Universidade do Minho
Escola de Engenharia

Nuno Alexandre Ramos de Carvalho

An Ontology Toolkit for Problem Domain Concept Location in Program Comprehension

The MAP Doctoral Program in Computer Science
of the Universities of Minho, Aveiro and Porto

universidade de aveiro

Universidade do Minho

U.PORTO

November 2014

Universidade do Minho
Escola de Engenharia

Nuno Alexandre Ramos de Carvalho

An Ontology Toolkit for Problem Domain Concept Location in Program Comprehension

The MAP Doctoral Program in Computer Science of the Universities of Minho, Aveiro and Porto

universidade de aveiro

Universidade do Minho

U.PORTO

Supervisors:
Professor Doutor José João Almeida
Professor Doutor Maria João Varanda

November 2014

Acknowledgments

I would like to thank all the persons that helped and supported me during this PhD, and dedicate the final work and results to all of them. I would like to express my gratitude in particular to:

- my advisors, Prof. José João Almeida and Prof. Maria João Varanda;
- a special thanks to Prof. Pedro Rangel Henriques;
- a word of thanks to Alberto Simões, Prof. Luís Soares Barbosa, Prof. José Nuno Oliveira, Luís Fernandes, Nuno Oliveira, the CROSS project team, and the Per-Fide project team.

This work was supported by the CROSS project[1], the Per-Fide project[2], and the University of Minho School of Engineering[3]. Finally, I would like to acknowledge the committee of the MAP-i Doctoral Program.

[1]FCT contract under reference PTDC/EIA-CCO/108995/2008.

[2]FCT contract under reference PTDC/CLE-LLI/108948/2008.

[3]This work was partially funded by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project PEst-OE/EEI/UI0752/2014.

An Ontology Toolkit for Problem Domain Concept Location in Program Comprehension

Software maintainers are often challenged with source code changes in unfamiliar programs to improve software systems, e.g., eliminating defects, introducing new features, adapting to reality shifts. To undertake these tasks a sufficient understanding of the system (or at least a part of it) is required. One of the most time consuming activity during the understanding process is locating which parts of the code are responsible for which key functionality or feature - concept (or feature) location. Details inherent to the different languages involved (natural languages used to describe concepts in the real world, *versus* the programming languages used to implement programs), and their different levels of abstraction, entail the major challenges during these activities.

This dissertation introduces the use of mappings for creating semantic bridges between the software system and its application domain, to enhance concept location, and other software understanding activities. The generic proposed approach for building mappings is divided in three main steps: (i) model, (ii) calculate, and (iii) devise views. The goal during the first step is to model relevant domains using ontologies to convey the information of interest, for example, model the software system (the program), or the application domain (the problem). Once the ontologies (models) are available, the second step implies performing arbitrary calculations to create, organize, or infer new information about each domain. During the final step, specific views are crafted based on information available in the different models, that emphasize elements and traits of interest.

During this work, a set of frameworks and libraries were developed, including generic methods and tool compositions, that allowed the implementation of the described approach in a elegant (simple but effective) way. This toolkit was used to develop an environment that features a set of applications that enhance program comprehension activities. A set of practical experiments were performed to measure tools (individual and composed) effectiveness, and a final overarching experience draws conclusions about the advantages of the featured techniques from a maintainers point of view, while performing software debugging tasks, and the benefits in general of exploring mappings between the program and the problem domain.

Um Kit Ontológico para Localização de Conceitos do Domínio do Problema na Compreensão de Programas

Programadores são muitas vezes confrontados com alterações do código de programas desconhecidos para melhorar os sistemas de software, eliminar defeitos, introdução de novas funcionalidades, ou adaptação a novas realidades. Para realizar estes tarefas é necessário uma compreensão suficiente do sistema (ou parte dele). Uma das actividades mais demoradas durante o processo de compreensão é localizar as partes do código responsáveis por recursos chave ou funcionalidades - localização de conceitos. Detalhes inerentes às diferentes linguagens envolvidas (linguagens naturais utilizadas para descrever conceitos no mundo real, *versus* linguagens de programação utilizadas nos programas), e os seus diferentes níveis de abstração, são responsáveis pelos principais desafios durante estas atividades.

Esta dissertação introduz o uso de mapeamentos para a criação de pontes semânticas entre o sistema de software e o seu domínio de aplicação, para melhor localizar conceitos, e outras actividades de compreensão de programas. A abordagem proposta para a criação de mapeamentos está dividida em três etapas principais: (i) modelar, (ii) calcular, e (iii) criar pontos de vista. O objectivo da primeira etapa é modelar domínios relevantes usando ontologias para representar a informação de interesse, por exemplo o modelo do sistema de software (programa), ou o domínio da aplicação (o problema). Assim que as ontologias (modelos) estejam disponíveis, o segundo passo implica a realização de cálculos para criar, organizar ou inferir novos dados sobre cada domínio. Durante a etapa final, são criadas vistas específicas com base em informação disponível nos diferentes modelos, que enfatizam elementos e características de interesse.

Durante este trabalho, desenvolveram-se um conjunto de *frameworks* e bibliotecas, incluindo métodos genéricos e composições de ferramentas, que permitiram a implementação da abordagem descrita de uma forma elegante (simples mas eficaz). Este kit de ferramentas foi utilizado para desenvolver um ambiente que disponibiliza um conjunto de aplicações que ajudam em atividades de compreensão de programas. Foram realizadas um conjunto de experiências práticas para medir a eficácia de algumas ferramentas (individualmente e compostas), e uma experiência final mais abrangente permite tirar conclusões sobre as vantagens da utilização das técnicas discutidas do ponto de vista do programador, durante a execução de tarefas de depuração de software, e os benefícios em geral de explorar mapeamentos entre o domínio do programa e o domínio do problema.

Preface

This document is a thesis in Computer Science (area of Program Comprehension) submitted to Universidade do Minho, Braga, Portugal.

Document structure

Part I - Introduction

Introduces the subject, presenting some basic concepts and ideas, the main challenges and goals, including the major research hypothesis.

Part II - Background and State-of-the-Art

Reviews some background concepts, methods and techniques used during this work, and discusses some related work available in the literature.

Part III - Ontology-based Concept Location

These chapters describe in detail the major contributions of this work, including the description and analysis of the implemented libraries and frameworks.

Part IV - Conclusion

Closing chapters presenting some concluding remarks and discussion, including some trends for future work.

Appendices

Complementary information is presented on the appendices.

Contents

Acronyms

AOIG Action-Oriented Identifier Graph.

ASDG Abstract System Dependence Graph.

CSV Comma-Separated Values.

DFT Dynamic Feature Traces.

FCA Formal Concept Analysis.

FEAT Feature Exploration and Analysis Tool.

IEEE Institute of Electrical and Electronics Engineers.

IR Information Retrieval.

kPSS *kind-of* Probabilistic Synonyms Set.

LSI Latent Semantic Indexing.

NLP Natural Language Processing.

OO Object Oriented.

OTK Ontology Toolkit.

OWL Web Ontology Language.

PC	Program Comprehension.
PTD	Probabilistic Translation Dictionary.
RDF	Resource Description Framework.
SBP	Scenario Based Probabilistic.
SDG	System Dependence Graph.
SE	Software Engineering.
SKOS	Simple Knowledge Organisation System.
SPARQL	Simple Protocol and RDF Query Language.
SVD	Singular Value Decomposition.
tf-idf	Term Frequency - Inverse Document Frequency.
TT	Template Toolkit.

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Introduction

Programmers have become part historian, part detective, and part clairvoyant.

T.A. Corbi

Reality shifts, bug fixes, updates or introduction of new features often require source code changes. These software changes are usually undertaken by software maintainers that may not be the original writers of the code, or may not be familiar with the code anymore. In order to carry out these changes, programmers need to first understand the source code [158]. This task is probably the main challenge during software maintenance activities [37]. The programmer is able to understand the program when he or she can explain the source code, and relate the code with the concepts in its problem domain [10].

Reverse engineering is the process of discovering how an object or system works through detailed and careful analysis of its structure and how available operations are performed. This approach has its origin in the analysis of hardware, mainly for military purposes, and has been used for years to infer how, and why, a mechanic system is assembled without any prior knowledge about its original design and/or implementation. For many years now that this discipline has also proven useful under the umbrella of software engineering, where the object of study is commonly a program. A similar approach can be adopted: deduce how the software works by analyzing and inspecting its building blocks and how they interact together to perform their role. This area of interest is typically called Program Comprehension [111, 33].

Program Comprehension (PC) is a field of research concerned with the analysis and understanding of applications and software source code, without previous knowledge about it, or its design goals. It is also concerned with the study of how humans analyze and comprehend source code, and the mental processes behind these activities, with the aim of devising methods and techniques that can assist during these processes. Many methods and techniques are widely available, and new ones are continuously being researched.

Comprehension and understanding activities, and knowledge in general, are strongly related with words and natural languages, and their active role as a *transport protocol* of information in and out of the human brain. When natural language words are weaved together using mathematical proprieties, objects in the family of the ontologies start to emerge. Hence, activities in the field of PC are often coupled with the study of languages, not only programming languages used to write source code, but also natural languages that are used foremost to discuss and reason about the original problem and related domains of interest. Many of the techniques used in PC also rely on mappings between human oriented concepts (described using natural language), and program elements (implemented using programming languages) [125]. These are often used to locate which parts of the program are responsible for addressing specific domain concepts [10], and are usually referred in the literature as feature location techniques [41].

The adoption of tools in the family of program comprehension allows a better and quicker understanding of software programs being analyzed. Which helps fault detection and bug discovery, and improves the development of better software and also helps maintaining it [113]. The use of these valuable approaches, and corresponding tools, is a very valuable help while analyzing software for finding or fixing bugs, discovering faulty sections of the software, improving software maintainability and overall quality.

This PhD thesis introduces a new approach to devise an effective program comprehension. It introduces the use of ontologies for describing program elements and concepts, and also other related domains of knowledge. The presented approach is concerned with first collecting and organizing, on one side concepts from the program domain, and on the other side concepts from the application domain. Which elements to explore, and concepts to capture, and which of these are conveyed to the program and problem higher level representation. Once concepts are available on both the program and the problem domain, the next step encompasses the creation of bridges between domains, and which relations and elements can be explored as clamps. The definition of these artifacts helps closing the gap between the different levels of abstractions of the languages used in both domains – programming languages in the program domain, and natural languages in the problem domain. Finally it explores the use of these bridges to create mappings, and other fine-tuned views, of the program (and sometimes also the problem) domain, that provide maintainers with insight and information that can aid in the understanding process and related activities.

The remaining sections of this chapter discuss the problem that this approach is addressing; it briefly motivates for studies in this area of research; and, discusses the main contributions of this work. The chapter concludes with an outline of the remaining chapters of this document.

1.1 The Problem

In order to change a program, either to add a new feature or to fix a problem, the maintainer needs to understand it. The programmer understands the program when he or she can explain the source code, relate the code with the concepts in its application domain, and is able to comprehend the concepts in the program and the problem domain. The challenge becomes to relate concepts described in natural languages – the problem domain – and the formal programming languages that are used to write source code – the program domain. For example, the task described in natural language as: *”reserve an airline ticket”* can be implemented as shown in the following snippet of pseudo code [10]:

```
if ( seat = request(flight) && available(seat) ) then
      reserve(seat, customer)
```

Once the programmer knows which real operations the code is implementing, he or she can understand the meaning and context of the program elements and reason about them, including verifying if it is working as intended. In this specific case as soon as the programmer understands that this code is about reserving an airline ticket, he understands the expected behavior of the *"request"*, *"available"* and *"reserve"* functions. This example is trivial for the human brain to process, but teaching a machine to do this systematically is not easy.

The main problem to solve is accurately compute which real concepts a specific program element is addressing. Common software applications are just too big and complex, to apply a top-down strategy, or have knowledge about the complete system [118]. Another major challenge, when addressing the problem of creating concept mappings between different domains, is the different level of abstractions used. An evidence of this situation is the abstraction level of programming languages used to develop computer programs to address the initial problem, and the natural languages used by humans to discuss, describe, and plan the problem outside the scope of source code [134]. Figure 1.1 cartoon emphasizes this gap.

The main research hypotheses has been defined as:

Can an ontological mapping between the problem domain, the program domain, and the real world effects of running the program, potentially provide additional benefits over existing approaches for collecting and relating information available in source code to enhance program comprehension?

1.2 Motivations

Program comprehension is an area that helps programmers better understand software programs [14]. Most of the times generally adopted techniques like program slicing and chopping are combined together to illustrate views of the program to give a better,

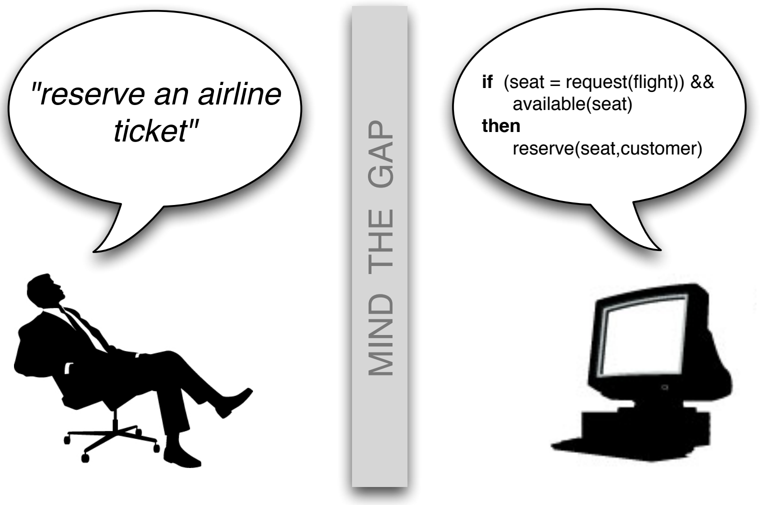

"reserve an airline ticket"
MIND THE GAP
if (seat = request(flight)) && available(seat) then reserve(seat,customer)

Figure 1.1: Natural language *versus* formal languages levels of abstraction gap cartoon.

or quicker, understanding of the program. Several known tools can be used to analyze source code. These technologies are important in software engineering as they improve program maintenance, debugging, testing, optimization and reuse.

The Linux Kernel[1] is one of the most successful open source projects ever and is a good example where the use of tools in the family of program comprehension can be very helpful. The Kernel release 2.6.35 has 33 335 source files, which makes an impressive total of 13 468 253 lines of code [36]. Imagine the initial effort required to dwell in the source code trying to fix a bug without any previous help or guidance.

In other examples of application, it has been demonstrated how a tool that analyses software helped finding areas of source code responsible for application bugs. This clearly reduced the time to find and fix the application problem [67]. This is a common scenario in many companies that offer services in the area of software refactoring, improvements, updates or maintainability. Murphy summarizes this general concern as: “developers may be spending more time looking for relevant information amongst the morass presented than working with it” [110].

Nowadays, software systems rule many aspects of our lives, and these systems are

[1]Available from: `http://www.kernel.org`, (Last accessed: 12-09-2014).

getting more complex every day. Faulty software, and related evolution and maintenance activities, have a direct impact on our day to day life, not only financial impact, but also creating life-threatening situations [170]. Maybe one of the most famous accident caused by software defects was the Ariane 5 explosion, less than a minute after launch [42]. Another example is the crash of a British Royal Air Force helicopter in 1994 costing 29 lives, where sufficient evidence was found for the cause of the accident to be related with a software defect [133].

1.3 Main Contributions

The overarching goal of this work is to contribute to enhance program comprehension tasks. In a broader sense, the main contributions of this work are:

- an increase awareness of the domain knowledge relevance, besides the source code, in the field of PC, and also about the approaches adopted for knowledge representation;
- a methodology for representing knowledge, including an ontology oriented formalism to describe knowledge from heterogenous domains;
- a methodology for creating relations between knowledge in different domains;
- a set of libraries and frameworks, to implement the devised methods and approaches to ease software maintenance and evolution tasks.

Some of the contributions have already been materialized in the following publications (in no particular order):

- "An Ontology Toolkit for Problem Domain Concept Location in Program Comprehension" [18], describes the general approach of using ontologies to devise methods and approaches in the context of PC, its' problems and motivations.
- "From Source Code Identifiers to Natural Language Terms" [20], discusses a new approach to convey program indentifiers to full sets of terms, by splitting mutiterm identifiers, and expanding abbreviations.

- “Open Source Software Documentation Mining for Quality Assessment” [26], and “DMOSS: Open Source Software Documentation Assessment” [27] are concerned with documentation analysis and extracting information from non-source code files usually available in software packages.

- “A Framework for Modular and Customizable Software Analysis” [101], illustrates the integration of the DMOSS application (described in detail in Chapter 8) in a broader system for software analysis.

- “Defining a Probabilistic Translation Dictionaries Algebra” [149], describes how to create required resources to build some of the artifacts used to compute scores between terms, used later for scoring elements in searching and mapping operations.

- “The Per-Fide Corpus: A New Resource for Corpus-Based Terminology, Contrastive Linguistics and Translation Studies” [4], discusses and illustrate many tools that are used in the background to produce the resources required to build *synsets* discussed in Chapter 7 and 8.

- “Conclave: Ontology-driven Measurement of Semantic Relatedness Between Source Code Elements and Problem Domain Concepts” [19], discusses the creation of mappings between source code elements and domain concepts to measure the semantic relatedness between them.

- “Conclave: Writing Programs to Understand Programs” [23], illustrates how systems like Conclave enhance PC activities, and how a set of well designed building blocks eases building complex systems.

- “OML: A Scripting Approach for Manipulating Ontologies” [25], defines a language to describe operations over ontologies.

- “Weaving OML in a General Purpose Programming Language” [24], describes an approach, and required tools, to weave the definition of ontology-*aware* operations inside a more general purpose programming languages to implement applications.

- “PFTL: A Systematic Approach For Describing Filesystem Tree Processors” [28], structural processing of filesystem trees to produce arbitrary side effects, a software package is as a tree of directories and files.
- “Structural alignment of plain text books” [138], text alignment is one of the main processes for obtaining parallel corpora, required to build Natural Language Processing (NLP) resources.
- “Generating flex lexical analyzers for Perl Parse::Yapp” [150], extends the syntax of a well known parser generation to automatically build lexical analyzers.
- “Probabilistic SynSet Based Concept Location” [21], discusses how synonyms sets can be used in concept location tasks.

Besides publications, contributions also include a set of tools and libraries developed during this work, and that are publicly available under an open source license, in ready to install libraries or applications. A set of the major frameworks and libraries developed and made available follows, including the public repository where development is being tracked, and everyone is invited to download the tool, browse the code, and submit comments (or issues found):

Conclave OTK

The ontology toolkit, that includes the ontology definition, storage backends, and methods for manipulating and retrieving information.

Public repository: `https://github.com/nunorc/Conclave-OTK`

Conclave Concept Mapper

The framework for performing searches and building maps, exploring data stored used Conclave OTK.

Public repository: `https://github.com/nunorc/Conclave-Concept-Mapper`

Conclave Utils

Set of specific features in the scope of software engineering used in the Conclave environment.

Public repository: `https://github.com/nunorc/Conclave-Utils`

Lingua IdSplitter

Generic application for splitting textual identifiers into words.

Public repository: `https://github.com/nunorc/Lingua-IdSplitter`

DMOSS

Application for analyzing non-source code content.

Perl distribution: `http://search.cpan.org/dist/DMOSS/`

1.4 Document Outline

A brief outline of the remaining of this document follows:

Chapter 2 presents some currently available knowledge in the literature about PC, describing some state-of-the-art theories and methods closely related with this work.

Chapter 3 introduces some ontology concepts, and state-of-the art representation formats and manipulation tools.

Chapter 4 describes some Information Retrieval (IR) techniques and common metrics used during this work.

Chapter 5 describes some NLP concepts, algorithms and tools that are used by some of the implemented tools.

Chapter 6 discusses the ontology oriented knowledge representation approach devised to store knowledge about different domains.

Chapter 7 describes in detail the method for creating semantic relations between domains and searching features.

Chapter 8 describes in detail some of the tools implemented during this work and Conclave, the final proof-of-concept system.

Chapter 9 presents experimental validations that help to draw conclusions about the effectiveness of some of the tools implemented during this work.

Chapter 10 concludes this document with some final remarks and trends for future work.

Appendix A introduces the Haskell notation that is used in some chapters to describe relevant functions, and how they are composed together.

Appendix B introduces the Template Toolkit templating engine, used in Chapter 6 to define templates for some common operations.

Appendix C introduces the SPARQL query language, and briefly describes how OTK uses it to perform operations on ontologies.

Appendix D presents the template used to bootstrap the program domain ontology.

Appendix E illustrates the survey proposed during the mappings experimental validation discussed in Chapter 9.

A couple of remarks regarding the content of this document. Mainly during Chapter 6 and 7, to clearly define functions (signatures, body or data types) the Haskell programming language is used. Most of the times the Haskell syntax is followed strictly, i.e., the code can be executed by an Haskell compiler, but sometimes in order to increase readability some details are simplified, resulting in non-valid code, but hopefully easier to be read by humans. Appendix A provides a brief introduction to the Haskell notation, emphasizing the most common expressions and statements used in this document. Algorithms description, mainly along Chapter 8, are written using a more algorithmic natural approach, and together with the text should be easy to follow. Most example programs are written in C, and are not simplified in any way.

Part II

Background And State-of-the-Art

Chapter 2

Program Comprehension

Program Comprehension (PC) is an area of software engineering concerned with the study of how software engineers understand and maintain programs [158]. When a programmer or maintainer is able to explain the program structure and behavior, its operations and effects, including the relations between source code elements and its application domain, the program (or a part of it) is understood [10]. The understanding activity is a fundamental stage to devise software changes during software maintenance and evolution activities [158, 37]. *Program evolution* and *software evolution* are used to encompass the progression of a software system through maintenance, including enhancements [86].

Software Maintenance is defined by the Institute of Electrical and Electronics Engineers (IEEE) Standard 1219 as:

Definition 1 *"The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment." [2]* ◊

Researchers have identified five tasks that require the understanding of a program during Software Maintenance: (i) adapt to new requirements; (ii) improve performance, efficiency or maintainability; (iii) correct errors or problems; (iv) identify and integrate reusable components; and, (v) code refactoring [158]. Understanding the code is among the first activities for any of these tasks, thus increasing the motivation for studies in this field of research.

Understanding source code is a challenging activity, mainly because of the gap between the application domain, and the programming domain languages and levels of abstraction [134]. For example, a maintainer analyzing the following function definition f_1:

$$f_1\ a\ b\ =\ sqrt\ (a^2\ +\ b^2)$$

understands the elementary algebraic operations involved, but in order to fully understand the program, it needs to know that this function uses the Pythagorean theorem to compute the hypotenuse. This enables the relation between the program elements and real world concepts, e.g., a and b are variables that represent sides of a triangle. This allows the programmer to validate if the function is working properly, since the result of such function is well defined outside the scope of the program.

A significant body of research exists on cognitive models to describe how programmers create mental representations of the code during software maintenance and evolution. The next sections describe some key concepts and models.

2.1 Mental Models and the Cognitive Process

During the understanding process, through observation, inference or interaction with the software, the maintainer creates and maintains a mental structure of the knowledge about the program.

Definition 2 *"A mental model describes a maintainer's mental representation of the program to be understood." [153]* $\Diamond$

Definition 3 *"A cognitive model describes the cognitive processes and information structures used to form the mental model." [153]* $\Diamond$

There is a varied collection of cognitive models available, with different characteristics. Some models are categorized as bottom-up or top-down approaches, others use some kind of hybrid or opportunistic approach, where the maintainer shifts between the most suitable approach according to available information, or previous knowledge (e.g., Mayrhauser and Vans approach [158]). The following sections introduce some key cognitive theories available in the literature.

2.1.1 Bottom-Up Theories

Bottom-up approaches propose that, the cognitive process of creating a model starts at the bottom, and moves upwards, i.e., the representation of the program is built by analyzing the source code (the bottom) and then building higher level abstractions referred as *chunks* [144, 107].

Shneiderman and Mayer bottom-up model [145], distinguishes between syntactic and semantic knowledge of programs. The syntactic knowledge refers to language dependent knowledge and is concerned with elements in the code. The semantic knowledge refers to the application domain, is language independent, and is built in progressive layers, starting from the code until a mental model of the program is formed. The final mental model is achieved by *chunking* (abstracting, by joining, conceptually related lower level units [107]) and aggregating semantic components and syntactic elements.

In Pennington bottom-up model [115], the programmer defines two mental structures: the program and the situation model. The program model captures the sequence of operations and procedures creating a control-flow abstraction of the program. The model is developed by chunking and *cross-referencing* (connecting elements at different levels of abstraction [158]) program units with textual structure abstractions. The situation model includes knowledge about data-flow and functional abstractions, requiring knowledge about the application domain.

2.1.2 Top-Down Theories

Brooks top-down theory [14], assumes that the understanding process is first based on reconstructing the domain knowledge and then mapping this knowledge to the source code. This is achieved by a successive formalization and validation of hypothesis. The presence of hypothesized structures or operations (beacons) refutes or confirms the initial, or subsidiary refined, hypothesis.

Soloway and Ehrlich in [152] claim that programmers use two distinct types of knowledge during the understanding process: programming plans, generic fragments of code that represent typical scenarios in programming; and rules of programming discourse, which capture coding standards, algorithm implementations, and other programming conventions. Rules of discourse and beacons are used to decompose plans and goals, until a hierarchy of plans (built top-down) is formed. Expert programmers, i.e., programmers with more years of experience and practice, are bound to have a richer knowledge of programming plans and rules of discourse, and are prone to work their way faster down the understanding process. Although, when expected rules of discourse are violated, the understanding time of expert programmers is roughly the same as novice programmers.

2.1.3 Other Approaches

Mayrhauser and Vans approach [158], describes a hybrid approach, where a top-down or bottom-up strategy is adopted by the programmer according to the situation. Namely, Soloway's top-down model and Pennington's bottom-up model. When the programmer or maintainer is familiar with the code, beacons are identified and the understanding approach follows a top-down approach, otherwise a bottom-up approach is adopted.

In Letovsky cognitive model [89], a maintainer understanding a program uses a combination of top-down and bottom-up approaches, as best fits the current knowledge. The model defines three elements: (i) a knowledge base, programmers existing knowledge of the application domain, programming domain, programming plans, goals and rules of discourse; (ii) a metal model, the program mental representation; and, (iii) an assimilation process, the process used to evolve the mental model. During the assimilation process the maintainer performs *inquiries*, i.e., formulates questions, conjectures about the answers, and then searches the code to verify these answers,

Littman *et al.* describe an opportunistic approach [91], where the maintainers use either a systematic or as-needed comprehension strategy. A systematic approach allows the creation of a mental model of the program based on information about the code (static knowledge) and interactions between elements of the program when executed (casual knowledge). Maintainers adopting an as-needed approach mental model is only based on static knowledge, resulting on a weaker mental representation of the program.

2.2 Concept and Feature Location

Concept location is the process of locating relevant domain concepts in source code, and is a key task in the area of PC. This is typically the first step a programmer undertakes in order to devise a code change.

Definition 4 *A concept is a principle, an abstract idea, a single unit of knowledge, and that typically has representation in a language, a term for example, or a symbol.* ◊

Probably being a maintainer, and not the original developer, the programmer in charge of updating the code knows the domain concepts of the the problem, but is unaware where the implementation of these concepts is written in the code. This is a required task during most software maintenance and evolution activities, and has been identified as a real problem by software engineering researchers and practitioners, thus many techniques have been developed to address this specific problem.

grep is a common approach to find relevant keywords in source code. This utility is used to find matches of terms which represent concepts that the current software maintainer is searching for. More advanced versions of this approach use regular expressions to match terms. This helps the programmer when looking for specific areas of the code that address the concepts that require updating or revision. A major drawback of *grep* is that is basically a technique that finds a string in a text, it is not context or program elements aware (meaning that it does not distinguish if looking for a function or a variable, or looking inside a specific class or module), and it does not take in consideration any semantic value that might be available (it is purely syntactic). Such drawbacks, shared by approaches like *grep*, motivated researchers to devise more complex strategies, able to provide a richer set of features to programmers and maintainers.

Concept location techniques can be categorized by type of analysis: (i) dynamic analysis, which is based in software execution traces, and examines programs runtime (e.g. [161, 164]); (ii) static analysis, based on static source code information, such as slicing, control or data flow graphs (e.g. [31, 129]); and, (iii) textual analysis, that explore natural language text found in programs like comments or documentation. This last type can be based on IR methods (e.g. [4,5,26]), NLP (e.g. [18,41]), or pattern matching (sometimes also referred as grep-*like*) based approaches (e.g. [14]).

The following section introduces Formal Concept Analysis (FCA) an underlying technique for Concept Location, and Section 2.2.2 throughout 2.2.6 introduce some state-of-the-art techniques available in the literature, organized by type of analysis.

2.2.1 Formal Concept Analysis

Formal Concept Analysis (FCA) [54] is a branch of mathematical lattice theory that provides means to identify meaningful groupings of objects that share common attributes, and provides a theoretical model to analyze hierarchies of these groupings. The main goal of FCA is to define a *concept* as a unit of two parts: (i) the *extension*, set of objects that belong to the concept, and (ii) the *intension*, set of attributes that are common to the objects under consideration.

The *formal context* is required by FCA, which indicates which attributes objects have.

Definition 5 *A formal context is a triplet (O, A, R), where O is a set of objects, A is a set of attributes, and R is a binary relation between objects and attributes ($R \subseteq O \times A$).* $\Diamond$

Formally, a set of concepts can be generated from the formal context using Def. 6, where X is the *extent*, and Y is the *intent*.

Definition 6 *A concept is a pair of sets (X, Y), such that: $X = \{o \in O \mid \forall a \in Y : (o, a) \in R\}$ and $Y = \{a \in A \mid \forall o \in X : (o, a) \in R\}$.* $\Diamond$

The set of all concepts of a given formal context forms a partial order via the superconcept-subconcept ordering: $\leq$: $(O_1, A_1) \leq (O_2, A_2) \Leftrightarrow O_1 \subseteq O_2$ or, dually $(O_1, A_1) \leq (O_2, A_2) \Leftrightarrow O_1 \subseteq O_2$. The set of all concepts of a given formal context and the partial order $\leq$ form a *concept lattice*. The concept lattice is the starting artifact for further analysis, it can be represented graphically or subject of more algebraic operations.

FCA has been used in the context of Software Engineering (SE) [151], for example, to support requirement analysis (e.g., [44]), software maintenance activities (e.g, [5, 121]), among others, a comprehensive survey is available in [155].

2.2.2 Static Analysis

Static analysis techniques inspect the source code, and explore its dependencies and structure, without executing the program.

Chen and Rajlich present a computer assisted search method for locating features and concepts related with a maintenance request [31]. This approach proposes the use of Abstract System Dependence Graphs (ASDGs), an higher level abstraction representation of a program based on System Dependence Graphs (SDGs) [65, 64]. In an ASDG vertices represent the components in the program (functions and global variables), *call* edges describe function calls and *data flow* edges describe flows of data between functions and global variables. In each step of the search process a component is selected, the *search graph* [30] is updated. During the search process the programmer role is to make decisions from a possible set (e.g., choose a component to visit, check if goal is reached), while the supporting tools perform other tasks (e.g., update the search graph, extract dependencies graphs). The process ends when the maintainer is satisfied with the set of relevant components visited.

Robillard and Murphy developed the Concern Graphs representation [129, 132], that abstracts the implementation details of concerns, by storing only key structures, and explicitly emphasizing the relationships between the different elements of a concern. Concern Graphs are automatically extracted from source code, or intermediate representations. Feature Exploration and Analysis Tool (FEAT) is the tool introduced to create and explore Concern Graphs.

Walkinshaw presents an approach [159] that combines *landmark* methods – methods that have a key role in a particular feature – with slicing, to create a call graph of the code related with a feature of interest.

Saul *et al.* in [139] present a set of algorithms to analyze rich APIs to find and recommend functions related with a given function, based on a random-walk approach. This approach relies only on the source code, which is always available.

Trifu approach [156], uses static data flow information (using directed graphs) for identifying concerns in Object Oriented (OO) programming systems, to support software understanding activities. This approach has been applied to the *JHotDraw* case study and was able to identify a significant number of concerns.

Robillard technique [128], from a set of program elements, produces a set of methods and fields of potential interest, based on topology of structural dependencies analysis. Each element in any of these sets, has associated a value that measures its relevance. Relevance values are based on two metrics: (i) *specificity*, inversely proportional to the number of elements related to it; and, (ii) *reinforcement*, directly proportional to the number of elements related to it. This implies that a program element becomes more specific, as fewer program elements are related to it. This approach was applied to two medium sized software systems, which has shown that it can help developers selecting program elements more worth investigating.

2.2.3 Dynamic Analysis

Opposed to static analysis, dynamic analysis requires the source code to be executed, and relies on collecting information about the program during runtime.

Wilde *et al.* introduced the *Software Reconnaissance* technique [161, 162]. This method executes the program using test cases that exercise the feature of interest, and another set that does not. The program elements that implement the feature of interest are identified by analyzing the differences between the executing traces of the two sets.

Wong *et al.* describe an execution slice-based technique [164], where an execution slice is the set of program components executed by a test input. The general idea is to re-use the tests written for the application, to identify which program elements are unique to a specific feature, and provide a programmer or maintainer with starting points for understanding large software systems.

Eisenbarth and De Volder introduce Dynamic Feature Traces (DFT) [50], a technique for feature locating based on execution-trace analysis. This approach uses the same approach of comparing executions traces of tests that exercise, or not, specific features, followed by [162] and [164], but tries to overcome the most significant problem of dynamic analysis – coming up with execution scenarios that exercise one specific feature.

Eisenberg *et al* introduce a technique based on execution traces for different scenarios, where each scenario represents the invocation of a single feature, identifying the program components executed for each feature [48, 49]. A concept analysis technique is then applied to the components list and the set of features, highlighting the relations between features and components – the feature component map.

Safyallah and Sartipi introduce an approach that explores frequent patterns in execution traces, to identify features in source code [137]. A set of case scenarios that share specific features is used to produce execution traces. A sequential pattern mining algorithm is then applied to the execution traces to highlight frequent execution patterns. After a set of refinements, the result are continuous fragments of execution traces that correspond to a particulars features.

Edwards *et al.* introduce a feature location technique for distributed systems [46]. To overcame the problem of the stochastic nature of distributed systems, and the difficulty of correctly ordering events (time-*wise*), a definition of time intervals based on causal relationships between events is proposed. Besides the execution traces, the first and last events related with a specific feature are also required, in order to identify the event set related to that specific feature. The final output is a ranked list of program elements inside this interval, related with a specific feature.

Bohnet *et al.* describe an analysis technique [11] that combines information from dynamic execution traces with information on the hierarchical structuring of implementation units. The program is executed using a scenario that exercises the feature of interest. The execution trace is analyzed and several views of the trace are created that highlight various characteristics. These views are synchronized, i.e., they simultaneously present the information extracted from the trace from different perspectives, and when the maintainer focus the point of view on a specific detail of the trace, all the views update to provide their view of the same detail.

2.2.4 Textual Approaches

Textual analysis techniques explore natural text found in software systems (e.g., comments, identifiers, documentation) to gather information about program elements.

Petrenko *et al.* in [118] propose the use of ontology fragments, an ontology based on the partial knowledge about features, during the understanding process. These ontologies fragments help the maintainer in formulating queries (e.g., proving terms, attributes and concepts) while searching the code for relevant components.

Marcus *et al.* in [99] explore a IR technique called Latent Semantic Indexing (LSI) (introduced in Chapter 4), to map concepts expressed in natural language by programmers to the source code components. This approach uses LSI to find semantic similarities between queries and software, to locate concepts of interest. Identifiers and comments are extracted and pre-processed to build a corpus, which is partitioned to create documents representing related program elements. The query and the documents are transformed in vectors, and the similarity measure between vectors is used to rank documents (relevance-*wise*) against queries.

Cleary *et al.* introduce the cognitive assignment technique in [34], an approach also based in IR techniques, but including information derived from non-source code artifacts.

Abebe and Tonella in [3] present an approach for extracting concepts and relations from source code identifiers, based on NLP techniques. Sets of sentences, built from lists of terms in the code, are parsed using NLP, and then used to extract concepts and relations. This information is stored in an ontology, which is used to enhance queries, and increase the precision of concept location activities.

Hill *et al.* in [62] also explore the use of NLP techniques, to automatically extract and categorize phrases from source code identifiers to categorize and organize search results. Once phrases are extracted from method and field names (sometimes also looking at methods parameters), they are grouped into a hierarchy based on partial matching, and are linked to the original source code. A maintainer when formulated a query (e.g., searching for a feature), is presented with is a hierarchy of phrases and contextualized program elements, which help to identify relevant program components.

Sheperd *et al.* introduce the Action-Oriented Identifier Graph (AOIG) in [143], a natural language representation of the source code, which help linking actions scattered in the program, built using NLP techniques. This work also demonstrates how a AOIG can be used during software maintenance activities.

Würsch *et al.* in [165], introduce a framework to query information about a software system, using a natural language similar to plain English. Program elements information is modeled in an ontology, which is later used by a guided-input natural language interface, to answer questions formulated by the maintainer.

2.2.5 Combined Techniques

In order to achieve an overall better result, some approaches combine several techniques, so that limitations of one technique can be compensated by another. This section introduces some examples of using combinations of techniques.

For example, Antoniol and Guéhéneuc, in [6] introduce a technique using both static and dynamic analysis, for feature identification in large object oriented programs. Using a set of scenarios that exercise specific features, sets of static data are filtered, allowing the relation between classes, features, and scenarios. Maintainers can use this approach to build micro-architectures (subsets of program architectures), for feature identification and comparison.

Poshyvanyk *et al.* in [119] combine static and dynamic analysis, by applying LSI to source code, and a Scenario Based Probabilistic (SBP) ranking of events extracted while executing the program under specific scenarios. The case studies discussed show that the combined techniques provide complementary results.

Eaddy *et al.* in [45] introduce prune dependency analysis, a hybrid feature location technique. An IR technique is used to extract terms from comments and identifiers; execution traces are used to determine which program elements are executed when a concern is exercised; and, then a prune dependency analysis technique is used to find the impact of removing relevant elements to infer additional elements. According to the authors the synergy between the different techniques produces more accurate results than other combinations of techniques.

2.2.6 Other Approaches

Other approaches explore information available in other artifacts outside the scope of the program itself (e.g., bug reports, source version control systems, development environments), this section introduces some of these techniques.

For example, Yao, in [167] introduces a tool that extracts information from the CSV version control system. It features an algorithm for mapping commit comments to the source code, and related program elements. Given a query, the tool produces a ranked set of lines of code, including a score of how better each line matches the query.

In another example, Robillard and Murphy, in [130], introduce a technique for concept location based on source code investigation undertaken during maintainers and developers activities. The results of these activities are documented using a set of concerns description, extracted by an algorithm from the investigated elements. This algorithm is based on elements visited during an investigation session, and how the maintainer navigates between elements.

2.2.7 Tools

Some of the work discussed and presented in previous sections, and other, spawned concrete tools for concept location. This section briefly enumerates some of these tools.

FEAT (Feature Exploration and Analysis Tool) is an plug-in for eclipse based on the Concern Graphs approach [131].

Featureous a plug-in for the NetBeans IDE, provides views for legacy Java software analysis [112].

Google Eclipse Search is a search engine that integrates Google Desktop Search into the Eclipse development environment, providing improved searches [122].

Ripples a tool for concept location based in ASDG [32].

STRADA helps software maintainers to explore traces links to source code through testing [47].

Suade an Eclipse plug-in for automatic generation of program investigation suggestions using static analysis [160].

TraceGraph a tool for execution traces analysis for concept location, using a simple visual interface [93].

2.3 Program Identifiers Normalization

Program identifiers are one of the major source of information about program elements [16, 17], and their meaningfulness has a direct impact on future comprehension tasks [83]. Most of the programming communities promote the use of best practices and coding standards, that usually include rules and naming conventions which tend to improve the quality of identifiers used (e.g., the style guide for the Python programming language[1]).

Program identifiers have been greatly explored in the context of program understanding: for concept and concern location (e.g., [142, 99, 3, 92]), relating documentation with source code (e.g., [7, 166, 98]), and other assorted software analysis applications (e.g., [84, 79, 51, 22, 27]). All this work can benefit from better program identifiers handling, and in many cases results can be improved [40].

Programming languages grammars constrain the strings that can be used as identifiers, not allowing spaces and other special characters (e.g., commas). These also tend to be short and easy to remember. Thus, acronyms and abbreviations are frequently used to represent real world concepts.

Identifiers created using a single word (or abbreviation) are easier to relate with domain terms. The real challenge are compound identifiers, i.e., identifiers assembled using more than one string (each representing a term), because these strings need to be correctly isolated before they can be linked with domain concepts. Moreover, these strings can be abbreviations or acronyms, and not actual words, increasing the tokenization process difficulty. Sometimes an explicit mark is used to delimit the strings used, for example, the identifier "*insert_user*" uses the underscore as an explicit mark

[1] Available from: `http://legacy.python.org/dev/peps/pep-0008/` (Last accessed: 31-03-2014).

to clearly distinguish the word "*insert*" and the word "*user*". Another common explicit technique is the CamelCase notation, for example in the identifier "*insertUserData*" the words used are explicitly delimited with an uppercase letter. This trend of explicit word compounds are referred in the literature as *hard splits* (or *hard words*). Many times no explicit mark is used to delimit the words, for example the identifier "*timesort*", was formed by joining the words "*time*" and "*sort*", but there is no explicit mark where one word ends, and the next word begins. This is usually referred as *soft splits* (or *soft words*). Splitting *soft words* is more complex that *hard words*, and the complexity increases when acronyms or abbreviations are used instead of complete words [83, 81, 82]. The next section introduces some state-of-the-art techniques to address this problem.

2.3.1 Current Approaches

The work by Caprile *et al* [16], describes their lexical, syntactical and semantic analysis of function identifiers. In this work the creation of a dictionary based on information extracted from the software (source code mainly) was also a concern, and a valuable source of information. It also helps to highlight the relevance of NLP techniques applied in the context of Program Comprehension.

Enslen *et al* in [51] describe Samurai, an automatic approach to split identifiers that uses a scoring function based on program-specific and global frequency tables. These tables are built by mining strings frequency in source code. The main intuition behind this algorithm is that sub-strings used as part of an identifier are likely to be used in other identifier from the same software, or even in other programs. A similar concern is behind our proposed custom corpus-based dictionaries, the expressions and terms found in natural language text belonging to the software domain are prone to be used as identifiers.

TIDIER [94, 57] is another approach for identifiers splitting. This algorithm is based in the Dynamic Time Warping algorithm, initially devised to compute distances in the context of speech recognition. And tries to achieve the correct split by computing distances between the identifier and words found in a set of dictionaries. The algorithm takes advantage of dictionaries, including domain specific dictionaries, and the inference of abbreviations is based on computing some kind of metric between the identifier and words found in dictionaries. A possible short-coming of this approach (and the previous one – Samurai) is that both can produce a different split for the same identifier in different iterations. TIDIER also does not handle splitting identifiers that contain single letter abbreviations.

TRIS [58] is a more recent technique for splitting and expanding program identifiers proposed by the same authors of TIDIER. It also uses a set of dictionaries, general and domain specific. TRIS handles the splitting and expansion as an optimization problem, divided in two stages. During the first stage a set of dictionary word transformations is created including corresponding costs, and during the second phase the goal is to find the optimal path in the expansion graph. The resulting split and expansion corresponds to the one with the minimal cost.

The GenTest normalization algorithm proposed by Lawrie *et al* described in [80] and [79] involves vocabulary normalization found in software artifacts (e.g., source code, documentation) to improve Information Retrieval software analysis tools. This algorithm starts by scoring all the possible splits, and the resulting split is the one with the highest score. The scoring function is based in a set of metrics, based on internal information (e.g., word characteristics), and external information (e.g., dictionaries).

LINSEN [35] is an approach for splitting identifiers, and expanding abbreviations, proposed by Corazza *et al*. The authors propose the use of the Baeza-Yates and Perleberg, an approximate string matching technique, and the use of several general and domain specific dictionaries, to find a mapping between program identifiers and the corresponding set of dictionary words.

The work by Sureka [154], is a more recent approach for splitting identifiers using the Yahoo web search and image search similarity distance. The main idea is that strings used as identifiers represent concepts in real life, and documents indexed in search engines include images and text, providing information to compute possible splits scores.

Butler *et al* in [15] describe the INTT algorithm, a technique for identifiers names automatic tokenization, with special focus on single case identifiers, and identifiers containing digits. INTT also takes advantage of a pre-defined set of dictionaries, including commonly used abbreviations and acronyms.

These approaches (more details in [52]) help to highlight the relevance of processing program identifiers, in the context of software analysis. The usage of NLP techniques and various types of dictionaries is a common trend in modern approaches, and the corresponding empirical studies help to highlight their added value and benefits.

2.4 Knowledge Domains

Knowledge is a common element in most of the cognitive models described in Section 2.1. The process of understanding a program implies the forming (or re-forming) of mental models based on maintainers previous knowledge, and new knowledge acquired during the understanding process.

The most typical domains of knowledge defined in PC are the problem domain, and the program domain, introduced in the next sections.

2.4.1 The Problem Domain

The problem (or application) domain [13] is concerned with real world concepts and problems the application is solving. The knowledge about the application domain plays an important role during the understading process [141].

The domain model of the application domain, which represents real world concepts that are addressed by a specific application, captures information about the application domain (e.g., terminology, objects, relationships). The domain model provides the maintainer with relevant knowledge about the domain, for example, provide initial sets of elements to query, or keywords to search, that are expected to be present in a program in a specific domain. The relations described in the application model can guide the understanding process providing the maintainer with information to be expected about a specific domain [135].

2.4.2 The Program Domain

The program domain [13] binds knowledge concerned with programming and source code concepts. This includes general programming concepts (e.g., programming paradigm, programming languages), and in some cases, may include information about actual components or elements in the source code.

The program contains an assorted amount of technical information (e.g., algorithms, data structures), but also, scattered throughout the program, knowledge about the application domain is available. The concepts to which this knowledge is related, is not always explicit. Modeling the source code, abstracting some specific programming concepts, and discarding specific technology details, may promote a better construct for understanding activities. Not only for the maintainer, but also for the application of other techniques. The program model provides a high level abstraction of the source code [126, 136]. Some techniques presented in Section 2.2, introduce several program models, with different aims, and heterogenous levels of abstraction.

Chapter 3

Ontologies

3.1 Introduction and Definition

One of the major goals of this work it to represent information gathered from software analysis and related domains using ontologies, for later reasoning. To achieve this a formalism to represent the information in a ontology is required, and also some methods (and tools) to reason about the available information. This chapter overviews some of the current trends for ontology representation and manipulation.

The term *ontology* has its origin in the field of philosophy. Ontologies are one of the solutions found in computer science to represent knowledge about a well defined domain in a structured way. Ontologies can be used to represent knowledge about any kind of domain or area of interest. The use of the term ontology in computer science was first introduced in the area of artificial intelligence reasoning [102]. An ontology was used to represent the things that existed in a given domain. The idea still persists today. An ontology is used to represent knowledge about a domain, by representing things that exist in that domain.

Another important term that we have been using but have not yet defined is *domain*. An ontology is always an artifact on a given domain. Again, this term is used in a wide range of sciences which makes it hard to define. But, it can be stated that a domain is a way of refereeing a particular well defined area of knowledge. Sometimes this knowledge may not be clearly bounded [68]. From the Oxford English Dictionary, the domain definition:

Definition 7 *A sphere of thought or action; field, province, scope of a department of knowledge, etc. [1]* $\diamondsuit$

During this work the following definition is assumed:

Definition 8 *a domain ontology is an engineered artifact that informally defines concepts from a specific domain, representing and organizing them as conceptualizations which a set of systems working cooperatively with each other agree to share [68].* $\diamondsuit$

Formal definitions, more common in Computer Sciences, are devised to clearly define what an ontology is. An example is the definition proposed by Serra *et al.* [140], where an ontology is defined by the following tuple:

$$O = (C, H, I, R, P, A)$$

Where, C represents the set of concepts in the domain; H the set of taxonomic relationships between concepts; I the set of relationships between classes and instances; R the set of other relationships; P the set of properties of classes; and, A the set of axioms.

Regardless of the concrete approach chosen, most formal definitions usually share some characteristics, namely the use of:

- classes (or concepts) to create groups of elements that share properties;
- instances to group elements (or individuals) in classes
- properties to define broader classes or instances characteristics;
- relations between classes and instances.

The specific definition that is used during this work is presented and discussed in Chapter 6.

Sometimes other structures that belong to the ontology family, but most of the cases they are quite distant cousins, are used. Nevertheless they are useful when some kind of conceptualization is required. Some examples of these structures are [12]:

Glossaries: lists of terms and definitions. In some cases it is a relation between terms and equivalents, not exactly definitions.

Thesaurus: networks of well defined interrelations, or associations, between terms. Given a particular term, a thesaurus will indicate other terms with the same meaning, which terms denote a broader category, which denote a narrower category, and which are related in some other way.

Taxonomies: traditional structures that arrange terms into groups and subgroups based on predetermined rules. Groups also need to follow well defined hierarchical relations.

When adopting this broad family of structures, most of the times tools that were originally designed to work with ontologies can be used. Which means that these data structures can be represented using languages that were designed to describe ontologies. Some languages are already specialized in describing some of these structures.

Ontologies are a popular approach to represent knowledge. Besides the formal definitions, to actually store and use them in a modern computer system, a concrete solution (format) for representing the information is required. The next section introduces and discusses some well known languages and formats to represent ontologies.

3.2 Representation and Formats

There are several ways to represent, and therefore store and share, ontologies. Some of them are more suitable to some kind of particular tasks or operations. The next sections briefly introduce some examples of families of languages that are used to describe ontologies, or some well defined subsets. Most of these languages use some kind of XML notation. This is mainly a portability issue, it makes information exchange between different systems easier.

```
<rdf:RDF xmlns="http://ontologies/example" />
  <owl:Class rdf:about="Felis" />
    <rdfs:subClassOf rdf:resource="#Felidae" />
  </owl:Class>
  <owl:ObjectProperty rdf:about="eatsMeat">
    <rdfs:domain rdf:resource="#Felis"/>
  </owl:ObjectProperty>
  <owl:Class rdf:ID="house\_cat" />
    <rdfs:subClassOf rdf:resource="#Felis" />
  </owl:Class>
</rdf:RDF>
```

Figure 3.1: OWL example.

OWL

The Web Ontology Language (OWL) is a family of languages for publishing and sharing ontologies on the World Wide Web [63]. This language is mainly developed and maintained by the World Wide Web Consortium (W3C). The OWL specification includes the definition of three variants:

- OWL Lite, supports basic needs of a classification hierarchy and simple constrains.
- OWL DL (Description Logic), supports maximum expressiveness.
- OWL Full, meant for maximum expressiveness and syntactic freedom of RDF.

OWL is intended to provide a language that can be used to describe the classes and relations in Web documents and applications [103].

The basic elements in OWL are: classes, objects, individuals, and properties. These basic elements can be put together to create ontologies. Figure 3.1 illustrates a simplified version of the definition of a class, and an instance, in OWL usig the RDF/XML syntax (a common format used to store OWL ontologies).

There are a lot of characteristics that are used to further specify properties of classes. Examples of proprieties are: transitive, symmetric, functional, and inverse. Although OWL is a complex language to use, and representations can quickly become complicated and confusing; this complexity translates in a more accurate knowledge representation. Ontologies are used among many sciences, OWL might be hard to use for

```
<Felis> rdf:type skos:Concept ;
  skos:prefLabel "Felis" ;
  skos:broader <Felidae> .

<Lucky> rdf:type skos:Concept ;
  skos:prefLave "Lucky" ;
  skos:broader <house_cat> .
```

Figure 3.2: SKOS example.

someone without a background in computer science. This can be an obstacle for the spread of the language between different communities.

OWL can be used to represent ontologies, taxonomies, thesaurus, etc. In 2009 the OWL working group published the OWL 2 Web Ontology Language, an extension of the previous version adding even more features. New features include additional property and qualified cardinality constructors, extended datatype support, simple metamodeling and extended annotations [109].

SKOS

Simple Knowledge Organisation System (SKOS) is a family of languages that are used for expressing the basic structure and content of concept schemes. It is published and maintained by the W3C Semantic Web Best Practices and Deployment Working Group. SKOS basic elements are classes and properties. In opposition to OWL, SKOS is not a formal knowledge representation language. It can be used to create thesaurus, taxonomies, classification schemes and terminologies for example, and it can be used together with OWL to create more demanding things as ontologies [105].

A simple example of this language representation is illustrated in Figure 3.2. There is no specific way to represent instances of concepts, so one way to go is to represent individuals as new concepts. SKOS is a very powerful vehicle already being used in many situations instead of OWL [104].

Topic Maps

Topic Maps is a specification that provides a grammar and a model for representing the structure of information resources [116]. Topic maps resolves around three basic concepts:

- Topics, that can represent anything, a fact, a person, an entity, concepts, etc.
- Associations, relations between one or more topics.
- Occurrences, any information relevant to a given subject.

In Topic Maps documents, real world subjects are represented using topics. Topics have names, and can also have occurrences that can be used to specify any information being relevant to a subject. For example for a person occurrences might be his home address, phone number, picture, etc. Topics can also have roles, roles can be used to define the role that a topic has in a association. Relationships between topics are modeled with associations. Topics can have different roles for different associations, for example a person can play a role of author in a association with a paper and a role of editor in a association with a book. Associations also have types. Figure 3.3 illustrates an example of using Topic Maps.

Topic maps can be used to create rich documents that contain structured information. Topic Maps can be used to represent and manage any kind of subjects and relationships between them, which means that any data structure discussed before (glossaries, thesaurus, ontologies, etc) can be represented [157].

ISO 2778

`Biblio::Thesaurus`[1] is a module that was created to store information based on ISO 2778. This ISO standard is developed and maintained by the International Organization for Standardization (ISO).

[1]Available from: `http://search.cpan.org/dist/Biblio-Thesaurus/` (Last accessed: 10-09-2014).

```
<topic id=”Felidae”>
  <baseName>
    <baseNameString>Felidae</baseNameString>
  </baseName>
</topic>
<associantion id=”broader—than”>
  <member>
    <roleSpec>
      <topicRef xlink:href=”#bigger”/>
    </roleSpec>
    <topicRef xlink:href=”#Felis”/>
  </member>
  <member>
    <topicRef xlink:href=”#Felidae”/>
  </member>
</association>
```

Figure 3.3: Topic Maps example.

```
Animal
BT Carnivora

Carnivora
BT Canidae, Felidae

Felida
BT Panthera, Felis
```

Figure 3.4: ISO 2778 example.

This module was initially created to provide a set of tools to maintain thesaurus files. We already discussed how a thesaurus can be defined as a sub set of an ontology. But this module has grown and now is prepared to work with more abstract and complex structures, like ontologies for example. It still maintains the name, but that is bound to change in the future.

The internal representation for the ontology follows ISO 2788. This means that it can interact with other sources that follow the same standard. Note that the module was changed to work with more complex structures, and the standard defines standard features to be found on thesaurus files. An example of the ISO representation looks like:

```
  $ontology->addTerm('termA');
  $ontology->addRelation('termA','relation','termB');
```

Figure 3.5: Biblio::Thesaurus API example.

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:terms="http://purl.org/dc/terms/">
   <rdf:Description rdf:about="urn:x-states:NewYork">
         <terms:alternative>NY</terms:alternative>
   </rdf:Description>
</rdf:RDF>
```

Figure 3.6: RDF example.

This module has already been successfully used to translate other resources into ontologies. A well defined API allows the manipulation and access to various information in a very simple way. Adding or deleting information can be as simple as illustrated in Figure 3.5 [148].

Clearly is a very different approach form the other representations discussed before, which has advantages and disadvantages.

RDF

RDF is a W3C language for representing information about resources in the World Wide Web. RDF is based on the idea of identifying things using Web identifiers (called Uniform Resource Identifiers, or URIs), and describing resources in terms of simple properties and property values [97]. The underlying structure of any expression in RDF is a collection of triples, each consisting of a subject, a predicate and an object.

Most of the times we see RDF graphs encoded in XML, this syntax is defined in RDF/XML syntax specification [9]. An example of the standard RDF/XML representation is illustrated in Figure 3.6. RDF by itself is not suitable to represent structures like ontologies, but, as already has been illustrated, it is very common for other languages and models to use RDF.

```
(#$genls #$Canidae #$Carnivora)

(#$isa #$Lucky #$house_cat)
```

Figure 3.7: CycL example.

```
 (#$implies
   (#$isa ?APPLE (#$FruitFn #$AppleTree))
   (#$colorOfObject ?APPLE #$RedColor) )
```

Figure 3.8: CycL inference expression example.

CycL

One of the first languages to aim for knowledge representation was CycL [88]. This formal language is mainly used by the Cyc knowledge based. Cyc is a project to create a comprehensive ontology and knowledge base of everyday common sense knowledge. The language itself is a bit different from those shown before. Figure 3.7 illustrate this language, this examples states that all members of the `Carnivora` group are also members of the `Mammalia` group, individuals are represented using the predicate `isa`.

CycL also supports variables in expressions, variable names start with a question mark (?). Knowledge can also be inferred, the example illustrate in Figure 3.8 states that every fruit of the apple tree has color red. This expression in loosen English reads, that every fruit of the `AppleTree` named `?APPLE`, which is acting as a variable, has color `RedColor`.

One big problem with CycL is that the predicates for the expressions are all predefined. Which means that to be able to represent any arbitrary relation, a predicate for that relation is required.

Notes Regarding Representation

All the languages discussed are interesting and provide useful instruments to represent knowledge. The common use of XML is a valuable asset, it makes easier the job of transporting and storing data formats. Languages like OWL are very complete and allow for accurate representations, but they tend to easy became complex, and the concepts being represented tend to be lost in the language syntax.

Languages variables, and ways to infer new knowledge from the existing information, are interesting feature of some of these languages. Defining a new concept using other concepts or concepts proprieties enhances the language, and allows for users to create expressions for adding more complex information. It can also save a lot of work, instead of enumerating for every individual, or concept, or any given propriety, an expression can be defined that implicitly adds that propriety for the class of individuals, or concepts, that share them. For example, every time an animal is added to an animal classification ontology, that belongs to the `Carnivora` class, a fact that states that the newly added animal eats meat could be added. Instead a generic fact stating that every animal that belongs to this class eats meat, could be added.

Of course there are a lot more ways to represent ontologies, or similar knowledge structures, than the ones presented here. But the illustrated set shows the most popular and adopted approaches, and is enough to emphasize most common traits and details.

3.3 Tools and Libraries

Since there are several ways to represent ontologies, there are also different approaches to manipulate them. Several software packages offer methods to change and manipulate information in a ontology. Once agreed on which representation to use, a set of readily tools and libraries is immediately available.

A small list of examples of tools that can be used to manipulate information in an ontology follows:

Protégé [2] is an open-source platform that provides a suit of tools for building knowledge representations based on ontologies. It has a specific extension to work with OWL. This extension allows for a visual editing of information stored in a OWL ontology, and other well known formats.

[2]Available from: `http://protege.stanford.edu/` (Last accessed: 10-09-2013).

Jena Framework [3] is a framework building semantic web applications in Java. This framework includes a wide set of classes for use in Java development. Among many things, this framework has an interface called *OntModel* that can be used with other tools in the framework as interfaces to underlying models, written in OWL for example.

SWOOP [4] is another tool for creating and editing OWL ontologies. This project is also hosted on Google Code. This tool has a nice look and feel and it is very intuitive. Simple interface with concise operations over ontologies and a plug-in option for quick development of new features.

ThManager [5] is an open source tool that is able to manage thesauri stored in SKOS, allowing their visualization and edition [76].

OWL Visual Editor [6] is, as the name clearly shows, a visual editor for OWL.

Biblio::Thesaurus is module that can be used to build ontology-*aware* applications in Perl. It can be used as a library, and provides a rich API that is used to do many elaborated operations over ontologies. Another plus for these approaches, in opposition to the graphical tools, is that operations can be automatically created and executed without human intervention.

Manipulation Tools Summary

- Table 3.1 summarizes some of the features of previously described tools for easier reference.
- Regarding the editors that implement a graphical interface for manipulating information, with more or less features or operations, they are all much alike. They are suitable for humans to use. This is not exactly our aim since we plan on using our manipulation approach to build complex tools, and most of the manipulation operations should be decided in runtime.

[3]Available from: `http://jena.apache.org/` (Last accessed: 10-09-2014).
[4]Available from: `http://code.google.com/p/swoop/` (Last accessed: 10-09-2014).
[5]Available from: `http://thmanager.sourceforge.net/` (Last accessed: 10-09-2014).
[6]Available from: `http://sourceforge.net/projects/owlve/` (Last accessed: 10-09-2014).

- On the other end of the spectrum, there are tools that do not provide a graphical interface but offer APIs or libraries for executing operations. This approach allows for a wider range of possible applications, since more complex tools can be built using the provided interfaces. This approach suits our needs best, providing a complete module to manipulate information through a well defined API. This way is more flexible for writing more complex applications using this API from a high level layer.

Tool	Version	Description	Platforms	Formats
Protége	5.0	graphical editor	All	OWL+SKOS
Jena	2.12.0	Java classes	All (w/ Java)	-
SWOOP	2.3	graphical editor	All	OWL
ThManager	2.0	graphical editor	All (w/ Java)	SKOS/RDF
OWL VE	1.1.0	graphical editor	Linux/Source	OWL
Biblio::Thesaurus	0.43	API	All	ISO 2778++
SquishQL	-	SQL-ish language	-	RDF

Table 3.1: Summary of ontologies editing tools and libraries.

3.4 Ontologies and Software Engineering

The adoption of ontologies in the context of software engineering is being used for different purposes, and in different stages of software development and evolution [55]. For example, in requirements engineering [85], software modeling [75], model transformations [70], software maintenance [73], and software comprehension [163].

Ontologies provide rich semantics that can cope with the heterogeneity of information sources usually present during software development. They provide a unambiguous mediating mechanism to improve collaborating environments not only for software engineers, but also other stake holders in the development process [39].

Chapter 4

Information Retrieval

Information Retrieval (IR) [96] is a field of study concerned with finding resources (usually unstructured documents, e.g., text) from a large collection of resources, that satisfy an information need. One popular modern example of an IR application is a web search engine. The task of the this engine is to find the web pages from the Internet, that satisfy the query submitted by the user, i.e., find the relevant documents from a huge collection of documents that satisfy a search query.

This chapter briefly introduces some topics in the branch of IR addressed throughout this work. Some techniques in the this field are being used, or adapted, in the context of PC to find relevant elements (e.g. code, documents, reports) in a program, during, or to enhance, software analysis. This subject is also discussed in Chapter 2.

4.1 Precision and Recall

In general, the goal of an IR technique is to find the relevant documents in a collection of documents, for a given definition of relevant. For example, for a web search engine, the set of retrieved documents is the set of pages computed for a search query, and the relevant documents set is the set of web pages that actually address the topic described in the search query. Precision and recall are two measures usually used to evaluate the performance of some technique, and to compare performances of techniques that address the same problem. The remaining of this section defines precision and recall as used in the context of IR, which may differ from other branches of science.

Precision

Precision measures the fraction of retrieved documents that are relevant and is calculated using the following formula:

$$P = \frac{|d_{relevant} \cap d_{retrieved}|}{|d_{retrieved}|}$$

Where, $d_{relevant}$ is the set of retrieved documents considered relevant, $d_{retrieved}$ is the set of retrieved documents, and $|x|$ means the cardinality of x. Precision P is calculated as the cardinality of the intersection between the relevant retrieved documents set, and the retrieved documents set, normalized by the cardinality of the retrieved documents set.

Recall

Recall measures the fraction of the relevant documents query that are successfully retrieved, and is calculated using the following formula:

$$R = \frac{|d_{relevant} \cap d_{retrieved}|}{|d_{relevant}|}$$

Where, $d_{relevant}$ is the set of retrieved documents considered relevant, $d_{retrieved}$ is the set of retrieved documents, and $|x|$ means the cardinality of x. Recall R is calculated as the cardinality of the intersection between the relevant retrieved documents set, and the retrieved documents set, normalized by the cardinality of the relevant documents set.

F-Measure

Precision and recall are often combined to produce measures about some technique. These can be analyzed independently or combined into a single value. A common measure that combines precision and recall values is the $f\text{-}measure$ (also known as F_1 measure). This measure represents the weighted harmonic mean between precision and recall, and is calculated using the following formula:

$$F = \frac{2 \cdot P \cdot R}{(P + R)}$$

Where, the $f\text{-}measure$ F is the result of multiplying the precision P, the recall R, and 2, normalized by the sum of precision P and recall R.

4.2 LSI

Latent Semantic Indexing (LSI) [38, 77] is an indexing and retrieval statistical technique for analyzing relationships between queries and unstructured collections of text using a term by document frequency matrix. Each column in the matrix represents a document (a text), or a query, and each row in the matrix stands for a unique word. The cells of the matrix contain the frequency with which the term is found in the corresponding text.

A mathematical technique called Singular Value Decomposition (SVD) is used to create a vector representation of the documents and the query by normalizing and decomposing the matrix. The similarity between a document and a query is typically measured by the co-sine between their corresponding vectors.

LSI has been used in the context of SE, for example to cluster source code elements to support program understanding [95], or to map feature descriptions (expressed in natural language) to source code [99].

4.3 tf-idf

Term Frequency - Inverse Document Frequency (tf-idf) is a statistical measure of how relevant a term is to a specific document in a collection of texts (*corpus*). It is commonly used by IR techniques as a weighting factor.

The frequency of a term $tf(t, d)$ counts the number of times term t occurs in document d. The *inverse document frequency* for a given term t in a collection of documents D, $idf(t, D)$, is calculated as:

$$idf(t, D) = \log \frac{|D|}{|\{d \in D : t \in d\}|}$$

Where, $|D|$ is the total number of documents in the collection D, and $|\{d \in D : t \in d\}|$ is the number of documents where the term t occurs in the collection D. The Term Frequency - Inverse Document Frequency (tf-idf) of a term t, in document d, in a collection of documents D , tf-idf(t, d, D), is then calculated as:

$$\text{tf-idf}(t, d, D) = tf(t, d) \times idf(t, D)$$

Chapter 5

Natural Language Processing

Natural Language Processing (NLP) is a field of computer science mainly concerned with the interaction between computers and human languages [100]. In practice, one common example of these interactions is a machine trying to understand text written in a natural language. This process may imply a narrower definition of understanding, e.g., extracting some specific information, or extracting arbitrary facts. These processes often imply an algorithm that applies a sequence of well defined operations over a text, where the output of one operation is handed to next operation, and the combined set of operations entails the final result. Some of these operations can be simple and straightforward, others can be more complex.

An example of a common operation is tokenization, which refers to the process of dividing a text into its constituent tokens. The most common token size is words, but other sizes can be used (e.g., letters, sentences). At first glance it seems a trivial operation, splitting text into words, but in real world applications several subtleties can produce some unexpected results or misbehaviors. For example, different handling of punctuation tokens, changing text language, or how to handle words composed using an hyphen (e.g., “*user-generated*” can be a single word, or the composition of two words).

The remaining sections of this chapter briefly introduce some topics or common operations that are closely related, or actual steps, with the algorithms introduced in upcoming chapters.

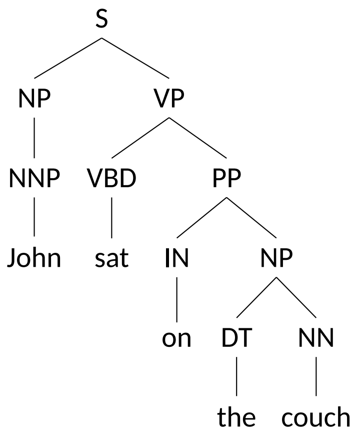


Figure 5.1: Parsing tree example.

5.1 Parsing

Parsing is the process of syntactically analyzing a sequence of symbols. In NLP, the goal of a parser is to derive the grammatical structure of a sentence, and which groups of words are part of each phrase, usually this structure is represented using a parsing tree. Most of the parsers expect a tokenized version of the text.

Figure 5.1 illustrates a parsing tree example, for the sentence: "*John sat on the couch*". The NP acronym stands for a noun phrase, and VP for a verb phrase. Due to natural languages grammars often having ambiguous rules, some sentences may have more than one possible parsing tree.

5.2 Lemmatization

Lemmatization refers to the process of finding the *lemma* of a word. The lemma of the word, also refereed to as *dictionary form*, is the base form of the word. For example, the words: "*walks*", "*walking*", "*walked*", all have the same lemma (base form) "*walk*". The main idea is to be able to group together different forms of a word so that they can be analyzed as the the same word. This is particular relevant in, for example, IR techniques. The total count of occurrences of all the possible tenses and inflections of the verb "*walk*" is more important than the separate count of the various possible

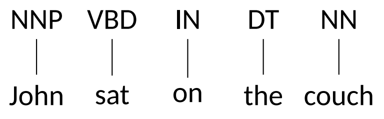


Figure 5.2: POS tagging example.

words. This also enables to find a match of a the query string “*reading*”, in a text that contains “*reads*” or “*read*”. Lemmatization is a context-aware process, meaning that the context of the word, i.e., the phrase where the word is used, has a weight while choosing the word lemma.

5.3 Part-of-Speech Tagging

Part-of-speech tagging (or POS tagging) corresponds to the operation that marks words in a sentence with a particular part-of-speech (commonly designated as tags). This allows to figure out if a specific form of a word is acting as a verb or as a noun, since many words have different meanings according to context. Although, this operation in strictly related with how the sentence is parsed, and how the words are grouped in phrases, its still possible to tag single words, outside the scope of a complete sentence.

Figure 5.2 emphasizes how the words in the sentence used in the parsing example illustrated in Figure 5.1 are tagged. The specific set of tags used in this example, tags the word “*John*” as a singular proper noun (NNP), “*sat*” as a verb in the past tense (VBD), and so on.

5.4 Parallel Corpora

Informally, a corpus is a collection of texts, usually representative of a given domain or subject. These constructs are used to build other common linguistic artifacts in the field of NLP [59, 72, 100]. Corpora is a term to refer a collection of texts, usually in a specific language.

Parallel corpora is a collection of texts in different languages where one of them is the original text and the other are translations [146]. Using alignment tools, the parallelism between units of text in a parallel corpora is increased. Usually, aligners perform the alignment at sentence or word level.

Parallel corpora are commonly used in teaching activities, terminological studies, automatic translation, cross language information retrieval engines. They are used as an input to produce other useful artifacts, an example is described in the next section.

5.5 Probabilistic Translation Dictionaries

A Probabilistic Translation Dictionary (PTD) [147] is a translation dictionary between two different languages. A PTD is built using a statistical method on a parallel corpora. The following example illustrates the PTD entry for the portuguese word "*codificada*", extracted from the EuroParl[1] parallel corpora:

$$\mathcal{T}\left(\text{codificada}\right) = \begin{cases} \text{codified} & 62.83\% \\ \text{uncoded} & 13.16\% \\ \text{coded} & 6.47\% \\ \ldots \end{cases}$$

This example should be read as: in the EuroParl corpus, the portuguese word "*codificada*" is highly co-related with the english words "*codified*", "*uncoded*", and "*coded*". The percentage value measures the degree of certainty for the actual translation.

The dictionaries are extracted automatically from parallel corpora, meaning that the vocabulary present in the dictionary is, up to a certain level, controlled by the text available in the corpora. This allows the creation of dictionaries (commonly not available) for specific domains. These dictionaries are also used to build another natural language artifacts.

[1]Parallel corpus built from the proceedings of the European Parliament.

5.6 Semantic Relatedness

Semantic relatedness is a metric used to measure the conceptual distance between terms, sentences or documents. This helps measuring how close a set of textual elements are referring the same concept or idea. For example, the semantic relatedness of the word "*car*" and "*vehicle*" by a given measure, is expected to be higher (assuming that the higher the relatedness score, the higher the terms are conceptually related), than for the terms "*vehicle*" and "*carrot*". Because, in probably most of the contexts the words "*car*" and "*vehicle*" can be used to refer to the same concept (object) – a means of transportation – they are closer at a semantic level, than the words "*vehicle*" and "*carrot*".

This process is often used in tasks of word disambiguation, plagiarism detection, summary creations, and in the design of query-answering systems [60].

5.7 Tools and Frameworks

This section briefly introduces some popular tools and frameworks currently available to aid the implementation of NLP features.

Jspell is a morphological analyzer, that can be used as a library in C or Perl, and supports several languages[2].

FreeLing is suite of tools for language analysis, it provides many common tasks (e.g., parsing, named entity recognition), and supports various languages (e.g., english, spanish, german)[3].

NLTK is a NLP toolkit for the Python programming language. It provides common tasks as stemming, tagging, or parsing, amongst others [4].

[2]Available from: `http://search.cpan.org/dist/Lingua-Jspell/` (Last accessed: 14-09-2014).
[3]Available from: `http://nlp.lsi.upc.edu/freeling/` (Last accessed: 14-09-2014).
[4]Available from: `http://www.nltk.org/` (Last accessed: 14-09-2014).

OpenNLP is machine learning based toolkit for processing natural language text implemented in Java. It supports most common NLP tasks (e.g., tokenization, segmentation, part-of-speech tagging)[5].

These tools are to be used as building blocks to ease the development of real world applications. The programmer, uses a combination of the provided operations, to weave an algorithm without spending time implementing well known tasks (e.g. tokenization, parsing, lemmatization).

5.8 NLP and Software Engineering

Feature location techniques often explore natural language text found in software packages to perform arbitrary analysis. These involve processing, for example, program identifiers, comments, or documentation, using NLP techniques. Similar techniques are also used to process queries devised by maintainers when analyzing software. Chapter 2 introduces some techniques that explore these approaches.

[5]Available from: `https://opennlp.apache.org/` (Last accessed: 14-09-2014).

Part III

Ontology-based Concept Location

Chapter 6

Domain Knowledge Representation

Knowledge is power.

Sir Francis Bacon

The goal of this chapter, is to describe the approach used to model required knowledge about the object of analysis for reasoning, and, which design goals were behind major decisions. The typical object of analysis during program understanding activities is a software system, this means that the source code, and other files that may be included in the software package (e.g., documentation, tests, examples), convey knowledge required to create useful models. Also, some other external sources can contribute with information, for example, information about the application domain, or area of interest, can be found elsewhere outside the scope of the package. The knowledge concerning a software package is devised by domain; important domains identified by the literature in the context of PC, are introduced in Section 2.4 (e.g., the problem domain, the program domain). Therefore, at least these two are (usually) available, and are discussed in detail during this chapter. However, the method introduced is valid for any domain, even outside the context of PC.

The first challenge when devising a model of some sub-set of the real world, is to choose a formalism to represent the information. In the context of this work, ontologies are adopted for knowledge representation mainly because of the following reasons:

- Ontologies can represent knowledge in any domain, they do not enforce any constrain on the kind of information or application domain, they can be regarded as having an universal type.
- Reasoning engines and algorithms are available to infer new information in a ontology setting. This allows to infer new information, and validate constrains for the already available data, for the created models.
- Technologies for querying, storing, sharing, and managing ontologies are widely available, providing a set of tools and applications that can be explored, and used to enhance current features, or build new applications.

These also allow that knowledge about distinct domains can be captured in different ontologies, but information can be linked between different ontologies, in order to devise bridges between domains.

The goal is to have a generalized way for building an ontology (a model) for a given domain, this can be abstracted by the signature of the following function, named *model*:

$$\mathit{model} :: \mathit{Domain} \ \rightarrow \ \mathit{Ontology}$$

The *model* function, given a domain, builds a model of the domain using an ontology. The *Domain* type is defined as a collection of artifacts (of heterogeneous types) that convey knowledge about the given domain (e.g., a program source code, software documentation, initial specification, taxonomies of concepts). The *Domain* type is defined as a set of artifacts of any given type, this heterogeneity is captured by the *Any* type:

$$\begin{aligned}&\textbf{data}\ \mathit{Domain} = \mathit{Domain}\ \{\ \mathit{artifacts} \ :: \ [\ \mathit{Any}\ ]\ \}\\ &\textbf{data}\ \mathit{Any} \quad\ \ = \mathit{Text\ String} \mid \mathit{Source\ String} \mid \mathit{Doc\ String} \mid \ldots\end{aligned}$$

For now an ontology is simply defined as a set of triples, where a triple is used to represent a relation between two elements.

$$\textbf{data}\ \mathit{Ontology} = \mathit{Ontology}\ \{\ \mathit{triples} \ :: \ [\ \mathit{Triple}\ ]\ \}$$

$$\textbf{data}\ \mathit{Triple} \quad = \mathit{Triple}\ \{\ \mathit{subject} :: \mathit{String},\ \mathit{relation} :: \mathit{String},\ \mathit{range} :: \mathit{String}\ \}$$

A more accurate and precise definition of the *Ontology* data type actually used throughout this work is introduced and discussed in Section 6.4.1.

In order to simplify the process of creating the domain ontology, and to improve its generalization, function *model* is decomposed in two functions: *process* and *convey*, with the following signatures:

$$\mathit{process} :: \mathit{Any} \rightarrow \mathit{Resource}$$

that, given an element of the domain (of any type), builds an arbitrary resource of information, and:

$$\mathit{convey} :: \mathit{Resource} \rightarrow [\ \mathit{Triple}\ ]$$

that, given an arbitrary source of information (a resource), conveys the required information to a set of triples. Given *process* and *convey*, the function *model* is defined as:

$$\mathit{model}\ d \ = \ \mathit{Ontology}\ (\ \textbf{concat}\ [\ \mathit{convey}\ (\mathit{process}\ a_i) \mid a_i \leftarrow \mathit{artifacts}\ d\ ]\ )$$

where the ontology that represents the domain, is created by composing together the sets of triples computed by applying *convey* to the resources created by applying *process* to the set of available domain artifacts. Most of the times, the *process* and *convey* functions are to be regarded as *dispatch functions*, i.e., that according to the given artifact type call the correct set of tools to process it, and according to the resulting resource type call the correct functions to create the triples sets.

For example, given a program source code file (an artifact of the program domain), the *process* function can call a tool that builds the table of identifiers for the program (produced resource), and *convey* calls a specific function to convey the information from a table of identifiers (the resource) to the ontology (set of triples).

This means that any arbitrary tool, from any particular discipline, can be used to create a resource that captures information from the domain, being only necessary to define the rules that convey the knowledge captured in the resource to the ontology. The domain ontology is then the result of composing together all the triple sets resulting from executing all the intended tools on the corresponding artifacts from the domain.

These functions definitions and composition captures the method for building the ontologies (the models) for each domain. Having a set of tools, that produce a collection of resources, to create the ontology it is only required to define how the information available in these resources is conveyed to the final domain ontology. The next sections describe in detail how this process is defined for two important domains of interest in the context of PC: the program (source code and related artifacts), and problem domain (real world knowledge about the application domain).

6.1 The Program Domain

The *program domain* (introduced in Section 2.4) captures knowledge in the scope of the program itself. The main source of information about a program is the source code, which precisely describes all the concepts and operations from the application domain. Source code is written in a specific formal language, a programming language, usually following a programming paradigm (e.g., object oriented, imperative, functional). Modern complex systems often use a heterogenous combination of programming languages and paradigms to implement different sub-systems. Using one or more programming languages the underlying intuition is the same: source code is written using a formal language and a strict grammar, which makes the sentences that compose a program follow a well defined set of rules. This well known set of rules allows tools to extract information about a program with a high degree of confidence, in the same sense a compiler needs to unambiguously interpret code.

To model the program domain, at least an abstract representation of (some) statements in the source code is required. The intended level of abstraction is high enough to hide details about the code that do not contribute to the calculations performed later, but still keeps the required information. This means that not every basic element in the source code is conveyed to the program abstract representation. Only elements that are prone to contribute for later calculations are transported from the source code.

Besides the challenge of choosing which elements are represented, there is also the heterogeneity of programming languages and paradigms currently available.

To simplify the creation of the source code abstraction, only a small set of program elements are elected to be represented. All the selected elements have parallel concepts in most programming languages. If in some abstract representation the programming languages are vertical lines in a matrix, and these concepts are represented in horizontal lines, it is expected that for most programming language, the two lines cross, i.e., the programming language implements some elements that are related to these concepts. The following concepts are analyzed and represented in the program domain ontology:

Files are mostly used for organization purposes. It is important to keep at least some information about the original structure of the program, so that when a view is created of some program element, its counterpart in the original source code can be directly pinpointed. This is the same reason why, for most elements, the line number is also stored.

Classes, modules or objects are common concepts in programming languages, as a way to organize the program, source code and functionalities inside these are usually related with the same concepts from the application domain.

Functions or methods in the sense of a way to encapsulate a set of instructions, to be executed at any given time. This means that it can be used to represent a function (or sub-routine) in imperative languages, or methods in OO programming, etc.

Variables encompass all the ways of defining data representations, including global and local variables, function parameters, etc., according to the programming language.

Identifiers are used in most programming languages to assign labels to elements so that they can be referenced elsewhere (e.g, function and variable names).

Function or method calls as a simple way to store some information about the possible execution flows of the program, this kind of information can be more or less complete according to the information available. For example, execution traces

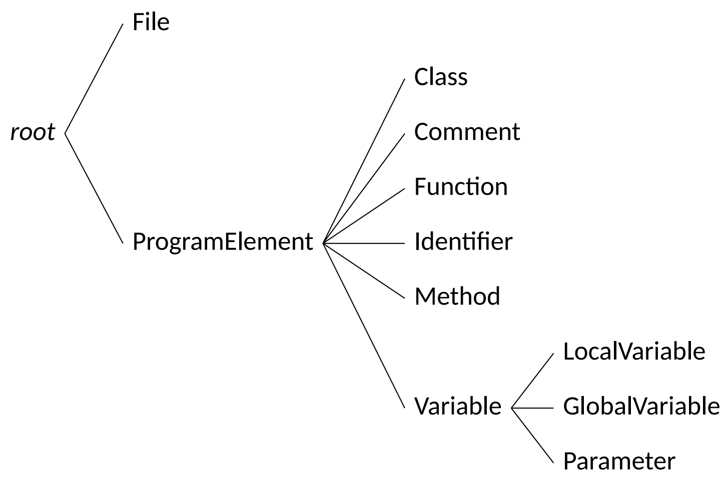


Figure 6.1: Class hierarchy of concepts transported from the source code to the program model.

of the program can provide exact function calls for given execution scenarios, while static analysis can only provide information about possible graphs of execution.

These elements convey knowledge information that can be later used to perform calculations in the program domain, and when searching for connections between domains. This elements set is used to define a hierarchy of classes (illustrated in Figure 6.1), used to organize elements extracted from the source code. This is not the only possible class hierarchy, other definitions are possible, and the only details that this particular definitions establishes (opposed to other definition) are the class names and corresponding sub-classes. This means that changing the set of classes is possible at anytime, and other tools can cope with class hierarchy updates.

Thus, the process of populating the program domain ontology simply implies classifying the extracted elements (according to the specified class hierarchy) and creating the required triples to convey the required information. Following the strategy defined in the previous section modeling the program domain is defined as:

$$\begin{aligned} &\mathit{model}_P \quad :: \mathit{Domain}_P \to \mathit{Ontology}_P \\ &\mathit{model}_P\ d_P = \textbf{concat}\ [\,\mathit{convey}\ (\mathit{process}\ s_i) \mid s_i \leftarrow \mathit{sources}\,] \end{aligned}$$

where
$$\textit{sources} = [\, e \mid e@(\textit{Source}\ \_) \leftarrow d_P]$$

To create the program model (the program ontology), every artifact in the program domain (for now only the source code is analyzed, so corresponds to the set of source files) is processed by the *process* function to create a resource for each source file that captures the intended information, and this resource is processed by the *convey* function to compute the corresponding set of triples. The information that is extracted, and how this information is conveyed to a set of triples, defines how the knowledge in the ontology reflects the source code. Different implementations of these functions can be used for different analysis, the next sections describe in more detail the implementation of these functions in the context of this work.

The *process* Function

The goal of this function is, given a source code file, extract the information required for further concept location analysis. The class hierarchy definition gives a set of elements of interest that can be found in the program. The signature of the function *process* is the same as before:

$$\textit{process} :: \textit{Any} \rightarrow \textit{Resource}$$

In the particular case of the program domain, all the arguments passed to this function are of type *Source String* (source code file content). The set of relevant information is defined by the class hierarchy illustrated in Figure 6.1, and by the set of concepts of interest described earlier in this section. In practice, the function that does the source code analysis is concerned with capturing elements related with these concepts: variable declarations, function definitions and calls, etc. The output of this function is a *Resource*, that represents arbitrary data, stored in a format that is easily processed (e.g, CSV, XML) or in plain text, simple to parse. Storing the resource information in this way, allows for other tools to process the resource data and convey information to the program representation, according to the program ontology semantics.

As an example, Program 6.1 illustrates a possible definition for a function to compute the factorial of a number in C. And Table 6.1 summarizes the computed resource

Id	Type	Identifier	Context	File	From	To
1	Function	*factorial*	-	fact.t	1	8
2	Parameter	*n*	1	fact.t	1	1
3	LocalVariable	*result*	1	fact.t	2	2
4	hasFunctionCall	*factorial*	1	fact.t	6	6

Table 6.1: Resulting resource of processing Program 6.1 with the C source analyzing tool.

that captures the intended information from Program 6.1. This information includes the function definition, the parameter used by the function, the local variable used inside the function, and the function call inside the function. For every captured element the identifier string, file and line where the element begins and ends is recorded. Once this resource is available, the next step is to convey the information to sets of triples that are used later to populate the ontology. This step is carried out by the *convey* function, discussed in the next section.

```
1 int factorial(int n) {
2   int result;
3   if (n == 0)
4     result = 1;
5   else
6     result = n * factorial(n-1);
7   return result;
8 }
```

Program 6.1: Function to compute the factorial of a number recursively, written in C.

The *convey* Function

The goal of this function is, given a resource file (created by the *process* function), compute a set of triples that convey the intended information, computed by an arbitrary tool, to the ontology. Table 6.2 describes the full sets of triples computed for every element in the resource created by the *process* function illustrated in Table 6.1. This step conveys the information from the resource to the semantics of the ontology. The first

Id	Triple Set
1	*Triple ”uid_1” ”hasClass” ”Function”* *Triple ”uid_1” ”hasFile” ”fact.c”* *Triple ”uid_1” ”hasLineBegin” ”1”* *Triple ”uid_1” ”hasLineEnd” ”10”* *Triple ”uid_1” ”hasIdentifier” ”uid_2”* *Triple ”uid_2” ”hasClass” ”Identifier”* *Triple ”uid_2” ”hasString” ”factorial”*
2	*Triple ”uid_3” ”hasClass” ”Parameter”* *Triple ”uid_3” ”hasFile” ”fact.c”* *Triple ”uid_3” ”hasLineBegin” ”1”* *Triple ”uid_3” ”hasLineEnd” ”1”* *Triple ”uid_3” ”hasIdentifier” ”uid_4”* *Triple ”uid_4” ”hasClass” ”Identifier”* *Triple ”uid_4” ”hasString” ”n”*
3	*Triple ”uid_5” ”hasClass” ”LocalVariable”* *Triple ”uid_5” ”hasFile” ”fact.c”* *Triple ”uid_5” ”hasLineBegin” ”2”* *Triple ”uid_5” ”hasLineEnd” ”2”* *Triple ”uid_5” ”hasIdentifier” ”uid_6”* *Triple ”uid_6” ”hasClass” ”Identifier”* *Triple ”uid_6” ”hasString” ”result”*
4	*Triple ”uid_1” ”hasFunctionCall” ”uid_1”*

Table 6.2: Resulting resource of processing Program 6.1 with the C source analyzing tool.

detail to note is the presence of the strings *uid_n* in the triples, these represent unique identifiers that unambiguously identify every element available in the resource. In the example, *uid*_1 represents the function definition, which is related with an instance of identifier (the function has a name – *factorial*) which is described as an instances of the class *Identifier*, with a data propriety that stores the actual name of the function.

The use of the unique identifiers created for every element is just to prevent problems when materializing the ontology to well known formats (e.g. OWL) that have rules to enforce the situation where each individual (instance) has a unique name. Although these set of triples look confusing, they are created in a systematic way, i.e., it is possible to define a pattern of relations that are created for each type of element present in the resource. Once the patterns are defined they can be implemented in a simple template and instantiated according to the element information. Table 6.3 illustrates a possible template that could have been used to create the triple set for *uid*_1 in Table 6.2. The problem of converting the entire resource is simply choosing the correct template for the current element. This systematic approach is used successfully in some tools described in Section 6.4.

In order to build the set of templates for every pattern, the set of classes and pos-

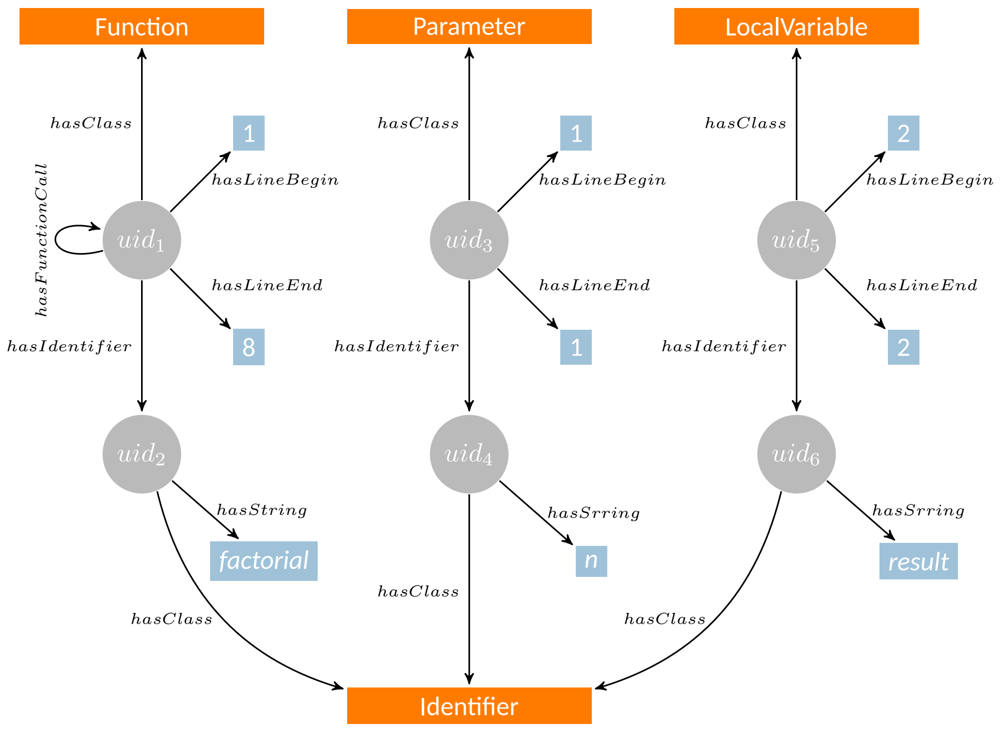


Figure 6.2: Graph representing the knowledge described by the triples described in Table 6.2.

Template
add element
Triple "[% **id** %]" "hasClass" "[% **type** %]"
Triple "[% **id** %]" "hasFile" "[% **file** %]"
Triple "[% **id** %]" "hasLineBegin" "[% **from** %]"
Triple "[% **id** %]" "hasLineEnd" "[% **to** %]"
Triple "[% **id** %]" "hasIdentifier" "[% **iid** %]"
add element identifier
Triple "[% **iid** %]" "hasString" "Identifier"
Triple "[% **iid** %]" "hasclass" "[% **identifier** %]"

Table 6.3: Possible definition of a template to create sets of triples for function definitions, the variable elements (inside **[%** and **%]**) are instantiated with information from the resource.

sible relations between instances needs to be well defined. The next section describes in more detail the sets of classes and proprieties used to implement the systems described in Section 6.4. This is not a closed and finished set, more classes and relations can be added when required, and the remaining tools in the workflow cope with such updates.

6.1.1 Classes and Instances

This section describes the set of concrete classes available in the program ontology. The top classes are *File* and *ProgramElement*. Every program element of interest that needs to be conveyed to the abstract representation is an instance (member) of a sub-class of *ProgramElement*. The set of available classes should be enough to capture the most relevant information, but other classes can be added if required. Table 6.4 enumerates the available sub-classes of *ProgramElement*, and describes which program elements should be instances of each class.

6.1.2 Data and Object Proprieties

After the elements are added to the ontology, as instances of the classes described in the previous section, proprieties can be defined for each element (relations with other elements, or arbitrary data). There are two main types of proprieties (relations): (i) a propriety that relates two instances, and (ii) a propriety that relates an instance and arbitrary information of a given type (e.g., string, integer). An example of the first is the relation *inFunction*, between a function and a local variable, describing that the variable is declared inside the function, this is a relation between two elements. An example of the second type of relation is the *hasName* propriety, which is a relation between an instance (element) and arbitrary data, a string in this particular case. Table 6.5 describes the possible object proprieties to describe relations between elements, and Table 6.6 describes the available data proprieties to describe relations between objects and arbitrary data. Again, more relations of both types can be added when required.

Ontology Class	Description
Class	The goal of this class is to capture collections of variables and function closely related (e.g., a *Class* in Java).
Function	Capture a set of statements wrapped in a single executable program unit (e.g., function in C, a procedure in Pascal).
Identifier	Captures labels in the source code, names of references to program elements (e.g., variable names, function names).
Method	Similar to the *Function* class but intended to be used in OO programming paradigms (e.g., method definitions in Java).
Variable	This class is used to represent a variable in a program. The more specific next three classes should be used when possible, this class should only be used if none of the more specific next three classes applies.
LocalVariable	Used to represent variables only defined in a given scope (e.g., C variable declared inside a function).
GlobalVariable	Used to represent a global variable, that exists throughout the entire program (e.g., global variable in C).
Parameter	A variable passed as an argument to a function call (e.g. parameters passed to a method call in Java).

Table 6.4: Available classes in the program ontology for instantiating program elements.

Propriety	Description	Range Class
hasFunctionCall	Describes static function calls.	*Function*
hasIdentifier	Relates a program element with its identifier.	*Identifier*
inFile	Describes in which file the element is defined.	*File*
inFunction	Describes elements that are defined inside a function scope (e.g. local variables).	*Function*

Table 6.5: Available object proprieties for program elements (relations between instances), and corresponding range classes.

Propriety	Description	Data Type
hasName	Describes the name of the element.	*String*
hasLineBegin	Describes in which line the element begins.	*Int*
hasLineEnd	Describes in which line the element ends.	*Int*
hasSplits	Describes the splits of an *Identifier* instance.	*String*
hasTerms	Describes the set of terms of an *Identifier* instance.	*String*

Table 6.6: Available data proprieties for program elements (relations between instances and data of an arbitrary type), and corresponding data types.

6.2 The Problem Domain

The *problem*, or *application*, *domain* (introduced in Section 2.4), encompasses the real world knowledge about the area of interest where the software is acting, usually by describing relevant concepts and the relations between these concepts. This domain is usually described and discussed using natural languages, opposed to the main source of information about the program domain (the source code) which is written using formal languages (programming languages).

The goal of this ontology is mainly to capture concepts used in the application, and sets of relations between these concepts. There is no set of classes and relations defined at the outset for this ontology. The ontology creator is free to devise the best approach to model the domain.

The creation of this ontology is not one of the goals of this work, it is assumed that an application domain ontology is available. However, in order to have problem ontologies for the analyzed case studies, a simple strategy was devised to automatically bootstrap a problem ontology for a software system. The generated ontologies satisfy the following invariants:

- A domain concept is represented using a class.
- Every class that represents a concept, is an instance of the class *Concept*, or instance of a sub-class of *Concept*.

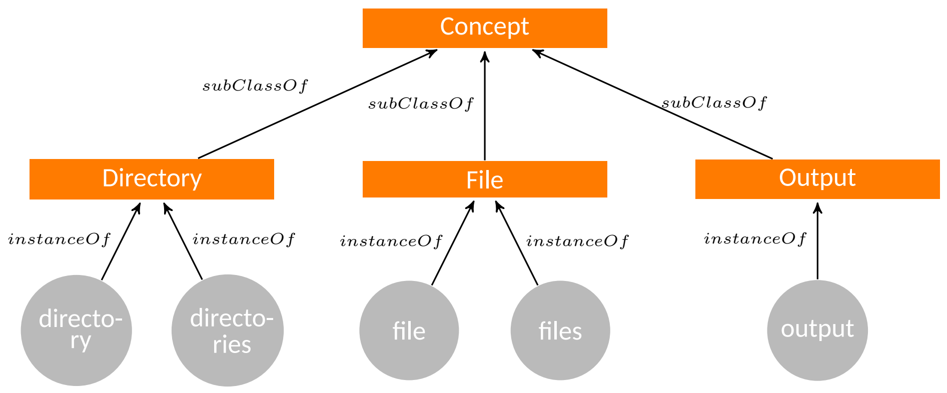


Figure 6.3: Graph representing a sub-set of the problem ontology automatically generated for the *tree* package.

- The linguistic representation of each concept, i.e., terms used to refer to the concept, are instances of the class representing the concept.

Figure 6.3 illustrates an example of a program ontology created automatically from the software system. The orange boxes represent concepts in the application domain, and grey circles are instances of the different classes, representing possible textual representations of the concept. This reads as *Directory* is a concept in this domain, and can be described using the terms *directory* or *directories*, and in a similar fashion for the *File* and *Output* concepts. All the classes representing concepts are sub-classes of the class *Concept*. The graph represents only a sub-set of the knowledge described in the complete ontology.

These artifacts capture relevant concepts about the application domain, and allow the realization of an application model. Although, ontologies manually crafted by experts provide more accurate knowledge, they are not always available, or are expensive to build. Hence, this approach provides a way to bootstrap an ontology to be directly used, which can be enhanced at a later time.

6.3 Query Domains

Previous sections address the problem of capturing information about a domain in an ontology, creating a model of the domain. Another relevant issue, is after the data is organized and stored, how can we query the model for information. Querying ontologies is straight-forward, in the sense that, there are clear and well defined languages, with different levels of expressiveness, for devising efficient expressions to retrieve information stored in an ontology.

Querying the ontology (the domain model) implies defining questions about the domain, and retrieving information, with well defined semantics, to answer these question. This approach allows to obtain information about the domain in a efficient and accurate way. Also, ontologies are easier to query than, for example, the program source code. Lacking a query language that targets source code, makes performing specific queries a complex task.

One of the major advantages of a common knowledge representation for different domains is that the same technology can be used to formulate queries that cross domains boundaries. This means that, using the same language for describing the query, or even in the same query, it is possible to query different domains of interest. For example, get the list of program identifiers (from the program domain), and also the list of concepts from the application domain.

The next section introduces the Ontology Toolkit that provides a set of methods to populate and query domain ontologies.

6.4 Conclave OTK – The Ontology Toolkit

This section describes the Ontology Toolkit (OTK), a toolkit developed in the context of this work, that provides an abstraction layer on top of ontology related technologies, to develop ontology-*aware* applications using the approach described in previous sections of this chapter. OTK is implemented as a library (a set of modules) for the Perl programming language. Outside the scope of this work, OTK can be used in any application that has to deal with ontologies. In the scope of this work, OTK is used to develop the applications that implement the methods and techniques described throughout this document, e.g., how to model (build an ontology) for a given domain of knowledge.

In practice, when applications developers want to perform an ontology related operation, instead of using a specific format or underlying technology (e.g., triple-store low level primitives), they can use the abstraction layer. To motivate for the development of this abstract framework, consider the modern Object-Relational Mappers (ORM) frameworks in the context of relational databases. Which provide an abstraction layer and interface for programming languages to handle data (stored in databases) as objects, allowing the development of applications regardless of the underlying database technology used. Also developers use object methods to perform operations on the data, instead of writing SQL statements. This allows writing more optimized and more portable code. More optimized because the SQL queries actually executed are generally well optimized by the ORM. More portable because the framework deals with different details that may exist between the actual underlying database technology used, and the system where the program is running. These are some of the advantages that OTK also features.

6.4.1 Formal Definitions

Given the previous discussion, and elements of interest in the domain models, the devised ontology formal definition follows. This definition refines the simple set of triples approach introduced in the beginning of this chapter to discuss the model building approach.

```
data Ontology = Ontology {
                    classes   :: [ Class ],
                    instances :: [ Instance ],
                    objProps  :: [ ObjProp ],
                    dataProps :: [ DataProp ]
                }
```

Where, an ontology is defined by describing its set of classes, its set of instances, the set of relations between elements (classes or instances) using object proprieties, and the set of relations between elements and arbitrary data using data proprieties (e.g., strings and integers). The *Class* type is a string, that stores the name of the class. The type of an *Instance* is also a string, that stores the name of the instance, or a more complete unique identifier for the instance (e.g., URI).

$$\begin{aligned}&\textbf{type}\ \mathit{Class} &&= \mathit{String}\\ &\textbf{type}\ \mathit{Instance} &&= \mathit{String}\end{aligned}$$

An object propriety, a relation between two elements of the ontology (e.g., two classes, two instances, a class and an instance), is defined using a triple: (i) the source element of the relation, (ii) the relation, and (iii) the range (or target element) of the relation:

$$\begin{aligned}&\textbf{data}\ \mathit{ObjProp} = \mathit{ObjProp}\ \{\\ &\qquad \mathit{source} &&:: \mathit{Element},\\ &\qquad \mathit{relation} &&:: \mathit{ObjRelation},\\ &\qquad \mathit{target} &&:: \mathit{Element}\\ &\quad \}\end{aligned}$$

A data propriety, a relation between an element and arbitrary data follows a similar approach, it is defined using a triple: (i) the source element, the element that is related, (ii) the relation, and (iii) the target data, including the data type:

$$\begin{aligned}&\textbf{data}\ \mathit{DataProp} = \mathit{DataProp}\ \{\\ &\qquad \mathit{source} &&:: \mathit{Element},\\ &\qquad \mathit{relation} &&:: \mathit{DataRelation},\\ &\qquad \mathit{target} &&:: \mathit{Data}\\ &\quad \}\end{aligned}$$

A set of predefined object proprieties is available. But this is not a close set, and relations can be added when required. Relations for describing class hierarchies (*subClassOf*), or classes instances (*instanceOf*) are available:

$$\textbf{type}\ \mathit{ObjRelation} = \mathit{subClassOf} \mid \mathit{instanceOf} \mid \ldots$$

A similar situation takes place with data relations, although, Data relations are more closely related with the details of the domain at hand. For example, for the program domain, there are a set of available relations to store information about the path of the file, or at which start the element begins and ends. But, again, the set of relations can be enriched to provide new semantics whenever required.

$$\textbf{type}\ \mathit{DataRelation} = \mathit{hasFullPath} \mid \mathit{hasLineBegin} \mid \mathit{hasLineEnd} \mid \ldots$$

Data relations are usually typed. This type is defined as alternation of all possible available types of data. Usual types are strings, and numbers, but more complex data types can be used.

$$\textbf{data}\ \mathit{Data} = \mathit{S\ String} \mid \mathit{I\ Int} \mid \ldots$$

To illustrate these definitions a brief example of a simple ontology follows.

For example, family trees are often available in ontological formats, to represent individuals and their family relationships. The following classes could be available to capture gender:

$$\mathit{classes} = [\,\text{"}\mathit{Female}\text{"},\ \text{"}\mathit{Male}\text{"}\,]$$

Each individual is represented as one instance. For example:

$$\mathit{instances} = [\,\text{"}\mathit{Ann}\text{"},\ \text{"}\mathit{Peter}\text{"}\,]$$

Individuals gender is captured by relating each instance with the corresponding class, and family relationships using object proprieties:

$$\begin{aligned}\mathit{objprops} = [\,&\mathit{ObjProp}\ \text{"}\mathit{Ann}\text{"}\quad \mathit{instanceOf}\ \text{"}\mathit{Female}\text{"},\\ &\mathit{ObjProp}\ \text{"}\mathit{Peter}\text{"}\ \mathit{instanceOf}\ \text{"}\mathit{Male}\text{"},\\ &\mathit{ObjProp}\ \text{"}\mathit{Ann}\text{"}\quad \mathit{hasParent}\ \ \text{"}\mathit{Peter}\text{"}\,]\end{aligned}$$

Extra information can be included, for example individuals age, using data proprieties:

$$\begin{aligned}\mathit{dataprops} = [\,&\mathit{DataProp}\ \text{"}\mathit{Ann}\text{"}\quad \mathit{hasAge}\ (I\ 4),\\ &\mathit{DataProp}\ \text{"}\mathit{Peter}\text{"}\ \mathit{hasAge}\ (I\ 28)\,]\end{aligned}$$

The final ontology is defined as:

$$onto = Ontology\ classes\ instances\ objprops\ dataprops$$

6.4.2 Operations and Information Handling

In order to provide an useful toolkit for building ontology-*aware* applications, the ontology types discussed in the previous section are not enough, a set of operations to handle (query and update) information is also required (e.g., creation of classes and instances, description of proprieties). In sum, operations for populating ontologies, and retrieving data, are required. The remaining of this section describes some common operations available in Conclave OTK.

Section 6.1 discusses how data about a resource is translated into a specific format using templates, i.e., how to transport the information from an arbitrary resource to an ontology. Although, this approach works well, it has major drawbacks. The templates produce sets of arbitrary triples, but in order to build real world scalable applications, a way to manage, store and share these triples is required. To overcome this drawback, instead of using templates to generate simple triple sets, Ontology Toolkit (OTK) uses a slightly different approach: templates create Simple Protocol and RDF Query Language (SPARQL) queries. This means that the result of applying a template is a SPARQL query that describes the intended operation. SPARQL is a language to describe queries on Resource Description Framework (RDF) documents. Its roughly the same as SQL is to relational databases, but to RDF documents. Most of the engines and technologies, typically adopted to store ontology formats, provide a SPARQL end-point, i.e., a service (most of the times via web) to perform queries. This allows the adoption of a centralized and manageable mechanism to store, retrieve, and share data. Appendix C provides an introduction to the SPARQL query language. This allows to devise templates that describe queries to perform operations, independent of the actual technology used to store the ontology, allowing the implementation of generalized operations in the toolkit.

SPARQL provides a mechanism to define queries based on triples. Although, an ontology can be represented as a set of triples, this is not enough. A language that defines the proprieties of the triples is needed, for example, to define clearly how classes and instances, are represented, etc. OTK use the OWL family of languages, introduced in Chapter 3, to define this semantics. OWL was adopted, not only because of the inherent characteristics of the language itself (e.g., clear, concise, good expressive power), but also due to its popularity, and the vast set of resources and tools currently available. But, as most of the design options in OTK, this is a well defined step in the workflow, other modules can be added to the toolkit to feature other languages.

For example, to add a new class to an ontology, OTK provides the *add_class* method, it can be used to add a class in a application. Adding the *Female* class to the ontology from a program written in Perl:

```
$ontology->add_class('Female');
```

This statement calls the *add_class* method, available in the object that represents the ontology passing the name of the new class as argument. A simplified version of the template for creating the SPARQL query that implements this operation is as follows:

```
INSERT DATA {
  GRAPH <[% graph %]> {
    [% name %] rdf:type owl:Class .
  }
}
```

The operation is an *INSERT*, since the goal is to add new information to the ontology (a new class). The *GRAPH* keyword defines in which ontology the query is to be executed. And finally the template describes the new triple to add, using the *rdf* : *type* relation to define a new *owl* : *Class* (following the OWL semantics). Processing this template for the specific example illustrated before would render the following query:

```
INSERT DATA {
  GRAPH <http://conclave/example/ontology> {
    <http://conclave/example/ontology#Female> rdf:type owl:Class .
  }
}
```

This query is then executed on the SPARQL end-point set during the definition of the object representing the ontology, adding a new class named

Female (an element of type *owl*:*Class*).

OTK provides a set of operations that allow common ontologies manipulation, and new operations can be easily added by defining new templates to build new queries. The following sections describe some of these operations in more detail, and illustrate how the OTK library can be used from an application written in Perl.

Classes and Instances

Adding classes and instances to an ontology are common operations. Instances (or individuals) are a major part on an ontology, they describe the entities that exist in the domain. Adding a class is done using the *add_class* method, that allows two arguments: (i) the new class name, and (ii) an optional set of parent classes for the new class. For example, adding the *Male* class to an ontology, as sub-class of the *Person* class:

```
$ontology->add_class('Male','Person');
```

Once classes are available, instances can be created. For example, creating an instance named *Ann*, of the *Female* class:

```
$ontology->add_instance('Ann','Female');
```

New classes and arbitrary instances are added to populate the ontology.

Once the information is populated, OTK provides methods for retrieving information from an ontology, related with classes and instances. For example, to get all the classes available in the ontology:

```
my @classes = $ontology->get_classes();
```

Or, to get the set of all sub-classes of a given class:

```
my @subclasses = $ontology->get_subclasses($parent);
```

More closely related with instances, for example, get the set of available instances for a given class:

```
my @instances = $ontology->get_instances($class);
```

Object and Data Properties

OTK also provides methods for handling properties. To add a new object propriety, a relation between two instances, the method *add_obj_prop* is available. For example, to add a relation between the instance *Ann* and *Peter* named *hasParent*:

```
$ontology->add_obj_prop('Ann','hasParent','Peter');
```

To retrieve the list of proprieties for an element from the ontology, the method *get_obj_props*, giving as argument the element of interest:

```
my @proprieties = $ontology->get_obj_props('Ann');
```

A data propriety is a relation between an element and arbitrary data. For example to add the relation between the instance *Ann* and the integer 5, to represent the person age:

```
$ontology->add_data_prop('Ann','hasParent',5,'Int');
```

The set of data proprieties for a given element is retrieved using the *get_data_props* method, that has a single argument, the element of interest. For example to get the data proprieties set for the *Ann* instance:

```
my @proprieties = $ontology->get_data_props('Ann');
```

Table 6.7 summarizes the set of operations on ontologies provided by Conclave OTK, most commonly used. The composition of these operations is enough to build an assorted array of heterogenous applications. Some examples are illustrated in the following chapters. And, since the operations are performed on a SPARQL end-point, anyone can build a query to perform a custom operation, and directly explore the service, or use the operations provided by OTK.

Classes and Instances	Object and Data Proprieties
add_class	*add_obj_prop*
add_instance	*get_obj_props*
get_classes	*add_data_prop*
get_subclasses	*get_data_props*
get_instances	

Table 6.7: Summary of most commonly used operations on ontologies provided by Conclave OTK.

Summary

- This chapter describes an approach for modeling a domain of knowledge, and the toolkit for implementing this method.

- A model of a domain is an artifact that helps to quickly study and better understand an application domain. The model discards less relevant real world details, and emphasizes more relevant domain concepts.

- This chapter introduces a domain agnostic approach for creating a model, divided in two main steps: (i) process domain artifacts; and (ii) convey relevant information.

- The first step, summarized in Figure 6.4 for the program domain, processes a set of domain artifacts in order to abstract specific information about the domain, the final result is a set of resources that capture specific information.

- The second step, summarized in Figure 6.5 for the program domain, processes a set of resources in order to convey (transport) relevant information to an ontological format, with a concrete semantics.

- The final domain model is implemented using an ontology. In the context of PC heterogenous domains may be available (e.g., the application domain, the program domain), abstraction of each one of them is captured in a different ontology.

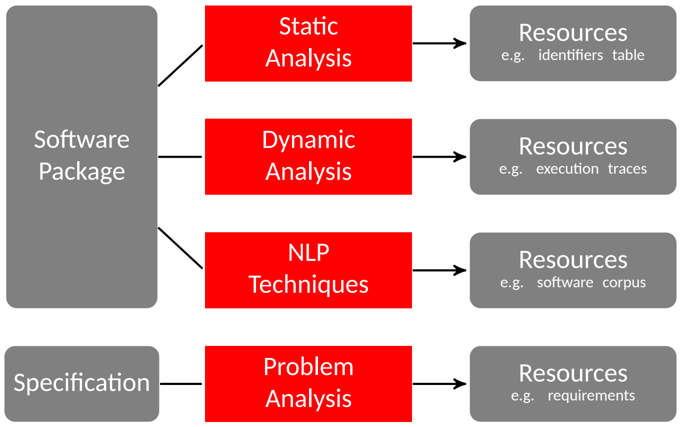

Software Package
Static Analysis
Dynamic Analysis
NLP Techniques
Resources e.g. identifiers table
Resources e.g. execution traces
Resources e.g. software corpus
Specification
Problem Analysis
Resources e.g. requirements

Figure 6.4: Process the software system, and related artifacts, using a set of heterogeneous techniques, to build resources that contain information about different domains.

- Conclave OTK is a toolkit for implementing ontology-aware applications, developed in the context of this work, and its used to implement the techniques discussed in this chapter.

- OTK uses a template based engine to build SPARQL queries, to describe and perform operations. The toolkit stores the ontology using OWL notation.

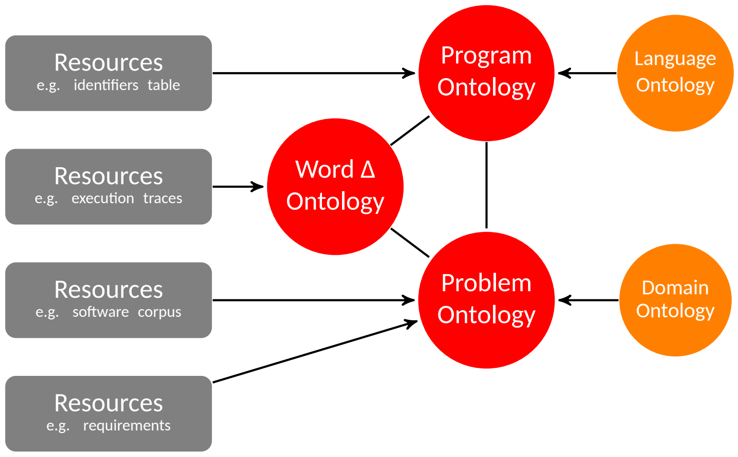


Figure 6.5: Convey the information available in the resources to a set of well defined ontologies.

Chapter 7

The Concept Mapper

The discussion in Chapter 6 revolves around on how to describe concepts about a software system, and related domains, using ontologies. For example, building a model for the program, or application domain. Relating real world concepts (commonly described in the problem or application ontology) with source code (the program ontology) is a crucial task during program comprehension activities, in order to understand a program. Concept Mapper main task is to build bridges between ontologies, to create relations between elements in the different domains.

This chapter describes the Conclave Concept Mapper framework. Its goal is to provide a reasoning layer, on top of the ontologies representation described in the previous chapter, that:

- provides a searching mechanism, for arbitrary keyword based searches;
- allows the creation of views that emphasize particular traits of interest;
- and, allows the creation of mappings between elements in different ontologies.

To create a view or a mapping, the reasoning layer queries the ontologies for information about their corresponding domain, and specific elements. This information is then used to build an artifact that emphasizes some particular trait being analyzed.

For example, query the program ontology for a list of function definitions available in the program, and query the problem domain for a list of concepts used in the application domain. Measure the relatedness between each concept and every function, and build a graph, or a tree-*like* structure, that emphasizes which functions are related to which concepts.

The abstract signature of the generic function provided by this framework is:

$$mapper :: [\,Ontology\,] \rightarrow [\,Query\,] \rightarrow View$$

i.e., given a set of ontologies and a set of queries, build some kind of view of the intended data. The *Ontology* data type captures domain models, and is defined in Section 6.4. The *Query* type defines the data of interest for the intended search or analysis, and how this data is compared when required. The *View* type encompasses the possible results of the *mapper* function. The view can be achieved in different ways, for example, a ranked set of elements when performing a plain search, new information to add to some ontology (i.e., infer new knowledge), a graph, a matrix of related elements of different domains, etc. Sometimes the resulting view can be less specific, or an intermediate representation, before presenting the data to the programmer or maintainer. A generic definition of the *View* type for now is:

$$\textbf{data}\ View = Rank \mid Mapping \mid [DataProp] \mid [ObjProp] \mid \ldots$$

More detailed definitions are introduced when more concrete applications of this function are discussed in future sections. In some situations, the *mapper* function can also compute a set of triples, more knowledge about the domain, to be added to the ontologies. In some cases an intermediate resource is created, that is further processed before being presented to the final recipient.

As with the *model* function in the previous chapter, these more generic definitions are used as a generic method for defining functions with a more straightforward practical application. An example of this are functions for locating elements of interest in a domain, that compute a rank. For this family of problems the generic definition of the *locate* function is:

$$locate :: Query \rightarrow Rank$$

which is discussed in detail in Section 7.2. Another example is the creation of mappings (matrix like resources) that describe possible relations between elements defined in the same domain, or in different domains. This family of problems is captured by the *map* function:

$$map :: Query \rightarrow Query \rightarrow Mapping$$

defined in detail in Section 7.3. The *locate* and *map* functions are specific cases, of the more generic *mapper* function. The ontologies input is omitted in both definitions for simplicity, it can be safely assumed that the set of ontologies is always available in these functions.

Defining queries is required for both functions, so the *Query* type and the domain specific language devised to write queries are discussed in the next section, before discussing the *locate* and *map* functions.

7.1 The Query Language

A query is used to describe which are the elements of interest in the domain. For example, when performing a simple search for a set of keywords, the class of the element of the domain can be used to restrict the elements being searched (e.g. search only function definitions). This is a detail that can be complex to accomplish using common approaches (e.g. *grep*). Most of the times when searching some comparison is performed between elements, to grade the relevance of the element being processed, for example comparing the function name with some search keyword. The functions used to compute relevance values for these comparisons are also defined in the query. Since more than one domain of knowledge (ontology) is available, the query allows to choose from which set of ontologies to perform queries.

Given this description of a query, the formal type is defined as:

$$\begin{aligned}&\textbf{data}\ Query\ = Query\ \{\ params\ ::\ [Param]\ \}\\&\textbf{data}\ Param = Param\ \{\ name\ ::\ String,\ value\ ::\ String\ \}\end{aligned}$$

A query is defined by a set of parameters, that describe the required proprieties, that entail the query semantics. Another advantage of this definition is that, to enrich the

Name	Description
word	set of keywords if required
class	classes of interest, i.e. select instances of these classes
aggr	relation to use to aggregate elements
score	scoring function used when elements need to be compared
onto	ontology to use

Table 7.1: Available parameters to define query properties.

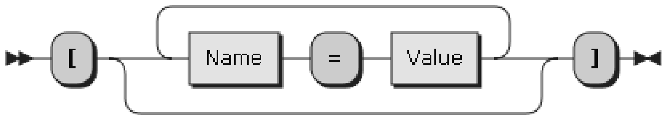


Figure 7.1: Syntax diagram for the query string grammar.

query with new proprieties, the query data type does not require any update. The currently defined parameters, that can be used to write a query are summarized in Table 7.1.

The *word*, *class* and *aggr* parameters can be used more than once in a query to define a set (e.g. a list of keywords, a list of classes). Figure 7.1 illustrates the syntax diagram, that defines the language to describe queries using a string: A query starts with a open square bracket ([), a list of pairs conjugated using the equals sign (=) follows, and ends with a closing square bracket (]). Each of these pairs defines a parameter, where the string on the left side of the equals sign defines the parameter name, and the string to the right side defines its value. To better describe how a query is used and to emphasize some query features, the remaining of this section is dedicated to the analysis of some examples.

The program maintainer when initially addressing a possible bug fix, may be interested in searching functions (or methods) only. The *class* propriety can be used to constrain the class of elements that are being retrieved from the ontology. The following query performs a search for the words "*color*" and "*schema*", but only analyses elements that are instances of the class *Function* (remember the ontology definition in Section 6.1, and that the program ontology is the default ontology for selecting elements):

> [word=color word=schema class=Function]

This means that the elements of the resulting rank are only instances of functions, because the query constrains the search domain of the *locate* function. Another example, can be searching for variables, by selecting the class *Variable*, this includes all the members that are instances of the class *Variable* and also instances of all sub-classes of *Variable* (e.g. *Parameter*, *LocalVariable*), i.e. the resulting rank includes all kinds of variables in the original program:

> [word=color class=Variable]

Particular types of variables can be selected, for example searching only local variables:

> [word=color class=LocalVariable]

Another important propriety that can be defined is the scoring function, i.e. the function that will compute the semantic relatedness score between the keywords searched and each of the selected elements, this is done using the *score* propriety. For example, the query:

> [word=color class=Variable score=levenshtein]

uses the Levenshtein word distance metric [90], to compute the score. By default, a scoring function based on kPSS is used (details about scoring functions are discussed in Section 7.4).

So far, the illustrated queries have been computing scores between elements (e.g. functions, variables) and a set of words. But more complex comparisons may provide more accurate rankings. The `aggr` propriety allows a query to define the name of a relation (defined in the ontology) to compute a score not only between each selected element, but also a set of related elements. For example, the query:

> [word=color class=Function score=levenshtein aggr=inFunction]

analyses all the functions, and for each function also considers all the elements that are related with that function by the relation *inFunction* (defined in the ontology). This relation is used to link all the local variables and parameters to the functions (or methods depending on programming language) where they are defined and used. In practice, the score for each element (function) is the mean between computing the score for the element itself, and the score for every local variable and parameter defined in that function.

7.2 The *locate* Function

The *locate* function addresses the problem of performing straightforward searches. Working in a similar fashion to a traditional web search engine, the user supplies a set of keywords that he or she wants to search, and the system builds a list of documents related with the keywords, sorted by relevance (using some kind of score). The signature for the generic *locate* function is:

$$locate :: Query \to Rank$$

i.e., given a query, compute a sorted rank of elements. The result of this function is always of type *Rank*, defined as:

$$\begin{array}{l} \textbf{data}\ Rank = Rank\ \{\ entries\ ::\ [Entry]\ \} \\ \textbf{data}\ Entry = Entry\ \{\ elem\ ::\ Element,\ score\ ::\ Float\ \} \end{array}$$

i.e., a rank is a set of entries, where for each entry the element and the relevance score are available. An element is used to represent any instance of any class in any ontology (e.g., a program function definition, a domain concept). The *score*, is computed by a scoring function (discussed in more detail in Section 7.4) and is used to sort the element set by relevance, the higher the score, for the majority of cases the higher is the degree of relatedness of the element with the search query.

Given the *Query* and *Rank* type definitions, the *locate* function is defined as:

$$locate\ \ :: Query \to Rank$$

```
$locate\ q =$ **let**
        $elements = getElements\ q$
        $entries\ \ = [\,Entry\ e\ (computeScore\ q\ e) \mid e \leftarrow elements\,]$
    **in**
        $Rank\ (sortByScore\ entries)$
```

i.e., given a query this function builds a rank (a set of entries) where an entry is created for every element in the set elements defined by the query, and each entry contains the element itself and the element score. The way the score is calculated is also defined by the query, details about this are discussed in Section 7.4. The *sortByScore* function simply sorts a list of elements by their corresponding score.

The *conc-locate*[1], a possible implementation for the *locate* function, is a command line tool for performing simple search queries in a software system. The tool takes two arguments: (i) the software system to use, and (ii) the query string. This and other related tools are discussed in more detail in Chapter 8, but some examples of usage are presented in this section just to illustrate some queries and corresponding results. In these specific examples, the queried ontology is always the program ontology, and the software package *tree-1.5.3* is the object of analysis.

```
$ conc-locate tree-1.5.3 "[ word=color ]"
tree.c: 1553 | int color(u_short mode, char *name, char orphan, char islink)
tree.c:  153 | int color(u_short, char *, char, char), cmd(char *),
 patmatch(char *, char *);
tree.c:  633 |   char *path, nlf = FALSE, colored = FALSE;
tree.c:  204 |   int i,j,n,p,q,dtotal,ftotal,colored = FALSE;
 (...)
```

The result includes the line of code where the elements were found, and is sorted by score, from highest to lowest. The prefix of each line includes the original source file, and line number. The query can constrain the search domain, for example by selecting only function defitions. The following example uses the same keyword search, but contrains the search domain to instances of the class *Function* only (i.e., function definitions):

[1]*conc-locate* is a tool available in the Conclave system introduced in the next chapter.

```
$ conc-locate tree-1.5.3 "[ word=color class=Function ]"
tree.c: 1553 | int color(u_short mode, char *name, char orphan, char islink)
tree.c:  153 | int color(u_short, char *, char, char), cmd(char *),
 patmatch(char *, char *);
tree.c: 1411 | void parse_dir_colors()
tree.c:  162 | void parse_dir_colors(), printit(char*), free_dir(struct _info**),
 indent(int maxlevel);
 (...)
```

The lines concerning variables are no longer present in the rank, because the corresponding elements (instances of variables classes) are no longer included in the computed rank. Of course the domain constrain could be on the variables class, or any other class available in the ontology.

7.3 The *map* Function

The *map* function goal is to build relations between elements, normally described in different ontologies. For example to related concepts in the application ontology, with elements from the program domain (e.g., functions). The generic signature for the *map* function is:

$$map :: Query \rightarrow Query \rightarrow Mapping$$

i.e., given a pair of queries, compute a set of scored relations between elements from both queries. The result of this function is usually a matrix-*like* structure, and is captured in the *Mapping* data type definition:

$$\begin{aligned}
&\textbf{data}\ Mapping = Mapping\ \{\\
&\qquad rows :: [\,Element\,],\\
&\qquad cols :: [\,Element\,],\\
&\qquad cells :: [\,Cell\,]\\
&\quad \}\\
&\textbf{data}\ Cell \quad = Cell\ \{
\end{aligned}$$

```
        $row$   $::$ $Element$,
        $col$   $::$ $Element$,
        $score$ $::$ $Float$
      }
```

i.e., a *Mapping* is a matrix of scores, where each cell stores the score (a real number) that measures the relatedness between the corresponding column and row element. Besides the cells, the set of row elements, and the set of column elements, are also stored. The *map* function is defined as follows:

```
$map$           $:: Query \rightarrow Query \rightarrow SFunction \rightarrow Mapping$
$map\ q_1\ q_2\ f =$ let
                   $rows = getElements\ q_1$
                   $cols\ \ = getElements\ q_2$
                   $cells = [\,Cell\ e_r\ e_c\ (f\ e_r\ e_c) \mid e_r \leftarrow rows,\ e_c \leftarrow cols\,]$
                 in
                   $Mapping\ rows\ cols\ cells$
```

Given a set of queries, that define two sets of elements, and a scoring function (more details about scoring in the next section), the *map* function computes a matrix (a set of cells), where each cell stores the relatedness score between the cell row and column elements. Examples of results for this function are illustrated in Chapter 8.

7.4 The Scoring Function

The *locate* and *map* functions, defined in previous sections, describe two generic methods for searching and building relations between elements. In either case a way to compare single elements is required, and the result of this comparison needs to be quantitative. In most cases, the goal is to have a numeric measure of how two elements are closely related, in the sense that they deal with the same real world concept(s). This idea has been entailed in the *semantic relatedness* expression used previously. Hence, whenever this expression is referred, its empiric measure is computed using a scoring function. This section describes scoring functions in general, and discusses some particular options for calculating scores.

Before analyzing possible scoring functions and their definitions, a more concrete definition of what actually is an element is required. An element is represented as an instance in an ontology, this means that an element is used to represent, for example, a function definition, a variable in the program ontology, or a concept in the problem ontology. The important detail is that every element has a textual representation, usually captured in a string. For example, a function definition is textually referred as the name of the function, a concept in the problem domain is also refereed by its name. Thus, the problem of computing the relatedness between two elements is reduced to the problem of comparing the two strings that represent the elements. A string is composed of a set of terms, at least one is required to convey some semantic meaning. So, the problem of comparing two strings is reduced again, to comparing two sets of terms. Ideally, comparing the semantic conveyed in the term set, and not only their syntactic resemblance.

Every element in the ontologies has a propriety that defines the string that represents the instance (see the *hasString* propriety in the program domain for example, in Section 6.1). The *Element* data type, used to store an element from any ontology is defined as:

$$\textbf{data}\ \mathit{Element} = \mathit{Element}\ \{\ \mathit{id}\ ::\ \mathit{UID},\ \mathit{hasString}\ ::\ \mathit{String}\ \}$$

An element in the ontology is identified by its unique identifier, and has a textual representation described by a string. When required, every other propriety can be extracted from the respective ontology, using the instance unique identifier (also unique across different ontologies). The string used to represent the instance, for now, is the only required information from the element.

Any scoring function used to measure the relatedness between elements respects the *SFunction* signature, i.e., it takes two elements as argument, and computes a real number (the score):

$$\textbf{type}\ \mathit{SFunction} = \mathit{Element}\ \rightarrow\ \mathit{Element}\ \rightarrow\ \mathit{Float}$$

The generic function *compare*, that actually compares two elements, is defined as:

$$\mathit{compare}\qquad ::\ \mathit{SFunction}$$

```
$compare\ e_1\ e_2 =$ let
        $t_1 = terms\ (\ hasString\ e_1\ )$
        $t_2 = terms\ (\ hasString\ e_2\ )$
    in
        $score\ t_1\ t_2$
```

where the *score* function is responsible for computing a float that compares both sets of terms. The *terms* function, simply breaks the string in terms, usually by splitting the string by the comma character, or empty spaces, depending on how the original data was stored. Different approaches for implementing the *score* function result in different ranks and mappings.

The general idea, is to find a way of computing the proximity between arbitrary terms, to figure out if they represent related concepts. A simple first approach is, for example, to verify if the terms are equal, which would provide a binary answer: if the terms are equal then the two concepts are related, and if the terms are not equal they represent different concepts. But this would fail with simple and obvious cases, like for example the terms "*write*" and "*wrote*". Although they are not textually equal, they represent the same verb "*write*", so it is possible (for some degree of concept definition) to claim that they represent the same concept, the act of writing. These, and similar situations, like comparing inflections (e.g., "*book*" versus "*books*") are overcome using some linguistic based approach, e.g., compare lemmas instead of actual terms, or use measures like the Levenshtein distance. These approaches allow finding more relations between terms, but still they rarely find semantic relations between terms that are textually completely different. For example, the terms "*car*" and "*vehicle*", in some contexts are used to represent the same concept but they are clearly different textually. To overcome these challenging situations other approaches are required, a common trend is to build some kind of fuzzy synonyms set for each term, and measure its intersection in some way (e.g., number of common terms). The next section, discusses the scoring function used by default in Concept Mapper, based on kPSSs.

Order (n)	Synset	Probability
$n = 1$	insert	0.333333
$n = 2$	insert	0.166667
	inserts	0.166667
$n = 3$	buffer	0.074937
	enter	0.054291
	insert	0.068407
	inserts	0.067033
	insertion	0.019270

Table 7.2: kPSS of order 3 for the term “*insert*”.

7.4.1 kPSS Based Scoring Function

This section describes the algorithm that measures the relatedness between two sets of terms using *kind-of* Probabilistic Synonyms Sets (kPSS). A kPSS is a data structure that stores synonyms for a given seed word organized in orders, including a probability for each synonym. Table 7.2 illustrates an example, the kPSS created for the term “*insert*”.

Given two (or more) kPSSs it is possible to calculate a similarity score between them. This score is used by aforementioned functions to measure relatedness between elements in the ontology, to create ranks or mappings. At this level the only information available of such elements is their textual representation, conveyed in a string, which is usually composed of a set of terms (words). Hence, the relatedness measure is computed based on these sets of terms. The function that scores this relatedness (assigns a real value) is defined as:

```
score         :: [Term] → [Term] → Float
score t₁ t₂ = let
                    kpss₁ = kpss t₁
                    kpss₂ = kpss t₂
               in
                    kpssScore kpss₁ kpss₂
```

i.e., given two sets of terms, first a kPSS is computed for each set of terms using the *kpss* function, and then the *kpssScore* function is used to computed the score between the two resulting kPSSs.

The *kpss* function is the first requirement to implement the scoring function. The goal of this function is to compute a kPSS for a set of terms. The strategy to compute this is divided in two major steps: (i) compute a kPSS for every individual term; and (ii) use an operation defined in a kPSS algebra to compose together the set of kPSSs, computing the final kPSS. Following this strategy two functions are required. One that given a term computes its kPSS, that has the following signature:

```
kpssBuild :: Term → kPSS
```

and a function to compose together a set of kPSSs into a single kPSS, for example kPSS union, with the following signature:

```
kpssUnion :: [ kPSS ] → kPSS
```

and, the *kpss* function, that builds a kPSS for a set of terms is then defined as:

```
kpss       :: [ Term ] → kPSS
kpss terms = kpssUnion (map kpssBuild terms)
```

i.e., build a kPSS by processing each term with the *kpssBuild* function, and composing together the resulting set using the *kpssUnion* function. Other composing functions can be used, for example kPSS intersection, that yields a different kPSS, and consequently a different score. Before discussing in detail the *kpssBuild* function, a data type for storing a kPSS is required.

A kPSS of order *n* is a list of *n synsets* (sets of synonyms). Each synset is composed of a set of terms, and for each term a corresponding probability, that measures the degree of confidence that the term is semantically equivalent (or related) to the original term (i.e., represents the same concept):

```
data kPSS  = kPSS { orders :: [ Order ] }
data Order = Order { n :: Int, synset :: [ Syn ] }
data Syn   = Syn { term :: String, prob :: Float }
```

Order *n*	Synset Content
1	the term *t* itself with $p = 1$
2	term *t* lemma and possible inflections, with each $p = \frac{1}{\lvert words \rvert}$
3	term *t* PSS, with corresponding probabilities

Table 7.3: Synset terms per order n in a kPSS.

A *kPSS* is a list of orders (of type *Order*), where each order stores the order number *n*, where $n \in N$, and a list of synonyms, for each synonym (of type *Syn*) the term itself and a probability is stored.

To build a kPSS the function *kpssBuild* is used, and has the following definition:

$$\begin{aligned} &\mathit{kpssBuild} \;::\; \mathit{Term} \rightarrow \mathit{kPSS} \\ &\mathit{kpssBuild}\ t = \mathit{normalize}\ [\ \mathit{Order}\ n\ (\mathit{buildSynset}\ t\ n) \mid n \leftarrow [1\,..\,3]\ ] \end{aligned}$$

i.e., given a term *t*, start by building the synset for orders 1 to 3, and then normalize the resulting synsets (i.e., maintain the invariant that the sum of all probabilities is less or equal to 1). To build a synsset for a given order, the *buildSynset* function is used, and has the following definition:

$$\begin{aligned} &\mathit{buildSynset} \;::\; \mathit{Term} \rightarrow \mathit{Int} \rightarrow [\mathit{Syn}] \\ &\mathit{buildSynset}\ t\ n = \langle \text{summary on how to build each synset in Table 7.3} \rangle \end{aligned}$$

i.e., given a term *t* and an integer *n* build the synset for term *t* of order *n*, that contains a set of synonyms with corresponding probabilities. Table 7.3 summarizes how this function builds the set of words that are members of the synset for a given order. For $n = 1$, the synset contains only the term *t* with a $p = 1$. For $n = 2$, the synset contains the lemma for the given term, i.e., the dictionary form has described in Chapter 5, and possible inflections (e.g., the plural or singular of the term), each word has the probability equal to 1 normalized by the total number of words in this synset. For $n = 3$, the synset contains terms extracted from the PSS for the term *t*, the probabilities are the same as defined by the PSS. More details about PSSs and how they are computed are presented in the next section.

Finally, the last requirement to compute the kPSS based relatedness score is the *kpssScore* function, defined as:

```
kpssScore       :: kPSS → kPSS → Float
kpssScore k1 k2 = sum [ min (prob x) (prob y) |
                        x ← flatten k1, y ← flatten k2,
                        word x == word y ]
```

This function iterates over the flattened version of the kPSS, and sums the minimum probabilities for terms that are common. The flattened version of the kPSS is simply a single list of terms. The *flatten* function is defined as:

```
flatten      :: kPSS → [ Syn ]
flatten kpss = unique $ concat [ synset order | order ← kpss ]
```

The *unique* function removes duplicate terms from a synset list, adding their probabilities. A possible definition for this function follows:

```
unique     :: [ Syn ] → [ Syn ]
unique set = let
                uniq = nub [ term s | s ← set ]
             in
                [ Syn t (sum $ map prob $ filter ((== t).term) set) | t ← uniq ]
```

The *kpssScore* function can be directly used as the *score* function, in the *compare* function definition, to quantify relatedness between elements, to compute ranks or mappings. It is important no note that looking at this score as absolute value does not convey much interesting information, its main goal is to rank (sort) sets of elements.

For example, given the kPSS illustrated in Figure 7.2 for the term "*insert*", and the following kPSSs for the terms "*enter*" and "*delete*" respectively.

	"*enter*"		"*delete*"	
Order (n)	Synset	Probability	Synset	Probability
$n = 1$	enter	0.333333	delete	0.333333
$n = 2$	enter	0.166667	delete	0.166667
	enters	0.166667	deletes	0.166667
$n = 3$	between	0.070727	delete	0.087397
	enter	0.063716	deletes	0.022420
	indicates	0.067441	excludes	0.029492
	insert	0.044695	is	0.081242
	provide	0.030405	removal	0.032690
	specify	0.027150	remove	0.080092
	type	0.029200		

The relatedness score between the terms can be computed using the kPSS based scoring function. The following tables summarizes the results of the *kpssScore* function for several combinations of input kPSSs.

$kpss_1$	$kpss_2$	*kpssScore*
"*insert*"	"*enter*"	0.098986
"*insert*"	"*delete*"	0.000000
"*enter*"	"*delete*"	0.000000

The table shows that there is no relation between "*insert*" and "*delete*", and "*enter*" and "*delete*", because the score is 0. But the score achieved between the terms "*insert*" and "*enter*" is 0.098986, higher than 0, showing that there is a semantic relation between the two terms. In the context of PC this is a plausible conclusion, since most of the times, "*insert*" and "*enter*" both refer to the same action (real world concept) of providing (inserting or entering) some kind of data. A clearly different action is deleting data, captured by the "*delete*" term. Semantically, "*insert*" and "*enter*" are much closely related than "*insert*" and "*delete*", or "*enter*" and "*delete*".

Term	Probability
patches	0.137035
regression	0.184512
sample	0.189865
test	0.203093
tests	0.107999
testing	0.177496

Table 7.4: PSS for the term "*testing*".

7.4.2 Probabilistic Synonyms Sets

In linguistics, and in the field of terminology, a set of synonyms, for a given term, is commonly called a *synset*. A popular example of *synsets* is provided by WordNet[2] [108, 106].

A Probabilistic Synonyms Set (PSS) is a set of terms closely related with the original seed word, it is similar to a *synset*, but it may contain terms that are not actual synonyms but are semantically related. A PSS is automatically built from PTDs (introduced in Section 5.5), and each synonym in the *synset* has an associated probability. This probability measures the similarity distance between each synonym and the original term. A PSS can contain terms in several languages if required. Table 7.4 illustrates an example of a PSS, for the term "*testing*".

A PTD contains dictionaries in a pair on languages (e.g., portuguese – english), which means that, there is a dictionary from one language to another and vice-versa (e.g., from portuguese to english, and from english to portuguese). A PSS is built by performing a triangulation between a pair of dictionaries, i.e., first get the set of translations from one language to another, and then for each possible translation determine the set of translations to the original language. Usually there is a probability constrain on the possible translations, to control the cut level of terms that are added to the final *synset*. Formally, and given the definition of *Syn* presented earlier, the PSS data type is defined as a set of synonyms:

$$\textbf{type}\ \mathit{PSS} \quad = [\, \mathit{Syn}\,]$$

[2]Available from: `http://wordnet.princeton.edu/` (Last accessed: 24-10-2014).

```
type $Prob = Float$
```

A probability, captured by the type *Prob*, is simply a real number. The following function, named *pssBuild*, defines how a PSS is built, given a pair of PTDs (one from language *A* to *B*, and another from language *B* to *A*), a term, and a cutting probability.

```
$pssBuild :: PTD_{A,B} \to PTD_{B,A} \to Term \to Prob \to PSS$
$pssBuild\ ptd_{A,B}\ ptd_{B,A}\ t\ p =$
 let
   $terms_{A,B} = ptd\ ptd_{A,B}\ t$
   $terms_{A,A} = concat\ [\ ptd\ ptd_{B,A}\ t_i \mid (t_i, p_i) \leftarrow terms_{A,B},\ p_i > p\ ]$
 in
   $[\ Syn\ t_j\ p_j \mid (t_j,\ p_j) \leftarrow terms_{A,A}\ ,\ p_j > p\ ]$
```

Where, *ptd* is a function, that given a PTD from language *A* to *B*, and a term *t*, computes a set of pairs, where each pair contains a term that is a possible translation of *t*, and a probability that is translation is valid. The final synset found in a PSS is added to a kPSS, as the 3^{rd} order synset.

A PSS provides a set of terms, closely related with the seed word. These terms are not all synonyms in a linguistic sense, i.e., sometimes a term is semantically closely related with the seed word, but it is not a synonym that would appear in a dictionary. For example, in the PSS example illustrated in Table 7.4, the term "*regression*" is not exactly a synonym for the term "*testing*", but they are very closely related in the context of software engineering[3]. These kind of relations between terms are usually not found in more linguistic based *synsets*, but they help to enrich the PSS semantically, and increase the number of possible relations with other terms. The scope of these relations, and the general vocabulary present, is controlled by the original parallel corpora that was used to build the PTDs from which PSSs are created. PTDs can be created from generic parallel corpora resources, but they can also be created from domain specific corpora. The text used helps to control the translation dictionaries vocabulary, which in turns helps to control the vocabulary present in the PSSs. For example, the PSSs used throughout

[3]In software engineering, regression testing is a specific type of testing, to make sure a new version of a program maintains backwards compatibility for example.

this work, included the examples illustrated, are built from a set of PTDs created based on parallel corpora extracted from software documentation, i.e., text about software. This is a way of controlling the term sets available in the PSSs. This is the main reason why, for example, the PSS for the term "*fork*" does not contain terms related with cutlery, because the actual sense that is predominant in the parallel corpora used, is the one related with the operating system forking a process. The PTDs and some related resources used are available in the Per-Fide project website[4].

7.5 Conclave Concept Mapper

Conclave Concept Mapper is a framework, implemented as a library for the Perl programming language, that provides the functions described in this chapter. Its designed to work together with Conclave OTK. This library uses the abstraction layer provided by OTK to retrieve data for answering locate queries, and build maps, even outside the scope of PC.

The library features two general mechanisms: (i) for searching, following the *locate* function definition; and, (ii) for creating maps, following the *map* function; both functions discussed earlier in this chapter. As input the provided mechanisms expect a query, written as a string, like the examples illustrated before. The framework includes a parser-*like* tool that is responsible for processing the query and, using primitives provided by OTK, achieve the intended results. The scoring functions (e.g., *kpss*), are provided as plug-ins, and new scoring functions can be easily added.

Besides the library, the framework also features some command line tools (e.g, *conc-locate*) for more quick operations, or to compose textual results with other tools. The next chapter illustrates some tools implemented using this framework.

[4]Available from: `http://per-fide.di.uminho.pt/` (Last accessed: 24-10-2014).

Summary

- Locating concepts, and relating concepts with source code, are crucial tasks during PC activities, in order to understand a software system, or a part of it.
- This chapter describes a set of functions that are used to clearly define locating and mapping operations. These definitions are represented using a query string, with a well defined syntax.
- Locate and searching functions compute ranks, and mapping-*like* functions produce maps between elements. In either cases a scoring function is used to measure the semantic relatedness of elements.
- Every scoring function between two elements follows the pattern illustrated in Figure 7.3, every element in the ontology as a textual representation (a string), that can be composed of one or more terms. Hence, a scoring function given two sets of terms, computes a score (a real number).
- A scoring function, based on kPSS, is defined during this chapter. A kPSS is a data structure for capturing terms synonyms, and the score between two kPSS can be used to measure the semantic relatedness between two terms.
- A PSS captures a set of synonyms, and closely related terms, based on a collection of parallel corpora, which can be domain specific, that allow the creation of relations between terms.
- Conclave Concept Mapper implements the described operations as a Perl library, and its designed to integrate with OTK (the toolkit described in the previous chapter), to build applications that create views of the program (and related domains) to enhance PC activities. The Concept Mapper contribution to the general workflow is illustrated in Figure 7.2.

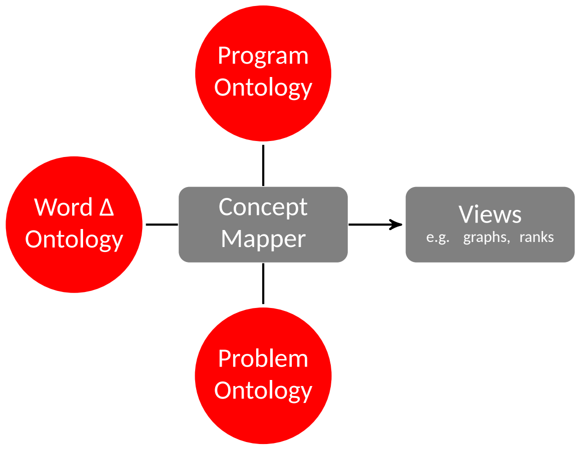


Figure 7.2: The Concept Mapper framework retrieves information from ontologies, and produces views of elements of interest.

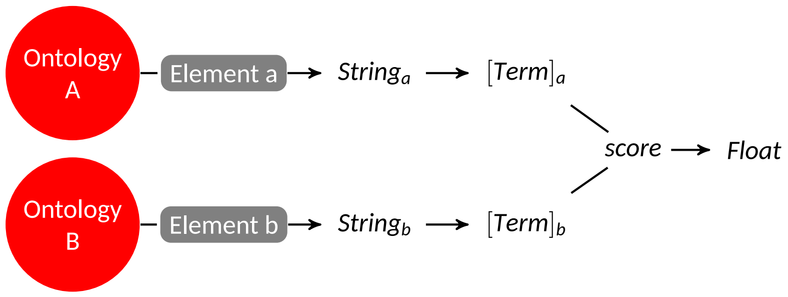


Figure 7.3: General pattern for computing scores between elements in ontologies.

Chapter 8

The Conclave Environment

There's an odd misconception in the computing world that writing compilers is hard. This view is fueled by the fact that we don't write compilers very often. People used to think writing CGI code was hard. Well, it is hard, if you do it in C without any tools.

Allison Randal

Conclave is an environment for software analysis, featuring a set of applications and tools for enhancing several PC activities. The methodology described in Chapter 6 is used to represent knowledge about domains (including the software system), and the approach described in Chapter 7 is used to build views of elements of interest, that paired together with other provided assorted tools, enhance software maintenance tasks. The system provides a web interface to be used by humans when analyzing software systems, and some of the tools include a web service that provides access to some features to be used more cleanly by machines.

Achieve an implementation as modular as possible is one major design goal during the development of Conclave, this entails dividing problems in several smaller problems and devising an independent solution to solve each smaller problem using an approach as generic as possible. This allows a more dynamic system, since current tools in the system are easy to update, and new tools can be added to the workflows without effort, to upgrade or provide new features. Also, generic tools are prone to be useful outside the scope of this system, or even outside the scope of software engineering.

The system provides a set of tools, that can be used independently, or composed together in previously defined workflows. These are divided in two main groups: (i) tools for performing tasks concerned with modeling the software system, and related domains, using ontologies; and, (ii) tools related with performing reasoning about the models, and providing views of the software system under analysis.

The remaining sections in this chapter discuss in detail some of these tools, and how they are composed together, to build the program representation and other models (e.g., the problem domain model) and, the interface provided to programmers and maintainers during software maintenance and evolution tasks.

Source Code Analysis

The major source of information of any software system is the source code itself. Assorted information can be extracted from analyzing the software system. This kind of information is rich enough to build a model dedicated entirely to the source code, this is refereed in previous chapters as the program ontology. This ontology conveys information about the program, that is explored by other tools to provide concept location and searching features. This representation is not complete, i.e., some of the details about the source code are discarded, and there is not enough information in the program ontology to build back the program.

Some elements of interest in the program and their representation in the ontology are discussed in Section 6.1. A set of tools were implemented to collect this information from the source code. Of course the ontology is not closed, and other tools can be used to extract more information, and produce other artifacts, that are later conveyed to the program ontology.

Analysis of Non-Source Code Artifacts

Besides source code, a fundamental source of information about software systems lies in documentation, and other non source code files, like *README*, *INSTALL*, or *How-To* files, commonly available in the software ecosystem. These documents, written in natural language, provide valuable information during the software development stage, but also in future maintenance and evolution tasks. Their content provides valuable information, that is conveyed to several models (e.g., application ontology, program ontology).

This chapter discusses tools and applications that act at different processing stages:

- with the goal of processing artifacts, creating and populating ontologies:
  - a tool to analyze source code and create intermediate resources that convey information of interest (e.g., Clang Conclave);
  - tools to create and populate ontologies with information extracted from source code and other artifacts (e.g., DMOSS, Conclave Utils);
  - tools that process ontologies and infer or create new information about domains (e.g., splitting and expanding identifiers using Lingua::IdSplitter);
- browsing ontologies information, or creating views about the software system:
  - allowing keyword based searching, or the creation of mappings to build bridges between domains (e.g., Conclave Concept Mapper);
  - and, the Conclave system in general.

8.1 From Code to Resources: Clang Conclave

Source code analysis, to extract information about program elements, is a crucial step when building the program domain ontology, the minimum required ontology for other tools to perform useful operations. This section describes Clang Conclave, a tool for source code static analysis for programs written in the C programming language. The major goal of this tool is to extract elements of interest from C programs to build the program ontology. The initial version of this tools was based on *Exuberant Ctags*[1], a tool for generating an index of language objects found in source code. Although it supports

[1]Available from `http://ctags.sourceforge.net/` (Last accessed: 28-10-2014).

a vast number of programming languages (the initial motivation to adopt it), it is based on processing the code as text, not using an approach based on a parser. Meaning that it does not extract some information of interest, e.g., the context where the program element is defined. Thus, the need of devising a tool using compilers technology, but the *Exuberant Ctags* based version is still useful to extract information from source code written in programming languages for which there is no specific front-end.

The main tasks performed by Clang Conclave are:

- Extract variable, functions and macro declarations, including: (i) original source file; (ii) context (i.e., where the element is in the code, for example inside a function); (iii) the name given to the element (i.e., the identifier); and, (iv) the source file line number where the element begins and ends.
- For each element extracted clearly identify to which class the element belongs, classes of elements are defined in Section 6.1.
- Extract static function calls, i.e., information about which functions are called throughout the code, even if functions are not called during runtime due to control flow decisions (e.g., *if* statements).

Following the strategy defined in Chapter 6 for modeling the program, this tool builds an intermediate artifact (acts as a *processing* function). The relevant information needs to be conveyed to the ontology later. This last step is performed by another tool, a PC specific module distributed with the Conclave Utils library.

An example of executing this tool, and the resulting artifact is illustrated in Chapter 6: Figure 6.1 mirrors the output of this tool for the source code illustrated in Program 6.1. The raw output of this tool is in Comma-Separated Values (CSV) format, where each line describes an element in the original source code. The following example of usage illustrates the raw output of this tool for the source code illustrated in Program 6.1 using the web service provided by Conclave:

```
$ curl -F "upload_file=@factorial.c" http://conclave.di.uminho.pt/clang/ws
# TYPE, UID, ID, CTX, FILE, LINESTART, LINEEND
```

```
Function,factorial.c::factorial::1,factorial,,factorial.c,1,8
Parameter,factorial.c::n::1,n,factorial.c::factorial::1,factorial.c,1,1
LocalVariable,factorial.c::result::2,result,factorial.c::factorial::1,factorial.c,2,2
hasFunctionCall,factorial.c::factorial::6,-,factorial.c::factorial::1,factorial.c,6,6
```

This output is loaded to the program ontology using the `conc-otk-load` tool, distributed with the Conclave Utils library. Given that the above output is stored in the `factorial.data`, and the *http://local/factorial* ontology is being used, the following command:

```
$ conc-otk-load http://local/factorial clang factorial.data
```

conveys the information computed by `clang-conclave` to the ontology, populating it with the required instances and data.

One of the advantages of splitting this task in two steps, and two distinct tools, is that although the first step is language dependent, the second tool, since it processes the created intermediate artifact in an intermediate representation, is less language specific and is used to process artifacts created from a heterogeneous set of programming languages. This is a practical example of the modularity that Conclave is, in general, aiming for.

Given that Conclave Utils provides this PC specific module for conveying resources built by Clang Conclave, the problem at hand is how to build the intermediate artifact. The first required component is a C parser. Instead of implementing a C parser from scratch, Clang Conclave was implemented as a CLang module using the CLang C parser[2]. CLang is the C/C++ (among others) front-end for the LLVM compiler [78]. The CLang library provides functions for parsing and parsing tree traversal performing arbitrary processing. The CLang library is used to: (i) parse the C code and build a parsing tree; and, (ii) traverse the parsing tree, and for each node of interest produce the corresponding entry in the intermediate representation. The nodes of interest set is defined by the elements of interest discussed in Section 6.1 – the program domain. In *libclang* each node (usually refereed as *cursor*) in the parsing tree has a well defined unique identifier (refereed as *kind*) that describes the type of node, the nodes types are defined in the

[2]Available from: `http://clang.llvm.org/` (Last accessed: 12-10-2014).

Node	Parent	Short Description
8	-	function declaration
9	8	variable declaration
9	300	variable declaration (local)
10	-	parameter declaration
20	-	type declaration
103	-	function call
501	-	macro declaration

Table 8.1: Nodes of interest identifiers in the libclang parsing tree.

`CXCursorKind` enumeration. Table 8.1 describes the node types that are processed in the parsing tree, each of these nodes spawns a line representing the element in the final artifact. Every other nodes are ignored.

Using this approach, the tool gathers not only information about the identifiers used throughout the code, but also information about elements of interest, as described in Section 6.1, for example: function definitions and corresponding parameters definitions, static function calls, local and global variables, etc.

This same strategy was also applied to develop a similar tool to process source code written in Java. Instead of using the libclang library, the Antlr parser generator[3] [114] was used to build the parsing tree, and transverse the tree to produce the intermediate artifact, analogously to the `clang-conclave` tool.

8.2 From Identifiers to Sets of Full Terms: Lingua IdSplitter

When writing code, programmers follow processes (e.g. use of abbreviations, multi-term composition) for devising identifiers that convey to programming language restrictions, but also are concise and convey semantic value. These processes are not always clearly identified, and often change during different stages of development. Nevertheless, bringing identifiers from the program domain to natural language full terms, i.e.,

[3]Available from: `http://www.antlr.org/` (Last accessed: 12-10-2014).

split muti-terms identifies, and expand abbreviations, can potentially improve concept location techniques. Guerrouj in [56] presents some studies on the impact of normalizing source code vocabulary on several feature location techniques.

Lingua IdSplitter (henceforth abbreviated LIdS)[4], is a simple and fast algorithm that addresses the problem of splitting multi-term identifiers, and can cope with abbreviations, acronyms, or any type of linguistic short-cuts (e.g., use only the first letter of a word). The algorithm calculates a ranked list of all the possible splits for an identifier, based on a set of dictionaries, and the top entry in the rank is proposed as the correct split. Besides the actual split, the result includes the set of full terms that compose the identifier, in case abbreviations were used for example. This technique can use an arbitrary set of dictionaries, but one of the major advantages of this approach is the use of a software specific dictionary computed automatically from the software corpus – computed automatically and specific to each software package – using a combination of Natural Language Processing (NLP) techniques. This dictionary enables the algorithm to correctly handle identifiers splitting using arbitrary abbreviations or combinations of terms specific to the application domain, not prone to be present in more general programming dictionaries.

This tool is used in the Conclave environment to split and expand identifiers extracted from software source code. Using the sets of full terms enhances the results of the scoring functions described in Chapter 7 to measure relatedness between elements.

8.2.1 The Splitting Approach

The goal of the technique described in this section is to split any combination of *hard* and *soft* terms (including abbreviations) found in an identifier. Figure 8.1 illustrates the intended process for the “*strcmp*” identifier. This identifier is composed of two abbreviations: “*str*” and “*cmp*”, so this is the first level of intended split. The next improved step, is to start expanding abbreviations to the full term they represent. The best possible answer is to have the list of all correct terms: $\{\, string,\ compare\, \}$, in this example.

To cope with cases where combinations of *soft* and *hard* words are used, the algo-

[4]LIdS is available under GNU General Public License in the official comprehensive Perl network (CPAN) from: `http://search.cpan.org/dist/Lingua-IdSplitter/` (Last accessed: 09-07-2014).

$$\begin{tikzcd}
 & strcmp \arrow[dl, "soft\_split"'] \arrow[dr, "soft\_split"] & \\
str \arrow[d, "expand"'] & & cmp \arrow[d, "expand"'] \\
string \arrow[dr] & & compare \arrow[dl] \\
 & string, compare &
\end{tikzcd}$$

Figure 8.1: Lattice for splitting the "*strcmp*" identifier.

$$\begin{tikzcd}
 & parse\_userstr \arrow[dl, "hard\_split"'] \arrow[drr, "hard\_split"] & & & \\
parse \arrow[dddrr] & & & userstr \arrow[dl, "soft\_split"'] \arrow[dr, "soft\_split"] & \\
 & & user \arrow[dd] & & str \arrow[d, "expand"'] \\
 & & & & string \arrow[dll] \\
 & & parse, user, string & &
\end{tikzcd}$$

Figure 8.2: Lattice for splitting the "*parse_userstr*" identifier.

rithm first applies a *hard split* technique, followed by a *soft split* to the strings resulting from the first split. Figure 8.2 illustrates an example.

The *hard_split* Function

This function is responsible for splitting strings when an explicit separator mark is present. Since this is not the main focus of this tool, a simple function that only detects two explicit cases: special common characters[5] and the CamelCase notation, is used. When these marks are found a simple split is made and the function returns the set of result-

[5]Currently these include: single dot, underscore and double colon.

ing strings. For splitting strings in CamelCase notation, the String::CamelCase (a Perl library) is used[6].

Algorithm 1 illustrates the *hard_split* function. The *matches* function (line 2 and 7) tries to match a string with a regular expression, returning a true value on success. *CamelCase* (line 7 and 8) represents a regular expression that matches the most common cases of CamelCase notation. The *re_split* function (line 3 and 8), returns a list of elements resulting from splitting a string using as delimiter a regular expression.

For example, the result of applying the *hard_split* function to the identifier "*insert_UserDataStr*" is the list: { *insert*, *user*, *data*, *str* }.

Algorithm 1 Compute *hard* splits.

```
Input: $id :: String$                                         // identifier to split
Output: $S :: [String]$
 1: $L \leftarrow [id]$
 2: if $matches(id, special\_marks)$ then
 3:   $L \leftarrow re\_split(id, special\_marks)$       // special marks are: '.', '_' and '::'
 4: end if
 5: $S \leftarrow \emptyset$
 6: for each $s_i \in L$ do
 7:   if $matches(s_i, CamelCase)$ then
 8:     $S \leftarrow S \cup re\_split(s_i, CamelCase)$
 9:   else
10:     $S \leftarrow S \cup s_i$
11:   end if
12: end for
13: return $S$
```

The *soft_split* Function

After the *hard* words in the identifier are split, the next step is to split *soft* words. The *soft_split* function, given a string to split, returns a list of pairs, each pair containing the string representing the cut and the full term (in case of abbreviations were used for example). A simplified version of the algorithm is illustrated in Algorithm 2.

[6]Available from: `http://search.cpan.org/dist/String-CamelCase/` (Last accessed: 03-03-2014).

Index	Set	Index	Set
0	$[t, ti, time, times]$	4	$[s, so, sort]$
1	$[i]$	5	$[o, or]$
2	$[m, me, mes]$	6	$[r, rt]$
3	$[e, es]$	7	$[t]$

Table 8.2: Dictionary valid words per string index for the identifier "*timesort*".

Lines 2-4 immediately return if the id to split is a valid string (i.e., a word or a known abbreviations[7]). Lines 6-14 compute all the possible valid strings that can be found starting in every position of the argument string. For example, Table 8.2 illustrates the possible strings per index for the "*timesort*" identifier. This means the set of valid words (according with the provided set of dictionaries) that start at every index. The actual computed set includes also the expanded terms (equal to the string if no abbreviation was used), and the weight assigned to the dictionary that validated the string. Once this set is computed, the next step (described in lines 16-20) is to build an automaton with all the words found, to calculate all the possible sequences of nodes (paths), that concatenate to rebuild the original identifier. An example of this automaton is illustrated in Fig. 8.3 for the "timesort" identifier.

The set of paths (sequence of nodes from the starting edge, to an end node) in the automaton, define the set of string sequences that are candidates to be the identifier correct splits. The *post_process* function, called in line 21, allows for some extra candidates to be created. Currently, a new candidate is added to the list when a sequence of 3 or 4 letters is found. The sequence of letters is added as a word with a weight lesser[8] than any dictionary, mainly to prevent over-splitting small unknown abbreviations and acronyms. Next, the algorithm computes the score for each candidate, creating a rank, where the top element (the sequence with the highest score) is returned as the resulting split. The top entries for the "*timesort*" identifier rank are illustrated in Table 8.3.

The *compute_word_graph* function, given a set of possible terms per original iden-

[7]Single letter strings (like "*a*" or "*x*") are valid english words. This technique can handle identifiers composed of such strings (e.g., "*xyfigure*" that splits to the set: $\{x, y, figure\}$).

[8]This weight is usually 0.1, since more specific dictionaries all have weights set to values above 0.1 (e.g., programming dictionary weight is 0.6, *custom corpus-based* dictionary weight is 0.6).

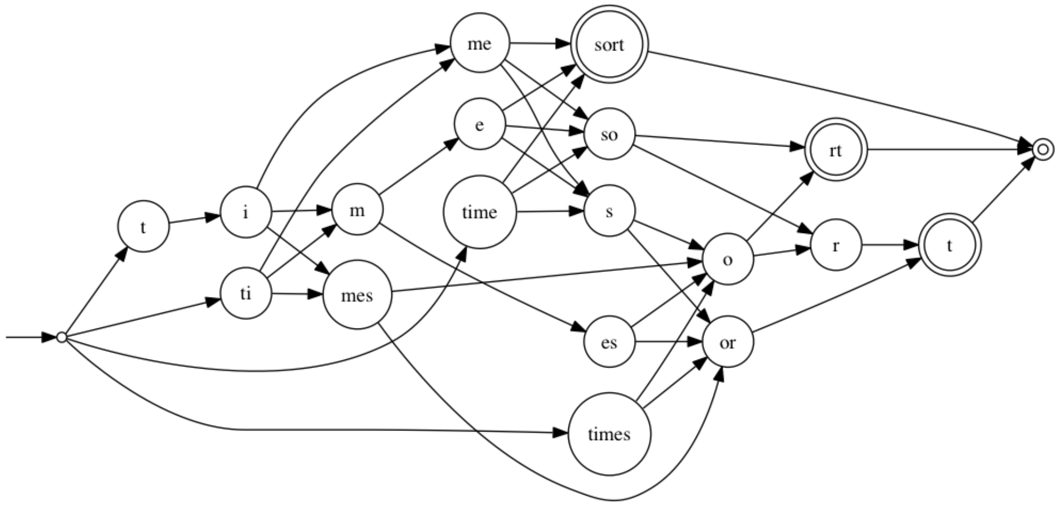


Figure 8.3: Word automaton for the "*timesort*" identifier.

Table 8.3: Top entries in the identifier "*timesort*" rank, sorted by score from highest to lowest.

Split	Score
$\{time, sort\}$	1.4400
$\{ti, me, sort\}$	0.1920
$\{time, so, rt\}$	0.1920
$\{times, o, rt\}$	0.1500
$(\ldots)$	

tifier index, builds an automaton (example illustrated in Fig. 8.3), and the *sort_by_score* function numerically sorts the candidates, using the entry scores, from highest to lowest. The *valid_term* and *score* functions are described in more detail later in this section.

The *split* Function

Given an identifier this function computes a list of pairs of type $(\textit{String}, \textit{Term})$ that represent the set of splits (and corresponding terms) for a single or multi-word identifier. It uses a combination of the *hard_split* and *soft_split* functions, and if a single word or known abbreviation is given as argument it returns the word or expanded abbreviation respectively.

This function is the entry point for the technique and is described in Algorithm 3. It starts by applying an $hard\_split$ to the argument, and then applies a $soft\_split$ to every resulting string. The final result is a list of pairs: each containing the split, and the full expanded term.

The *valid_term* Function

The *valid_term* function, used by the *soft_split* function, decides for a given string and a set of dictionaries if the string is a valid term. A string is considered valid if present in any of the dictionaries. If valid, the function returns a tuple including: (1) the original string, (2) the term the string represents (if an abbreviation is used for example), and (3) the dictionary (that validated the string) weight. Algorithm 4 illustrates this function implementation.

Generally, a dictionary is defined as a pair: (i) a function that given a string returns a word, and (ii) a weight:

$$\textbf{data}\ \textit{Dictionary} = \textit{Dictionary}\ \{\ \textit{words}\ ::\ \textit{String}\ \rightarrow\ \textit{String},\ \textit{weight}\ ::\ \textit{Float}\ \}$$

The weight is a float that expresses the dictionary degree of confidence. This attributes main purpose is to give dictionaries a preference order. For example, the english language dictionary is always used, and has a weight set to less than more specific programming dictionaries. So that programming terms, more common to be used as program

Algorithm 2 Compute *soft* splits.

```
Input: $id : String$                                   // identifier to split
Output: $S : [(String, String)]$                       // list of pairs split,term
1: // return if valid word or know abbreviation
2: if $(s, t, \_) = valid\_term(id)$ then
3:     return $[(s, t)]$
4: end if
5: // compute possible valid terms in $id$ per index
6: $terms \leftarrow \emptyset$
7: for $i = 0$ to $length(id)$ do
8:     for $j = i$ to $length(id)$ do
9:         $str \leftarrow splice(i, j, id)$
10:        if $(s, t, w) = valid\_term(str)$ then
11:            $terms[i] \leftarrow terms[i] \cup (s, t, w)$
12:        end if
13:    end for
14: end for
15: // compute every possible sequence of terms
16: $g \leftarrow compute\_word\_graph(terms)$
17: $candidates \leftarrow \emptyset$
18: for all $p_i \in paths(g)$ do
19:     $candidates \leftarrow candidates \cup p_i$
20: end for
21: $candidates \leftarrow candidates \cup post\_process(candidates)$
22: // compute score for each candidate
23: $scores \leftarrow \emptyset$
24: for $c_i \in candidates$ do
25:     $scores\{c_i\} \leftarrow score(c_i)$
26: end for
27: // sort candidates by score and select top ranked
28: $rank \leftarrow sort\_by\_score(candidates, scores)$
29: $top \leftarrow pop(rank)$
30: $S \leftarrow map\,(fst)\,top$
31: $T \leftarrow map\,(snd)\,top$
32: return $zip(S, T)$
```

Algorithm 3 Split an identifier.

```
Input: $id : String$                                   // identifier to split
Output: $S : [(String, String)]$                       // list of pairs split,term
1: $S \leftarrow \emptyset; T \leftarrow \emptyset$
2: $hard\_words \leftarrow hard\_split(id)$
3: for all $s_i \in hard\_words$ do
4:     $(s, t) \leftarrow unzip(soft\_split(s_i))$
5:     $S \leftarrow S \cup s$
6:     $T \leftarrow T \cup t$
7: end for
8: return $zip(S, T)$
```

identifiers, have a higher chance to be included in the result of the identifier split. And also, terms that share the same abbreviation, can use expansions more specific to the program domain. For example, "*directory*" is commonly abbreviated as "*dir*", but in the *AbcMidi* package, "*dir*" is more often used to abbreviate "*direction*"[9]. The weight is also used to calculate the sequence score (more details on this in the next sub-section).

Algorithm 4 Verify if a string is a valid term.

```
Input: $str :: String$                                 // term to be verified
Input: $D :: [Dictionary]$                             // dictionaries set
Output: $(s, t, w) :: (String, String, Float)$
1: for all $d \in sort\_by\_weight(D)$ do
2:     if $str \in domain(d\{words\})$ then
3:         return $(str, d\{words\}(term), d\{weight\})$
4:     end if
5: end for
6: return $\emptyset$
```

The *score* Function

The *score* function, is used by the *soft_split* function, to calculate a score for each possible sequence of strings (paths) found in the automaton. This score measures the likelihood that a given sequence of strings is the correct split for a multi-word identifier. The candidate sequences are sorted by score and the proposed solution is the sequence

[9]The direction is used to describe which way the note stem is oriented: upwards or downwards.

with the highest score.

The formula to calculate a score is analytically defined as:

$$score(S) = \frac{(\prod_{i=1}^{length(S)} factor(S_i)) + length(m)}{length(S)^2}$$

where the multiplicand of factors (a factor is calculated for each element in the sequence) plus the length of the longer string in the sequence, is normalized by the squared sequence length. Each factor is calculated according to the formula:

$$factor(s, t, w) = length(s) \times w$$

i.e., the length of the string found times the dictionary weight that validated the string.

Algorithm 5 illustrates the implementation of this function using these formulas. It simply iterates over the elements in the sequence, computing the multiplicand of element factors, adding the longest split length, and dividing by the squared sequence length.

Algorithm 5 The scoring function.

```
Input: $S : [(String, String, Float)]$            // split,term,weight triple
Output: $score : Float$
 1: $prod \leftarrow 1$
 2: $max \leftarrow 0$
 3: for all $s_i \in S$ do
 4:     $prod \leftarrow prod \times length(s_i\{split\}) \times s_i\{weight\}$
 5:     if $length(s_i\{split\}) > max$ then
 6:         $max \leftarrow length(s_i\{split\})$
 7:     end if
 8: end for
 9: $score \leftarrow (prod + max) / length(S)^2$
10: return $score$
```

8.2.2 Documentation Corpus

Informally, a corpus is a collection of texts, usually representative of a given domain or subject. These constructs are used to build other common linguistic artifacts in the field of NLP [59, 72, 100]. The first step in order to build some of the dictionaries used by the $valid\_term$ function is to create the *documentation corpus*, by collecting (natural language) text from all the files included in the software package.

The corpus is created using a tool distributed with the DMOSS[10] framework (discussed in the next section). This tool iterates over the files in the package, and some specific file types are processed to extract their content as plain text to be included in the corpus. The following heuristics are currently being used:

- Documentation files (that can be plain text files or other common formats like HTML, *man* or JavaDoc) content is included in the corpus, specific format files are implicitly pre-processed for plain text extraction.
- Text from files commonly available in software packages that usually convey domain information is also included (e.g., *README*, *INSTALL* files).
- All other plain text files content is included. The file type computed by the *DMOSS-Oracle* tool – also distributed with the DMOSS framework – is used to decide which files are plain text.

All this content is stored in a plain text file called the *documentation corpus*, specific for each software package. This file is later processed to build more linguistic artifacts, namely dictionaries, as described in the next section. Source code comments are extracted by a different process, and are not included in this corpus. More details about DMOSS are available in Section 8.3.

8.2.3 Custom Corpus-based Dictionary

Application domains tend to use a specific vocabulary, that uses terms, expressions and common abbreviations which are not easily found in general purpose dictionaries. This section describes the technique devised to automatically create a dictionary for domain specific abbreviations and multi-word expressions from the documentation corpus, specific to each software package.

[10]DMOSS is a framework for software packages (mainly non-source content) analysis, available from `http://search.cpan.org/dist/DMOSS/` (Last accessed: 27-03-2014).

The starting points are: (i) the documentation corpus, (ii) the set of program identifiers extracted from the the source code, and (iii) a general programming dictionary. The steps to create the *custom corpus-based* dictionary are:

1. Create the *srcIds* set, that includes the starting point (ii) - the set of identifiers collected from the source code – and, the explicit identifiers found in the documentation corpus. In order to be considered an explicit identifier, the string needs to combine one or more terms using an explicit mark (i.e., use *hard words*).

2. Split the *srcIds* set using *hard* split techniques, the resulting set is called *SimpleIds-bag*.

3. Search possible identifier expansions in the corpus. For each string in the *SimpleIds-bag* calculate a set of regular expressions to extract probable expansions and multi-word correspondences (matches) in the corpus. Rank them by occurrence frequency.

4. For every multi-word expansions found, calculate the single word correspondences, and the non-trivial ones are added to the custom corpus-based dictionary. By non-trivial we mean exact matches (equal strings), and that are known words in english. For example, the multi-word identifier "*timesig*", that expands to $\{$ *time*, *signature* $\}$ produces: (1) "*time*" $\rightarrow$ "*time*" (trivial), and (2) "*sig*" $\rightarrow$ "*signature*" (non-trivial, hence added to the dictionary).

5. To create the final dictionary, expansions and multi-words are included based on a set of filters (concerned with increasing precision, even if lowering recall). Filters include, for example, minimum length for abbreviated strings (3 characters), rejecting abbreviated strings with vowels, and expansions with a length of 15 characters or more.

For example, in the context of the *AbcMidi* package, many compound identifiers are found (e.g., "*mrest*" and "*timesig*"), and abbreviated terms (e.g., "*chan*"). Table 8.4 illustrates some example regular expressions created automatically to search the package corpus for the corresponding expansions. The top match for each expression is also illustrated, and the final expansion selected, either by filters or frequency count (when

String	Derived Regular Expressions	Top Match	
mrest	`m\w* rest\w*`	"*multibar rest*"	✓
	`m\w{,2}r\w{,2}est\w*`	$\emptyset$	
timesig	`t\w* imesig\w*`	$\emptyset$	
	`tim\w* esig\w*`	$\emptyset$	
	`time\w* sig\w*`	"*time signature*"	✓
chan	`c\w* han\w*`	"*chord handling*"	
	`c\w{,2}h\w{,2}an\w*`	"*channel*"	✓

Table 8.4: Derived regular expressions examples.

Id	Splits	Expands
mrest	*m* | *rest*	$\{$ *multibar*, *rest* $\}$
timesig	*time* | *sig*	$\{$ *time*, *signature* $\}$
chan	*chan*	$\{$ *channel* $\}$

Table 8.5: Top entries in the identifier "*timesort*" rank, sorted by score from highest to lowest.

different expansions are available). The heuristics to create the regular expressions involve iteratively filling gaps between characters with wildcards and spaces.

The first ranked corespondent occurrences in the corpus are: "*multibar rest*", "*time signature*" and "*channel*" respectively. The final corpus based dictionary (after all the inference process) includes: expansions (e.g., "*chan*" $\rightarrow$ "*channel*"), abbreviations (e.g., "*flg*" $\rightarrow$ "*flag*"), multi-word (e.g., "*timesig*" $\rightarrow$ $\{$ *time*, *signature* $\}$), some words not present in a general english dictionary but valid in the application context (e.g., "*lynx*");

The automatic creation of these dictionaries, which provide valuable information for splitting source code identifiers, is one of the major novelties introduced by this approach. These allow the correct split and expansion of strings difficult to achieve otherwise (e.g., the "*hornp*" expands to "*hornpipe*". And increases the relevance of specific domain terms, not found in general programming dictionaries (e.g., "*anacrusis*" or "*accidentals*", from the *AbcMidi* corpus) but frequent in this package as identifiers. This dictionary is used by the *valid_term* function, which allows the *soft_split* function to handle (single and multi-word) domain abbreviations found in identifiers.

8.2.4 Other Dictionaries

Since the custom corpus-based dictionary goal is to capture application domain specific vocabulary, more general programming common abbreviations and acronyms may not be present in this dictionary. Also, documentation may not be available to create the documentation corpus. To overcome this and similar situations another set of dictionaries are being used by LIdS:

programming: includes some general programming terms and abbreviations (e.g., "*msg*" $\rightarrow$ "*message*", "*param*" $\rightarrow$ "*parameter*") that have been collected over time by the authors, it has around 110 entries.

acronyms: a set of well known and common acronyms (e.g., HTML, XML, BSD, SQL), this dictionary has around 130 entries. These acronyms are not expanded to the full set of terms in this dictionary. In order to keep the setting for the experimental verification described in Section 9.3 as close as to the one described in [58], the acronyms dictionary provided by Guerrouj *et al.* was used to create the first version of this dictionary.

abbreviations: includes common abbreviations general to most programs (e.g., "*ctrl*" $\rightarrow$ "*control*", "*buff*" $\rightarrow$ "*buffer*"). Again, to keep the setting as close as possible to the one described in [58], the abbreviations dictionary provided by Guerrouj *et al.* is used. It has around 190 entries.

general: dictionary for the english language, the dictionary provided by *aspell*[11] is used. It has around 120 000 entries.

8.3 From Software File Tree to Ontologies: DMOSS Toolkit

Typically a software system is not a single a file, but a collection of files, including content not written using programming languages. The starting point is often an heterogenous file tree of files, including the source code and other assorted files.

[11]Available from: `http://aspell.net` (Last accessed: 12-02-2014).

The Documentation Mining Open Software Systems (DMOSS) toolkit main goal is to provide a set of tools that systematically process a software package, and produce a report with conclusions about the quality of the non-source code content found. This includes analyzing all the natural language text available in the documentation, comments in the code, and other non-source code files typically found in packages.

The main design goals for DMOSS are:

- Develop small autonomous tools, so that they can be useful in other contexts or environments. Applications that use these smaller tools are modular, so that new tools can be added without any additional effort, just like typical plugins.

- Many tools in DMOSS take advantage of known algorithms and techniques (for example the file *processors*). The main engine in the toolkit needs to be based on the usage of plugins, so that new processors and similar utilities can be added and improved easily.

- Represent the software package as a tree. This allows the implementation of the analysis algorithms as a set of tree traversals (for more details on tree data structures and traversal algorithms see e.g. [53, 43]). Keeping the implementation of the specific analysis algorithms self-contained in the plugins.

Let us stress again the importance regarding the way DMOSS represents a software package: an annotated tree. In this tree, nodes represent files and directories, and edges describe the hierarchical structure of the package. An example tree is illustrated in Fig. 8.4, for the *tree* software package. This tree is automatically generated by the toolkit, by traversing the filesystem file hierarchy recursively, adding nodes for each file and directory visited. For each node that is a known documentation format (e.g. man(ual) page, HTML) the plain text content is extracted and added to the corresponding node as an attribute[12].

Once this tree is available, the task of processing a software package is divided in two tree traversals:

[12]Known types are defined in the toolkit by a set of regular expressions that match file extensions, and a dispatch table that contains functions definitions for text extraction for each type.

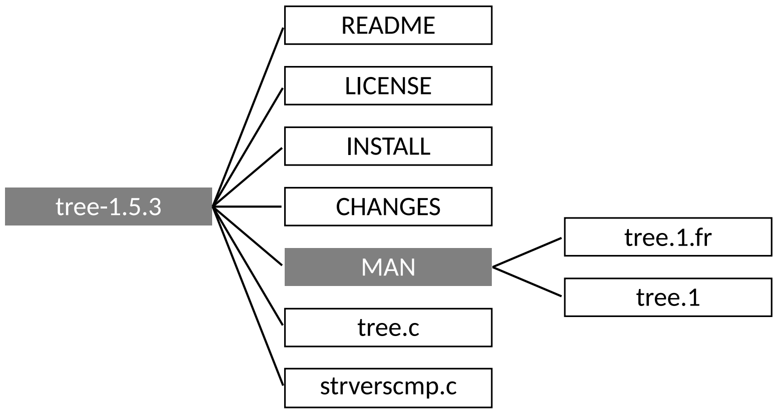


Figure 8.4: DMOSS software package tree like structure representation.

1. During the first pass the goal is to gather information about files and their content. Each plugin *processor* function is executed for each file individually, and the computed metrics are stored in the tree as node attributes.

2. In the second pass, results are aggregated (or reduced). For each directory node, the available features are reduced to a single result. In the end, the tree root node (the package top directory) contains the results of processing the entire package.

After these traversals, a final report with conclusions is produced using the data stored in the annotated tree. The plugins that perform the actual analysis and build conclusions need to implement three functions to be used in both tree traversals:

1. A *processor*, which is responsible for gathering information about a file and produce a set of features (a metric can be measured using one or more features) about its content. These features are stored in the tree as node attributes:

   $processor :: Node \rightarrow [Feature]$

2. A *reducer*, which is responsible for reducing features to produce either intermediate or final results. Results can be a single feature or a set of features:

   $reducer :: [Feature] \rightarrow [Feature]$

3. Finally, a *reporter*, which is responsible for building the final report given a set of features:

   $$reporter :: [Feature] \rightarrow Report$$

The only strictly required function is the *processor*, as there are default implementations for the other two functions, which are used when a plugin does not provide them. The default *reducer* reduces attributes using string concatenation or arithmetic sum depending on value type. The default *reporter* uses a pre-defined template to produce a simple report.

A feature is defined as a pair, consisting of a name and a value:

$$\begin{aligned}&\textbf{data}\ Feature = Feature\ \{\ name\ ::\ String,\ val\ ::\ Value\ \}\\&\textbf{data}\ Value\ \ \ = Int\ n \mid String\ s \mid \ldots\end{aligned}$$

where,

- *name* is the attribute identifier (a string);
- *val* can be an atomic value (a string or number for example), or a structured set for storing complex data structures.

A node in the tree is defined as:

$$\begin{aligned}\textbf{data}\ Node = Node\ \{&\\&path\ ::\ String,\\&isFile\ ::\ Bool,\\&text\ ::\ String,\\&features\ ::\ [Feature]\\\}&\end{aligned}$$

where,

- *path* stores the file name and its path;

- *isFile* is a boolean value stating if this node is a file or a directory;
- *text* stores the natural language text found in the file, and is computed before starting the tree traversal stages. During this step, content is extracted from files written in known formats (e.g. HTML, POD, *man*) and stored in the tree as plain text.
- *features* stores a set of features for each node.

8.3.1 First Pass: Gathering Information

When traversing the tree, each file node is processed, i.e. the files represented by each node are processed. These nodes are processed in two steps:

1. Determine the file type, either using its full media type [66], or using heuristics, like the file header or extension. The result of this step is the creation of an attribute named *type* with the corresponding file type (for example *plain/text*, *text/xml* or *text/html*) as its value.
2. Given the node *type* and a list of available *processors* for each file type[13] the next step is to process the current file with all the available processors that support it, and store each processor resulting feature as a new node attribute.

This workflow is executed for every single node that represents a file, and is illustrated in Algorithm 6[14]. The final result is a tree with a set of metrics calculated for each file node, and stored as attributes (including the file type).

Processors

compute attributes values for file nodes, the toolkit provides an heterogeneous set of processors. Each processor typically handles a single file, and produces a result that is

[13]The toolkit provides a set of plugins that implement several *processors* (more details in Section 8.3.4), and new plugins can be easily added.

[14]Algorithm 6 and 8 loop over the relevant nodes in the tree are simplified for illustration purposes, the actual implementation follows the traditional tree traversal algorithms described in the literature.

Algorithm 6 Decorate tree with *processor*s results.

```
Input: $tree$ : tree representing package content
Input: $processors$ : set of processors indexed by type
Output: list of pairs split, term
 1: for all $node \in tree : node.isFile = True$ do
 2:     $type \leftarrow typeOf(node)$                              // compute file type
 3:     for all $processor \in processors(type)$ do
 4:         $node.features.push(processor(node))$        // add resulting feature set to node
 5:     end for
 6: end for
 7: return $tree$
```

stored as an attribute in the tree. For example, the spell checker processor computes the total number of words in a text file, and the total number of words found in the dictionary (see Algorithm 7), the dictionary used is *aspell*[15].

Algorithm 7 Processor example: Spell Checker.

```
Input: $node$ : Node representing the file being processed
Output: New feature set to be added to the node
 1: $total \leftarrow 0$
 2: $found \leftarrow 0$
 3: for all $word \in split\_words(node.text)$ do
 4:     if $dictionary.valid(word)$ then
 5:         $found \leftarrow found + 1$                    //  word was found in the dictionary
 6:     end if
 7:     $total \leftarrow total + 1$
 8: end for
 9: $f_1 \leftarrow Feature($"spellCheckerTotal"$, total)$
10: $f_2 \leftarrow Feature($"spellCheckerFound"$, found)$
11: return $[f_1, f_2]$
```

New processors can be added or *plugged in* at any time. Each plugin is also responsible for defining which file types it wants to process. This information is used to build a dispatch table before any transversal, which keeps the traversing tree engine agnostic to which processors are available, and which files to process.

[15]Available from: `http://aspell.net` (Last accessed: 12-02-2014).

8.3.2 Second Pass: Reducing Results

The goal of the second tree traversal (depth-first [87]) is to produce the final feature set. This is achieved by combining (or reducing) the intermediate results for every level of the package tree, and adding new attributes (typically to the directories nodes) that store the result of combining the features for each subtree. Every plugin may provide a specific function to combine results. The default method for combining intermediate results is plain string concatenation, or arithmetic addition (depending on value type).

For example, the combining function for the spell checker processor is to add the total number of words, and the total number of words not found for the files on each directory. This means that after this pass, the `MAN` node (illustrated in Figure 8.4, which represents the file-system `man/` directory) has an attribute that stores the result of combining the spell checker processor result for files `man.1` and `man.1.fr`[16]. Later, this attribute value is used to calculate the totals for the package, stored in the top level directory.

Figure 8.5 illustrates this process for an arbitrary metric. The algorithm is also described in Algorithm 8.

Algorithm 8 Traversing the annotated tree, depth-first, to reduce nodes features.

```
Input: $tree$ : Tree representing package content
Input: $reducers$ : Set of available reducers
Output: Tree after adding reducers results to nodes as features
1: for all $node \in transverse\_depth\_first(tree)$ : $node.isFile = False$ do
2:     for all $reducer \in reducers$ do
3:         $features \leftarrow ...$                    // get features from children node set
4:         $result \leftarrow reduce(features)$         // call reduce function
5:         $node.features.push(result)$                 // add reducing result features to node
6:     end for
7: end for
8: return $tree$
```

[16]Although the two files are written in different languages the plugin uses a language identification algorithm before the spell checking task.

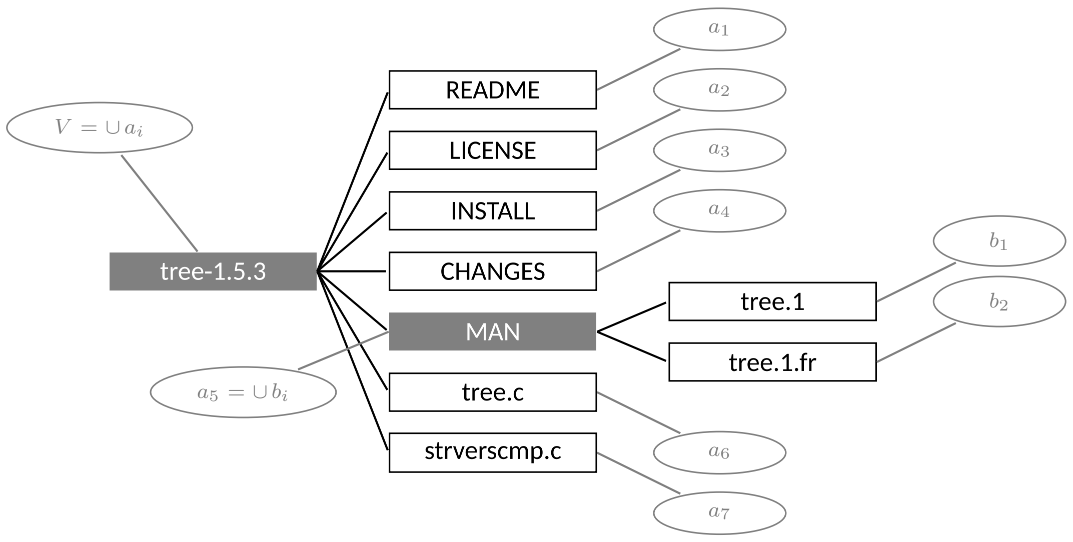


Figure 8.5: DMOSS software package tree like structure representation.

Reducers

are used to reduce intermediate results, *i.e.*, combine the results found by the processors in the subtree of the node currently being processed, and add this reduced result to the current node as new features. Algorithm 9 illustrates the reducer for the spell checker example.

8.3.3 Building Reports

After the package is processed, a tree representing the package is available. This tree is decorated with a set of features per node, that convey all the results gathered from processing each file node, and also the conclusions taken for each processor. This information is stored in the tree using attributes. The toolkit provides a tool that can build reports in several formats including HTML, and ontology style graphs in GraphViz[17] notation. Examples of a HTML formatted reports are illustrated in Figure 8.10.

After the tree traversal stages, the set of reporters functions can be used to produce a final report. In this step all the reporters functions are executed, and the results are aggregated to build the final report (Algorithm 10). Besides these structured reports,

[17]Available from: `http://www.graphviz.org/` (Last accessed: 12-02-2014).

Algorithm 9 Reducer example: Spell Checker.

```
Input: $features$ : Set of node features
Output: New set of features to be added to the node
 1: $total \leftarrow 0$
 2: $found \leftarrow 0$
 3: for all $feature \in features$ do
 4:     if $feature.name =$ "SpellCheckerTotal" then
 5:         $total \leftarrow total + feature.value$
 6:     end if
 7:     if $feature.name =$ "SpellCheckerFound" then
 8:         $found \leftarrow found + feature.value$
 9:     end if
10: end for
11: $f_1 \leftarrow Feature($"spellCheckerTotal"$, total)$
12: $f_2 \leftarrow Feature($"spellCheckerFound"$, found)$
13: return $[f_1, f_2]$
```

Algorithm 10 Build final reports.

```
Input: $tree$ : Tree representing package content
Input: $reporters$ : Set of available reporters
Output: Final HTML report
1: $report \leftarrow$ ""                          // start with an empty report
2: $features \leftarrow ...$                       // // collect features set from tree root
3: for all $reporter \in reporters$ do
4:     $curr \leftarrow reporter(features)$        // build each individual report
5:     $report \leftarrow concat(report, curr)$    // concatenate individual reports
6: end for
7: return $report$
```

the full tree is available as an associative array to be further processed by any other tool or application.

Reporters

Reporters process a specific set of features about the package and produce custom reports. They are mainly used for producing reports that require post processing computations to achieve the intended result in the report (averages computations, for example). Reporters usually compute a final grade for a specific analyzed feature (the formula for computing the grade is another responsibility of a reporter function). Reporters' output is usually a snippet of HTML, built using a default set of templates. The complete tree is always available inside any reporter function, to gather any required information to build a more detailed report.

8.3.4 Toolkit Plugins

This section gives a brief overview of the plugins currently included in the DMOSS toolkit, and used to produce the reports illustrated in the next section.

Validate Links gathers links found for known protocols (e.g. HTTP, FTP), and checks if the link is still working. To validate the link a simple request is made, and if a successful reply is received the link is considered valid. The plugin rates the package better, as more valid links are found.

Spell Checker performs word spell checking, using a general purpose dictionary, for every word found in the documentation, and other non-source code files. It automatically detect the language used, and chooses the dictionary accordingly. The dictionary engine used is *aspell*.

Verify Licenses gather possible license information found in the software package. It can verify some common open source license (e.g. GNU General Public License [18])

[18]General information about GNU licenses available from: `http://www.gnu.org/licenses/` (Last accessed: 12-02-2014).

used in software packages. Since this plugin was initially created for open source software, it grades the package if two criteria are met: (1) license information is found, (2) a known open source license was found. Of course, more licenses can be added.

Comment Lines counts the total number of source code lines, and the total number of comment lines found in the package source files. While the number of comment lines per lines of source code is above 19% [19] this plugin grades the files positively.

Identifiers Found in Docs attempts to measure documentation source code coverage. It measures the number of program identifiers (strings used as functions or variables names) found in the documentation. Mainly because most documentation formats (e.g. DoxyGen) use these strings to relate the documentation snippets with the source code. This can help to have an idea of which source code is covered by documentation.

Automatic Classification is used to automatically classify the software package using SourceForge taxonomy[20]. The classification algorithm is straight-forward, it measures the distance between the words found in the documentation (which are valid according to the english dictionary), and the terms in the taxonomy. A more robust version of this plugin should use a well established classification approach like Support Vector Machine (SVM) [69] or Naive Bayes related algorithms [168]. The correct automatic classification of the package can be a positive characteristic, because there's a good probability that the vocabulary used in the documentation is close with the vocabulary stored in the taxonomy index, which is usually in line with the vocabulary used in the software area of interest. This plugin grade is directly related with two factors: (1) classification was possible, and (2) the degree of confidence (distance) on the computed classification.

Changes Verification is used to analyze changes information, if available. This file typically describes major releases done for the software, including the date for the

[19]This particular threshold was chosen based on a study by Arafat *et al.* about source code comments practices in open source projects [8].

[20]Available from: `http://sourceforge.net/` (Last accessed: 12-02-2014).

release, and a list of topics that describe the major changes. The current goal of this plugin is to discover the date of the last release, and compare it to the current year. This is used in the report to grade positively packages with more recent releases. The lower possible grade is given when the set of regular expressions that are used to parse the file content are not able to return any information. Although, there is no standard format for these files, this can be an indicator that maybe some of the best practices were not followed.

New plugins can be easily added to analyze other features or characteristics of the package. New plugins just need to define the required functions as described in the previous sections. The set of plugins available in DMOSS do not cover all the measures and metrics described in the literature, and new analysis are proposed every day. One of the major goals of the proposed methodology is to provide the community with a framework that allows the quick development of new measurements, and integration with currently available ones.

8.3.5 Traits *Versus* Plugins

In line with the previous discussion, the described plugins only provide information to assess a limited number of features. These features are related with previously discussed traits, that tend to have a considerable weight in the overall package quality.

Plugins like *Spell Checker* are close related to readability, the increasing number of spelling errors can introduce noise in the text, making it harder to read or understand. Other features that can be measured to increase readability coverage are, for example, excessive use of acronyms and abbreviations, or punctuation analysis.

Completeness is a trait concerned with how much of the source code, and related concepts, are covered by documentation. The *Comment Lines* plugin measures the ratio of lines of comments found per lines of code, a low ratio may suggest that there are big portions of undocumented code. This feature is close related to the *Identifiers Found in Docs* plugin measure, by finding function definitions that lack corresponding documentation. This can be crucial when documentation generation systems are used (e.g. Doxygen). Both these traits are useful during software development, measures can be used during development stages to make sure documentation is keeping up with code implementation.

Plugins like *Validate Links* are more related to the actuality trait, because they provide clues that some elements in the documentation may be out of date, by finding links that are no longer active or have been moved elsewhere. Another possible clue can be given by the *Changes Verification* plugin, if a software package is stalled in time, i.e has not released a new version in the last years, it might be prone to have outdated content.

Other plugins tend to be less subjective, and provide accurate information about a specific propriety, and are not closely related to these traits. For example the *Verify Licenses* plugin attempts to answer a specific question: “*Which license is the software package released under?*”. This is a relevant detail in the context of open source software.

8.4 Generalizing the Creation and Population of Ontologies: Conclave Utils

While Conclave OTK toolkit is designed to be as general as possible, i.e., to allow the implementation of arbitrary applications that handle data using ontologies, some operations and features directly concerned with software engineering are required by the Conclave environment. This set of particular features are distributed with the Conclave Utils library. This allows keeping OTK as agnostic as possible, and provide an independent library to be developed for specific operations in the scope of this work. This section describes some major features provided by this library.

8.4.1 Initializing Ontologies

When a software system is added to the Conclave environment, the first task that is undertaken is creating the corresponding ontologies for the new system. These model the various domains of knowledge, as described in Chapter 6. But before starting to populate the ontology, a minimal bootstrap of domain is required. For example, when creating a new ontology to store a program domain, the initial ontology already has some classes defined (e.g., *Function*, Variable), and some available proprieties (e.g., *hasString*, *hasLineBegin*), etc. The information that bootstraps the ontologies is available in the Conclave Utils library. This information is stored in a set of Template Toolkit (TT) templates, so that information is easily updated, and new models are easily added.

The library provides an easy to use function called *conc_base_ontologies*, that given as argument the type of ontology that is being bootstrapped, it returns the information in RDF/XML format, which can be immediately conveyed to the actual ontology using Conclave OTK method *init*. The *init* method performs the initialization of an ontology and has one optional argument, the information to bootstrap the new ontology. The result of the *conc_base_ontologies* can be directly composed with the *init* method. The library provides a command line tool called `conc-otk-init` that takes advantage of this simple composition to quickly initialize new ontologies. This tool takes two arguments: (i) the base name for the ontology; and, (ii) the template to use for bootstrapping the new ontology.

For example, to initialize the program ontology for the *tree* software package the following command is used:

```
$ conc-otk-init http://conclave/tree/program program
Initialized: http://conclave/tree/program
```

In practice, the ontology name refers to its unique URI.

Base ontologies are distributed using templates, so they can be adjusted in runtime if required. For the sake of completeness, and as an example, Appendix D illustrates the template set used to bootstrap the program ontology.

8.4.2 Populating Ontologies

Once the ontology is properly initialized, and ready to use, the next step is to start populating it with information. Usually this information is available in heterogenous resources, and needs to be conveyed to the ontology semantics. Conclave Utils provides a set of functions that undertake this task, and features `conc-otk-load`, a command tool to populate ontologies using a well defined set of resources. An example is loading a resource created used the `clang-conclave` tool, this is illustrated in Section 8.1. LIdS, introduced in a previous section, is another example of a tool that produces a resource, `conc-otk-load` also provides a feature for loading sets of terms for identifiers.

The functions responsible for populating ontologies use Conclave OTK API for handling information, this allows a fast and clear implementation of the operations that convey the available data. The Conclave Utils is easy to extend to feature more functions for loading arbitrary resources to ontologies.

8.5 The Conclave Environment

The Conclave environment combines the tools described in this chapter, and the approaches described in Chapter 6 and Chapter 7 to provide single interface set of software analysis tools. The interface is provided via web so that it can be accessed easily, and all the tools are readily available to use server-side.

The main system workflow is divided in three main stages: (i) collecting data; (ii) processing collected data and loading ontologies; and, (iii) reasoning about data in the ontologies, and providing views of computed information. All the tools implemented in the context of this system are modular (or work as plugins), and some provide web-services, so that they can be used as standalone applications, or composed together to create more complex applications or other workflows.

Collecting Data

This is the first stage of the main workflow; its goal is to collect data from a software package, and any kind of problem specification if available. It takes as input the complete package (and other available documents) and produces as output an heterogeneous collection of resources. The processing tools involved in this stage can use different type of analysis: static source code analysis (e.g. parsing code to extract identifiers and static call graphs), dynamic analysis (e.g. execution traces), etc. Some of the tools described earlier in this chapter contribute to the data collecting process, but other tools, implemented outside the scope of this work are also used.

Normalizing Information, Populating Ontologies

The main goal of this stage is to convey the data collected during the previous stage into the system ontologies. The input of this stage is a collection of resources, and the output is a set of populated ontologies. Usually three ontologies are populated for each software package:

Program Ontology: abstract representation of some key program elements (e.g. methods, functions, variables, classes);

Problem Ontology: concepts and relations in the application domain;

World $\triangle$ Ontology: runtime effects of executing the program (e.g. program run traces).

The approach used during this stage follows the method described in Chapter 6, the data produced by arbitrary tools is conveyed to the ontology using Conclave OTK. But there are some important details to emphasize and discuss. The first one is the format and technology chosen to store the ontologies. A RDF based triple-store technology was adopted to store the data. This allowed for a scalable and efficient method for performing storing and querying operations, and also allows to export the data in several community accepted ontology formats (e.g. OWL, RDF/XML, Turtle) [74, 63]. Querying facilities are also readily available; for instance, SPARQL is a querying domain specific language for RDF triple-stores [124, 117].

Although these technologies provide scalable and efficient environments for handling information, development wise, they are far from the abstraction desired by the applications level implementation. To overcome this problem the Conclave OTK[21] was implemented, which provides an abstraction layer on top of the RDF technology, to develop ontology-*aware* applications. In practice, when applications developers wish to perform an ontology related operation, instead of using triple-store low level primitives,

[21]Implemented as a set of libraries for the Perl programming language.

they can use the abstraction layer. To motivate for the development of this abstract framework, consider the modern Object-Relational Mappers (ORM) in the context of relational databases. Which provide an abstraction layer and interface for programming languages to handle data (stored in databases) as objects, allowing the development of applications regardless of the underlying database technology used [71].

Reasoning and Views

During this stage more knowledge about the system is build and provided to the system end-user. The tools in this stage use as input the ontologies built during the previous stage, and generally fall in one of the two categories, either they: (i) process information to compute new information and knowledge about the system – usually in this case the tool output is new content added to the ontologies; or (ii) information or knowledge suitable for visualization is built – in this particular case the final output of the tool is a view for the package system. The main features provided by the Conclave environment are provided using the Concept Mapper described in Chapter 7, which includes searching and features the creation of mappings between available models.

8.6 Conclave Tour

The Conclave website is freely available (still under development) at:

`http://conclave.di.uminho.pt/`

This chapter briefly introduces the application from the user point of view, and illustrates some previously discussed features. Figure 8.6 illustrates the web interface front page, the system is divided in blocks, and most of the applications use resources produced by other blocks.

The ontology browser facility, illustrated in Figure 8.7, can be used to browse the information in any ontology available in the system. It also provides exporting mechanisms, that allow the user to locally store the ontology in some well know formats (e.g., OWL, RDF).

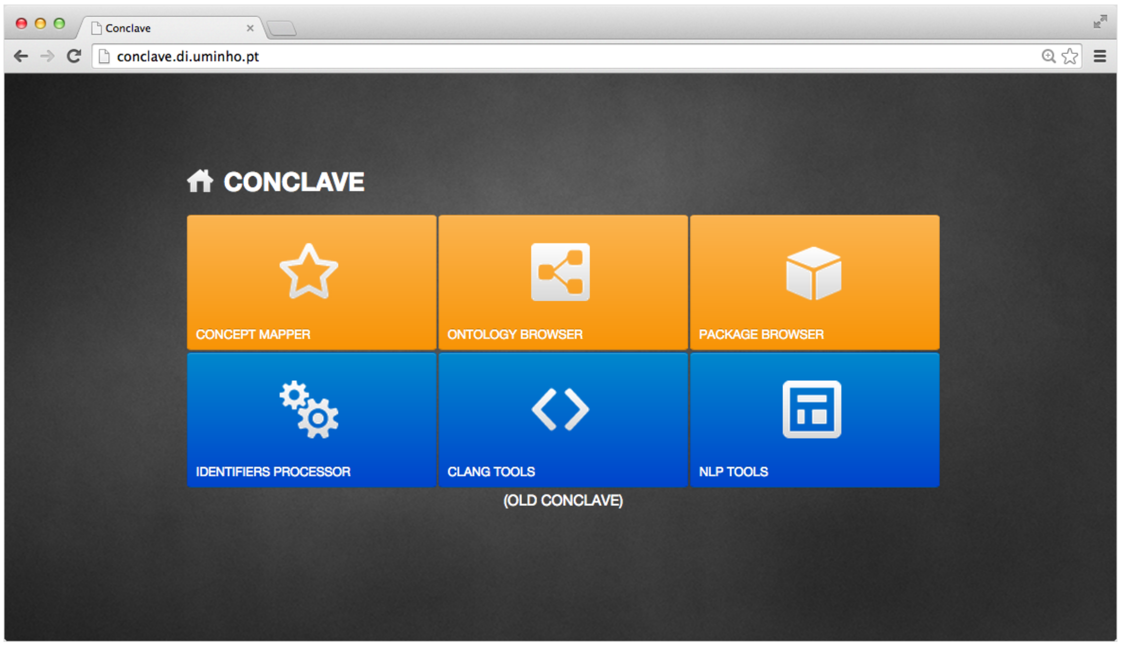
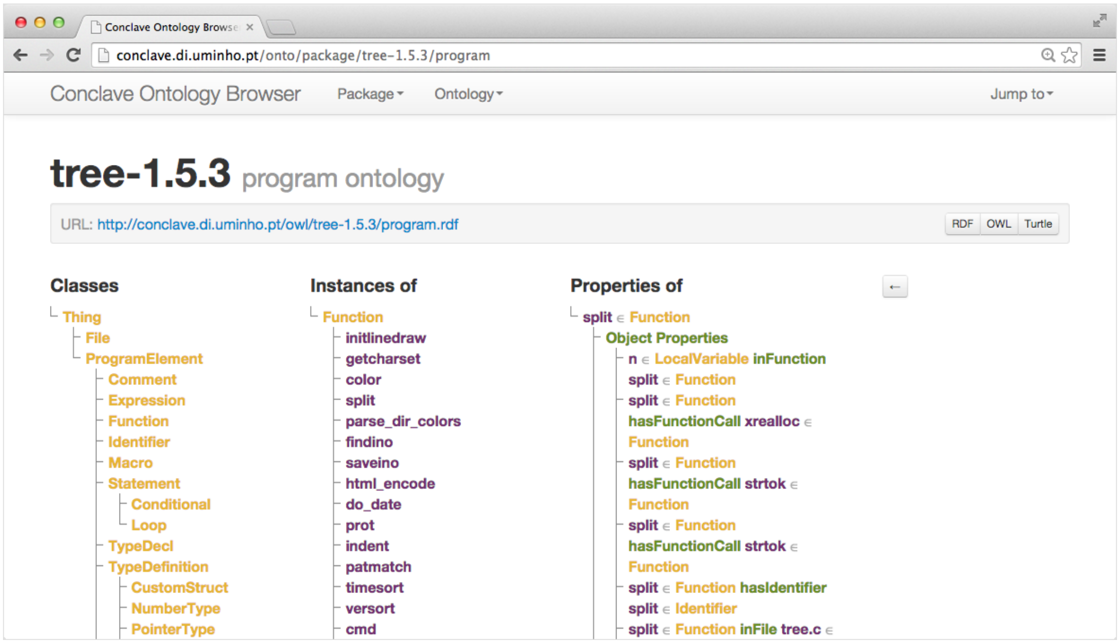


Figure 8.6: Conclave system web interface front page, main applications are divided in blocks.

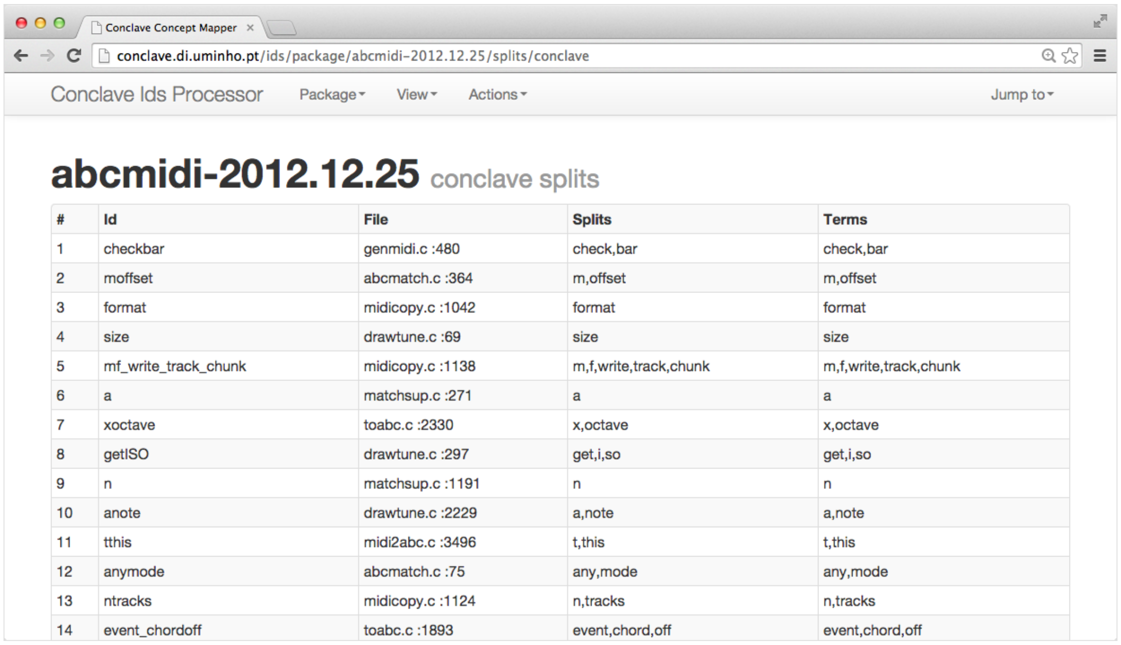

Conclave Ontology Browse
conclave.di.uminho.pt/onto/package/tree-1.5.3/program
Conclave Ontology Browser
Package
Ontology
Jump to
RDF
OWL
Turtle

tree-1.5.3 program ontology

URL: http://conclave.di.uminho.pt/owl/tree-1.5.3/program.rdf

```
Classes                 Instances of          Properties of
└ Thing                 └ Function            └ split ∊ Function
  ├ File                  ├ initlinedraw        ├ Object Properties
  └ ProgramElement        ├ getcharset            ├ n ∊ LocalVariable inFunction
    ├ Comment             ├ color                 ├ split ∊ Function
    ├ Expression          ├ split                 ├ split ∊ Function
    ├ Function            ├ parse_dir_colors      ├ hasFunctionCall xrealloc ∊
    ├ Identifier          ├ findino                 Function
    ├ Macro               ├ saveino               ├ split ∊ Function
    ├ Statement           ├ html_encode           ├ hasFunctionCall strtok ∊
    │ ├ Conditional       ├ do_date                 Function
    │ └ Loop              ├ prot                  ├ split ∊ Function
    ├ TypeDecl            ├ indent                ├ hasFunctionCall strtok ∊
    ├ TypeDefinition      ├ patmatch                Function
    │ ├ CustomStruct      ├ timesort              ├ split ∊ Function hasIdentifier
    │ ├ NumberType        ├ versort               ├ split ∊ Identifier
    │ ├ PointerType       ├ cmd                   ├ split ∊ Function inFile tree.c ∊
```

Figure 8.7: Conclave ontology browser, illustration of the program ontology for the tree package.



abcmidi-2012.12.25 conclave splits

#	Id	File	Splits	Terms
1	checkbar	genmidi.c :480	check,bar	check,bar
2	moffset	abcmatch.c :364	m,offset	m,offset
3	format	midicopy.c :1042	format	format
4	size	drawtune.c :69	size	size
5	mf_write_track_chunk	midicopy.c :1138	m,f,write,track,chunk	m,f,write,track,chunk
6	a	matchsup.c :271	a	a
7	xoctave	toabc.c :2330	x,octave	x,octave
8	getISO	drawtune.c :297	get,i,so	get,i,so
9	n	matchsup.c :1191	n	n
10	anote	drawtune.c :2229	a,note	a,note
11	tthis	midi2abc.c :3496	t,this	t,this
12	anymode	abcmatch.c :75	any,mode	any,mode
13	ntracks	midicopy.c :1124	n,tracks	n,tracks
14	event_chordoff	toabc.c :1893	event,chord,off	event,chord,off

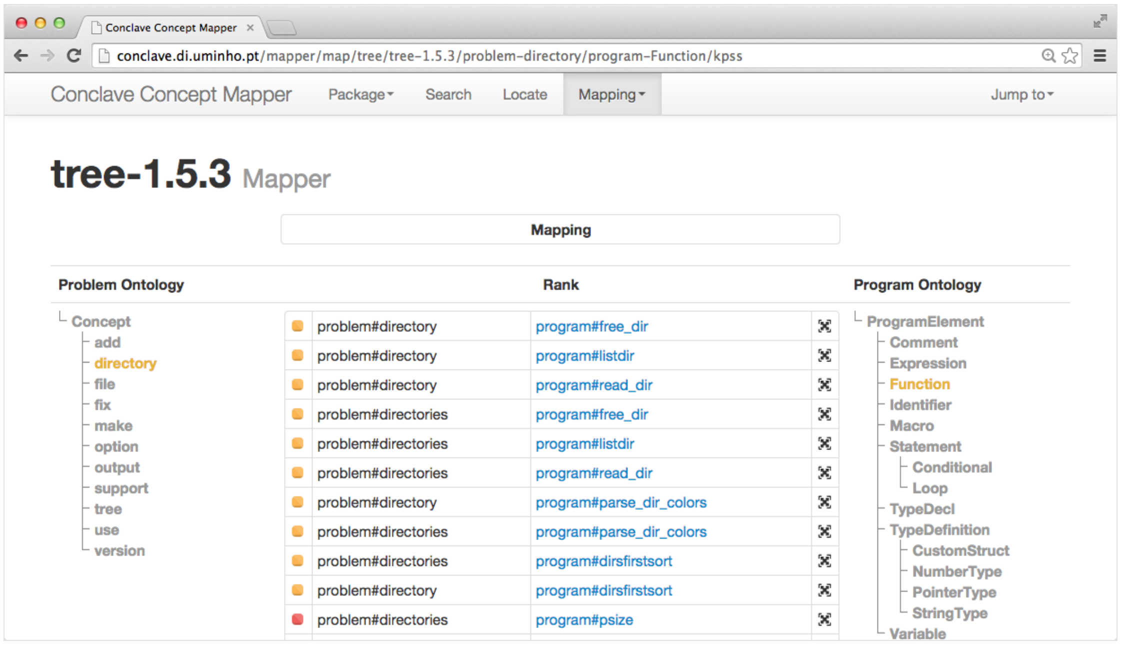
Figure 8.8: Conclave program identifiers processing using LIdS.

Conclave uses LIdS described in Section 8.2 to process identifiers, i.e. to split and expand abbreviations to more meaningful terms. Figure 8.8 illustrates these results.

Figure 8.9 illustrates a mapping between elements in different domains (ontologies). On the left the Problem Ontology is used to constrain the concepts being searched (directory), on the right the program ontology is used to constrain the range of program elements being analyzed (functions), and in the center the resulting rank sorted by relevance. The ranking is built using the approach described in Chapter 7 in the background. Although the interface provides some features to tinker mappings to specific needs, for example, clicking a class of the ontology to constrain rankings to instances of that class only, the user can built more complex mappings using the query language described in Section 7.1.

Figure 8.10 illustrates some reports created by DMOSS, from open source analyzed packages: (8.10a) *aspell 0.60*, a spell checker; (8.10b) *wget 1.9.1*[22], a package for retrieving files using several protocols; (8.10c) *tree 1.5.3*, a recursive directory listing com-

[22]Available from: `http://www.gnu.org/software/wget/` (Last accessed: 12-02-2014).



Conclave Concept Mapper

conclave.di.uminho.pt/mapper/map/tree/tree-1.5.3/problem-directory/program-Function/kpss

Conclave Concept Mapper | Package | Search | Locate | Mapping | Jump to

tree-1.5.3 Mapper

Mapping

Problem Ontology

- Concept
  - add
  - directory
  - file
  - fix
  - make
  - option
  - output
  - support
  - tree
  - use
  - version

Rank

🟠	problem#directory	program#free_dir
🟠	problem#directory	program#listdir
🟠	problem#directory	program#read_dir
🟠	problem#directories	program#free_dir
🟠	problem#directories	program#listdir
🟠	problem#directories	program#read_dir
🟠	problem#directory	program#parse_dir_colors
🟠	problem#directories	program#parse_dir_colors
🟠	problem#directories	program#dirsfirstsort
🟠	problem#directory	program#dirsfirstsort
🔴	problem#directories	program#psize

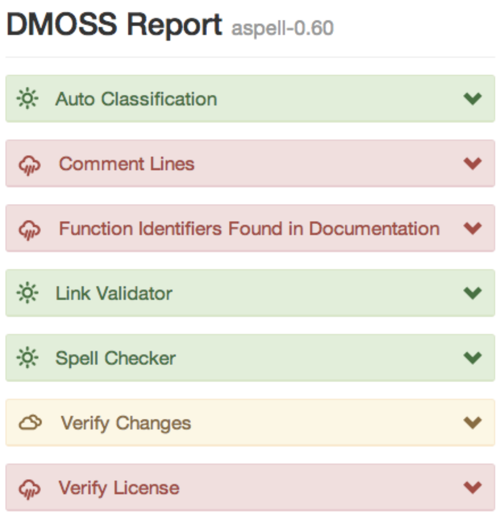
Program Ontology

- ProgramElement
  - Comment
  - Expression
  - Function
  - Identifier
  - Macro
  - Statement
    - Conditional
    - Loop
  - TypeDecl
  - TypeDefinition
    - CustomStruct
    - NumberType
    - PointerType
    - StringType
  - Variable

Figure 8.9: A mapping produced by Concept Mapper using the Conclave environment web interface.

mand; and, (8.10d) *grep 2.9*[23], a tool for searching patterns in plain text files.

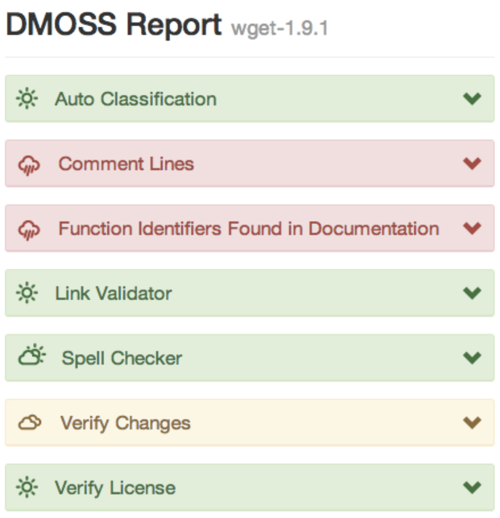
[23]Available from: `http://www.gnu.org/software/grep/` (Last accessed: 12-02-2014).



DMOSS Report aspell-0.60

- Auto Classification
- Comment Lines
- Function Identifiers Found in Documentation
- Link Validator
- Spell Checker
- Verify Changes
- Verify License

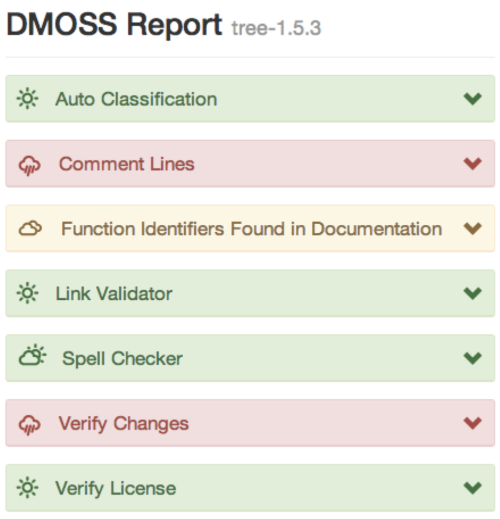
(a) *aspell 0.60*



DMOSS Report wget-1.9.1

- Auto Classification
- Comment Lines
- Function Identifiers Found in Documentation
- Link Validator
- Spell Checker
- Verify Changes
- Verify License

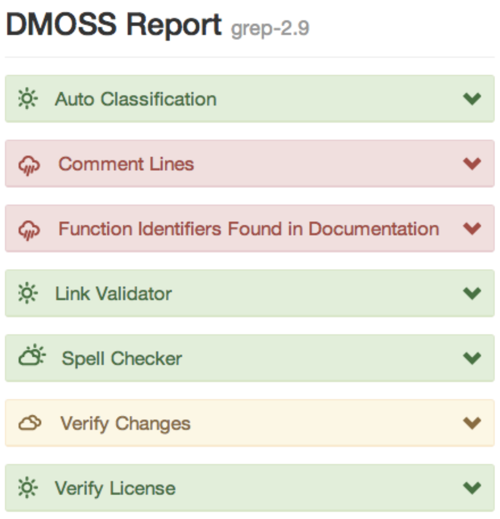
(b) *wget 1.9.1*



DMOSS Report tree-1.5.3

- Auto Classification
- Comment Lines
- Function Identifiers Found in Documentation
- Link Validator
- Spell Checker
- Verify Changes
- Verify License

(c) *tree 1.5.3*



DMOSS Report grep-2.9

- Auto Classification
- Comment Lines
- Function Identifiers Found in Documentation
- Link Validator
- Spell Checker
- Verify Changes
- Verify License

(d) *grep 2.9*

Figure 8.10: Screenshots of HTML reports produced using DMOSS for several software packages, including analyzed features and corresponding grades.

Summary

- Conclave provides an environment for software analysis, providing a set of tools and applications to aid in program comprehension activities. Searching and creating mappings between different domains, both using a domain specific language to describe search queries, enchace feature location activities.

- The knowledge domains about the software (including the source code) are represented using ontologies following the methodology described in Chapter 6. Searching code, and mappings between domains are built using the approach described in Chapter 7. Figure 8.11 illustrates how previously described techniques are combined to devise the system main workflow.

- Most of the tools described in this chapter act in specific areas of PC (e.g., concept location, documentation analysis), but they are still modular enough to be used as stand-alone applications outside of Conclave scope to implement other approaches. Some applications are based on plugins, that are extended by simply adding a new plugin. Other applications provide a web-service to be easily composed in other workflows without being installed. Generic toolkits are provided as libraries to devise and implement application, recall Chapter 1 for references.

- Concerning PC, source code vocabulary normalization, including multi-term splitting and abbreviation expansion, is a crucial step for most feature location (and other) activities; the application (or problem domain) helps the understanding process (e.g., providing a taxonomy for devising search queries, relating source code with application concepts); the synergy between domains allows the creation of views of the program that emphasize details of interest when performing concrete software maintenance or evolution tasks; and, the automatic creation of resources (e.g., program ontology, application ontology) reduces the initial overhead of adopting solutions in the family of Conclave, while keeping the option to tinker and manually improve the created artifacts.

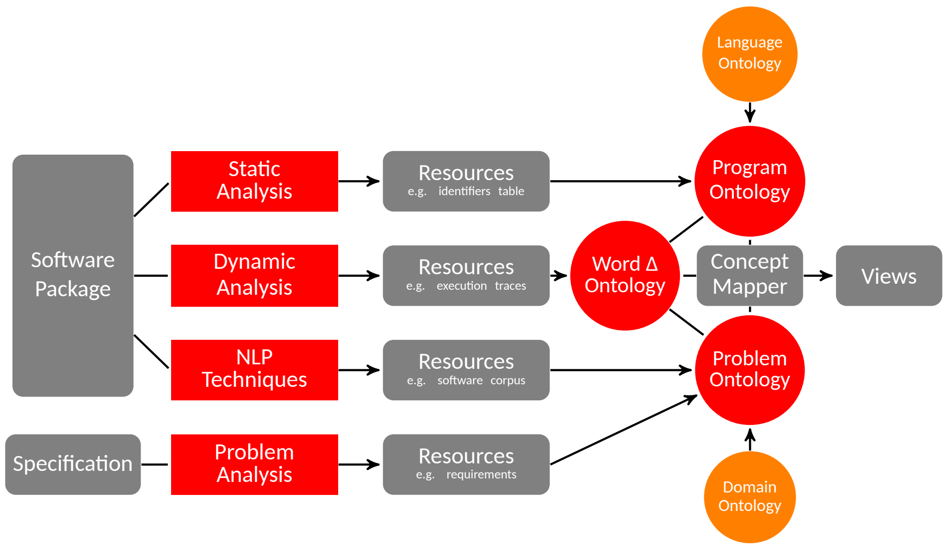


Figure 8.11: Conclave main workflow, combining techniques, tools, and approaches discussed in Chapter 6, 7 and 8.

Chapter 9

Experimental Validation

Divide et impera [a]

Philip II of Macedon

[a] "Divide and Rule" or "Divide and Conquer".

It is commonly accepted that experimental validation is a requirement to support new techniques or approaches for a given problem [169]. Sometimes authors do not use the correct accuracy, or do not present enough results, to convince others of the benefits of their work. To try to overcome such shortcomings, and given the complexity of Conclave (complex because it explores and combines a heterogenous set of applications and tools) the process of experimenting was often partitioned, i.e., when possible each application is experimented and validated individually. This also allows to use experimental validation as a tool to measure improvements on application developments. Comparing applications with other approaches currently available, besides measuring tools effectiveness, also pushes the tools state of maturity forward.

Choosing case studies is required for some experiences. The case studies should be complex enough to illustrate key aspects of the new methods, tools or techniques, but should also be simple enough for everyone to understand what is going on. The time devoted to the implementation of the tools in the context of research is sometimes not enough to develop mature and finished programs that can deal with every aspect.

Many times the tools used to demonstrate or validate results are still in a stage of heavy development and may not cover all the key points of complex case studies. This is another important reason why case studies need to be chosen carefully [123].

A good practice, is to perform validation of internal (often smaller) components, i.e., as often as possible evaluate a specific step (component) of the general approach in an isolated setting. These smaller evaluations tend do be less complex that evaluating the entire system, which usually simplifies the process of evaluation, and result analysis.

In the context of this work several tools are being implemented, at different stages of development. The development general guide line is to create small, self-contained tools, that are composed in more complex workflows to obtain the final results. Evaluation follows a similar tenet, performing evaluation on small components of the workflow when possible. This allows gathering evidence of the benefits of the smaller tools, most of them stand-alone applications, independently.

> For example, one possible claim about Conclave Concept Mapper is that, searching features uses the complete sets of splits and full terms (because they tend to convey more semantics), instead of the original program identifiers strings, to measure semantic relatedness between elements. One valid observation is that: "*then, searching results depend on the ability of the technique adopted to split and expand identifiers to correctly perform its task*". This is absolutely correct, and that is why LIdS ability to correctly split and expand identifiers is measured independently of the remaining of the workflow, including searching features. This allows that, when the searching ability is actually empirically measured, there is already a clear idea on how well the splitting technique is performing, and the results can be analyzed accordingly.

The remaining of this chapter describes some of the experimental validations undertaken to empirically evaluate Conclave components, including the discussions about the achieved results.

9.1 kPSS Experimental Validation

Scoring functions described is previous chapters are used to measure semantic relatedness between elements. An example of such a function is the kPSS based scoring function described in Section 7.4.1. The goal of this experiment is to measure the quality of the terms found in a kPSS. The following research question (RQ) was defined:

- **RQ1:** What is the percentage of terms in a kPSS that are semantically related with the original term used to build the kPSS?

To help answering this question the following experience was devised:

Step 1: collect bag of all terms resulting from splitting and expanding the program identifiers available in Conclave program ontologies from different software packages (e.g. *jEdit*, *tree*, *AbcMidi*);

Step 2: select 1000 random terms from the list built in *Step 1*;

Step 3: build a kPSS for every term t selected in *Step 2*;

Step 4: for every synonym t_S in every kPSS built in *Step 2*, if (i) t_S is different from the original term t, (ii) t_S is a valid english word in the dictionary, and (iii) the probability associated with t_S is greater or equal to 0.1, then:

4.1: get all *synsets* for term t_S available in WordNet;

4.2: if the original term t is present in any on the *synsets* collected in *Step 4.1*, t_S is considered *related*.

Step 5: calculate percentage of *related* terms during Step 4.

This five step experience was repeated ten times, the results are presented and discussed in the next Section.

	Terms		
Set	Total	Related	% Related
0	1102	758	69%
1	1091	754	69%
2	1092	746	68%
3	1106	758	69%
4	1129	758	67%
5	1092	748	68%
6	1097	737	67%
7	1105	776	70%
8	1115	776	70%
9	1110	749	67%

Table 9.1: Percentage of terms found in WordNet *synsets*.

9.1.1 Results and Discussion

Table 9.1 describes the results of this experiment. The table presents the terms compared, and the number of terms that were related with WordNet *synsets*, achieving an average of around 69% across all sets. This means that, on average, 69% of terms found in a kPSS, have a *synset* in WordNet, that also contains both the term in the kPSS, and the original seed term to build the corresponding kPSS. This implies that on average, 69% of semantic arbitrary relations found between terms using kPSS, are also present in WordNet.

The main reasons for not relating 100% of the terms with WordNet *synsets* are: (i) *synsets* in WordNet only have the dictionary form of a verb, e.g., the kPSS for the term "*compile*" includes the terms "*compiles*" and "*compiled*", but these are not present in any *synset* for the term "*compile*" in WordNet; (ii) term inflections, e.g., the kPSS for the term "*backup*" includes the term "*backups*", but not on the WordNet *synset*. These encompass most of the terms pairs that were not found in WordNet *synsets*. With more or less effort an extra validation layer could perform some normalization of terms, but this could introduce some bias on the results, since that the exact terms are used to build candidates for creating terms and splits sets in LIdS, for example, not the normalized version of the term. Still, some were not fond due to higher semantic distances between terms, e.g., the kPSS for the term "*click*" contains the term "*button*", which is plausible, in software systems, when you perform a click you usually click on *something*, this something can be a button, but there is no *synset* in WordNet for the "*click*" term that contains the term "*button*", which is also plausible, since from a more linguistic point of view, and depending on level of abstraction, both terms represent different real world concepts.

9.2 LIdS Experimental Validation

In order to measure its ability to correctly split and expand identifiers, LIdS was applied to two open source (so that the code is readily available) software packages: *tree* (version 1.5.3)[1], that implements the *tree* command, which can be used to list directories content hierarchically; and *AbcMidi* (version 2012.12.25)[2], a package that provides a set of tools to convert Abc[3] files to the Midi format. This last package was also chosen as case study, because it acts on a specific domain (music), that has a specialized vocabulary, which terms are prone to be used as program identifiers.

Both packages are written in C. Source code written in this particular language was chosen for the experimental validation because although this language is being used for many years, there is no universal guidelines for the techniques used to compose multi-word identifiers. Typically, many combinations of techniques are used. This is not the case for other languages, like Java for example, where there is a more traditional habit to use CamelCase for example [57]. Another relevant detail about these packages is they are quite old, and different programmers have changed the code, increasing the heterogeneity of ways to create identifiers (either by composition or abbreviation). These characteristics make the splitting process (even manually) harder, but allow to better conclude about the ability of the proposed technique to generalize for other software packages.

The goal of this experiment is to measure LIdS ability to correctly split and expand program identifiers. The following research questions (RQ) were defined:

[1]Available from: `http://mama.indstate.edu/users/ice/tree/` (Last accessed: 01-10-2013).
[2]Available from: `http://abc.sourceforge.net/abcMIDI/` (Last accessed: 01-10-2013).
[3]A text notation to represent music.

- **RQ1:** What is the percentage of identifiers in a program that LIdS can correctly split?
- **RQ2:** What is the percentage of identifiers in a program that LIdS can correctly split and expand in case abbreviations were used?
- **RQ3:** What is the gain of using the custom corpus-based dictionary when splitting and expanding identifiers with LIdS?

To answer these questions the splits and expansions computed by LIdS were compared with the correct split obtained from the *oracle* (the *correct* answer, more details about *oracles* in the next section). Besides splitting correctness, the set of calculated full terms was also compared, to validate the ability of expanding strings to terms in case abbreviations were used. Accuracy, precision and recall measurements were made in three different settings:

- *HardSplit*, in this setting only LIdS *hard_split* function is called, this acts as the baseline for other comparisons. No dictionaries are used by this function.
- *Split*, in this setting LIdS *split* function is used to compute splits and term sets. In this setting the following dictionaries are used: *programming*, *acronyms*, *abbreviations*, and *general* (details about these are discussed in Section 8.2.4).
- *CorpDict*, is equivalent to the previous setting, but the custom corpus-based dictionary, automatically built for each specific package, is also included in the dictionary set.

The identifiers correct split and abbreviations expansion are required for the evaluation, the next section describes the creation of the *oracle*.

9.2.1 Creating the Oracles

The *oracle* consists of two sets for each analyzed package: (i) the correct split, the list of strings in which a multi-word identifier is correctly split; and (ii) the correct terms

			Identifiers		Oracle	
Package	Files	KLOC[4]	total	multi-word	splits	terms
tree	10	$\sim$2	235	145 (62%)	161	147
AbcMidi	86	$\sim$33	$3\,437$	$2\,142$ (62%)	$1\,644$	$1\,565$

Table 9.2: Some software packages characteristics.

set, the list of terms that compose a multi-word identifier (in this set abbreviations are expanded). Single terms are also included in the set, because although the split is straightforward, the string used can still be an abbreviation. The steps to build the *oracle* were:

1. Collect identifiers from source code files.
2. Remove identifiers with two or less characters.
3. Remove duplicate identifiers.
4. For each identifier in the set, by analyzing the source code, we: (i) manually created the correct split for the identifier (e.g., the correct split set for the identifier "*wcount*" is: $\{w, count\}$); and, (ii) manually created the correct set of intended terms by the original programmer (e.g., the set of terms for "*wcount*" is: $\{word, count\}$ - this example function counts number of words).

Table 9.2 includes some characteristics about the software packages (number of files, and number of lines of code); and about identifiers found in each package: total number of identifiers, percentage of identifiers that are composed of several terms, and the number of identifiers in each *oracle* – the set of manual *splits* and the set of full abbreviations expanded *terms*. In some cases there was not a general consensus between the authors on how to split an identifier or expand an abbreviation, these cases were not included in the *oracle*. Mainly to try to reduce the final number of errors present in the *oracle*.

[4]Thousands Lines of Code.

		tree		*AbcMidi*	
Setting	Measure	splits	terms	splits	terms
HardSplit	accuracy	0.4907	0.2721	0.3668	0.3073
Split	accuracy	0.8571	0.6939	0.8832	0.7885
CorpDict	accuracy	0.8696	0.7007	0.9300	0.8281

Table 9.3: Correct split accuracy means and correct terms accuracy means for *tree* and *AbcMidi*, in the three settings.

9.2.2 Accuracy

Accuracy measures the ability to correctly split and expand the terms that compose an identifier. The function that validates if the split is correct returns a binary value: 1 in case the set of splits (or terms) is exactly equal to the set in the *oracle*, and 0 otherwise. It correctly measures the algorithm overall accuracy, but with a small draw-back: in case the algorithm misses one correct single string (or expansion) in the set, but all the others are correct, the validation function returns 0, even if some partial result is correct.

Table 9.3 illustrates the results of validating the accuracy measure on both packages, in the different settings for each set of unique identifiers in the corresponding oracle. Using the *CorpDict* setting, LIdS achieved an accuracy mean of 0.8696 when splitting identifiers, i.e., around 87% of the identifiers were split correctly; and an accuracy mean of 0.7007 when splitting and expanding identifiers, i.e., around 70% of the identifiers were correctly split and expanded to the set of terms in the *oracle* for the *tree* package. The same setting achieved a mean of 0.9300 for splitting terms, and 0.8281 for splitting and expanding terms for the *AbcMidi* package.

9.2.3 Precision and Recall

To overcome the draw-back of measuring using the binary validation function described in the previous sub-section, a precision and recall measure of the correct splits (and terms) was also made.

For a given identifier id to split let the *oracle* split set be: $o = \{o_1, o_2, ..., o_n\}$, and

$s = \{s_1, s_2, ..., s_n\}$ the computed split, then the precision and recall are calculated as:

$$precision = \frac{|o \cap s|}{|s|} \quad recall = \frac{|o \cap s|}{|o|}$$

where $|x|$ represents the cardinality of x. The same formulas are applied when calculating the measures for correct terms, but using the calculated sets of terms instead of splits.

Once precision and recall are computed the $f\text{-}measure$ can also be calculated. This measure represents a weighted average between precision and recall, and is calculated using the following formula:

$$f\text{-}measure = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

Table 9.4 summarizes the precision, recall and f-measure means for correct splits and sets of terms in the different settings for both packages. Using the *CorpDict* setting, LIdS achieved a precision mean of 0.8959 and a recall mean of 0.9027 when splitting identifiers for the *tree* package; and a precision mean of 0.9548 and a recall mean of 0.9552 when splitting identifiers for the *AbcMidi* package. For the correct set of terms, in the same setting, LIdS achieved a precision mean of 0.8041 and a recall mean of 0.8101 for the *tree* package, and a precision mean of 0.9100 and a recall mean of 0.9112 for the *AbcMidi* package. A f-measure mean of 0.8060 was achieved when splitting and expanding terms of the *tree* package, and a f-measure mean of 0.9101 for the *AbcMidi* package.

Figure 9.1 illustrates f-measure means for correct splits, and correct terms sets for the *tree* and for the *AbcMidi* package.

9.2.4 Results Discussion

Regarding **RQ1** and **RQ2**, the results obtained indicate that the proposed technique performed well in the analyzed programs, written in C, and that use an heterogeneous combination of techniques to create program identifiers (f-measure means in Table 9.4 illustrate this). The *HardSplit* setting achieves, at best, a f-measure mean of 0.5025

		tree		*AbcMidi*	
Setting	Measure	splits	terms	splits	terms
HardSplit	precision	0.5031	0.4150	0.4218	0.3887
	recall	0.5021	0.4138	0.4034	0.3754
	f-measure	0.5025	0.4143	0.4107	0.3807
Split	precision	0.8834	0.7973	0.9230	0.8782
	recall	0.8903	0.8033	0.9307	0.8856
	f-measure	0.8858	0.7992	0.9257	0.8810
CorpDict	precision	0.8959	0.8041	0.9548	0.9100
	recall	0.9027	0.8101	0.9552	0.9112
	f-measure	0.8982	0.8060	0.9544	0.9101

Table 9.4: Precision, recall, and f-measure means, for correct splits and correct terms sets, for *tree* and *AbcMidi* packages, in the three settings.

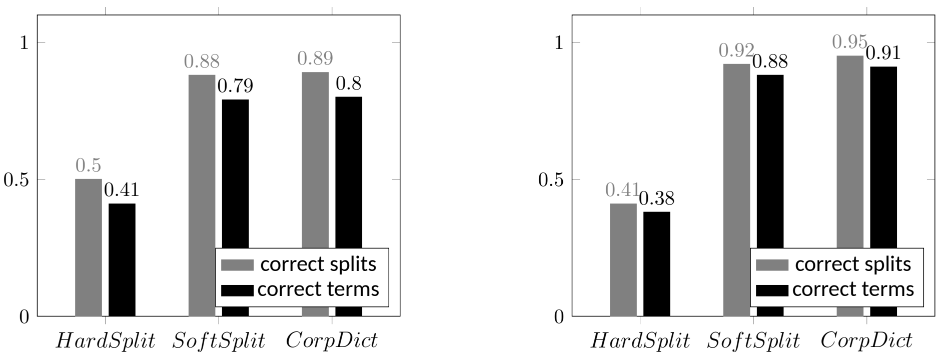


Figure 9.1: F-measure means for correct splits and correct terms sets, for the *tree* (left) and for the *AbcMidi* (right) packages.

when splitting identifiers for the *tree* package. This setting only splits *hard words*, which clearly is not enough for the analyzed packages identifier sets. The *Split* setting helps to illustrate that LIdS outperforms a simple hard splitting technique, accuracy means in Table 9.3 and f-measures means in Table 9.4 support this statement. Taking advantage of the custom corpus-based dictionary (the *CorpDict* setting) improves all the results, mainly because it introduces package specific abbreviations, i.e., abbreviations not found in the *abbreviations* or *programming* dictionaries (e.g, "*ana*" $\to$ "*anacrusis*", "*syll*" $\to$ "*syllable*"). This helps to answer **RQ3**, the empirical data shows that for every setting, accuracy, precision and recall means increase when using the custom corpus-based dictionary in the analyzed packages. This increase is higher for the *AbcMidi* package, mainly because the corpus is bigger ($\sim$ 30 000 words, versus the $\sim$ 2 000 words for the *tree* package), and it includes a more specialized vocabulary, allowing for the custom corpus-based dictionary to capture a set of terms more representative of the application domain.

The main reasons for not reaching 100% precision, in the *HardSplit* setting is straightforward: the analyzed software packages have many multi-term identifiers composed of *soft words*. Regarding the LIdS approach, the main reasons for failing splits are: over-splitting, splitting abbreviations or expressions used by the developers that are not present in the dictionaries (e.g., splitting "*downoct*", in the set $\{down, o, ct\}$, and the set in the *oracle* is $\{down, oct\}$); unexpected words found in the identifier that are validated by the general english dictionary (e.g., the *oracle split set for* "*gotends*" is $\{got, ends\}$, but LIdS resulting split set is $\{go, tends\}$); and invented words by the programmer that are not valid in any dictionary but are reasonably perceived by humans (e.g., "*chording*" is used to represent the action of making chords, but this is not a valid english word, so its split set ends up being erroneously $\{c, hording\}$). The LIdS algorithm can cope with an heterogeneous set of dictionaries, and the most natural solution to address these issues is to devise methods for creating new and improved dictionaries, that can better capture the specifics of the vocabulary used by programers in software development. Which means that improving the results is possible without changing the algorithm itself, but providing more accurate dictionaries. Furthermore, these improved dictionaries can be used by other dictionary based approaches for identifiers splitting.

Although all the measures illustrated in the previous sub-section show that LIdS performed well for the analyzed programs, there is still not enough evidence to generalize its effectiveness for all programs, or other programming languages. The data analyzed and the results presented along this section are available online, including all the dictionaries and oracles used [5].

9.2.5 Threats to Validity

One shortcoming of the validation approach is the existence of errors in the *oracle*, either errors by typos or misspelling, or because the manual approach (split or term expansion) was not exactly the same as the intent of the original programmer. When manually creating the oracle some terms were not included because there was not a clear consensus amongst the authors on how to split or expand a given identifier. Although these cases would provide good examples to evaluate the performance of the splitting technique, we were afraid to end up including errors in the oracle, which would end up by jeopardizing the evaluation results.

Another shortcoming of the evaluation is that sometimes the exact split or term chosen by the algorithm is not syntactically equal to the manual split but semantically equivalent, this is the case of plurals (e.g., "*chord*" versus "*chords*"), or transitive verbs (e.g., "*trim*" versus "*trimming*"). The evaluation uses a syntactic exact match, meaning that all these examples result in a incorrect split/expansion. This issue is mainly related with the evaluation of the resulting splits and terms.

Another shortcoming of this experiment is the fact that it was applied to a couple of software systems only, with a reduced number of identifiers. Therefore, the set of analyzed identifiers is not enough to generalize the results obtained for these specific software systems to all software packages.

[5] Available from `http://conclave.di.uminho.pt/articles/` (Last accessed: 10-08-2014).

9.3 LIdS Experimental Comparison

In order to verify the performance of LIdS against other state-of-the-art approaches, two more experiments were conducted. The following research question was defined:

- **RQ4:** What is the performance of LIdS compared with other state-of-the-art approaches for splitting and expanding identifiers?

To compare the performance of several techniques, all the approaches need to be applied in the same setting: same program identifiers, same *oracle*, same measures, etc. To hold true to these assumptions, instead of devising new experiences, some case studies described in [58] and [61] were used. The data provided by the authors includes the *oracles*, allowing other approaches to compute splits on the same identifiers, and assume the same correct answers. The goal of the following experiences is to re-create the original experience, but including the LIdS approach. All the LIdS results achieved in this section use the *CorpDict* setting, described in Section 9.2; unless stated otherwise, the documentation corpus and custom corpus-based dictionary were created for each analyzed package.

9.3.1 First Experiment

The subjects for the first experiment, described in [58], are two programs: *JHotDraw* [6], a framework for technical and structured graphics, written in Java; and *Lynx* [7], a text-based web browser written in C. Both projects are open-source, so the source code is readily available. Table 9.5 highlights some characteristics about these packages.

The study follows the design described by Guerrouj *et al.* in [58], so that the results can be compared. The main independent variable is the approach used to compute the split and expansion set, which is compared to the gold set provided by the *oracle*. The *oracles* (provided by Guerrouj *et al.*) were created by first extracting some random identifiers from each source package, and then manually split identifiers, and expand abbreviations to full terms, more details in [58]. LIdS was applied to the identifiers

[6]Available from: `http://www.jhotdraw.org/` (Last accessed: 07-03-2014).
[7]Available from: `http://lynx.isc.org/` (Last accessed 07-03-2014).

	JHotDraw	*Lynx*
Release	5.1	$2.8.5$
Files	155	247
Size (KLOC)	16	174
Identifiers (>2 chars)	$2\,348$	$12\,194$

Table 9.5: *JHotDraw* and *Lynx* main characteristics.

from the *oracle* for each package, precision, recall and f-measure were computed for the resulting splits. Table **??** and 9.7 summarize the results found in [58], including two new approaches: LIdS and INTT. This new data helps to answer **RQ4**. Comparing the ability to split and expand identifiers, the new approaches f-measure mean values are close to the ones from TRIS and TIDIER. For the *JHotDraw* package, LIdS achieved a f-measure mean of 0.9603, which is very close to TRIS and TIDIER. For the *Lynx* package, LIdS achieved a f-measure mean of 0.8593, lower than TRIS f-measure 0.9206, but close to TIDIER f-measure mean 0.8525. Results are better for the *JHotDraw* package mainly because of the same reasons already highlighted in [58]: this package follows coding standards and most of the identifiers are composed of hard words, opposed to the *Lynx* project, where a more ad-hoc set of rules were used to create multi-term identifiers, and hard words were less used.

The plot in Figure 9.2 illustrates the f-measure means for both packages, using the different approaches. For the *JHotDraw* package all the values are above 0.90, but for *Lynx*, there is a clear gap between CamelCase and Samurai approaches, and TIDIER, TRIS, LIdS and INTT approaches. Mainly because *Lynx* identifiers use more techniques to shorten identifiers, and create abbreviations, harder to split by techniques best suited to split hard words.

9.3.2 Second Experiment

The next experiment, based on [61], aims to compare the techniques ability to split a multi-term identifier strings. In their paper Hill *et al.* use the Ludiso oracle which contains a set of $2\,731$ identifiers from a collection of $2\,117$ open source program written in C, C++ and Java; and the manual splits created by human annotators. State-of-the-art

Metric	Approach	1Q	Median	Mean	3Q	σ
Precision	Camel Case	1.0000	1.0000	0.9244	1.0000	0.2424
	Samurai	1.0000	1.0000	0.9316	1.0000	0.2244
	TIDIER	1.0000	1.0000	0.9716	1.0000	0.1472
	TRIS	**1.0000**	**1.0000**	**0.9804**	**1.0000**	**0.2025**
	INTT	1.0000	1.0000	0.9623	1.0000	0.1704
	LIdS	1.0000	1.0000	0.9591	1.0000	0.1728
Recall	Camel Case	1.0000	1.0000	0.9203	1.0000	0.2502
	Samurai	1.0000	1.0000	0.9367	1.0000	0.2129
	TIDIER	1.0000	1.0000	0.8984	1.0000	0.2158
	TRIS	1.0000	1.0000	0.9084	1.0000	0.1213
	INTT	1.0000	1.0000	0.9606	1.0000	0.1760
	LIdS	**1.0000**	**1.0000**	**0.9641**	**1.0000**	**0.1583**
F-measure	Camel Case	1.0000	1.0000	0.9217	1.0000	0.2476
	Samurai	1.0000	1.0000	0.9325	1.0000	0.2200
	TIDIER	1.0000	1.0000	0.9233	1.0000	0.1791
	TRIS	1.0000	1.0000	0.9328	1.0000	0.1614
	INTT	**1.0000**	**1.0000**	**0.9607**	**1.0000**	**0.1733**
	LIdS	1.0000	1.0000	0.9603	1.0000	0.1670

Table 9.6: Precision, recall, and f-measure for several approaches on JHotDraw.

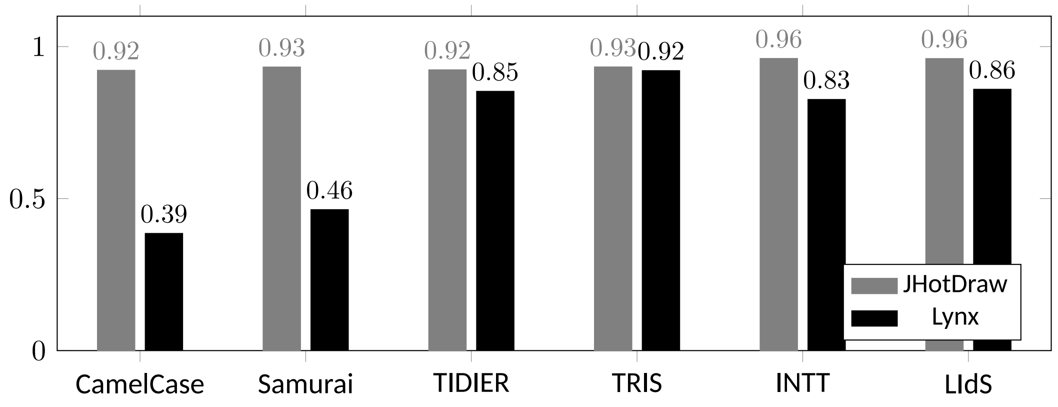


Figure 9.2: F-measure means for *JHotDraw* and *Lynx* using different approaches.

Metric	Approach	1Q	Median	Mean	3Q	σ
Precision	Camel Case	0.0000	0.5000	0.4065	0.7500	0.4147
	Samurai	0.0000	0.5000	0.4767	1.0000	0.4089
	TIDIER	0.8000	1.0000	0.8609	1.0000	0.2674
	TRIS	**1.0000**	**1.0000**	**0.9344**	**1.0000**	**0.1369**
	INTT	0.7500	1.0000	0.8294	1.0000	0.3215
	LIdS	0.8000	1.0000	0.8539	1.0000	0.2868
Recall	Camel Case	0.0000	0.3333	0.3705	0.6667	0.4066
	Samurai	0.0000	0.3333	0.4569	1.0000	0.4101
	TIDIER	0.7500	1.0000	0.8499	1.0000	0.2684
	TRIS	**1.0000**	**1.0000**	**0.9138**	**1.0000**	**0.2060**
	INTT	0.7500	1.0000	0.8244	1.0000	0.3269
	LIdS	1.0000	1.0000	0.8681	1.0000	0.2711
F-measure	Camel Case	0.0000	0.4000	0.3851	0.7273	0.4086
	Samurai	0.0000	0.4000	0.4634	1.0000	0.4084
	TIDIER	0.6667	1.0000	0.8525	1.0000	0.2664
	TRIS	**1.0000**	**1.0000**	**0.9206**	**1.0000**	**0.2055**
	INTT	0.7500	1.0000	0.8258	1.0000	0.3245
	LIdS	0.8333	1.0000	0.8593	1.0000	0.2796

Table 9.7: Precision, recall, and f-measure for several approaches on Lynx.

approaches are applied to the identifiers in the oracle, and their ability to correctly split the identifiers set is measured by means of accuracy, precision and recall, and analyzed by groups (e.g., programming language, identifier subsets).

Accuracy is a binary measure (as before), if the technique output split is exactly equal to the corresponding split in the oracle the accuracy value is 1, but if there is any difference between the technique output and the oracle the accuracy value is 0. For this experiment the *programming*, *acronyms*, and *abbreviations* dictionaries described in previous sections were used. The oracle is composed from thousands of programs, it was not feasible to compute the custom corpus-based dictionary for every software package. Instead, a documentation corpus was created that includes the natural language text for the top three programs found in the oracle (*mozilla-source*, *mysql* and *cinelerra*), a custom corpus-based dictionary was created from this corpus and used by LIdS.

Table 9.8 mirrors the results compiled in [61] but includes the LIdS approach results, which achieved an accuracy mean of 0.67 when splitting the identifiers from the

Technique	All	*C*	*C++*	*Java*
Samurai_all	0.82	0.79	0.85	0.83
Samurai_cpp	0.81	0.77	0.85	0.81
GenTest_lg_all	0.80	0.78	0.82	0.78
INTT	0.75	0.70	0.78	0.78
GenTest_med_java	0.74	0.75	0.77	0.71
CS	0.71	0.68	0.72	0.72
DTW	0.69	0.75	0.66	0.65
LIdS	0.67	0.69	0.72	0.60
Greedy_lg	0.60	0.59	0.66	0.54
Greedy_sm	0.56	0.58	0.59	0.51
Count	$2\,663$	885	887	891
% of data	100%	33%	33%	33%

Table 9.8: Mean per-identifier accuracy for each programming language subset for the Ludiso oracle.

Ludiso oracle. The major reason for LIdS accuracy to be lower than most of the other approaches is over-splitting. For example, the result of splitting "*GGGPP_CDMA2000*" with LIdS is the set $\{ggg, pp, c, dma, 2000\}$, the oracle correct split is $\{GGGPP, CDMA, 2000\}$. The excessive split occurs because the strings "*GGGPP*" and "*CDMA*" are not present in any dictionary used. The only way to overcome similar situations is using the custom corpus-based dictionary to gather such strings, but this dictionary was not created for the package where this identifier was extracted from. Although, there is no assurance that every abbreviation string would be added to every package custom corpus-based dictionary, at least some of them would be expected to, increasing the approach overall accuracy.

Besides accuracy, precision and recall measures were also made. These are calculated in a slightly different way than in the previous experience, since the goal now is to measure the correct splits, and not the resulting set of terms. Table 9.9 summarizes the intra-technique results for precision and recall using different approaches presented in [61], but including the LIdS approach. The newly introduced technique results achieved a precision mean of 0.90, a recall mean of 0.96, which translated in a f-measure mean of 0.92. These results are close and in line with other approaches. The main reason for having a precision under 100% is the over-splitting introduced by splitting specific abbreviations not presented in any dictionary as discussed previously. This data helps to answer **RQ4**: LIdS ability for correctly split identifiers is in line with other approaches.

9.3.3 Threats to Validity

Regarding both comparison experiments, the major threats to validity are concerned with the oracles, and how the resulting splits are compared. Even if a lot of effort is dedicated to making sure the oracles are accurate, some issues are always present: actual typos (e.g. "*buf*" $\rightarrow$ "*bufer*"); ambiguous splits - different programmers split the same identifier in a different ways (e.g, "*invalid-username*" $\rightarrow \{invalid, user, name\}$ versus $\{invalid, username\}$); semantically equivalent splits but syntactically different; linguistic issues (e.g., "*reparse*" is often considered a unique term but it is not an english word) for example. Regarding the oracles that contain the exact set of terms, i.e., abbreviations are expanded, there are some issues with the exact expansion chosen (e.g., "*auth*" $\rightarrow$ "*authenticate*" versus "*authentication*"), or lack of consistency, i.e., abbreviations lacking an expansion, just to make it clear abbreviations considered acronyms (e.g., HTML, XML, SQL) are usually not expanded in the oracles. A human looking at the resulting split for each identifiers can cope with most of these issues, but all the processes that compute metrics over the resulting splits are done automatically, introducing some measurement errors.

The extra data files (including results) required for both experimental comparisons discussed in this section are available online[8].

9.4 Concept Mapper Locate Experimental Validation

The previous chapters describe the underlying technique used in the Conclave system for feature location, based on kPSS. This section describes the preliminary evaluation done, to verify if this technique introduces benefits over other common techniques in the context of PC. In currently available IDEs, common search facilities provided to the users, are still match based approaches, so the following research question was formulated:

[8]Available from: `http://conclave.di.uminho.pt/articles/` (Last accessed: 10-08-2014).

Technique	Measure	All	*C*	*C++*	*Java*
Samurai_all	P	0.97	0.96	0.98	0.98
	R	0.96	0.94	0.97	0.97
	F	0.96	0.95	0.97	0.97
Samurai_cpp	P	0.98	0.98	0.98	0.98
	R	0.95	0.93	0.96	0.96
	F	0.96	0.94	0.97	0.97
GenTest_lg_all	P	0.97	0.97	0.97	0.96
	R	0.96	0.94	0.96	0.98
	F	0.96	0.95	0.96	0.96
INTT	P	0.98	0.99	0.99	0.98
	R	0.93	0.91	0.94	0.95
	F	0.96	0.95	0.95	0.96
GenTest_med_Java	P	0.95	0.97	0.98	0.98
	R	0.97	0.94	0.97	0.95
	F	0.95	0.95	0.97	0.96
CS	P	1.00	1.00	1.00	1.00
	R	0.90	0.88	0.90	0.91
	F	0.94	0.92	0.94	0.95
DTW	P	0.93	0.94	0.92	0.92
	R	0.94	0.96	0.91	0.95
	F	0.93	0.95	0.91	0.93
LIdS	P	0.90	0.90	0.91	0.87
	R	0.96	0.96	0.97	0.96
	F	0.92	0.92	0.93	0.91
Greedy_lg	P	0.89	0.89	0.91	0.86
	R	0.97	0.96	0.98	0.98
	F	0.92	0.91	0.94	0.91
Greedy_sm	P	0.88	0.89	0.90	0.86
	R	0.97	0.96	0.98	0.98
	F	0.92	0.91	0.93	0.91

Table 9.9: Mean precision (P), recall (R), and f-measure (F) for each programming language subset for the Ludiso oracle, sorted by mean overall accuracy.

RQ1: How does the *kpss* scoring function performs, when compared to a *match* scoring function, for finding relevant elements of the code given a search query?

To help answering this question the following experience was performed:

Step 1: in order to ease the process of replicating this experience, the benchmarks provided by Dit *et al*[9] were used, instead of devising a new data set. Each benchmark contains a set of bug reports, and corresponding function sets that was changed to resolve the bug (referred as the gold set) – more details about the benchmark in [41];

Step 2: the title for each bug report was extracted, stop words[10] were removed, and the resulting set was archived as keywords;

Step 3: for each bug report, the *locate* function to compute a rank was called, using the *match* scoring function, the keyword set computed in *Step 2*, and setting as range the *Method* program element;

Step 4: *Step 3* was replicated but using the *kpss* scoring function;

Step 5: effectiveness measure for each resulting rank was calculated.

The effectiveness measure is calculated by analyzing the computed rank in order, and its value is the first position of the rank that is a relevant function. Methods that are part of the set of functions changed to resolve the bug (the gold set) are considered relevant. The rank position can be compared for different scoring methods to measure which rank produced the best results. This approach was also used in [120] and [127] for comparing feature location techniques performance.

Table 9.10 presents some informative data about the analyzed software packages: *jedit*[11] (version 4.3), *mucommander*[12] (version 0.8.5), and *jabref*[13] (version 2.6); all written in Java. The results of the experience are presented in Table 9.11. They show that for

[9]Available from: `http://www.cs.wm.edu/semeru/data/benchmarks/` (Last accessed: 29-01-2014).

[10]Common words that tend to convey low semantics (e.g. "*the*", "*a*", "*to*") [100].

[11]Available from: `http://jedit.org/` (Last accessed: 29-01-2014).

[12]Available from: `http://www.mucommander.com/` (Last accessed: 29-01-2014).

[13]Available from: `http://jabref.sourceforge.net/` (Last accessed: 29-01-2014).

Software Package	Number of Source Files	Number of Identifiers	Number of Methods	Number of Issues
jedit-4.3	465	23606	4934	150
mu-0.8.5	1069	27501	7489	92
jabref-2.6	480	19921	3901	39

Table 9.10: Information about the analyzed packages during the Concept Mapper locate function empirical validation.

Software Package	Scoring Function	Analyzed Issues	Better Eff. Measure	Worst Eff. Measure
jedit-4.3	*match*	150	35	78
	kpss		78	35
mu-0.8.5	*match*	92	17	51
	kpss		51	17
jabref-2.6	*match*	39	4	20
	kpss		20	4

Table 9.11: Results of the empirical validation for the Concept Mapper locate function.

these software packages, the kPSS based scoring approach produced a better effective measure result, that the simple *match* function. The *match* function simply compares the two sets of words (search keywords, and terms from the identifier), and the score is computed by normalizing the number of common words and the total of words. For the *jedit* package, the kPSS based scoring approach achieved a better result 78 times, i.e., a better position in the rank for the first relevant method found. For the *mucommander* package, the kPSS based scoring function achieved a better result 51 times, and 20 times for the *jabref* package. The remaining times either both approaches scored the same, or none of the relevant functions were found in any resulting rank.

9.4.1 Threats to Validity

Although these results are satisfactory, they do not provide enough empirical data to generalize the performance of kPSS based techniques. Also, the keywords used to build the queries and the functions gold sets are a threat to validity because: (i) the keywords set was built automatically from reports titles that sometimes lack relevant terms, or use only ambiguous words (e.g. "*bug*"), a human would be more prone to devise a set of terms (after reading the report) that would create a more accurate rank; (ii) sometimes, when fixing bugs, the actual defect is really not related to the concepts functions are addressing, which translates in changing code unrelated to search queries.

Modern searching features, explore the use of regular expressions, and similar approaches, for improving text based search. The *match* function could explore such approaches to improve the number of word matching, instead of blindly comparing strings, to better mimic currently available solutions.

9.5 Mappings Experimental Validation

Previous experiences draw conclusions about the merits and benefits of using the Concept Mapper framework to perform concept location by searching elements in the ontologies, and comparing the relatedness of the elements using a kPSS based scoring approach. In general, given the previous results, the kPSS scoring function, exploring the terms extracted from each element of the source code (resulting from splitting and expanding identifiers), provides sound results for building relations between elements for the analyzed case studies. Besides searching, the Concept Mapper framework, allows the creation of mappings between elements of different ontologies (domains). A common example of interest is a mapping between the problem domain, and the program domain, illustrated in the previous chapter, to help relating concepts in the application domain with the source code elements that are responsible for addressing them.

The goal of this specific experience is to measure the benefits for the program maintainer of such mappings, while performing software maintenance activities. The following research question was formulated:

RQ1: Are there any benefits of building a mapping between the problem domain and program domain, from a maintainers point of view while performing software maintenance tasks?

To help answering this question the following experience was undertaken:

Step 1: a software system already included in Conclave was selected for this experience, in particular the *tree* software package, because of its relatively small size.

Step 2: an obvious and simple bug was introduced in the source code, and a bug report was created to describe the faulty behavior of the tool.

Step 3: two small and simple exercises were created, containing the bug report, a widget for submitting and verifying the correct answer, which is to determine the function that contains the bug. The only difference between the two exercises is the mechanism available to browse the source code:

Version A: in this version, the source code navigation window displays the complete set of functions available in the code using a one level straightforward list; the maintainer can select a function to inspect its source code, or choose to view the entire source code.

Version B: in this version, using the problem ontology available in Conclave a second browsing level is introduced. The maintainer can choose one of the concepts available in the ontology, and see the rank of functions (automatically generated) related with each concept. The maintainer can then select a function from the rank, to inspect its source code, or choose to view the entire source code.

Step 4: a simple webpage, including a small set of instructions was created, to comprise the survey, including the exercises described in *Step 3*. Appendix E illustrates the webpages. At the time of this writing, the survey is still available online[14].

Step 5: the link for the survey created in *Step 4* was passed along a set of people with background in software development, and they were invited to complete the proposed exercises and submit some feedback. Besides the feedback, the number of functions inspected, the number of failed tries, and the time required to complete each exercise were measured.

[14]Available from: `http://conc-survey.di-um.org/` (Last accessed: 29-10-2014).

	Total	Version A	Version B
Correct answer submitted	52	24	28
Average functions visited	6	10	2
Average failed tries	1	2	0
Average time (in seconds)	279	520	85

Table 9.12: Measures results for the mappings analysis survey.

9.5.1 Results Discussion

At the time of this writing, there were already around 130 visits to the exercises pages (for both versions), and around 50 correct answers submitted (also for both versions). Roughly, this means that around 30 people participated in the survey, 22 of whom submitted some feedback using the survey form.

Table 9.12 summarizes data gathered during survey participation concerning automatically collected measures: (i) number of functions visited, (ii) number of failed tries, and (iii) the time spent while performing the exercise. These measurements were gathered via accesses to the exercises pages, which means that may be some values that are not precisely correct. For example, the time spend to complete an exercise is measured by computing the difference between the absolute time when the participant first visits the exercises page, and the absolute time when the participant successfully submits the correct answer. Of course whoever is completing the exercise may be for example multi-tasking, i.e., performing other tasks at the same time, jeopardizing the absolute time spent to complete the exercise. Other problems are, for example, open the exercise page at a given time, but only actually attempt to complete the exercise later, or in the next day. Nevertheless, analyzing Table 9.12, there is a general trend of higher average values, concerning not only the time taken[15], but also functions visited, for the exercise version A, and a general trend of lower average values for version B. There are a general set of details concerning the exercises themselves that have some influence of these trends, which are discussed in the next section.

Besides automatic measurements, the participants were invited to submit feedback

[15]Due to the shortcomings of measuring time using pages access and HTTP sessions, time spans higher than 1 hour are not included for average calculations.

using a free text form available in the survey webpage. To draw conclusions about the participants point of view, while performing the role of software maintainers, the list of submitted feedbacks messages were analyzed and categorized. Table 9.13 describes the devised categories, and the distribution values for the set of submitted messages. A total of 22 participants submitted a feedback message, 21 of which describe a preference for the interface presented in version B, over the interface for version A. None of the participants submitted a message that would express their preference for version A, over version B, and only 1 participant submitted a message that does not include any explicit statement concerning this detail. Some examples of statements used to sustain the preferences of version B used by the participants include: "*The second step helps on organizing the functions on their meaning/semantic, making the debugging process easier*", "*The grouping of functions by concept greatly increased the speed of my problem resolution.*", "... *the key words shown in the first column were very important to find the correct answer faster.*", "... *because functions are well organized according to their role, thus it was possible to find the error in a short time*", and "*The concept arrangement of the code methods simplified the process of finding the Bug* ...", and "... *because it organizes the functions by topics and makes the task of finding the bug easier.*".

Although the participants knew nothing about the mappings between the program and problem domain acting in the background, given the submitted feedback, there is almost a general consensus that the organization and features used in version B clearly have benefits (e.g., faster pinpoint of buggy elements, enhanced source code browsing) when compared to version A. The conceptual navigation approach to browse source code featured in version B, is built on top of the mapping between the problem domain, and the program domain. This helps to answer the initial research question devised for this experiment. For the selected use case, and for the proposed task, the feedback gathered from the survey participants indicates that there are benefits in building and exploring a mapping between the problem domain to aid maintainers performing software maintenance tasks. Some of the participants also commented some problems with the exercises, for example: "*search after finding the first one influences directly the search of the second step*", this and other issues concerning this experiment are discussed in the next section.

Category	Total	Total (%)
A - Prefer Version A over version B	0	0%
B - Prefer Version B over version A	21	95%
N - None of the above	1	5%
Total	22	100%

Table 9.13: Feedback distribution for the mappings analysis survey feedback.

9.5.2 Threats to Validity

This section discusses some issues and threats to validity concerning the experiment described in the previous section.

The first detail to point out is the bug complexity. The bug introduced in the code is very simple, and immediately spotted. Although this can jeopardize the final results in the sense that it may be too easy too find the bug, the actual goal of the experience is not to measure the ability of the participants to debug the code, but to enhance the concept location tasks, i.e., find the culprit zone of code faster. Another concern regarding the introduced bug, is that it is the same in both exercises. If the participant successfully finds the answer in version A, of course its behavior is already biased for version B. The problem with providing different exercises to overcame this detail is in finding pairs of tasks with exactly the same complexity. In the next iteration of similar experiments, randomly choosing the first proposed exercise version would help addressing this shortcoming.

Concerning the automatic measures accuracy (e.g. number of functions inspected, time to complete exercise), although more complex approaches could be used to attempt to increase the accuracy of such measures, the underlying proprieties of the HTTP layer, and the fact that the person is not observed while performing the proposed tasks, are always an issue. The only way to completely overcome such issues is to have participants completing the survey in a controlled environment. This is also being pondered for a future iteration of similar surveys.

The described experiment, even with its shortcomings, already provides some insight on the general preference from the maintainers point of view of the benefits of exploring mappings between the program domains, while performing software maintenance activities for the studied cases. However, more studies are still required before generalizing these claims. Also, designing and using interfaces specially tinkered for maintenance tasks, provided a good insight and helped clarifying some ideas concerning software maintenance activities.

9.6 The Development Point of View

There is another relevant scope concerning this work, that does not have a straightforward empirically evaluation. The initial research question refers to the benefits of using ontology-*aware* applications, in the context of PC. But the benefits of adopting ontologies is not evident only on the final application results, but are also clear in the implementation of such applications. The method described for modeling information, and related tools, namely the topics discussed in Chapter 6 and 7, enable the implementation of techniques and algorithms in a clear and simple manner, with many advantages right out of the box. There is no straightforward way to measure this, and present empirical evidence of such benefits. Hence, the remaining of this section is a walkthrough of an example that helps to emphasize the advantages noted while using this approach to implement most of the applications describes in this document.

Program identifiers vocabulary normalization is a common task in program understanding approaches, i.e., split (and expand when required) program identifiers strings to sets of more meaningful terms. The Conclave environment also undertakes such a task, in order to improve searching and mapping results. Program 9.1 illustrates the code used to enrich the ontology with the sets of splits, and terms, calculated for each identifier extracted from the source code, written in Perl. LIdS, introduced in the previous chapter, is the tool used to split and expand identifiers.

Line 1-2 simply load the required libraries: Conclave OTK, and Lingua IdSplitter. Line 4-5 create instances to abstract the ontology, and for the splitting algorithm. In line 7, the ontology is queried to retrieve the set of identifiers available in the program domain, and line 9 iterates and processes each identifier individually. For each identifier, in line

```
1  use Conclave::OTK;
2  use Lingua::IdSplitter;
3
4  my $ontology = Conclave::OTK->new($base_uri);
5  my $splitter = Lingua::IdSplitter->new();
6
7  my @identifiers = $ontology->get_instances('Identifier');
8
9  foreach my $id (@identifiers) {
10   my $name = $ontology->get_data_prop($id, 'hasString');
11   my ($splits, $terms) = $splitter->split($name);
12
13   $ontology->add_data_prop($id, 'hasSplits', $splits);
14   $ontology->add_data_prop($id, 'hasTerms', $terms);
15 }
```

Program 9.1: Program, written in Perl, to populate the program ontology with the splits and terms, computed by LIdS, for each identifier found in the source code.

10, the specific identifier string is retrieved from the ontology, and in line 11 the sets of splits and terms are computed using LIdS. Finally, lines 13-14 store the resulting sets in the ontology. This snippet of code is clear, simple, and easy to read. Yet, implements a rather complex task.

There are some subtle advantages of such an approach. The LIdS tool, that splits the identifier is developed outside the scope of this code. Its effectiveness can be improved without having to change the approach for loading the information to the ontology. Also, other tools can be used, but everything else will just work. Such tools are not required to know nothing about the language being processed, or parsing techniques. The above code undertakes its task correctly without even knowing in which programming language the code is written. Other tools that take advantage of the splits and terms set can always take advantage of such information, once its available in the ontology. The task is defined using ontology operations, regardless of the backend where the ontology is being stored (e.g, file, triple store), and format (e.g., OWL), which means that even, because of other requirements, any shift in format or storage backend is performed in the background, this task will continue to be able to perform its work successfully. Also, splitting and expanding identifiers is a task that does not require understanding of source code, it can be executed with a small set of sub-elements extracted from the program. Abstracting such details allows for a general implementation of the task.

These advantages allow the definition of tasks, entailed in an application, in a generic way, and most of the discarding details that do not contribute for the operation at hand, e.g., the parsing technique used to parse source code and extract identifiers does not contribute to the split and expansion quality of the splitting technique adopted.

Summary

- This chapter describes some experimental validations undertaken to empirical measure the benefits of the devised approaches for some particular problems. Fully evaluate an application may take longer than building it, nevertheless evaluation pushes applications to a more mature and scalable state, and that can cope with real world scenarios.

- The results achieved for kPSS *synsets* analysis, and LIdS splitting approach, show that these techniques provide results that do not jeopardize the final experiments.

- Source code search based on keywords, can potentially enhance concept location activities. The ontology based concept location searching feature, for the studied cases, outperforms a simple string matching approach (based on the effective measure).

- The final experience, discussed in Section 9.5, helps drawing conclusions about the benefits of exploring mappings between domains, in particular the program and problem domain, to enhance programmers maintenance activities. The design of interfaces specially crafted to aid in software maintenance tasks can also potentially enhance PC activities in general.

- Besides the applications and tools, this work also contributes to the way such tools and applications are designed and implemented. Providing a set of tools and frameworks, that allow the implementation of clear, simple, and easy to read tasks in the context of PC.

Part IV

Conclusion

Chapter 10

Final Remarks

A story has no beginning or end: arbitrarily one chooses that moment of experience from which to look back or from which to look ahead.

Graham Greene

This dissertation describes work undertaken in the field of software engineering, in the domain of program understanding. This chapter discusses some final thoughts and reflexions about this work.

Program comprehension, and related fields of research, are becoming even more relevant in todays world, where the dependency on software systems, and the direct impact they have on everyday life is drastically increasing. Also, as time goes by software tends to increase in size and complexity. Software maintenance and evolution tasks are central activities in the scope of software engineering. In order to undertake these knowledge intensive activities, programmers and maintainers dwell most of their time in software comprehension and understanding processes, synthesizing information obtained from different sources. One of the most relevant steps to achieve the understanding of a program (or a part of it), is relating source code elements, with the real world concepts they are addressing. This is a challenging endeavor mainly due to the different levels of abstractions and languages used, on one side, the natural languages used to describe and discuss real world concepts, and the languages used to write software (programming languages). The study of the mental representations, and the underlying human cognitive processes, allows the development of techniques and implementation of tools that assist in this process, promoting a faster and better understanding. This work promotes the adoption of ontologies entailed semantics and knowledge representation techniques, to build models of the program, and related domains, in order to enhance software understanding activities.

In order to achieve this goal, the generic workflow illustrated in Figure 8.11 is proposed. The workflow defines a set of generic steps, starting from the original software system, and ending with a set of heterogenous views of the program (e.g., graphs, mappings, annotated code) that emphasize specific traits of interest. The generic workflow is split in three major stages: (i) process the program and related artifacts to create resources; (ii) process resources to convey relevant information to build and populate ontologies; and, (iii) process ontologies to infer knowledge and build views of the software.

During the first stage, the software system under analysis is processed using an arbitrary set of tools. The set of tools is not closed, i.e., any arbitrary tool can be added to process the system, and provide some kind of conclusion or analysis about the program, or other artifact included or related with the software. Besides allowing any type of tool to contribute to the process, it enables the combination of results of these tools without extra effort from the final application.

During the second stage, the information in the resources produced during the first stage is conveyed to an ontology. This data normalization, allows applications to explore the available information using a single notation and paradigm. Conclave OTK takes a vital role in this stage. Using OTK operations, relevant data is easily and effectively transported to an ontology, increasing its semantics. Higher level applications, i.e., applications for programmers and maintainers to study software systems, exploit the information available in ontologies using OTK. These applications benefit from data computed by other tools, and are easily used across programming languages, or paradigms, since all the information is readily available (e.g., not required to parse the source code, or normalize identifiers), of course sometimes, the required data to compute some result may not be available in the ontology. In this case, the best approach is to start by devising a tool that conveys the required information, and then go back and implement the final application exploring the data previously stored in the ontology. This allows for other applications to explore the same information later, and also makes composing results with other tools easier.

During the third and final stage, applications targeting end users[1], produce specific views of interest of the software system. The Conclave OTK toolkit allows building applications, that explore the information available in the ontologies, to aid programmers and maintainers. At this stage, all the information collected during the analysis stage is normalized and available in ontologies, applications are not concerned with performing low level operations (e.g., parsing) and can retrieve the required information using Conclave OTK operations, Program 9.1 illustrates an example application. The Conclave Concept Mapper, described in Chapter 7, is a more complex example of an application that performs searches and mappings exploring the information available in the models. This approach allows applications to be quickly developed, free from dealing with programming languages details, e.g., Conclave Concept Mapper explores program identifiers gathered from source code, but does not parse any program, or does not know how to build an identifiers table, such tasks are delegated to other tools or stages.

Some final remarks, and recalling the initial research question:

- Regarding program comprehension there is no doubt that it is a very important area of research with many benefits for the industry, and society in general. Most techniques are based in the same steps: gather information, reason about information gathered, provide a combined view for the original source code and the synthesized information. During this work, one general goal was to provide generic tools that perform common tasks (e.g., collect identifiers, gather function definitions, split and expand program identifiers) and make the information available in the model for other applications to explore. This allows the implementation of more language agnostic tools, and the composition of results easier.

- The adoption of ontologies allowed to store information, computed from heterogeneous sources, with a clear semantics. Mechanisms inherent to this technology allow to query and update information in a clear and effective way. Ontologies

[1]Programmers and maintainers performing software understanding activities.

are used as a universal type, to store arbitrary data, and provide a common language (semantics) for tools to share information and results.

- The use of ontologies does not break only the boundaries of produced information or inferred knowledge, i.e., the normalization of information to be explored by heterogeneous applications. But also allows to overcome domain boundaries, i.e., combine and explore information from different domains (e.g., mappings between the problem and program domain). Tools, using a single mechanism, are able to query different domains (e.g., the program domain, the application domain), in order to build semantic relations between elements at different levels of abstraction, and described using different languages. Using the mechanisms provided by OTK, applications can query ontologies for information in different domains, and relate elements in the program domain, with elements in the application domain, for example. This enhances the applications that can be build in the context of PC.

- Tools introduced in Chapter 8, and the experiments described in Chapter 9, provide evidence of the benefits of this work for enhancing program comprehension activities. Benefits concerning not only the final applications developed, but also during the development process. The presented case studies during the experimental evaluations, the introduced applications, and the mappings between domains, created using Conclave Concept Mapper, show that for the discussed analysis there are benefits on using the proposed approach for enhancing program comprehension.

- The generic tools developed during this work are available freely, for researchers interested in reproducing some of the results illustrated, or use them in their own research. There was a major concern, while developing the introduced frameworks, to implement them as generic and modular as possible, so they are easily applied in other domains, and are easily extended when required. The most prominent frameworks and tools, including corresponding web sites, are enumerated in Section 1.3.

- The work discussed in this document relates to many different research areas (e.g., Natural Language Processing, compilers technology, inference engines, ontology formats). Although, this situation can be daunting at first, the synergy between different disciplines can eventually provide better overall results. This synergy is in line with current state-of-the-art approaches in the context of software engineering.

In sum, mappings between the problem domain, the program domain, and other related domains, provide benefits to enhance program comprehension while performing software evolution and maintenance tasks. These mappings help building relations between real world concepts, and program elements, a required activity for software programmers and maintainers while building a mental model of the system, and consequently understanding the program, or a part of it.

10.1 Future Work

This section describes some trends for future work. Topics related with this work In particular include:

- Develop more tools during the first stage of the software analysis workflow, building more resources about the program with particular emphasis on non-source code content (e.g., documentation), and runtime analysis.

- Expand the set of classes, relations, and proprieties in the ontologies, to capture more information about the domains.

- Increase the information conveyed to the ontologies, from the currently generated resources, and also from adding the resources to the available toolchain.

- Expand Conclave OTK set of available methods, to provide more features for updating, and querying, information available in the ontologies.

- Extend the domain specific language and the Conclave Concept Mapper, to allow the creation of new mappings between elements, and to improve searching features.
- Still, in Conclave Concept Mapper, devise new scoring functions, to compute relatedness scores between elements in the different domains.
- Add more end-user applications to the Conclave environment, more tools to process programs and domain artifacts to convey more information, and increase the collection of software systems available to explore.
- Continue to devise more experiments to empirically measure the efficiency of the devised tools, and in particular how to compare applications that address the same problems.

And topics more likely to spawn new research endeavors:

- Devise the automatic creation of interfaces, from the domain representation of the program and other domains (e.g., the problem domain), to build tools that enhance programmers software maintenance and evolution activities.
- Given that information and knowledge about the program (and related domains) semantics is well defined, it is possible to export this information to a format outside the family of ontologies. This means transporting the information of interest back and forth to another modeling approach, to perform more arbitrary computations.
- Modern development communities explore and use a set of common tools, outside the scope of the software package itself, e.g., version control systems, wikis, issues and bug tracking systems. These can provide useful information and insight to improve already discussed models (e.g., the application domain), or spawn the creation of new models of interest. For instance, the creation of a model to represent the software changes as recorded by the version control system, including commits messages. A mapping between this domain and the problem domain could provide information about which changes are related to which concepts from the application domain.

Appendices

Appendix A

Introduction to the Haskell Notation

Haskell is a purely functional and strongly typed programming language. A very brief and summarized introduction to the language notation follows. More information about the language, and detailed resources about its notation are available in the Haskell official website[1]. Some of the illustrated examples use the Glasgow Haskell Compiler[2] interactive environment (ghci) to calculate expressions.

Functions Definitions and Signatures

A function in Haskell is clearly defined, and has a clear signature, that defines the type of inputs that the function takes, and the type of results the function computes. For example, the signature for a function called *square*, that given an integer computes its square, can be as follows:

$$\mathit{square} :: \mathit{Int} \rightarrow \mathit{Int}$$

This reads as: the *square* function takes and integer, and its result is an integer. Functions can have arbitrary numbers of arguments, for example a function for computing the maximum of four numbers can have the following signature:

1 Available from: `http://haskell.org/` (Last accessed: 20-09-2014).
2 Available from: `http://www.haskell.org/ghc/` (Last accessed: 20-09-2014).

$$max :: Int \rightarrow Int \rightarrow Int \rightarrow Int \rightarrow Int$$

To generalize this function to take as input an arbitrary list of numbers, the signature could be changed to take as input a list of integers, and the result is the maximum integer:

$$max :: [Int] \rightarrow Int$$

Most of the times, function signatures can be omitted, in which case the compiler will derive them. But still they are very useful, especially for humans reading the code, to explicitly state which are the arguments to the function and the final result.

The function body follows the signature, for example the complete definition for the *square* function could be as follows:

$$\begin{aligned} &square &&:: Int \rightarrow Int \\ &square\ n &&= n * n \end{aligned}$$

The integer value that the function takes as argument is referred using *n*. The result of the function its simply to compute the square of *n*. Intermediate values can be computed inside the function definition. For example, to define a function that computes the cube (n^3) of an integer value, using the *square* function:

```
cube   :: Int → Int
cube n = let
           sq = square n
         in
           sq ∗ n
```

Where, the *let* keyword is used to start a section of intermediate calculations, and the *in* keyword defines the final expression that calculates the function result.

The $ sign is used in function definitions to avoid the use of parenthesis, anything appearing after takes precedence. The following expressions are equivalent:

```
ghci> square (3 + 2)
25
ghci> square $ 3 + 2
25
```

Declaring New Data Types

New data types can be defined in Haskell using the *data* keyword. For example, the following statement defines a new data type called *Point2D*, which is composed of two integers (*x* and *y*, the point coordinates):

data *Point2D* = *Point2D* { *x* :: *Int*, *y* :: *Int* }

New members of this structure are created using the constructor (a function) that is automatically available, and passing the corresponding values is order:

```
ghci> Point2D 4 6
Point2D {x = 4, y = 6}
```

The names of the fields in the data type definition are used as *acessors* to the values. For example, to get the *x* coordinate of a point *p*:

```
ghci> let p = Point2D 4 6
ghci> x p
4
```

Sometimes, no new data structure is actually required, but to be more clear on values semantics, an alias can be given to other types. For example, in the previous *Point2D* data type definition, the *Int* value is the type for coordinates, to be more explicit an alias for the *Int* type named *Coordinate* can be created using the *type* keyword:

type *Coordinate* = *Int*

Now, the *Coordinate* type can be used, instead of *Int*. Updating the previous example, the data type definition can be written as:

data *Point2D* = *Point2D* { *x* :: *Coordinate*, *y* :: *Coordinate* }

A new data type can also be defined using possible alternatives. For example the boolean data type has two alternatives: true or false, different alternatives are grouped together using a vertical bar:

data *Bool* = *True* | *False*

Lists

A list stores a set of elements of the same type. A list is the denoted by a square brackets [and], and its elements are separated by commas. For example a list of integers:

```
[1,2,3,4,5,6]
```

Or a list of strings:

```
["hello", "world"]
```

Concatenating lists is done using the double $+$ sign operator:

```
ghci> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
```

And, strings are also lists of characters:

```
ghci> "hello" ++ " " ++ "world"
"hello world"
```

The .. operator is used to create lists:

```
ghci> [1..10]
[1,2,3,4,5,6,7,8,9,10]
```

A different step can be used:

```
ghci> [1,3..10]
[1,3,5,7,9]
```

And an infinite list can be created, if no upper bound is given:

```
ghci> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,
25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,
46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,...
```

Lists can also be defined by comprehension, using expressions similar to the common mathematical notation:

```
ghci> [ x | x <- [1..10] ]
[1,2,3,4,5,6,7,8,9,10]
ghci> [ x | x <- [1..10], x > 5 ]
[6,7,8,9,10]
ghci> [ square x | x <- [1..10] ]
[1,4,9,16,25,36,49,64,81,100]
ghci> [ (x,y) | x <- [1,2], y <- [3,4] ]
[(1,3),(1,4),(2,3),(2,4)]
```

The @ sigil has a special meaning in list comprehension expressions, it is used to select elements from a list when alternatives are available. For example, to select only the *True* elements from a list of elements of the *Boolean* type defined earlier:

```
ghci> [ x | x @ True <- [True, False, True] ]
[True,True]
```

Some Built-in Functions

This section describes some commonly used functions, available in Haskell. The *map* function, given a function and a list, applies the argument function to every element of the list. The result is a list containing the individual results of calling the function given as argument for each element in the original list. The following example applies the *square* function to a list of numbers:

```
ghci> map square [1,2,3,4,5]
[1,4,9,16,25]
```

The *filter* function takes as argument a filtering function (required to return a boolean value) and a list of elements, and returns the list of elements for which the filtering function returns a true value. The following example filters the set of numbers that are greater that 5:

```
ghci> filter (>5) [1..10]
[6,7,8,9,10]
```

The *zip* function returns the pairwise interleaving of the two lists given as arguments, for example:

```
ghci> zip ['a','b','c'] [1,2,3]
[('a',1),('b',2),('c',3)]
```

The *concat* function concatenates lists of lists in a single list, for example:

```
ghci> concat [[1,2],[3,4]]
[1,2,3,4]
```

Miscellaneous

Comments for a single line are declared using a double dash:

```
-- This is a single line comment
```

Multiline comments usa the following syntax:

```
{-
  Multiline
  comment.
-}
```

Appendix B

Introduction to Template Toolkit

Template Toolkit (henceforth abbreviated TT) is a fast, all purpose, template processing system. It is often used to produce HTML, but can be used to efficiently produce other formats (e.g., XML, plain text). TT has a simple templating language that is used to write templates. Other tools can use the templates to build their final (or any required intermediate output). For example, the OTK toolkit described in Chapter 6 uses templates to build SPARQL queries. Templates can be stored in their own files, so they can be edited by people that are not proficient with the programming language in which the actual tool is written (e.g., data experts, output format expert). TT is written in Perl. This appendix briefly introduces the templating notation for writing templates.

The major purpose of a template when processed, is to output a set of static data previously defined, and a set of dynamic data, computed in runtime. The template weaves both data sets together, and then produces the devised output. For example, the following template snippet can be used to build the header section of an HTML file:

```
<HEAD>
  <TITLE>[% title %]</TITLE>
</HEAD>
```

Where the `HEAD` and `TITLE` tags are static, they do not change between template processing, but the actual title of the page can change. Everything between `[%` and `%]` is expected to be written in TT templating language, and is processed by the template engine to produce the desired output, while everything outside is static data to be written directly to the output. In this template `title` is a variable, when the template is processed it is replaced by an arbitrary value set by the code that calls the template. The final output the snippet of HTML where the title variable (including the delimiter tags) is replaced by the title of the page, hence producing the final output. For example, the following Perl code can be used to call this template:

```
my $tt = Template->new;
my $vars = { title => "Template Toolkit Page" };
$tt->process('header.tt', $vars);
```

Where a new template processor is created The result of executing this code would be:

```
<HEAD>
  <TITLE>Template Toolkit Page</TITLE>
</HEAD>
```

Where the `title` variable was replaced with the value defined in `$vars`, and everything else in the template was simply piped to the output.

TT allows for more complex expressions to be used inside templates. In fact, there is almost a programming language to write templates [29]. The remaining of this introduction to TT introduces in more detail some directives commonly used in templates illustrated in this document. For more details and a complete description refer to the official documentation and related resources[1].

Include

The `INCLUDE` directive is used to insert the result of processing a template, inside another template. For example, the following template builds a HTML page by including some HTML directives, and including the result of processing the `header.tt` and `body.tt` templates to compose the entire output.

```
<!DOCTYPE html>
<HTML>
  [% INCLUDE 'header.tt' %]
  [% INCLUDE 'body.tt' %]
```

[1]Available from: `http://www.template-toolkit.org/` (Last accessed: 08-09-2014).

```
</HTML>
```

The directive is replaced with the result of processing the included template in the output. This is a useful approach to keep templates simple, and easy to maintain, and is also used to generate dynamic content in runtime, by selecting which templates to include dynamically.

Foreach

The `FOREACH` directive is used to iterate over lists, processing the foreach block for each element. For example, the following template builds an unsorted list using HTML, for the elements in the `persons` array. In every new iteration of the cycle, the `person` variable is instantiated with the next element in the array.

```
<UL>
  [% FOREACH person IN persons %]
    <LI>[% person %]</LI>
  [% END %]
</UL>
```

For example, the following snippet of Perl code, using the above template:

```
my $tt = Template->new;
my $vars = { persons => [ 'Ann', 'John', 'Peter', 'Sarah' };
$tt->process('foreach.tt', $vars);
```

Would produce the following output:

```
<UL>
  <LI>Ann</LI>
  <LI>John</LI>
  <LI>Peter</LI>
  <LI>Sarah</LI>
</UL>
```

Appendix C

Introduction to SPARQL and OTK Queries

SPARQL is a query language for databases, or documents, that handle data in RDF format. It is used to query and update information stored in RDF format, in the same sense as SQL is used to query and update information stored in a relational database. A SPARQL query consists in patterns of triples, including disjunctions and conjunctions of triples. A triple is defined by three elements that unambiguously define three elements, a triple definition ends with a single dot, or may end with a semicolon, in which case the source (the first element of the triple) is assumed to be the same as in the previous triple.

Conclave OTK uses a set of SPARQL queries to perform the actions required for the operations provided by its API, since the ontologies are stored in a RDF format. SPARQL queries are defined in templates, and the actual queries executed are built in runtime. There are two versions of SPARQL: (i) SPARQL 1.0 defines queries for retrieving information; and, (ii) SPARQL 1.1 defines queries to update and insert new data; both are recommended by W3C. SPARQL 1.0 specifics four query forms for retrieving data, OTK only explores one form: SELECT queries, which are used to retrieve data in a table like format (e.g., XML, CSV). SPARQL 1.1 specifies its own query forms, OTK only uses INSERT queries, to add new information to the ontology.

All query forms allow a block for setting prefixes, that define shortcuts to write elements. Variables are defined using a ? prefix. The graph, i.e., defines the triples data set to use, in OTK a graph is used to store one individual ontology (e.g., the program ontology, the problem ontology). For example, the following query retrieves the available set of classes from the ontology:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT ?class
FROM <[% graph %]>
WHERE {
  ?class rdf:type owl:Class .
}
```

It stars by defining two prefixes: “rdf” and “owl”, i.e., writing “rdf:type” is exactly the same as writing “http://www.w3.org/1999/02/22-rdf-syntax-ns#type”, and “owl:class” is exactly the same as writing “http://www.w3.org/2002/07/owl#Class”. Also in this query ?class acts as a variable, which means that in the query result (a set) this variable is the only variable being instantiated, once for each class available in the ontology, with the name of the class. The FROM clause defines from which graph (ontology) to retrieve the information.

Table C.1 describes some templates used in OTK, to render common queries. All templates assume the following set of prefixes is defined.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
```

Add Class	Get Classes

`INSERT DATA {`
`  GRAPH <[% graph %]> {`
`    [% name %] rdf:type owl:Class .`
`    [% FOREACH c IN parents -%]`
`    [% name %] rdfs:subClassOf [% c %] .`
`    [%- END %]`
`  }`
`}`
	`SELECT ?c`
	`FROM <[% graph %]>`
	`WHERE {`
	`  ?c rdf:type owl:Class`
	`}`

Add Instance	**Get Instances**

`INSERT DATA {`
`  GRAPH <[% graph %]> {`
`    [% name %] rdf:type [% class %] ;`
`      rdf:type owl:NamedIndividual .`
`  }`
`}`
	`SELECT ?i`
	`FROM <[% graph %]>`
	`WHERE {`
	`  ?i rdf:type [% class %]`
	`}`

Add Object Propriety

```
INSERT DATA {
  GRAPH <[% graph %]> {
    [% subject %] [% relation %] [% target %] .
    [% relation %] rdf:type owl:ObjectProperty .
  }
}
```

Add Data Propriety

```
INSERT DATA {
  GRAPH <[% graph %]> {
    [% subject %] [% relation %] "[% target %]"^^xsd:[% type %] .
    [% relation %] rdf:type owl:DatatypeProperty .
  }
}
```

Table C.1: OTK query templates for some common operations.

Appendix D

Program Ontology Template

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

<!DOCTYPE rdf:RDF [
    <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>

<rdf:RDF xmlns="[% base_uri %]"
     xml:base="[% base_uri %]"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <owl:Ontology rdf:about="[% base_uri %]"/>

    <!-- OBJECT PROPERTIES -->
    [% INCLUDE 'obj_props.tt' %]

    <!-- DATA PROPERTIES -->
    [% INCLUDE 'data_props.tt' %]

    <!-- CLASSES -->
    [% INCLUDE 'classes.tt' %]

</rdf:RDF>
```

Object Proprieties Template

```
<owl:ObjectProperty rdf:about="#hasFunctionCall">
    <rdf:type rdf:resource="&owl;TransitiveProperty"/>
    <rdfs:range rdf:resource="#Function"/>
    <rdfs:domain rdf:resource="#ProgramElement"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasIdentifier">
    <rdf:type rdf:resource="&owl;FunctionalProperty"/>
    <rdfs:range rdf:resource="#Identifier"/>
    <rdfs:domain rdf:resource="#Function"/>
    <rdfs:domain rdf:resource="#TypeDefinition"/>
    <rdfs:domain rdf:resource="#Variable"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasType">
    <rdfs:domain rdf:resource="#ProgramElement"/>
    <rdfs:range rdf:resource="#TypeDefinition"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#inFile">
    <rdf:type rdf:resource="&owl;TransitiveProperty"/>
    <rdfs:range rdf:resource="#File"/>
    <rdfs:domain rdf:resource="#ProgramElement"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#inFunction">
    <rdfs:range rdf:resource="#Function"/>
    <rdfs:domain rdf:resource="#LocalVariable"/>
    <rdfs:domain rdf:resource="#Parameter"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#inMethod">
    <rdfs:range rdf:resource="#Method"/>
    <rdfs:domain rdf:resource="#LocalVariable"/>
    <rdfs:domain rdf:resource="#Parameter"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#inClass">
    <rdfs:range rdf:resource="#Class"/>
    <rdfs:domain rdf:resource="#ClassVariable"/>
    <rdfs:domain rdf:resource="#Method"/>
</owl:ObjectProperty>
```

Data Proprieties Template

```
<owl:DatatypeProperty rdf:about="#hasIdString">
    <rdfs:domain rdf:resource="Identifier"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:about="#hasFullPath">
    <rdfs:domain rdf:resource="#File"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
```

```
<owl:DatatypeProperty rdf:about="#hasFileName">
    <rdfs:domain rdf:resource="#File"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:about="#isMain">
    <rdf:type rdf:resource="&owl;FunctionalProperty"/>
    <rdfs:domain rdf:resource="#Function"/>
    <rdfs:range rdf:resource="&xsd;boolean"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:about="#hasLineBegin">
    <rdfs:domain rdf:resource="#ProgramElement"/>
    <rdfs:range rdf:resource="&xsd;int"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:about="#hasLineEnd">
    <rdfs:domain rdf:resource="#ProgramElement"/>
    <rdfs:range rdf:resource="&xsd;int"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:about="#hasSplits">
    <rdfs:domain rdf:resource="#Identifier"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:about="#hasTerms">
    <rdfs:domain rdf:resource="#Identifier"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
```

Classes Template

```
<owl:Class rdf:about="#Comment">
    <rdfs:subClassOf rdf:resource="#ProgramElement"/>
</owl:Class>
<owl:Class rdf:about="#Conditional">
    <rdfs:subClassOf rdf:resource="#Statement"/>
</owl:Class>
<owl:Class rdf:about="#CustomStruct">
    <rdfs:subClassOf rdf:resource="#TypeDefinition"/>
</owl:Class>
<owl:Class rdf:about="#Expression">
    <rdfs:subClassOf rdf:resource="#ProgramElement"/>
</owl:Class>
<owl:Class rdf:about="#File">
    <rdfs:subClassOf rdf:resource="&owl;Thing"/>
</owl:Class>
<owl:Class rdf:about="#Macro">
    <rdfs:subClassOf rdf:resource="#ProgramElement"/>
</owl:Class>
<owl:Class rdf:about="#TypeDecl">
```

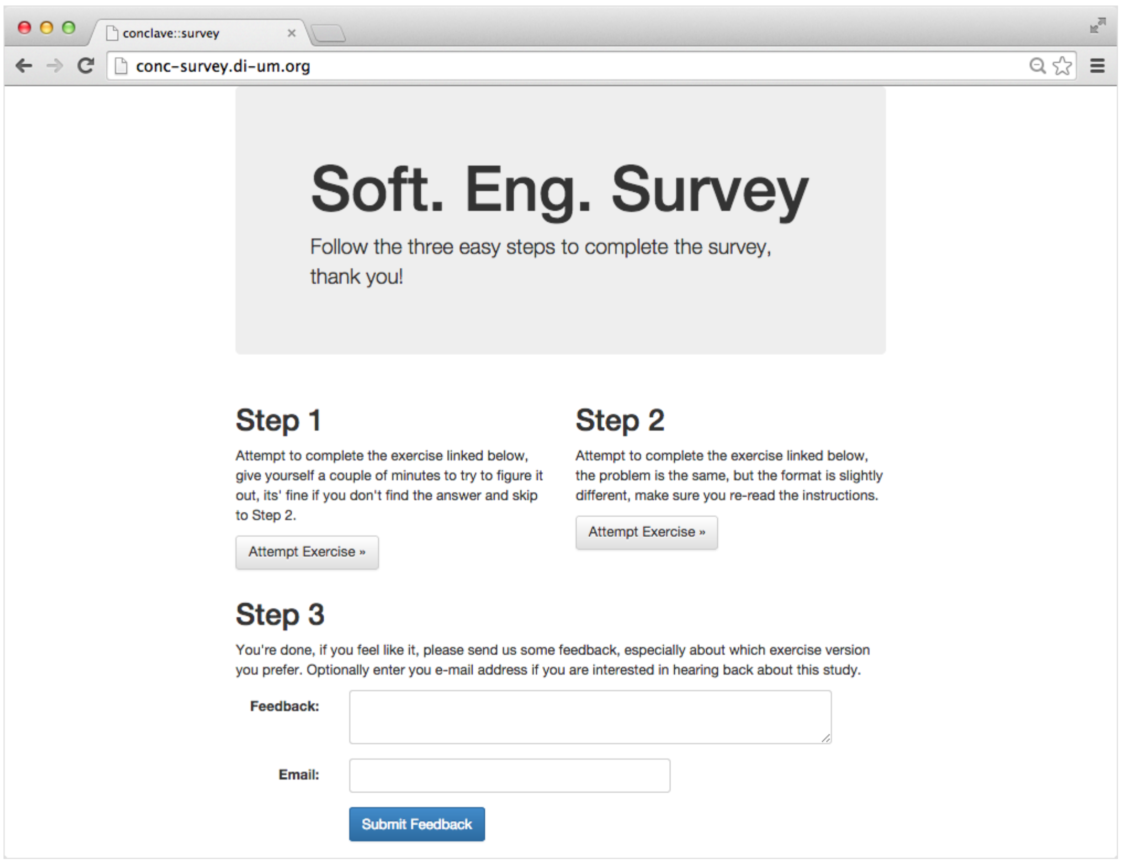
```
    <rdfs:subClassOf rdf:resource="#ProgramElement"/>
</owl:Class>
<owl:Class rdf:about="#Function">
    <rdfs:subClassOf rdf:resource="#ProgramElement"/>
</owl:Class>
<owl:Class rdf:about="#Method">
    <rdfs:subClassOf rdf:resource="#ProgramElement"/>
</owl:Class>
<owl:Class rdf:about="#Class">
    <rdfs:subClassOf rdf:resource="#ProgramElement"/>
</owl:Class>
<owl:Class rdf:about="#Constructor">
    <rdfs:subClassOf rdf:resource="#ProgramElement"/>
</owl:Class>
<owl:Class rdf:about="#GlobalVariable">
    <rdfs:subClassOf rdf:resource="#Variable"/>
</owl:Class>
<owl:Class rdf:about="#ClassVariable">
    <rdfs:subClassOf rdf:resource="#Variable"/>
</owl:Class>
<owl:Class rdf:about="#Identifier">
    <rdfs:subClassOf rdf:resource="#ProgramElement"/>
</owl:Class>
<owl:Class rdf:about="#LocalVariable">
    <rdfs:subClassOf rdf:resource="#Variable"/>
</owl:Class>
<owl:Class rdf:about="#Loop">
    <rdfs:subClassOf rdf:resource="#Statement"/>
</owl:Class>
<owl:Class rdf:about="#NumberType">
    <rdfs:subClassOf rdf:resource="#TypeDefinition"/>
</owl:Class>
<owl:Class rdf:about="#Parameter">
    <rdfs:subClassOf rdf:resource="#Variable"/>
</owl:Class>
<owl:Class rdf:about="#PointerType">
    <rdfs:subClassOf rdf:resource="#TypeDefinition"/>
</owl:Class>
<owl:Class rdf:about="#ProgramElement">
    <rdfs:subClassOf rdf:resource="&owl;Thing"/>
</owl:Class>
<owl:Class rdf:about="#Statement">
    <rdfs:subClassOf rdf:resource="#ProgramElement"/>
</owl:Class>
<owl:Class rdf:about="#StringType">
    <rdfs:subClassOf rdf:resource="#TypeDefinition"/>
</owl:Class>
<owl:Class rdf:about="#TypeDefinition">
    <rdfs:subClassOf rdf:resource="#ProgramElement"/>
```

```
</owl:Class>
<owl:Class rdf:about="#Variable">
    <rdfs:subClassOf rdf:resource="#ProgramElement"/>
</owl:Class>
```

Appendix E

Survey Pages

Front Page



Soft. Eng. Survey

Follow the three easy steps to complete the survey, thank you!

Step 1

Attempt to complete the exercise linked below, give yourself a couple of minutes to try to figure it out, its' fine if you don't find the answer and skip to Step 2.

Step 2

Attempt to complete the exercise linked below, the problem is the same, but the format is slightly different, make sure you re-read the instructions.

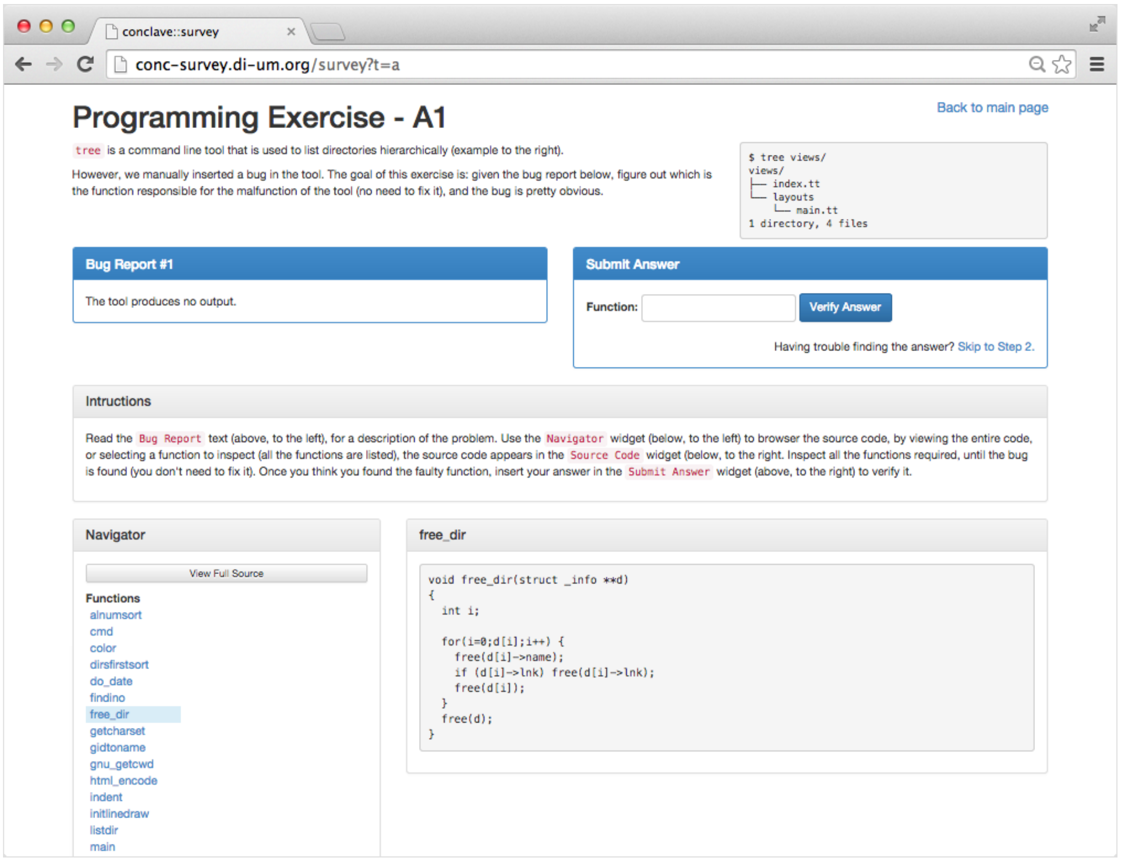
Step 3

You're done, if you feel like it, please send us some feedback, especially about which exercise version you prefer. Optionally enter you e-mail address if you are interested in hearing back about this study.

Feedback:

Email:

Exercise - Version A



conclave::survey

conc-survey.di-um.org/survey?t=a

Programming Exercise - A1

Back to main page

`tree` is a command line tool that is used to list directories hierarchically (example to the right).

However, we manually inserted a bug in the tool. The goal of this exercise is: given the bug report below, figure out which is the function responsible for the malfunction of the tool (no need to fix it), and the bug is pretty obvious.

```
$ tree views/
views/
├── index.tt
└── layouts
    └── main.tt
1 directory, 4 files
```

Bug Report #1

The tool produces no output.

Submit Answer

Function: Verify Answer

Having trouble finding the answer? Skip to Step 2.

Intructions

Read the `Bug Report` text (above, to the left), for a description of the problem. Use the `Navigator` widget (below, to the left) to browser the source code, by viewing the entire code, or selecting a function to inspect (all the functions are listed), the source code appears in the `Source Code` widget (below, to the right. Inspect all the functions required, until the bug is found (you don't need to fix it). Once you think you found the faulty function, insert your answer in the `Submit Answer` widget (above, to the right) to verify it.

Navigator

View Full Source

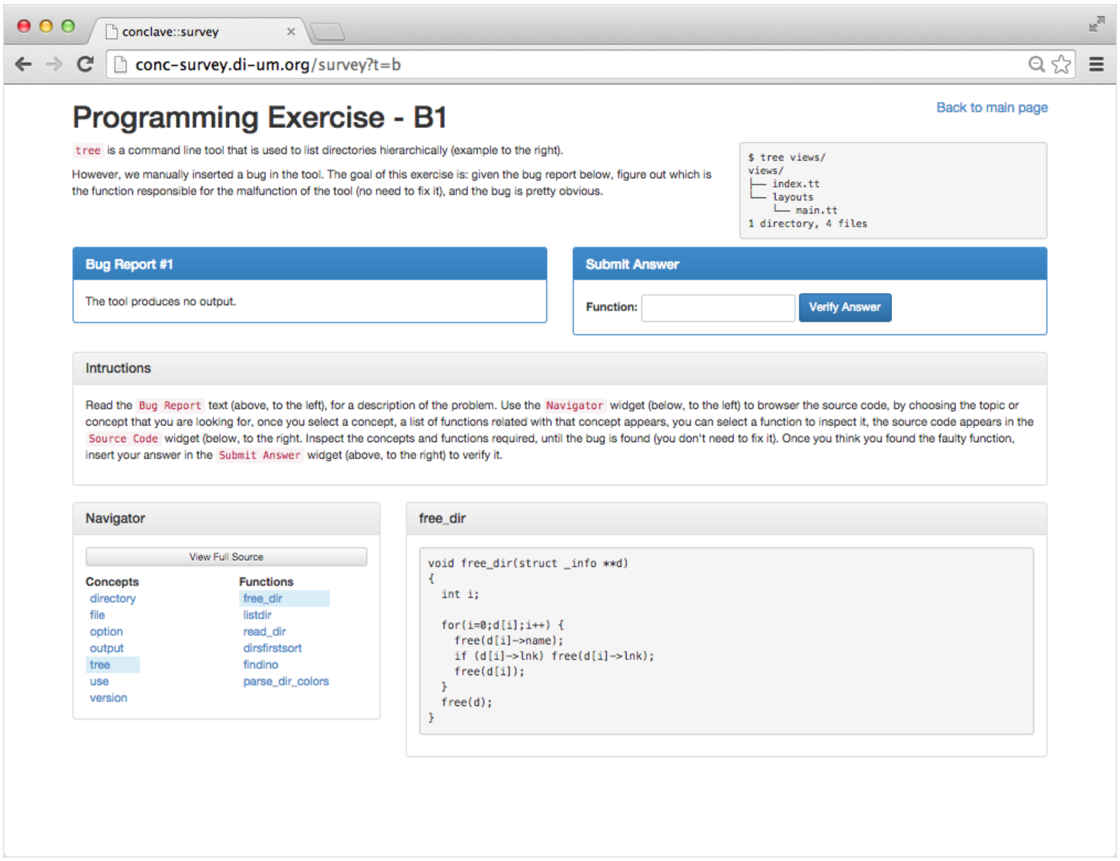
Functions
- alnumsort
- cmd
- color
- dirsfirstsort
- do_date
- findino
- free_dir
- getcharset
- gidtoname
- gnu_getcwd
- html_encode
- indent
- initlinedraw
- listdir
- main

free_dir

```
void free_dir(struct _info **d)
{
  int i;

  for(i=0;d[i];i++) {
    free(d[i]->name);
    if (d[i]->lnk) free(d[i]->lnk);
    free(d[i]);
  }
  free(d);
}
```

Exercise - Version B



Programming Exercise - B1

`tree` is a command line tool that is used to list directories hierarchically (example to the right).

However, we manually inserted a bug in the tool. The goal of this exercise is: given the bug report below, figure out which is the function responsible for the malfunction of the tool (no need to fix it), and the bug is pretty obvious.

```
$ tree views/
views/
├── index.tt
└── layouts
    └── main.tt
1 directory, 4 files
```

Bug Report #1

The tool produces no output.

Submit Answer

Function:

Verify Answer

Intructions

Read the `Bug Report` text (above, to the left), for a description of the problem. Use the `Navigator` widget (below, to the left) to browser the source code, by choosing the topic or concept that you are looking for, once you select a concept, a list of functions related with that concept appears, you can select a function to inspect it, the source code appears in the `Source Code` widget (below, to the right. Inspect the concepts and functions required, until the bug is found (you don't need to fix it). Once you think you found the faulty function, insert your answer in the `Submit Answer` widget (above, to the right) to verify it.

Navigator

View Full Source

Concepts	Functions
directory	free_dir
file	listdir
option	read_dir
output	dirsfirstsort
tree	findino
use	parse_dir_colors
version	

free_dir

```
void free_dir(struct _info **d)
{
  int i;

  for(i=0;d[i];i++) {
    free(d[i]->name);
    if (d[i]->lnk) free(d[i]->lnk);
    free(d[i]);
  }
  free(d);
}
```

Bibliography

[1] *Oxford English Dictionary, Second Edition*. 1989.

[2] IEEE Standard for software daintenance. *IEEE Std 1219-1998*, 1998.

[3] S.L. Abebe and P. Tonella. Natural language parsing of program element names for concept extraction. In *IEEE 18th International Conference on Program Comprehension (ICPC)*, pages 156–159. IEEE, 2010.

[4] J.J. Almeida, S. Araújo, N. Carvalho, I. Dias, A. Oliveira, A. Santos, and A. Simões. The Per-Fide corpus: A new resource for corpus-based terminology, contrastive linguistics and translation studies. In Tony Berber Sardinha and Telma de Lurdes São Bento Ferreira, editors, *Working with Portuguese Corpora*, pages 177–200. Bloomsbury Publishing, April 2014.

[5] Gi. Ammons, D. Mandelin, R. Bodík, and J.R. Larus. Debugging temporal specifications with concept analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 182–195, New York, NY, USA, 2003. ACM.

[6] G. Antoniol and Y.-G. Guéhéneuc. Feature identification: a novel approach and a case study. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM)*, pages 357–366, Sept 2005.

[7] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *Software Engineering, IEEE Transactions on*, 28(10):970–983, 2002.

[8] O. Arafat and D. Riehle. The commenting practice of open source. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 857–864. ACM, 2009.

[9] D. Beckett and B. McBride. RDF/XML Syntax Specification (Revised). *W3C Recommendation*, 10, 2004.

[10] T.J. Biggerstaff, B.G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th international conference on Software Engineering*, pages 482–498. IEEE Computer Society Press, 1993.

[11] J. Bohnet, S. Voigt, and J. Dollner. Locating and understanding features of complex software systems by synchronizing time-, collaboration- and code-focused views on execution traces. In *16th IEEE International Conference on Program Comprehension (ICPC)*, pages 268–271, June 2008.

[12] C.A. Brewster and Y. Wilks. Ontologies, taxonomies, thesauri learning from texts. 2004.

[13] R. Brooks. Using a behavioral theory of program comprehension in software engineering. In *Proceedings of the 3rd International Conference on Software Engineering*, ICSE '78, pages 196–201, Piscataway, NJ, USA, 1978. IEEE Press.

[14] R. Brooks. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies*, 18(6):543–554, 1983.

[15] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Improving the tokenisation of identifier names. In *25th European Conference on Object-Oriented Programming*. Springer, Jul 2011.

[16] B. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *6th Working Conference on Reverse Engineering*, pages 112–122. IEEE, 1999.

[17] B. Caprile and P. Tonella. Restructuring program identifier names. In *Proceedings of the International Conference on Software Maintenance*, pages 97–107. IEEE, 2000.

[18] N.R. Carvalho. An ontology toolkit for problem domain concept location in program comprehension. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1415–1418. IEEE Press, 2013.

[19] N.R. Carvalho, J.J. Almeida, P.R. Henriques, and M.J.V. Pereira. Conclave: Ontology-driven measurement of semantic relatedness between source code elements and problem domain concepts. In Beniamino Murgante, Sanjay Misra, AnaMariaA.C. Rocha, Carmelo Torre, JorgeGustavo Rocha, MariaIrene Falcão, David Taniar, BernadyO. Apduhan, Osvaldo Gervasi, and other, editors, *Computational Science and Its Applications – ICCSA 2014*, volume 8584 of *Lecture Notes in Computer Science*, pages 116–131. Springer International Publishing, 2014.

[20] N.R. Carvalho, J.J. Almeida, P.R. Henriques, and M.J. Varanda. From source code identifiers to natural language terms. *Journal of Systems and Software*, 2014 (forthcoming).

[21] N.R. Carvalho, J.J. Almeida, M.J.V. Pereira, and P.R. Henriques. Probabilistic synset based concept location. In Alberto Simões, Ricardo Queirós, and Daniela da Cruz, editors, *SLATE'12 — Symposium on Languages, Applications and Technologies*, volume 21, pages 239–253. OASIC – Open Access Series in Informatics, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany, June 2012.

[22] N.R. Carvalho, J.J. Almeida, M.J.V. Pereira, and P.R. Henriques. Probabilistic synset based concept location. In *SLATE'12 — Symposium on Languages, Applications and Technologies*, pages 239–253, June 2012.

[23] N.R. Carvalho, J.J. Almeida, M.J.V. Pereira, and P.R. Henriques. Conclave: Writing Programs to Understand Programs. In Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões, editors, *3rd Symposium on Languages, Applications and Technologies*, volume 38 of *OpenAccess Series in Informatics (OASIcs)*, pages 19–34, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[24] N.R. Carvalho, J.J. Almeida, and A. Simões. Weaving OML in a general purpose programming language. In Raul Barbosa and Luis Caires, editors, *INForum'11 — Simpósio de Informática (CoRTA2011 track)*, pages 184–197, Coimbra, Portugal, Setembro 2011.

[25] N.R. Carvalho, A. Simões, and J.J. Almeida. OML: a scripting approach for manipulating ontologies. In *CISTI'11 - 6ª Conferência Ibérica de Sistemas e Tecnologias de Informação*, pages 624–629, Chaves, Portugal, June 2011.

[26] N.R. Carvalho, A. Simões, and J.J. Almeida. Open source software documentation mining for quality assessment. In *Advances in Information Systems and Technologies*, volume 206 of *Advances in Intelligent Systems and Computing*, pages 785–794. Springer Berlin Heidelberg, 2013.

[27] N.R. Carvalho, A. Simões, and J.J. Almeida. DMOSS: open source software documentation assessment. *Computer Science and Information Systems*, 2014 (forthcoming).

[28] N.R. Carvalho, A. Simões, J.J. Almeida, P.R. Henriques, and M.J.V. Pereira. PFTL: A systematic approach for describing filesystem tree processors. In Raul Barbosa and Luis Caires, editors, *INForum'11 — Simpósio de Informática (CoRTA2011 track)*, pages 222–233, Coimbra, Portugal, Setembro 2011.

[29] D. Chamberlain, D. Cross, and A. Wardley. *Perl Template Toolkit*. O'Reilly Media, Inc., 2011.

[30] E. Charniak and D. McDermott. *Introduction to Artificial Intelligence*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1985.

[31] K. Chen and V. Rajlich. Case study of feature location using dependence graph. *International Conference on Program Comprehension*, 0:241, 2000.

[32] K. Chen and V. Rajlich. Ripples: Tool for change in legacy software. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, ICSM '01, Washington, DC, USA, 2001. IEEE Computer Society.

[33] E.J. Chikofsky and J.H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE software*, 7(1):13–17, 1990.

[34] B. Cleary, C. Exton, J. Buckley, and M. English. An empirical analysis of information retrieval based concept location techniques in software comprehension. *Empirical Software Engineering*, 14(1):93–130, 2009.

[35] A. Corazza, S. Di Martino, and V. Maggio. Linsen: An efficient approach to split identifiers and expand abbreviations. In *28th IEEE International Conference on Software Maintenance (ICSM)*, pages 233–242. IEEE, 2012.

[36] J. Corbet, G. Kroah-Hartman, and A. McPherson. Linux kernel development. White Paper - The Linux Foundation, December 2010.

[37] T.A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[38] S.C. Deerwester, S.T. Dumais, T.K. Landauer, G.W. Furnas, and R.A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[39] T.S. Dillon, E. Chang, and P. Wongthongtham. Ontology-based software engineering - software engineering 2.0. In *19th Australian Conference on Software Engineering (ASWEC)*, pages 13–23, March 2008.

[40] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol. Can better identifier splitting techniques help feature location? In *IEEE 19th International Conference on Program Comprehension (ICPC)*, pages 11–20. IEEE, 2011.

[41] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[42] M. Dowson. The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2), March 1997.

[43] A. Drozdek. *Data Structures and algorithms in C++*. Cengage Learning, 2012.

[44] S. Düwel. Enhancing system analysis by means of formal concept analysis. In *Conference on advanced information systems engineering 6th doctoral consortium*, 1999.

[45] M. Eaddy, AV. Aho, G. Antoniol, and Y.-G. Guéhéneuc. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *16th IEEE International Conference on Program Comprehension (ICPC)*, pages 53–62, June 2008.

[46] D. Edwards, S. Simmons, and N. Wilde. An approach to feature location in distributed systems. *Journal of Systems and Software*, 79(1):57 – 68, 2006.

[47] A. Egyed, G. Binder, and P. Grunbacher. Strada: A tool for scenario-based feature-to-code trace detection and analysis. In *Companion to the Proceedings of the 29th International Conference on Software Engineering*, ICSE COMPANION '07, pages 41–42, Washington, DC, USA, 2007. IEEE Computer Society.

[48] T. Eisenbarth, R. Koschke, and D. Simon. Derivation of feature component maps by means of concept analysis. In *5th European Conference on Software Maintenance and Reengineering*, pages 176–179, 2001.

[49] T. Eisenbarth, R. Koschke, and D. Simon. Feature-driven program understanding using concept analysis of execution traces. In *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*, pages 300–309, 2001.

[50] A.D. Eisenberg and K. De Volder. Dynamic feature traces: finding features in unfamiliar code. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 337–346, Sept 2005.

[51] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 71–80. IEEE, 2009.

[52] H. Feild, D. Binkley, and D. Lawrie. An empirical comparison of techniques for extracting concept abbreviations from identifiers. In *Proceedings of the International Conference on Software Engineering and Applications*, 2006.

[53] W. Ford, W. Topp, and W.H. Ford. *Data Structures with C++ Using STL, 2/e*. Pearson Education India, 2002.

[54] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1997.

[55] D. Gašević, N. Kaviani, and M. Milanović. Ontologies and software engineering. In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 593–615. Springer Berlin Heidelberg, 2009.

[56] L. Guerrouj. *Context-Aware Source Code Identifier Splitting and Expansion for Software Maintenance*. PhD thesis, École Polytechnique de Montréal, 2013.

[57] L. Guerrouj, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc. Tidier: an identifier splitting approach using speech recognition techniques. *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.

[58] L. Guerrouj, P. Galinier, Y. Gueheneuc, G. Antoniol, and M. Di Penta. Tris: A fast and accurate identifiers splitting and expansion algorithm. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 103–112. IEEE, 2012.

[59] M. A. K Halliday. Language as system and language as instance: The corpus as a theoretical construct. 65:61–77, 1992.

[60] S. Harispe, S. Ranwez, S. Janaqi, and J Montmain. Semantic measures for the comparison of units of language, concepts or entities from text and knowledge base analysis. *CoRR*, 2013.

[61] E. Hill, D. Binkley, D. Lawrie, L. Pollock, and K. Vijay-Shanker. An empirical study of identifier splitting techniques. *Empirical Software Engineering*, 19:1–27, 2013.

[62] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 232–242, Washington, DC, USA, 2009. IEEE Computer Society.

[63] I. Horrocks, P.F. Patel-Schneider, and F. van Harmelen. From SHIQ and RDF to OWL: the making of a Web Ontology Language. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(1):7–26, 2003.

[64] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Proceedings of the 14th International Conference on Software Engineering*, ICSE '92, pages 392–411, New York, NY, USA, 1992. ACM.

[65] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, January 1990.

[66] IANA. MIME Media Types. Web site: http://www.iana.org/assignments/media-types/index.html (Last accessed: 02-09-2014), 2014.

[67] T. Janssen, R. Abreu, and A.J.C. van Gemund. Zoltar: A toolset for automatic fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 662–664. IEEE, 2009.

[68] D. Jin. *Ontological Adaptive Integration of Reverse Engineering Tools*. PhD thesis, 2004.

[69] T. Joachims. *Text categorization with support vector machines: Learning with many relevant features*. Springer, 1998.

[70] G. Kappel, E. Kapsammer, H. Kargl, G. Kramler, T. Reiter, W. Retschitzegger, W. Schwinger, and M. Wimmer. Lifting metamodels to ontologies: A step to the semantic integration of modeling languages. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 528–542. Springer Berlin Heidelberg, 2006.

[71] W. Keller. Mapping objects to tables. In *Proceedings of the European Conference on Pattern Languages of Programming and Computing*, volume 19, 1997.

[72] G. Kennedy. Introduction to corpus linguistics (studies in language and linguistics). 1998.

[73] C. Kiefer, A. Bernstein, and J. Tappolet. Analyzing software with iSPARQL. In *Proceedings of the International Workshop on Semantic Web Enabled Software Engineering (SWESE)*, 2007.

[74] G. Klyne, J.J. Carroll, and B. McBride. Resource description framework (rdf): Concepts and abstract syntax. *W3C recommendation*, 10, 2004.

[75] H. Knublauch. Ontology-driven software development in the context of the semantic web: An example scenario with protege/owl. In *1st International workshop on the model-driven semantic web (MDSW2004)*. Monterey, California, USA.[WWW document] http://www. knublauch. com/publications/MDSW2004. pdf, 2004.

[76] J. Lacasta, J. Nogueras-Iso, F.J. Lopez-Pellicer, P.R. Muro-Medrano, and F.J. Zarazaga-Soria. ThManager: An open source tool for creating and visualizing SKOS. *Information Technology and Libraries*, 26(3):39–51, 2007.

[77] T.K. Landauer, P.W. Foltz, and D. Laham. An introduction to latent semantic analysis. *Discourse processes*, 25(2-3):259–284, 1998.

[78] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization*, pages 75–86, March 2004.

[79] D. Lawrie and D. Binkley. Expanding identifiers to normalize source code vocabulary. In *27th IEEE International Conference on Software Maintenance (ICSM)*, pages 113–122. IEEE, 2011.

[80] D. Lawrie, D. Binkley, and C. Morrell. Normalizing source code vocabulary. In *17th Working Conference on Reverse Engineering (WCRE)*, pages 3–12. IEEE, 2010.

[81] D. Lawrie, H. Feild, and D. Binkley. Syntactic identifier conciseness and consistency. In *6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 139–148. IEEE, 2006.

[82] D. Lawrie, H. Feild, and D. Binkley. Extracting meaning from abbreviated identifiers. In *7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 213–222. IEEE, 2007.

[83] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *14th International Conference on Program Comprehension (ICPC)*, 2006.

[84] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering*, 3(4):303–318, 2007.

[85] S.W. Lee and R.A Gandhi. Ontology-based active requirements engineering framework. In *Software Engineering Conference, 2005. APSEC '05. 12th Asia-Pacific*. IEEE, Dec 2005.

[86] M.M. Lehman and F.N. Parr. Program evolution and its impact on software engineering. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, pages 350–357, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[87] C.E. Leiserson, R.L. Rivest, C. Stein, and T.H. Cormen. *Introduction to algorithms*. The MIT press, 2001.

[88] D.B. Lenat. CYC: a large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11):33–38, 1995.

[89] S. Letovsky. Cognitive processes in program comprehension. *Journal of Systems and Software*, 7(4):325 – 339, 1987.

[90] V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

[91] D.C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. *Journal of Systems and Software*, 7(4):341 – 355, 1987.

[92] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 234–243. ACM, 2007.

[93] K. Lukoit, N. Wilde, S. Stowell, and T. Hennessey. Tracegraph: immediate visual location of software features. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 33–39, 2000.

[94] N. Madani, L. Guerrouj, M. Di Penta, Y. Gueheneuc, and G. Antoniol. Recognizing words from source code identifiers using speech recognition techniques. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 68–77. IEEE, 2010.

[95] J.I. Maletic and A. Marcus. Using latent semantic analysis to identify similarities in source code to support program understanding. In *Tools with Artificial Intelligence, 2000. ICTAI 2000. Proceedings. 12th IEEE International Conference on*, pages 46–53, 2000.

[96] C.D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.

[97] F. Manola, E. Miller, et al. RDF Primer. *W3C Recommendation*, 10, 2004.

[98] A. Marcus and J.I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 125–135. IEEE, 2003.

[99] A. Marcus, A. Sergeyev, V. Rajlich, and J.I. Maletic. An information retrieval approach to concept location in source code. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 214–223. IEEE, 2004.

[100] J.H. Martin and D. Jurafsky. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, 2nd Edition*. Prentice Hall, 2009.

[101] P. Martins, N.R Carvalho, J.P. Fernandes, J.J. Almeida, and J. Saraiva. A framework for modular and customizable software analysis. In *Computational Science and Its Applications–ICCSA 2013*, pages 443–458. Springer, 2013.

[102] J. McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial intelligence*, 13(1):27–39, 1980.

[103] D.L. McGuinness, F. van Harmelen, et al. OWL Web Ontology Language Overview. *W3C Recommendation*, 10:2004–03, 2004.

[104] A. Miles, B. Matthews, D. Beckett, D. Brickley, M. Wilson, and N. Rogers. SKOS: A language to describe simple knowledge structures for the web. In *XTech 2005 Conference Proceedings*, 2005.

[105] A. Miles, B. Matthews, M. Wilson, and D. Brickley. SKOS Core: Simple Knowledge Organisation for the Web. In *Proceedings of the International Conference on Dublin Core and Metadata Applications*, pages 12–15, 2005.

[106] G. Miller and C. Fellbaum. Wordnet: An electronic lexical database, 1998.

[107] G.A. Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological review*, 63(2):81, 1956.

[108] G.A. Miller. WordNet: a lexical database for English. *Communications of the ACM*, 38(11):39–41, 1995.

[109] B. Motik, P. Patel-Schneider, and B Parsia. OWL 2 Web Ontology Language. 2009.

[110] G.C. Murphy, M. Kersten, M.P.. Robillard, and D. Čubranić. The emergent structure of development tasks. In AndrewP. Black, editor, *ECOOP 2005 - Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 33–48. Springer Berlin Heidelberg, 2005.

[111] M.L. Nelson. A survey of reverse engineering and program comprehension. *Arxiv preprint cs/0503068*, 2005.

[112] A. Olszak and B.N. Jørgensen. Featureous: A tool for feature-centric analysis of java software. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pages 44–45, June 2010.

[113] D.L. Parnas and M. Lawford. Inspection's role in software quality assurance. *Software, IEEE*, 20(4):16 – 20, 2003.

[114] T. J. Parr and R. W. Quong. Antlr: A predicated-ll(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[115] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, 19(3):295–341, 1987.

[116] S. Pepper. The TAO of Topic Maps: Finding the Way in the Age of Infoglut. In *Proceedings of XML Europe 2000 Conférence*, 2000.

[117] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. In *The Semantic Web-ISWC 2006*, pages 30–43. Springer, 2006.

[118] M. Petrenko, V. Rajlich, and R. Vanciu. Partial domain comprehension in software evolution and maintenance. In *16th IEEE International Conference on Program Comprehension (ICPC)*, pages 13–22, June 2008.

[119] D. Poshyvanyk, Y.-G. Gueheneuc, A Marcus, G. Antoniol, and V. Rajlich. Combining probabilistic ranking and latent semantic indexing for feature identification. In *14th IEEE International Conference on Program Comprehension (ICPC)*, pages 137–148, 2006.

[120] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, 2007.

[121] D. Poshyvanyk and A. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *15th IEEE International Conference on Program Comprehension (ICPC)*, pages 37–48, June 2007.

[122] D. Poshyvanyk, M. Petrenko, A. Marcus, Xinrong X., and Dapeng L. Source code exploration with google. In *22nd IEEE International Conference on Software Maintenance (ICSM)*, pages 334–338, Sept 2006.

[123] C. Potts. Software-engineering research revisited. *Software, IEEE*, 10(5):19–28, 2002.

[124] E. Prud'Hommeaux, A. Seaborne, et al. Sparql query language for rdf. *W3C recommendation*, 15, 2008.

[125] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 271–278. IEEE, 2002.

[126] D. Ratiu. Reverse engineering domain models from source code. In *International Workshop on Reverse Engineering Models from Software Artifacts (REM'09)*, pages 13–16, 2009.

[127] M. Revelle, B. Dit, and D. Poshyvanyk. Using data fusion and web mining to support feature location in software. In *IEEE 18th International Conference on Program Comprehension (ICPC)*, pages 14–23. IEEE, 2010.

[128] M.P. Robillard. Topology analysis of software dependencies. *ACM Trans. Softw. Eng. Methodol.*, 17(4):18:1–18:36, August 2008.

[129] M.P. Robillard and G.C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 406–416, New York, NY, USA, 2002. ACM.

[130] M.P. Robillard and G.C. Murphy. Automatically inferring concern code from program investigation activities. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 225–234, Oct 2003.

[131] M.P. Robillard and G.C. Murphy. Feat: A tool for locating, describing, and analyzing concerns in source code. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 822–823, Washington, DC, USA, 2003. IEEE Computer Society.

[132] M.P. Robillard and G.C. Murphy. Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol.*, 16(1), February 2007.

[133] S. Rogerson. The chinook helicopter disaster. *IMIS Journal*, 12(2), 2002.

[134] S. Rugaber. Program comprehension for reverse engineering. In *AAAI Workshop on AI and Automated Program Understanding, San Jose, California*, pages 106–110, 1992.

[135] S. Rugaber. The use of domain knowledge in program understanding. *Annals of Software Engineering*, 9(1-2):143–192, 2000.

[136] S. Rugaber and K. Stirewalt. Model-driven reverse engineering. *Software, IEEE*, 21(4):45–53, July 2004.

[137] H. Safyallah and K. Sartipi. Dynamic analysis of software systems using execution pattern mining. In *14th IEEE International Conference on Program Comprehension (ICPC)*, pages 84–88, 2006.

[138] A. Santos, J.J. Almeida, and N.R. Carvalho. Structural alignment of plain text books. In Nicoletta Calzolari et al., editors, *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*, Istanbul, Turkey, may 2012. European Language Resources Association (ELRA).

[139] Z.M. Saul, V. Filkov, P. Devanbu, and C. Bird. Recommending random walks. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 15–24, New York, NY, USA, 2007. ACM.

[140] I. Serra and R. Girardi. A process for extracting non-taxonomic relationships of ontologies from text. *Intelligent Information Management*, 3:119, 2011.

[141] T.M. Shaft and I. Vessey. The relevance of application domain knowledge: Characterizing the computer program comprehension process. *J. Manage. Inf. Syst.*, 15(1):51–78, June 1998.

[142] D. Shepherd, Z.P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to find and understand action-oriented concerns. In *Int. Conf. on Aspect-oriented Software Development*, 2007.

[143] D. Shepherd, L. Pollock, and K. Vijay-Shanker. Towards supporting on-demand virtual remodularization using program graphs. In *Proceedings of the 5th International Conference on Aspect-oriented Software Development*, AOSD '06, pages 3–14, New York, NY, USA, 2006. ACM.

[144] Ben Shneiderman. *Software psychology: Human factors in computer and information systems*. Winthrop Publishers, 1980.

[145] Ben Shneiderman and Richard Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, 8(3):219–238, 1979.

[146] A. Simões. Parallel corpora word alignment and applications. Master's thesis, Escola de Engenharia - Universidade do Minho, 2004.

[147] A. Simões. *Extracção de Recursos de Tradução com base em Dicionários Probabilísticos de Tradução*. PhD thesis, Escola de Engenharia, Universidade do Minho, Braga, 2008.

[148] A. Simões and J.J. Almeida. Library::* — a toolkit for digital libraries. In *ElPub 2002 - Technology Interactions*, 2002.

[149] A. Simões, J.J. Almeida, and N.R. Carvalho. Defining a probabilistic translation dictionaries algebra. In Luís Correia, Luís Paulo Reis, José Cascalho, Luís Gomes, Hélia Guerra, and Pedro Cardoso, editors, *XVI Portuguese Conference on Artificial Inteligence - EPIA*, pages 444–455, Angra do Heroismo, Azores, September 2013.

[150] A. Simões, N.R. Carvalho, and J.J. Almeida. Generating flex lexical analyzers for perl parse::yapp. In Alberto Simões, Ricardo Queirós, and Daniela da Cruz,

editors, *SLATE'12 — Symposium on Languages, Applications and Technologies*, volume 21, pages 239–253. OASIC – Open Access Series in Informatics, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany, June 2012.

[151] Gregor Snelting. Concept lattices in software analysis. In B. Ganter, G. Stumme, and R. Wille, editors, *Formal Concept Analysis*, volume 3626 of *Lecture Notes in Computer Science*, pages 272–287. Springer Berlin Heidelberg, 2005.

[152] Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *Software Engineering, IEEE Transactions on*, pages 595–609, 1984.

[153] M.-A.D Storey, F.D Fracchia, and H.A Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44(3):171–185, 1999.

[154] A. Sureka. Source code identifier splitting using yahoo image and web search engine. In *Proceedings of the First International Workshop on Software Mining*, pages 1–8. ACM, 2012.

[155] T. Tilley, R. Cole, P. Becker, and P. Eklund. A survey of formal concept analysis support for software engineering activities. In B. Ganter, G. Stumme, and R. Wille, editors, *Formal Concept Analysis*, volume 3626 of *Lecture Notes in Computer Science*, pages 250–271. Springer Berlin Heidelberg, 2005.

[156] M. Trifu. Using dataflow information for concern identification in object-oriented software systems. In *Proceedings of European Conference on Software Maintenance and Reengineering (CSMR'08)*, pages 193–202. IEEE, 2008.

[157] B. Vatant. Ontology-driven topic maps. In *XML Europe*, pages 03–03, 2004.

[158] Anneliese Von Mayrhauser and A Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[159] N. Walkinshaw, M. Roper, and M. Wood. Feature location and extraction using landmarks and barriers. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 54–63, Oct 2007.

[160] F.W. Warr and M.P. Robillard. Suade: Topology-based searches for software investigation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 780–783, Washington, DC, USA, 2007. IEEE Computer Society.

[161] N. Wilde, J.A. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 200–205, Nov 1992.

[162] N. Wilde and M.C. Scully. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[163] René Witte, Yonggang Zhang, and Jürgen Rilling. Empowering software maintainers with semantic web technologies. In Enrico Franconi, Michael Kifer, and Wolfgang May, editors, *The Semantic Web: Research and Applications*, volume 4519 of *Lecture Notes in Computer Science*, pages 37–52. Springer Berlin Heidelberg, 2007.

[164] W.E. Wong, S.S. Gokhale, J.R. Horgan, and K.S. Trivedi. Locating program features using execution slices. In *Proceedings of the IEEE Symposium on Application-Specific Systems and Software Engineering and Technology*, pages 194–203, 1999.

[165] M. Würsch, G. Ghezzi, G. Reif, and H.C. Gall. Supporting developers with natural language queries. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 165–174, New York, NY, USA, 2010. ACM.

[166] Suresh Yadla, Jane Huffman Hayes, and Alex Dekhtyar. Tracing requirements to defect reports: an application of information retrieval techniques. *Innovations in Systems and Software Engineering*, 1(2):116–124, 2005.

[167] A.Y. Yao. Cvssearch: Searching through source code using cvs comments. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, ICSM '01, Washington, DC, USA, 2001. IEEE Computer Society.

[168] S. Yong-feng and Z. Yan-ping. Comparison of text categorization algorithms. *Wuhan university Journal of natural sciences*, 9(5):798–804, 2004.

[169] M.V. Zelkowitz and D.R. Wallace. Experimental models for validating technology. *Computer*, 31(5):23–31, 2002.

[170] M. Zhivich and R.K. Cunningham. The real cost of software errors. *IEEE Security and Privacy Magazine*, 2009.