found using a graph embedding technique by transforming vertices into points in a multidimensional space, thus converting the problem into a distance-based multi-way join over the vector space. At the end, each candidate match is checked for approximate subgraph match. The algorithm that handles the distance-based join, called D-join, uses block-nested loop join and hash join to handle the high-dimensional space. This work considers reachability queries, that check whether a node is available from another node in a large graph, which is a different user need with respect to ours.

The work [1] introduces the notion of similarity skyline of a graph query defined by the subset of graphs of a graph database most similar to the query in a Pareto sense. The similarity between graphs is modeled by a vector of scalars in order to achieve a d-dimensional comparison between graphs in terms of d-distance measures and retrieve only results that have maximal similarity.In other words, the skyline contains all graphs in the graph database which are not dominated by any other graph with respect to a query. A graph dominates another graph if it is not less similar to a query in all the dimensions, and strictly more similar to the query than the other graph in at least one dimension. The approach suggests usage of several indexes, each of which should be dedicated to measure a local distance between two graphs related to one aspect in the graph structure. However, it does not give any concrete details concerning indexing, concentrating only on the querying part.

Corese search engine [38] 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 the 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 anotations 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. This work is limited to RDF graphs only.

The following top-k approaches propose efficient search of top-k graphs considering a query graph.

The work [134] explores the problem of finding the most similar top-k graphs with respect to a query graph using a new maximum common subgraph measure, capturing the common and different structure of the graphs. The approach uses two distance lower bounds, based on edge frequency, with different computational costs to reduce the number of MCS computations. An index based on the triangle property of graph similarities is used in order to obtain tighter lower bound. Three types of indexes are considered with different trade-offs between construction cost and pruning, ordered by their increasing pruning power: (Disjoint Partition Index) DPIndex partitions the graphs into non-overlapping clusters choosing random graphs as cluster centers, and assigning the non-center graphs to its nearest center; Overlapping Partition Index that allows each graph to belong to multiple clusters, and General Similarity Index which treats each graph as a center. In this work, the search is based on edge frequency only, without considering the nodes. Neighborhood Based Similarity Search (Ness) proposed in [72], is a method that uses an information propagation model to convert a large network into a set of multidimensional vectors that finds the top-k set of approximate matches with respect to a query based on the vertex labels and vertex neighborhood information. The similarity measure discounts how the graph vertices are exactly connected, but focuses on the proximity of the vertex labels in the graph, an information encoded in a neighborhood vector. The search algorithm is a scalable and iterative approach that finds the top-k query embeddings in a graph based on subgraph similarity search, such that it first matches individual nodes of the query graph with the individual nodes of the big graph and while propagating only the labels of the matched nodes, it recomputes their neighborhood vectors. The approach uses two types of indexing: hash table corresponding to each label, and an index built on the neighborhood vector for each vertex used in the case when the labels are not very selective. The difference of this approach with respect to our multidimensional scaling graph work, is that it is not designed for graphs with labeled edges and the search is performed to find the top-k embeddings in one large network graph. Top-k embeddings might also be a new direction in our future work.

The following approaches propose specific algoirthms for graph search.

$Turbo_{ISO}$ [58] is a subgraph search method which uses candidate region exploration and combine and permute strategy (Comb/Perm). The candidate region exploration finds candidate subgraphs that contain embeddings of the query graph and computes a robust matching order for each candidate region that has been explored. The Comb/Perm strategy is based on the concept of neighborhood equivalence class (NEC), where each query vertex belonging to the same NEC has identically matching data vertices by label and a set of adjacent query vertices. Thus the query graph is rewritten into a NEC tree by performing breadth-first search (BFS) from a given start query vertex. The Comb/Perm strategy avoids finding all possible enumerations during subgraph isomorphism, considering only the combination for each NEC, thus making the approach efficient. If a chosen combination does not contribute to the solution, all its possible permutations are also pruned. This method considers only exact matches in labels, while in our work based on the user need expressed as a content-based query, we consider similar matches.

The approach [89] uses similarity flooding as a graph matching algorithm to perform schema matching. The algorithm compares two graphs as input, and produces a mapping between the corresponding graph nodes. The main criteria used in 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. The algorithm output is checked and corrected by a human, and its accuracy is measured by the number of adjustments. This algorithm operates and is restricted to directed labeled graphs, and does not produce optimal results on graphs with unlabeled edges. Furthermore, it requires human participation in the verification of the obtained results.

The keyword graph approaches, use keywords as queries, and they are not appropriate for expressing the query-by-example need necessary in our work.

The work in [70] proposes a bidirectional search algorithm for schema-agnostic text search on weighted directed graphs, which efficiently extracts from a data graph, a small number of the best answer trees. An answer tree for a keyword query that contains a set of terms, is a minimal rooted directed tree, embedded in the data graph, containing at least one node from each set of nodes that matches a query term. The bidirectional algorithm can simultaneously exploit forward paths from nodes that represent potential roots of answer trees towards other keyword nodes, and backward paths from the keyword nodes to the roots of the answer trees. In order to achieve that, the approach uses prioritization scheme based on spreading activation, i.e., favoring expansion of those paths that have less branching.

The work [133] explores the diversity of user information need when searching graphs differentiating between exploratory search, where the user is unfamiliar with the graph structures, and known-item search, where the user has as a target a set of trees, or particular pattern. The problem of known-item search is addressed by expressing the query as a set of keywords. The answers to the query represent minimum connected trees, that represent subtrees of an unlabeled directed weighted graph containing at least one matched vertex for every query keyword. Matched Vertex Pruning (MVP) index is used to capture the query-independent local neighborhood information in the graph by pruning matched vertices that do not participate in the answer trees with heights less than a threshold. The approach is independent on the graph search algorithm and minimizes the index access times.

6.2.1 Graph 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 [99]. It is a declarative language, which 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 scheme, which is the scheme of the instances that can be given as input to the rule, and the target scheme, which is the scheme of the output instances of the rule.

XML-GL is a graphical query language for querying XML data [35]. 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.

GraphDB [56] is an object-oriented based query language orginally designed for spatially embedded networks. It provides explicit path objects and it allows to define graph operations whose argument graphs (subgraphs of the database graph) can be specified by regular expressions over link class names. Four fundamental tools are defined necessary to formulate a query: *derive* statement which extends the standard "select...from...where" for graphs, *rewrite* for manipulations of extended sequences, *union*for transfroming heterogenous collections of objects into homogenous collection,and a collection of graph operations such as *shortest path search*.

Another object-oriented query language is Graph-Oriented Object Database Model (GOOD) [57], which uses graph transformations for querying, such as node insertion/deletion and edge insertion/deletion as a function of pattern matches. It also supports object identity and it allows abstraction over objects that share the same set of properties. The language provides method mechanism for executing operation sequences.

These visual query languages establish the rules to build a query, which requires from the users to be knowledgeable about a specific language. Moreover, in the query-by-example scenario of our work, using an existing graph query language would not be an appropriate solution to define a query in form of a model or partial model.

6.3 Overview of Related Work

Our approach to keyword-based search is inspired by information retrieval techniques and it is related to works like [84]. With respect to this work, we focus on the comparison with content-based search and thus adopt a rather straightforward approach to indexing and search, which uses only the knowledge present in the text content and in the metamodel. The extension to a semantically richer treatment of the domain knowledge, e.g., for term expansion and domain-driven clustering of projects, can be easily envisioned for our approach. As for the A-star algorithm(s) for graph-based search, our approach mostly draws inspiration from graph-based works in the context of business process models, most notably, [105, 39]. With respect to BPM-oriented content-based search, DSL-oriented search shares the mix of label and structural knowledge exploited in indexing and searching, but must cope with a richer language syntax and semantics, which we have considered in the design of parameter configurations. The multidimensional scaling algorithm for graph search is motivated by the works that propose efficient search of multi-labeled graphs, such as [77, 126]. Although these approaches are designed for graphs where at least the vertices contain multiple labels, and they include a mechanism for neighborhood check, they only work for exact labels, and do not consider the concept of graph similarity. Moreover, these approaches are not created to search for models.

Chapter 7

Conclusions and Future Work

In this thesis we have addressed the problem of designing search systems for repositories of models. We have contrasted two major approaches for the implementation of search: keyword-based and content-based. Keyword-based search employed classical information retrieval techniques expanded with injection of metamodel knowledge. For content-based search we focused on graph search approaches, and we propose and implement two of them: A-star graph-based search and multidimensional scaling graph-based search. Extensive experimentation has been conducted with a sample of 10 queries against a real-world repository of WebML models, for which a gold standard set has been constructed that embodies what experts consider good responses to both keyword-based and content-based queries. Keyword-based search has also been configured and evaluated on a repository of UML models for a sample set of 20 queries, for which the ground truth has been built considering the same criteria used in the WebML groundtruth construction.

Experiments have shown that even traditional text indexing techniques can deliver good performance for keyword-based queries, especially evident in the UML repository, while adding metamodel knowledge to the index can improve accuracy. For A-star graph-based search, the conclusion is that matching the textual content of projects is still important, but the system benefits from an appropriate injection of metamodel knowledge regarding both the types of the elements and the structure and topology of models. Furthermore, A-star graph-based search results exhibited greater variability and dependency on the queries than keyword-based results. Considering the multidimensional scaling graph-based seach, incorporating additional metamodel information improves performance and generates alternative matching patterns. The multidimensional scaling graph search has increased efficiency with respect to the A-star graph-based search, due to the use of similarity-based indexing, at the price of performance loss.

These results have been gathered in the context of a mid-scale experiment and cannot be generalized in an absolute way. They provide insight about (i) what expert modelers consider suitable queries and responses and (ii) the way in which two different classes of information retrieval systems can be configured to respond to the expectations of these searchers. Nonetheless, we believe that the results presented in this thesis provide a number of interesting observations about the usage of keyword-based and content-based techniques for model search and therefore respond to the research questions we initially defined in Section 1.2.

Future work will develop along several complementary lines:

- On the search systems side, both keyword-based and content-based approaches will be further explored. The keyword-based approach will be expanded with a better exploitation of semantics and domain knowledge, both at query time (e.g., by means of keyword expansion), and at indexing time (e.g., by means of text feature extraction and project topical clustering). The content-based approach also lends itself to several investigation directions: exploring new graph similarity functions and new graph matching algorithms, providing more eficient and scalable search, that will be compared to the presented approaches. More specifically, we plan to explore other scoring functions to improve multidimensional scaling graph-based search performance. Automatic fine-tuning of the parameters of the Chalmer's algorithm, used in the indexing phase of multidimensional scaling graph-based search, will be performed by exploring optimization methods.

- On the usage of metamodel information, several additional options for embodying such knowledge in the search system can be evaluated. Besides using metamodel knowledge to segment projects and to influence the matching and ranking of the IR system, it is also possible to use it for mining relevant information from the project repository, such as term distribution, and for automating concept weighting based on the analysis of concept centrality in the collection of model element graphs.

- On the Web engineering side, it would be interesting to proceed with the analysis of other DSLs, and to compare search techniques for other general purpose and domain specific languages. We also plan to investigate how the introduction of explicit reuse-oriented constructs, e.g., WebML reusable modules, alters the structure of projects and the modeling style of developers, and thus impacts content-based search.

- On the evaluation side, a general-purpose task crowdsourcing system could be employed to design user studies and deploy them on top of open social networks and/or closed groups [20]. Several types of search result evaluation questions could be formulated as crowd tasks, to gather a large scale collection of queries and expert-validated result relevance scores, exploiting both open groups (e.g., LinkedIn MDE groups) and closed communities (e.g., the WebRatio developers network).

Bibliography

[1] K. Abbaci, A. Hadjali, L. Liétard, and D. Rocacher. A similarity skyline approach for handling graph queries-a preliminary report. In *IEEE 27th International Conference on Data Engineering Workshops (ICDEW), 2011*, pages 112–117. IEEE, 2011.

[2] R. Acerbis, A. Bongio, M. Brambilla, and S. Butti. Webratio 5: An eclipse-based case tool for engineering web applications. In L. Baresi, P. Fraternali, and G.-J. Houben, editors, *ICWE*, volume 4607 of *Lecture Notes in Computer Science*, pages 501–505. Springer, 2007.

[3] C. C. Aggarwal and H. Wang. *Managing and mining graph data*, volume 40. Springer, 2010.

[4] D. Akehurst and B. Bordbar. On querying uml data models with ocl. In M. Gogolla and C. Kobryn, editors, *UML 2001 The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, pages 91–103. Springer Berlin Heidelberg, 2001.

[5] B. Anda, K. Hansen, I. Gullesen, and H. Thorsen. Experiences from introducing uml-based development in a large safety-critical project. *Empirical Software Engineering*, 11(4):555–581, 2006.

[6] 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.

[7] Artech Consultores S.R.L. Genexus Marketplace. http://marketplace.genexus.com, Last accessed October 2013.

[8] C. Atkinson and T. Kuhne. Model-driven development: a metamodeling foundation. *Software, IEEE*, 20(5):36–41, 2003.

[9] AtlanMod Group. AtlanMod Zoos. `http://www.emn.fr/z-info/atlanmod/index.php/Zoos`, Last accessed October 2013.

[10] 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.

[11] 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.

[12] 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.

[13] K. Belhajjame and M. Brambilla. Ontological description and similarity-based discovery of business process models. *International Journal of Information System Modeling and Design (IJISMD)*, 2(2):47–66, 2011.

[14] H. Ben Khalifa, O. Khayati, and H. Ghezala. A behavioral and structural components retrieval technique for software reuse. In *Advanced Software Engineering and Its Applications, 2008. ASEA 2008*, pages 134–137. IEEE, 2008.

[15] D. Berardi, D. Calvanese, and G. De Giacomo. Reasoning on uml class diagrams. *Artificial Intelligence*, 168(1):70–118, 2005.

[16] R. Bergmann and Y. Gil. Retrieval of semantic workflows with knowledge intensive similarity measures. In *Case-Based Reasoning Research and Development*, pages 17–31. Springer, 2011.

[17] R. K. Bhatia, M. Dave, and R. C. Joshi. Ant colony based rule generation for reusable software component retrieval. *ACM SIGSOFT Software Engineering Notes*, 35(2):1–5, 2010.

[18] 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.

[19] M. Bilenko, R. Mooney, W. Cohen, P. Ravikumar, and S. Fienberg. Adaptive name matching in information integration. *Intelligent Systems, IEEE*, 18(5):16–23, 2003.

[20] A. Bozzon, M. Brambilla, and S. Ceri. Answering search queries with crowdsearcher. In *Proceedings of the 21st international conference on World Wide Web*, WWW '12, pages 1009–1018, New York, NY, USA, 2012. ACM.

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

[22] M. Brambilla and P. Fraternali. Large-scale model-driven engineering of web user interaction: The webml and webratio experience. *Science of Computer Programming*, 2013.

[23] 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.

[24] A. Buja, D. F. Swayne, M. Littman, N. Dean, and H. Hofmann. Xgvis: Interactive data visualization with multidimensional scaling. *Journal of Computational and Graphical Statistics*, pages 1061–8600, 2001.

[25] H. Bunke. Graph matching: Theoretical foundations, algorithms, and applications. In *International Conference on Vision Interface*, pages 82–88, May 2000.

[26] A. Calì, G. Gottlob, G. Orsi, and A. Pieris. Querying uml class diagrams. In *Foundations of Software Science and Computational Structures*, pages 1–25. Springer, 2012.

[27] 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.

[28] 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.

[29] M. Chalmers. A linear iteration time layout algorithm for visualising high-dimensional data. In *Visualization'96. Proceedings.*, pages 127–131. IEEE, 1996.

[30] N. N. Chan, W. Gaaloul, and S. Tata. Assisting business process design by activity neighborhood context matching. In *Service-Oriented Computing*, pages 541–549. Springer, 2012.

[31] S. Chatterjee, S. Juvekar, and K. Sen. Sniff: A search engine for java using free-form queries. In *Fundamental Approaches to Software Engineering*, pages 385–400. Springer, 2009.

[32] E. Chávez, J. L. Marroquín, and G. Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications*, 14(2):113–135, 2001.

[33] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 857–872. ACM, 2007.

[34] D. Chiu, T. Hall, F. Kabir, and G. Agrawal. An approach towards automatic workflow composition through information retrieval. In *Proceedings of the 15th Symposium on International Database Engineering & Applications*, pages 170–178. ACM, 2011.

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

[36] J. Conallen. *Building Web applications with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[37] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[38] 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.

[39] R. Dijkman, M. Dumas, B. Van Dongen, R. Käärik, and J. Mendling. Similarity of business process models: Metrics and evaluation. *Information Systems*, 36(2):498–516, 2011.

[40] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[41] 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.

[42] K. M. Fairchild. Semnet: Three-dimensional graphic representation of large knowledge bases. *Cognitive science and its applications for human-computer interaction*, pages 201–233, 1988.

[43] 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.

[44] R. France, J. Bieman, and B. H. C. Cheng. Repository for model driven development (remodd). In *Proceedings of the 2006 international conference on Models in software engineering*, MoDELS'06, pages 311–317, Berlin, Heidelberg, 2006. Springer-Verlag.

[45] R. France, J. Bieman, S. Mandalaparty, B. Cheng, and A. Jensen. Repository for model driven development (remodd). In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1471 –1472. IEEE Press, june 2012.

[46] 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.

[47] A. Goderis, P. Li, and C. Goble. Workflow discovery: the problem, a case study from e-science and a graph-based solution. In *International Conference on Web Services, 2006. ICWS'06.*, pages 312–319. IEEE, 2006.

[48] J. Goguen, D. Nguyen, J. Meseguer, D. Zhang, V. Berzins, et al. Software component search. *Journal of Systems Integration*, 6(1-2):93–134, 1996.

[49] P. Gomes, F. C. Pereira, P. Paiva, N. Seco, P. Carreiro, J. L. Ferreira, and C. Bento. Using wordnet for case-based retrieval of uml models. *AI Communications*, 17(1):13–23, 2004.

[50] J. Gómez, A. Bia, and A. Parraga. Tool support for model-driven development of web applications. In *Web Information Systems Engineering–WISE 2005*, pages 721–730. Springer, 2005.

[51] J. Gómez and C. Cachero. *Information Modeling for Internet Applications*, chapter OO-H Method: Extending UML to Model Web Interfaces, pages 144–173. Idea Group Publishing, Hershey, PA, USA, 2003.

[52] 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.

[53] L. Gregory and J. Kittler. Using graph search techniques for contextual colour retrieval. *Structural, Syntactic, and Statistical Pattern Recognition*, pages 193–213, 2002.

[54] D. Grigori, J. C. Corrales, M. Bouzeghoub, and A. Gater. Ranking bpel processes for service discovery. *IEEE Transactions on Services Computing*, 3(3):178–192, 2010.

[55] J. Grundy. Storage and retrieval of software components using aspects. In *23rd Australasian Computer Science Conference, 2000. ACSC 2000.*, pages 95–103. IEEE, 2000.

[56] R. Gtiting. Graphdb: Modeling and querying graphs in databases. In *20th Very Large Data Bases Conference (VLDB)*, 1994.

[57] M. Gyssens, J. Paredaens, J. Van den Bussche, and D. Van Gucht. A graph-oriented object database model. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):572–586, 1994.

[58] W.-S. Han, J. Lee, and J.-H. Lee. Turbo iso: towards ultra-fast and robust subgraph isomorphism search in large graph

databases. In *Proceedings of the 2013 international conference on Management of data*, pages 337–348. ACM, 2013.

[59] 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.

[60] R. Hoffmann, J. Fogarty, and D. S. Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, pages 13–22. ACM, 2007.

[61] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 117–125, New York, NY, USA, 2005. ACM.

[62] O. Hummel, W. Janjic, and C. Atkinson. Code conjurer: Pulling reusable software out of thin air. *Software, IEEE*, 25(5):45–52, 2008.

[63] J. Hutchinson, M. Rouncefield, and J. Whittle. Model-driven engineering practices in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 633–642, New York, NY, USA, 2011. ACM.

[64] J. Hylton. *Identifying and merging related bibliographic records*. PhD thesis, Massachusetts Institute of Technology, 1996.

[65] 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.

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

[67] T. Jin, J. Wang, and L. Wen. Efficient retrieval of similar business process models based on structure. In *On the Move to Meaningful Internet Systems: OTM 2011*, pages 56–63. Springer, 2011.

[68] T. Jin, J. Wang, N. Wu, M. La Rosa, and A. H. Ter Hofstede. Efficient and accurate retrieval of business process models through indexing. In *On the Move to Meaningful Internet Systems: OTM 2010*, pages 402–409. Springer, 2010.

[69] H. Joho. Diane kelly: Methods for evaluating interactive information retrieval systems with users - foundation and trends in information retrieval, vol 3, nos 1-2, pp 1-224, 2009, isbn: 978-1-60198-224-7. *Inf. Retr.*, 14(2):204–207, 2011.

[70] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *Proceedings of the 31st international conference on Very large data bases*, pages 505–516. VLDB Endowment, 2005.

[71] I. Keivanloo, L. Roostapour, P. Schugerl, and J. Rilling. Semantic web-based source code search. In *Proc. 6th Intl. Workshop on Semantic Web Enabled Software Engineering*, 2010.

[72] A. Khan, N. Li, X. Yan, Z. Guan, S. Chakraborty, and S. Tao. Neighborhood based fast graph search in large networks. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 901–912. ACM, 2011.

[73] C. Kiefer, A. Bernstein, H. Lee, M. Klein, and M. Stocker. Semantic process retrieval with isparql. In E. Franconi, M. Kifer, and W. May, editors, *The Semantic Web: Research and Applications*, volume 4519 of *Lecture Notes in Computer Science*, pages 609–623. Springer Berlin Heidelberg, 2007.

[74] A. G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[75] W. Kling, F. Jouault, D. Wagelaar, M. Brambilla, and J. Cabot. Moscript: A dsl for querying and manipulating model repositories. In *Software Language Engineering*, pages 180–200. Springer, 2012.

[76] A. Kraus, A. Knapp, and N. Koch. Model-driven generation of web applications in uwe. In N. Koch, A. Vallecillo, and G.-J. Houben, editors, *MDWE*, volume 261 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[77] V. Krishna, N. Ranga Suri, and G. Athithan. Mugram: An approach for multi-labelled graph matching. In *International Conference on Recent Advances in Computing and Software Systems (RACSS), 2012*, pages 19–26. IEEE, 2012.

[78] M. Kunze and M. Weske. Metric trees for efficient similarity search in large process model repositories. In M. Muehlen and J. Su, editors, *Business Process Management Workshops*, volume 66 of *Lecture Notes in Business Information Processing*, pages 535–546. Springer Berlin Heidelberg, 2011.

[79] O. A. Lazzarini Lemos, S. K. Bajracharya, and J. Ossher. Codegenie:: a tool for test-driven source code search. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 917–918. ACM, 2007.

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

[81] W. Lin, X. Xiao, J. Cheng, and S. S. Bhowmick. Efficient algorithms for generalized subgraph query processing. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 325–334. ACM, 2012.

[82] H. Liu, G. Liu, Y. Wang, and D. Liu. A novel behavioral similarity measure for artifact-oriented business processes. In *Technology for Education and Learning*, pages 81–88. Springer, 2012.

[83] R. Lu and S. Sadiq. Managing process variants as an information resource. In S. Dustdar, J. Fiadeiro, and A. Sheth, editors, *Business Process Management*, volume 4102 of *Lecture Notes in Computer Science*, pages 426–431. Springer Berlin Heidelberg, 2006.

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

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

[86] 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.

[87] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering*, 38(5):1069–1087, 2012.

[88] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 111–120, New York, NY, USA, 2011. ACM.

[89] 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.

[90] Mendix. The Mendix App Store. https://appstore.mendix.com, Last accessed October 2013.

[91] J. Mendling, B. F. van Dongen, and W. M. van der Aalst. On the degree of behavioral similarity between business process models. In *EPK*, volume 303, pages 39–58, 2007.

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

[93] MIT. MIT process handbook. http://ccs.mit.edu/ph/, Last accessed October 2013.

[94] P. Mohagheghi and V. Dehlen. Where is the proof? - a review of experiences from applying mde in industry. In *Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications*, ECMDA-FA '08, pages 432–443, Berlin, Heidelberg, 2008. Springer-Verlag.

[95] A. Morrison, G. Ross, and M. Chalmers. Fast multidimensional scaling through sampling, springs and interpolation. *Information Visualization*, 2(1):68–77, 2003.

[96] M. Niemann, M. Siebenhaar, S. Schulte, and R. Steinmetz. Comparison and retrieval of process models using related cluster pairs. *Computers in Industry*, 63(2):168–180, 2012.

[97] OMG. UML Resource Page. http://www.uml.org, Last accessed September 2013.

[98] Outsystems Inc. The Agilenetwork Component Store. https://www.outsystems.com/NetworkSolutions/Home.aspx, Last accessed October 2013.

[99] 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.

[100] E. G. M. Petrakis and A. Faloutsos. Similarity searching in medical image databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):435–447, 1997.

[101] D. Pilone and N. Pitman. *UML 2.0 in a Nutshell.* O'Reilly Media, Inc., 2009.

[102] C. Platzer and S. Dustdar. A vector space search engine for web services. In *Third IEEE European Conference on Web Services, 2005. ECOWS 2005.*, pages 9–pp. IEEE, 2005.

[103] M. Qiao, R. Akkiraju, and A. J. Rembert. Towards efficient business process clustering and retrieval: combining language modeling and structure matching. In *Business Process Management*, pages 199–214. Springer, 2011.

[104] S. P. Reiss. Semantics-based code search. In *IEEE 31st International Conference on Software Engineering, 2009. ICSE 2009.*, pages 243–253. IEEE, 2009.

[105] 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.

[106] ReMoDD Team. ReMoDD The Repository for Model-Driven Development. `http://www.cs.colostate.edu/remodd/v1/`, Last accessed October 2013.

[107] G. Rossi and D. Schwabe. Modeling and implementing web applications with OOHDM. In G. Rossi, O. Pastor, D. Schwabe, and L. Olsina, editors, *Web Engineering: Modelling and Implementing Web Applications*, Human-Computer Interaction Series, chapter 6, pages 109–155. Springer, London, 2008.

[108] 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.

[109] A. Sanfeliu and F. King-Sun. A distance measure between attributed relational graphs for pattern recognition. *IEEE transactions on systems, man, and cybernetics*, 13(3):353–362, 1983.

[110] R. C. Seacord, S. A. Hissam, and K. C. Wallnau. Agora: A search engine for software components. *Internet Computing, IEEE*, 2(6):62, 1998.

[111] H. Shang, X. Lin, Y. Zhang, J. X. Yu, and W. Wang. Connected substructure similarity search. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 903–914. ACM, 2010.

[112] 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.

[113] L. Shapiro and R. Haralick. Structural descriptions and inexact matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (5):504–519, 1981.

[114] L. Song, J. Wang, L. Wen, W. Wang, S. Tan, and H. Kong. Querying process models based on the temporal relations between tasks. In *Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2011 15th IEEE International*, pages 213–222. IEEE, 2011.

[115] 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.

[116] A. P. Street and D. J. Street. *Combinatorics of Experimental Design.* Oxford University Press, 1987.

[117] T. Syeda-Mahmood, G. Shah, R. Akkiraju, A.-A. Ivan, and R. Goodwin. Searching service repositories by combining semantic and ontological matching. In *IEEE International Conference on Web Services, 2005. ICWS 2005. Proceedings. 2005*, pages 13–20. IEEE, 2005.

[118] Y. Tian, R. C. Mceachin, C. Santos, J. M. Patel, et al. Saga: a subgraph matching tool for biological graphs. *Bioinformatics*, 23(2):232–239, 2007.

[119] Y. Tian and J. M. Patel. Tale: A tool for approximate large graph matching. In *IEEE 24th International Conference on Data Engineering, 2008. ICDE 2008.*, pages 963–972. IEEE, 2008.

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

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

[122] P. Weber, P. Taylor, B. Majeed, and B. Bordbar. Comparing complex business process models. In *IEEE International Conference on Industrial Engineering and Engineering Management–IEEM 2012*, 2012.

[123] WebRatio s.r.l. The WebRatio Store. http://www.webratio.com/portal/content/-/store, Last accessed August 2013.

[124] D. W. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *IEEE 23rd International Conference on Data Engineering, 2007. ICDE 2007.*, pages 976–985. IEEE, 2007.

[125] Z. Yan, R. Dijkman, and P. Grefen. Fast business process similarity search. *Distributed and Parallel Databases*, 30(2):105–144, 2012.

[126] J. Yang, S. Zhang, and W. Jin. Delta: indexing and querying multi-labeled graphs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1765–1774. ACM, 2011.

[127] 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.

[128] YesSoftware, Inc. CodeCharge Marketplace. http://www.codecharge.com/marketplace, Last accessed October 2013.

[129] D. Yuan and P. Mitra. Lindex: a lattice-based index for graph databases. *The VLDB JournalThe International Journal on Very Large Data Bases*, 22(2):229–252, 2013.

[130] L. Zhang, L. Chen, L. Pan, and Y. Zhang. A novel approach of components retrieval in large-scale component repositories. In *IEEE 3rd International Conference on Software Engineering and Service Science (ICSESS), 2012*, pages 220–223. IEEE, 2012.

[131] X. Zhang, H. Chen, and T. Zhang. An uml model query method based on structure pattern matching. In *Trustworthy Computing and Services*, pages 506–513. Springer, 2013.

[132] X. Zhao, C. Xiao, X. Lin, and W. Wang. Efficient graph similarity joins with edit distance constraints. In *IEEE 28th International Conference on Data Engineering (ICDE), 2012*, pages 834–845. IEEE, 2012.

[133] M. Zhong, M. Liu, Z. Bao, X. Li, and T. Qian. Mvp index: Towards efficient known-item search on large graphs. In *Database Systems for Advanced Applications*, pages 193–200. Springer, 2013.

[134] Y. Zhu, L. Qin, J. X. Yu, and H. Cheng. Finding top-k similar graphs in graph databases. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 456–467. ACM, 2012.

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

[136] L. Zou, L. Chen, M. T. Özsu, and D. Zhao. Answering pattern match queries in large graph databases via graph embedding. *The VLDB JournalThe International Journal on Very Large Data Bases*, 21(1):97–120, 2012.