Towards Extracting Domain knowledge from C Code

Author:
Mohammed Tarabain

January 27, 2014

Advisors:
Prof. Dr. rer. nat. habil. Gunter Saake
Dr.-Ing. Jochen Quante, Dr.-Ing. Janet Siegmund
"Writing code is not the problem, understanding the code is the problem" - this saying [7] summarizes how important is domain knowledge in software development and maintenance. Gathering this knowledge is an expensive process, which requires an investment of time, money, resources, and which is very demanding, because knowledge is scattered over various locations within source code. In this work, we propose a method to recover domain knowledge from C code using identifiers. To this end, we extract identifiers from source code, and we use them to generate domain concepts and investigate their interrelations. We describe four use cases, in which the domain concepts and their interrelations are typically used. To evaluate the performance of our approach, we conduct two experiments. Both experiments show promising result, in that our approach misses only few relevant concepts, and rarely generates irrelevant concepts. Currently, our approach is not fully automated, because the user has to traverse through a short list that contains both domain concepts and general concepts, and manually remove the general ones.
Acknowledgement

It would not have been possible to write this thesis without the help and support of the kind people around me, to only some of whom it is possible to give particular mention here.

Above all, I would like to thank my family and especially my mother for their infinite patience and unequivocal support during the preparation of this thesis.

This thesis would not have been possible without the help, support, and patience of my advisors: Gunter Saake, Jochen Quante, and Janet Sigmund. I cannot find words to express my gratitude to them. Their guidance helped me in all the time of research and writing of this thesis.

Last but not the least, I would like to thank Alexander Sung from Robert Bosch GmbH for his fruitful discussions about my evaluation results.
# Contents

List of Figures vi  
List of Tables vii  
List of Code Listings ix  
List of Acronyms x  

## 1 Introduction  
1.1 Goals and Tasks 3  
1.2 Structure 4  

## 2 Background  
2.1 University Management System Source Code Example 6  
2.2 Domain knowledge 8  
2.2.1 Definition and Description of Concepts 8  
2.2.2 Interrelations among Concepts 12  
2.2.2.1 Language-Defined Relations 12  
2.2.2.2 Ontologies 14  
2.2.3 University Domain Knowledge 16  
2.3 Identifiers 17  
2.3.1 What is an Identifier? 17  
2.3.2 Importance of Identifiers 17  
2.4 Naming Conventions 18  
2.4.1 Potential Benefits for using Naming Conventions 18  
2.4.2 Approaches Used for Naming Conventions 19  

## 3 Domain Knowledge Recovery  
3.1 Overview 20  
3.2 Part 1: Extracting and Resolving Identifiers from Source Code 22  
3.2.1 Identifiers in the Present Work 22  
3.2.2 Extracting Identifiers 23  
3.2.3 Resolving Identifiers 24  
3.2.4 Issues with the Identifiers 25
3.3 Part 2: Generating Domain Concepts ............................ 28
   3.3.1 List of Nouns ........................................... 29
   3.3.2 Introduce Candidates with Multiple Words .......... 29
      3.3.2.1 First Method: a Greedy Algorithm .......... 30
      3.3.2.2 Second Method: Considering Abbreviations . 31
   3.3.3 Stemming ............................................... 32
   3.3.4 Refinement ............................................. 33
      3.3.4.1 Term Frequency Inverse Document Frequency
              (TFIDF) Technique ............................. 33
      3.3.4.2 Document Frequency (DF) Technique ........ 34
      3.3.4.3 Term Frequency (TF) Technique ............. 34
      3.3.4.4 Manual Step .................................... 36
3.4 Part 3: Relationship Creation ................................. 38
   3.4.1 Language-Defined Relations ............................ 38
   3.4.2 Ontology .............................................. 38
   3.4.3 Creating Relationships Based on Identifier Descriptions 39

4 Use Cases 42
   4.1 Concept Distribution ..................................... 42
   4.2 Relevance of a Concept to each Component ........... 45
   4.3 Shared Domain Concepts .................................. 47
   4.4 Related Concepts ......................................... 48
   4.5 Graphical User Interface .................................. 49

5 Experiments 52
   5.1 First Experiment .......................................... 54
      5.1.1 Experimental Setup ................................. 54
         5.1.1.1 Objective ..................................... 54
         5.1.1.2 Subjects ....................................... 54
         5.1.1.3 Tasks .......................................... 54
         5.1.1.4 Execution ..................................... 54
      5.1.2 Threats to validity .................................. 55
      5.1.3 Results ............................................. 55
      5.1.4 Interpretation ...................................... 56
   5.2 Second Experiment ......................................... 56
      5.2.1 Experiment Setup .................................... 56
         5.2.1.1 Objective ..................................... 57
         5.2.1.2 Materials ....................................... 57
         5.2.1.3 Subjects ....................................... 57
         5.2.1.4 Tasks .......................................... 57
         5.2.1.5 Execution ..................................... 58
      5.2.2 Threats to validity .................................. 58
      5.2.3 Results ............................................. 58
      5.2.4 Interpretation ...................................... 59
6 Related Work 61
7 Summary and Future Work 67
  7.1 Contribution .......................... 69
  7.2 Future Work ............................ 69

Bibliography 71
List of Figures

2.1 Example of concepts lattice [23] . . . . . . . . . . . . . . . . . . . . 11
2.2 Part of the family ontology [22] . . . . . . . . . . . . . . . . . . . . 14
2.3 Abstract graph in University domain . . . . . . . . . . . . . . . . . 16

3.1 Method used to extract domain knowledge . . . . . . . . . . . . . 21
3.2 Relationships between the two concepts “nox” and “lambda” 40

4.1 Distribution of domain concepts over the components . . . . . . 43
4.2 Tree map for usage intensity of the concept “pressure” . . . . . 44
4.3 List of all domain concepts . . . . . . . . . . . . . . . . . . . . . . . 45
4.4 List of the domain concepts relevant to “BstSet” component . 46
4.5 List of all components that contain the concept “pressure” . 47
4.6 Common concepts between “LamDd”, “PFlt” and “ThS” . . . . 48
4.7 Related concepts to the concept “torque” . . . . . . . . . . . . . 49
4.8 The start page of the Graphical User Interface (GUI) . . . . . . 50
4.9 Detailed information about the given concept . . . . . . . . . . . 51
4.10 Common concepts between the selected components . . . . . . 51
List of Tables

2.1 Relationships between persons and sweets [23] . . . . . . . . . . . 10
2.2 WordNet 3.0 ontology statistics [11] . . . . . . . . . . . . . . . . . 16
2.3 Sample of identifiers . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

3.1 Statistics about the extracted variables . . . . . . . . . . . . . . . . . 23
3.2 A sample of the identifiers . . . . . . . . . . . . . . . . . . . . . . . . 24
3.3 Set of nouns that are mapped to “AirDvp” . . . . . . . . . . . . . 30
3.4 Groups obtained from a sentence . . . . . . . . . . . . . . . . . . . . 31
3.5 Sample of concepts created by both methods . . . . . . . . . . . . . 31
3.6 Sample of candidates excluded by DF technique . . . . . . . . . . 35
3.7 Set of candidates for the component “AirDvp” . . . . . . . . . . . 35
3.8 Relevant candidates to AirDvp component . . . . . . . . . . . . . . 36
3.9 Sample of manually excluded candidates . . . . . . . . . . . . . . . 37
3.10 Reduction of concept-candidate list throughout steps . . . . . . 37
3.11 Sample of domain concepts . . . . . . . . . . . . . . . . . . . . . . . 38

5.1 Reduction of concept-candidate list throughout steps . . . . . . 52
5.2 Sample of Components along with their frequencies . . . . . . . 53
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>Generated domain concepts for the “AirDvp” component</td>
<td>53</td>
</tr>
<tr>
<td>5.4</td>
<td>The result of validating five components</td>
<td>56</td>
</tr>
<tr>
<td>5.5</td>
<td>The result of validating 10 components</td>
<td>59</td>
</tr>
</tbody>
</table>
Listings

2.1 Source code of university management system .............. 6

3.1 Resolving an identifier ..................................... 25

3.2 An identifier that was partially resolved incorrectly ....... 28
List of Acronyms

**DF** Document Frequency
**TF** Term Frequency
**TW** Term Weight
**TFIDF** Term Frequency Inverse Document Frequency
**LDA** Latent Dirichlet Allocation
**PLSI** Probabilistic Latent Semantic Indexing
**NLP** Natural Language Processing
**EGR** Exhaust Gas Recirculation
**NSC** Nox Storage Catalyst
**ECU** Electronic Control Unit
**LSU** Lambda Sensor Unit
**OBD** Onboard Diagnosis
**DSL** Domain Specific Language
**GPL** General Programming Language
**FCA** Formal Concepts Analysis
**POS** Part Of Speech
**TP** True Positive
**FP** False Positive
**FN** False Negative
**HTML** HyperText Markup Language
**GUI** Graphical User Interface
Chapter 1

Introduction

Empirical studies show that software maintainers spend most of their time understanding the software being involved [6]. As a result, software maintainers have to browse through a huge amount of source code to re-establish the linkage between their domain knowledge and its counterpart in source code before any modification of the source code can take place [17]. According to software maintainers, domain knowledge is the soul of software systems. The following situation illustrates this: Suppose that there is a program which your boss requests you to understand. One time he gives you no explanation, another time he gives you a 10 minute introduction on the program. Certainly in the second situation, you need much less time [22]. This raises the question about content of the introduction. According to previous research on program comprehension, this is mainly knowledge about the basic concepts represented in the program and their interrelations [27, 29], so in other words, it is a part of domain knowledge.

Domain knowledge must be learned from domain experts rather than software developers. A domain expert is a specialist who is familiar with main functions of a certain domain. He is responsible to answer the following questions: What objects make up a certain domain? How are these objects related? How to liaise marketing (business requirements) and developers? In contrast, a developer is supposed to understand and implement domain functionality provided by a domain expert. For instance, if the domain is a university, then an expert must be familiar with main objects of the university (e.g., library, professors, students, classes) and the relationships between these objects (e.g., students borrow books from library, a library does not sell books), whereas a programmer is supposed to determine what the necessary classes and their attributes are. He must decide whether to create two separated classes -one for professors and the other one for students- or whether one class with an attribute labeled job is sufficient. In general, a programmer cares more about implementation details and less about do-
main functionality. Going back to the university example, a programmer is responsible for writing a program that allows a student to borrow books from the library, but he is not interested in knowing why students are not allowed to buy books from the library.

The interaction between a developer and a domain expert can be summarized as follows: A domain expert delivers knowledge about a certain domain to a developer, and the developer encodes this knowledge in a program. During the process of encoding, a developer uses currently available programming languages such as Java, C, and C#. Each language has its own set of constructs that can be used to describe real world objects. That is, each real object can be abstracted to a concept. To implement this concept, we need a language construct. Thus, the language constructs can be used to (indirectly) refer to things from reality [26].

Unfortunately, most currently available programming languages are not domain specific, but too general. That is, a General Programming Language (GPL) does not distinguish between building an application for different domains (e.g. managing a university, a bank, or transportation means). As a consequence, domain knowledge is not only scattered all over the program, but also weakly defined in it. In contrast to GPL, a Domain Specific Language (DSL) is applicable for one or at most few domains, and it provides specialized features to a particular domain. For example, HyperText Markup Language (HTML) is tailored to writing webpages, and it provides domain-specific features, such as embedding links, images and formatting text.

The main reasons for not using DSL are: Their limited usability outside the specific domain they were created for, the difficulty to learn a DSL for each domain, because of the high cost of designing, implementing, and maintaining a DSL, and finally, there is no adequate tool support [13]. However, we can compensate for the absence of DSLs by creating and manipulating problem-level abstraction as first class entities [23]. That is, a programmer must start with basic principles, such as algorithms, and then implement procedures defined in the algorithm from scratch without any source code. However, this is rarely true today, because most of the time, developers build on existing code available in form of libraries or collection of classes, which model both domain and implementation concepts (e.g., software libraries, design patterns, and frameworks). By reusing existing code with predefined infrastructure that does not fit perfectly to a target project, the developers have to perform additional programming tasks, such as implementing further methods, with which they bring extra information -additional concepts -to the implemented domain( known as the “interleaving phenomenon”).
Domain experts can no longer capture domain knowledge presented in the source code. Now the knowledge is weakly defined, distributed all over the program, and combined with knowledge coming from outside. One way to overcome this problem is re-constructing domain knowledge from source code.

1.1 Goals and Tasks

In this thesis, we developed a method to recover domain knowledge from C code. We call this domain-knowledge recovery. The key to recover domain knowledge is the identifiers that are presented in the source code.

Our main goals are:

1. **Recover domain knowledge from C code.** Domain knowledge is very important. However, recovering it is a challenge, because it is weakly defined in the program and interleaved with knowledge coming from other domains.

2. **Exploit knowledge once it is gained.** For example, it can be useful for concept distribution/logical cohesion analysis, showing the relevance of a concept to each component (a component is a part of software), finding shared concepts between components, and finding related concepts.

To fulfill our goals, we define the following tasks:

1. **Extract and resolve identifiers:** Extracting identifiers is essential, because domain knowledge contained in a program is manifested in identifiers. Identifiers need to undergo resolving, because they follow naming conventions. That is, the terms that form an identifier are not explicitly, but implicitly separated. Furthermore, an identifier, most often, contains abbreviations. Thus, the resolving process consists of splitting an identifier into single terms and expanding the involved abbreviations.

2. **Identify domain concepts:** The domain concepts are the most important part of the domain knowledge. A domain concept is designed only for understanding by developers and experts; it is an abstraction of a real-world object in a certain domain.

3. **Create relationships between domain concepts:** The relationships are the complementary part of the domain knowledge. A relationship connects a couple of relevant concepts. That is, if two objects are related in reality, then their corresponding concepts must also be related.
4. **Identify several use cases:** To show what domain knowledge can be used for, we need to define use cases, and investigate each use case separately. For example, assessing the logical cohesion—how concepts are distributed—is useful to evaluate the software design. Moreover, studying the relevance of a concept to each component is useful to find out where each concept has been implemented, and where it has been used only. Furthermore, determining the common concepts between components is useful to recognize how the parts of the same software are related.

1.2 **Structure**

We structure the remainder of this thesis as follows:

**Chapter 2 Background:**

In Chapter 2, we introduce the background knowledge necessary to understand the rest of the thesis, such as identifiers, naming convention, and domain knowledge.

**Chapter 3 Domain-knowledge Recovery:**

In Chapter 3, we propose a semi-automatic approach to allow experts and developers to recover domain knowledge from the C code. The approach consists of three parts: Extract and resolve identifiers from source code, generate set of domain concepts, and create relationships between domain concepts.

**Chapter 4 Use Case:**

In Chapter 4, we discuss four use cases for the extracted concept and relationship information that are relevant in practice. In particular, we show their use for concept distribution/logical cohesion analysis, reflecting the relevance of a concept to each component, finding the shared concepts between components, and finding related concepts.

**Chapter 5 Experiments:**

In Chapter 5, we conduct two experiments, in which we evaluated the generated concepts. In the first experiment, a domain expert has been involved to evaluate the result, whereas in second experiment, the documents that are used for documentation have been involved to validate the result.

**Chapter 6 Related Works:**
In Chapter 6, we provide an overview of existing approaches that extract domain concepts and their interrelations from source code.

**Chapter 7 Summary and Conclusion:**

In chapter 7, we summarize the outcome of our thesis and our contribution, and we present future work.
Chapter 2

Background

In this chapter, we introduce background knowledge necessary to understand the rest of this thesis. For illustration purposes, we prepare a running example, and we introduce it in Section 2.1. Since the main goal of this thesis is recovering domain knowledge, we give an overview about domain knowledge and its main components in Section 2.2. Because identifiers are the key to recover domain knowledge from source code, we explain in Section 2.3, what the identifiers are. Finally, in Section 2.4 we present several conventions used for naming identifiers, and we explain in detail the benefits for using them.

2.1 University Management System Source Code Example

For better illustration, we use a running example throughout this chapter, which is a source code for managing a university system. We choose this example, because it is easy to understand. Although it does not have a complicated structure, it is still sufficient to explain the information presented in this chapter.

The source code consists of four classes: Person, Student, Library, and Book. Some classes are arranged in hierarchy (e.g., the Student class is derived from the Person class). For each class, we present only few methods and attributes. Furthermore, we skip the body of all methods, because they play no role in our situation. Some classes contain instances of other classes. For example, the Library class has two attributes students and books, which are instances of the Student class and Book class, respectively.
package university;

public class Person {
    private String id;
    private String name;
    ...
    public Person(String id, String fullName)
    {
        this.id = id;
        this.name = fullName;
    }
    public String getID() {...}
    public void setID(String id) {...}
    public String getName() {...}
    public void setName(String name) {...}
}

class Student extends Person {
    private String major;
    private Double gradePointAverage;
    ...
    public double getGradePointAverage() {...}
    public void setGradePointAverage(double gpa) {...}
    ...
}

class Library {
    private ArrayList<Book> books =
    private ArrayList<Student> student =
    ...
    public void studentRegistration(Student s) {...}
    public void bookReservation(String stID, Book b) {}
    public void blockStudentAccount(String stID){...}
    ...
}

class Book {
    ...
}

Listing 2.1: Source code of university management system
2.2 Domain knowledge

Domain knowledge is valid knowledge used to refer to an area of human endeavor, an autonomous computer activity, or other specialized discipline [14].

In software engineering, domain knowledge is knowledge about the environment in which the target system operates [14]. In general, people with expertise in a certain domain are called domain experts, and the knowledge they possess is known as domain knowledge. Experts and specialists are responsible for building domain knowledge rather than software developers.

Domain knowledge consists of two main components: domain concepts and their interrelations [1,23], which we explain next.

2.2.1 Definition and Description of Concepts

There are several definitions of the term “concepts” in the literature of reverse engineering. We introduce them, because our definition of a domain concept is based on them.

Concepts as words: Some researchers [2] have considered concepts as words contained in the labels of identifiers, in the comments, or in the names of files. For example, Anquetil and Lethbridge classify programs into conceptual modules based on the words contained in the names of source code files: The files that share a common word in their name belong to the same conceptual module [2].

If we apply this definition on the source code presented in Listing 2.1, we get the following list of concepts: University, Person, Student, Grade, Point, Id, Name, Average, Major, Library, Book, Registration, Block, Reservation, Account.

The definition of concepts based only on words has the following advantages and disadvantages:

Advantages:

1. It is a simple definition: easy to understand and easy to apply.

2. Implementation needs short time working: Writing a program based on this definition is straightforward. A developer can accomplish this task in short time. Furthermore, the size of the program is relatively small and has low complexity. As a consequence, we get a better performance.
Disadvantages:

1. Concepts that contain multiple words are ignored: According to this definition, a concept can only consist of one word, where as in reality, a concept may be described through multiple words. For example, the concept "grade point average" is not presented in the above obtained list of concepts, but it is replaced by three individual concepts which are "grade", "point", and "average". The meaning of "grade point average" is no longer reflected in any of the obtained concepts.

2. The polysemy problem is not considered: This problem is common in the English language. It means that, the same word may have different meaning in different contexts. For example, the word “crane” may stand for a bird or a type of construction equipment. Separating such words from their context makes the process of realizing their exact meaning very cumbersome, if not impossible.

The definition of concepts based on words is not sufficient for our work, because it restricts a concept to one word. In contrast to this, the next definition allows a concept to contain several words.

Concepts as clusters of words: There is a significant body of research in reverse engineering that considers concepts as clusters of words [16, 18]. These clusters are created based on a similarity metric (e.g., words that recurrently occur together in the labels of program elements belong to the same cluster).

Below, we show a sample of 9 concepts obtained from applying this definition on the source code presented in Listing 2.1. We have identified these concepts by analyzing words that occur at least once together in the labels of methods and classes. Usually, the similarity metric is much more complicated, but here we have chosen a simple metric for better illustration.

1. get, Grade, Point, Average
2. block, Student, Account
3. Set, Grade, Point, Average
4. set, Name
5. set, ID
6. book, Reservation
7. student, Registration
8. get, Name
9. get, ID

Considering concepts as clusters of words has the following advantages and disadvantages:

Advantages:

1. Concepts are not restricted to one word.
2. Words that form a concept are strongly related: The words are semantically related, because they belong to the same program element.

Disadvantages:

1. An optimal similarity metric does not always exist: As a result, we would miss some concepts. For example, based on the chosen metric, we have missed a couple of interesting concepts (e.g., Person, Library, Student and major).

2. There is no hierarchy between concepts: Concepts are isolated, and there is no structure among them. For example, even if we chose a complicated metric, and we succeeded in retrieving the two concepts Person and Student, we would not consider the inheritance relationship between them.

The definition of concepts as clusters of words is not sufficient for our work, because it assumes no hierarchy between concepts. In contrast to this, the next definition defines partially a structure between concepts.

**Concepts in Formal Concepts Analysis (FCA):** The term concept associates with the use of FCA [3, 9, 25]. The input for FCA is a set of objects and their attributes. These objects are clustered into concepts based on their common attributes [12]. A concept is a group of objects that share common attributes, and consists of two sets: the set of objects and the set of attributes. The concepts are arranged hierarchically in a concept lattice. Nodes in the lattice represent concepts, whereas edges represent the “is-a” relationship. A concept C1 which contains a set of attributes A1 is a sub concept of C2, which in turn contains set of attributes A2, if A1 is included in A2.

In Figure 2.1, we show an example of applying FCA on a set of persons (acting as objects) and a set of preferred sweets (acting as attributes). The relationships between persons and sweets are shown in the Table 2.1. For instance, the concept ({Adrian, Stefan} {ice cream, coke}) contains the objects: Adrian and Stefan and the attributes: ice cream and coke.

<table>
<thead>
<tr>
<th>Likes</th>
<th>Chocolate</th>
<th>Ice cream</th>
<th>Juice</th>
<th>Coke</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adrian</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td></td>
</tr>
<tr>
<td>John</td>
<td>True</td>
<td></td>
<td>True</td>
<td></td>
</tr>
<tr>
<td>Stefan</td>
<td>True</td>
<td>True</td>
<td></td>
<td>true</td>
</tr>
</tbody>
</table>

Table 2.1: Relationships between persons and sweets [23]
CHAPTER 2. THE QUEST FOR ABSTRACTION IN SOFTWARE ENGINEERING

The definition of concepts as clusters of words that recurrently occur together has the following drawbacks:

1. It offers a weak and ambiguous definition of concepts. The interpretation of concepts depends exclusively on the human who reads the cluster of words. Two different persons can understand different concepts—e.g., is the first cluster from above referring to a table or to more general graphical containers?

2. The concepts are isolated and there is no structure among them.

3. The relation between concepts and program parts is weakly defined since the program parts are regarded as flat text documents. By linking a cluster to a program part, it is not clear how the concept is implemented in that part.

Concepts in formal concept analysis.

The notion of "concept" also occurs in reverse engineering with the use of formal concept analysis (FCA) (Tilley et al., 2003; Arévalo et al., 2005; Eisenbarth et al., 2003). FCA takes as input a set of objects, each object having its own attributes. These objects are clustered according to their common attributes. A concept is a maximal collection of objects that share common attributes and is described through a pair of sets—the set of objects (its extent) and the set of attributes (its intent). The concepts are arranged hierarchically in a concept lattice. Each node of the lattice represents a concept and the concepts are arranged in an is-a hierarchy. The concept that contains a set of attributes A has as sub-concepts all those concepts that contain a set of attributes B that is included in A.

Example 2.12: Computing a concepts lattice using FCA

In Figure 2.10 we present an example of performing concepts analysis on a set of persons (acting as objects) and a set of preferred sweets (acting as properties). The relation between the objects and their attributes is presented in the table on the left-hand side and the concepts lattice on the right-hand side. For example, we have the concept described by the pair $\{\{\text{Adrian, Stefan}\}, \{\text{ice cream, cake}\}\}$. This concept contains the objects Adrian and Stefan that have the common attributes ice cream and cake.

Figure 2.1: Example of a concepts lattice

The definition of concepts based on FCA has the following advantages and disadvantages:

Advantages:

1. A concept may contain several terms.

2. Attributes are considered along with objects to create concepts: As a consequence, it is partially possible to define structure between concepts.

Disadvantages:

1. Weakly defined structure between concepts (defines only the "is-a" relationship). A more complicated structure is not defined (e.g., the part-of relationship).

2. Analyzing attributes associated with objects is not sufficient to assign the concepts a domain meaning. That is, a connection between the obtained concepts and the modeled domain is not clear, whereas in reality, the concepts are highly related to the domain.

3. This approach is computationally expensive. That is, the task of constructing the lattice is known to be computationally expensive, due to the inherent complexity of the structure [28].

Generating concepts using only FCA is not sufficient for our work, because it is computationally expensive, and it does not distinguish between human-oriented and programming-oriented concepts. In contrast to this, the next definition clearly separates between them.

"Programming-oriented" vs. "human-oriented" concepts: The work of Biggerstaff [4] classifies concepts into two categories: programming oriented (e.g., class, array, and loop) and human oriented (e.g.,...}
grade point average, student, and university). The programming concepts can be identified by using techniques, such as parsing. For example, to investigate the presence of the programming concept *Array* in a C program, we should design a pattern that is able to recognize all possible declarations of array in the C language. After that, we check whether there is a match between the designed pattern and the C program. If yes, then the concept *Array* has been implemented in the program. In contrast, identifying human-oriented concepts requires the use of informal information that is contained in the program. According to Biggerstaff, the identification process depends heavily on a-priori knowledge about the application domain, the domain entities and the typical relationships among them.

Thus, the domain concepts, which form the first part of domain knowledge, can be defined in several ways. Next, we discuss the second part: interrelations among concepts.

### 2.2.2 Interrelations among Concepts

A relation specifies in what sense a concept is related to other concepts [23]. Some of the relation types in domain knowledge are specific to a domain. That is, they are used to store facts or to answer particular questions (e.g., a student reserves a book, a student studies in a university). Other common types are the “is-a” and “part-of” relations. The “is-a” relation is used among concepts and their sub concepts (e.g., a student is a person), where as the “part-of” relation (e.g., a person is part of a university, a library is part of a university, a book is part of a university) shows how concepts can be combined together to form composed concepts (e.g., *Person*, *Library*, and *Book* combined together to form *university*).

Similar works have used either language-defined relations or ontologies to create relations among domain concepts. Below, we explain briefly how to do that.

#### 2.2.2.1 Language-Defined Relations

Ratiu [22] classifies the language-defined relations between program elements according to two criteria:

Relations generated by the module system:

1. **memberOfPackage**: This relation holds between a package and all its classes (e.g., between *Person* and *university*).

2. **memberOfClass**: This relation holds between a class and all its attributes (e.g., between *Student* and *major*)
Relations generated by the type system:

1. **subTypeOf**: This relation holds between a class and all its super classes (e.g., Student is subTypeOf Person).

2. **hasType**: This relation holds between a variable and its declared type. Each variable in a program is associated with a declaration. In the declaration, we specify explicitly the variable type (e.g., the variable students hasType Student).

3. **assignedTo**: This relation holds between a named member of an expression in the right side of an assignment and the assigned variable (e.g., the variable fullName is assignedTo variable name).

4. **boundWith**: This relation holds between a member of an expression in the place of an actual parameter and the formal parameter. For example, to create an instance of class Person, we need to call the constructor: Person ("123445","Mohammed Tarabain"); during the call, the string "Mohammed Tarabain" is boundWith fullName.

According to Ratiu and Deissenboeck, generated relations based on the first criterion are equivalent to “part-of” relations, whereas generated relations based on the second criterion are equivalent to “is-a” relations.

Creating relations based on the language-defined relations has the following advantages and disadvantages:

**Advantage:**

1. We can easily create the “is-a” and “part-of” relations. That is, we only need to traverse through the source code, and extract their equivalent language-defined relations.

**Disadvantages:**

1. It is only applicable on source-code files that are written in object-oriented programming languages, such as Java and C#. That is, some of the language-defined relations (e.g., memberOfClass, memberOfPackage, and subTypeOf) cannot be matched to C files. For example, in the C language, there are no classes. Thus, we cannot define the memberOfClass relation.

2. Using language-defined relations, we can define only two kinds of relations, which are “is-a” and “part-of” relations. Thus, we would miss the relations that are specific to the domain, such as a student reserves a book.
2.2.2.2 Ontologies

Another way to create relations among domain concepts is based on ontologies [1, 22]. There is a wide spectrum through which ontology can be seen from the point of view of the specification details [19]. That is, the ontology with the lowest level of details is merely a list of concepts. In the next level of details, the ontology contains relations among domain concepts. A more detailed ontology assumes hierarchy between concepts. The best known hierarchical relation in an ontology is the “is-a” relation. The most detailed ontology allows value restrictions for properties of a concept. In Figure 2.2, we present an ontology at this level of detail. It shows a few members of the family and their relations. Some concepts have a set of properties and constraints defined over them. For example, the concept “parent” must have at least one child. The “is-a” relation is represented through direction of arrows.

![Figure 2.2: Part of the family ontology][22]

Relations Based on Ontology

Creating relations based on ontology requires in advance generating domain concepts. That is, we assume that, the list of the domain concepts has been generated. To establish relations, first we have to create all possible pairs of domain concepts. Then we examine for each pair whether the concepts that made up the pair exist in the ontology or not. If yes and if the concepts are related in the ontology, then the relationship among them is exactly the same as in the ontology. Similar works that used this method to create relationships have either created in advance their own ontology, or used a general exiting ontology, such as WordNet 3.0 [1, 22].

---

[22] Figure 2.2: Part of the family ontology [22]

Legend

- **Class**
  - **Instance**
    - **Constraint**
      - **Type**
Creating relations based on ontology has the following advantages and disadvantages:

**Advantages:**

1. In contrast to language-defined relations method, using ontology, we can create relations which belong to different kinds. That is, domain-specific relations are no longer ignored.

2. This method is straightforward, if the ontology has been created in advance. That is, the process of extracting relations from the ontology is not complicated.

**Disadvantages:**

1. The used ontology may not contain all the domain concepts. As a consequence, we would fail in generating a few relationships, because either their target concept or their source concept is not presented in the ontology.

2. Using a general ontology, we cannot handle the polysemy problem (i.e., the same word may have a different meaning in different contexts). That is, if the same word occurs in both the generated list of concepts and the used ontology, this does not mean they have the same meaning. For example, the word “crane” may stand for a bird or a type of construction equipment.

3. Creating ontology requires too much time and effort.

**Example on General Ontology: WordNet 3.0**

We present the WordNet 3.0, because it has been used in similar works that used a general ontology to create relations between domain concepts. WordNet 3.0 is a large lexical database of English words [11]. Nouns, verbs, adjectives, and adverbs are grouped into sets of synsets, each expressing a distinct concept. Each concept contains a group of terms that have the same meaning (e.g., \{shut, close\} belong to the same concept). WordNet 3.0 contains over 155,000 words (out of which more than 75% are nouns), grouped in more than 117,000 sets of synonyms (see Table 2.2). Each of WordNet’s synsets is connected to other synsets by means of a small number of “conceptual relations”. The most frequently used relation among synsets is the super-subordinate relation (also known as the “is-a” relation). For example, general synsets like \{furniture\} are linked to specific ones like \{bed\} and \{bunkbed\}. 
As a conclusion, the relations show relevant domain concepts and in what sense they are related. That is, we use them to show how objects that are abstracted to domain concepts are connected in reality. Although creating relations is very difficult, it is an essential part to build domain knowledge.

### 2.2.3 University Domain Knowledge

In this section, we show an example on domain knowledge (Figure 2.3). In particular, we show a few domain concepts that form university-domain knowledge along with their interrelations. Nodes of the graph represent domain concepts, whereas edges represent relationships between concepts. For clarity, not all relationships are shown. As Figure 2.3 shows, we used the language-defined relations to create the “is-a” and “part-of” relations (e.g., a student is a person). Throughout this work, we will specify exactly, how to extract domain concepts from the C code and generate relationships that are specific to a domain.

![Figure 2.3: Abstract graph in University domain](image)


2.3 Identifiers

Identifiers are informal source of information [23]. According to Ratiu, knowledge contained in a program is manifested in identifiers, because they reflect the meaning of program elements, such as classes, methods, attributes, and interfaces [23]. If we remove identifiers from source code, we get a text that contains keywords related to the programming language (e.g., `main`, `class`, `public` and `private`). Programming keywords have no influence on domain knowledge [1]. Therefore, identifiers are the key to recover domain knowledge from source code.

2.3.1 What is an Identifier?

An identifier is a label of a program element [10]. Table 2.3 shows a sample of the identifiers that are contained in the source code presented in Listing 2.1. Identifiers constitute about 33% of all tokens in the source code of programs, such as Eclipse, Sun JDK or Tomcat [8].

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person</td>
<td>Student</td>
</tr>
<tr>
<td>getId</td>
<td>getName</td>
</tr>
<tr>
<td>StudentRegistration</td>
<td>bookReservation</td>
</tr>
<tr>
<td>blockStudentAccount</td>
<td>Students</td>
</tr>
<tr>
<td>Book</td>
<td>Id</td>
</tr>
<tr>
<td>Name</td>
<td>Books</td>
</tr>
<tr>
<td>gradePointAverage</td>
<td>getGradePointAverage</td>
</tr>
</tbody>
</table>

Table 2.3: Sample of identifiers

2.3.2 Importance of Identifiers

Although identifiers are only labels for program elements, they carry important information and have strong impact on program comprehension [22]. For example, we assume that both a human and a complier would consider the following code: `Area= width*length;` to be acceptable. If we modify the code and replace the variable `length` by another variable, say `temperature`, we assume everybody would consider the new piece of code to be at least awkward, if not wrong, even though it is still conforming to the programming-language semantics [22].

Identifier scrambling method -also known as identifier obfuscation- is additional evidence on the strong influence which identifiers have on program
comprehension [22]. It is the process of substituting identifiers by meaningless characters, and storing the meaningless characters along with initial identifiers in a separate file. This modification is sufficient to make the process of understanding a program very difficult, if not impossible. Usually, this method is used by companies to prevent competitors from understanding their data (in case competitors get access to source code).

2.4 Naming Conventions

As we presented in the previous section, an identifier name reflects the meaning of a program element. Therefore, a label must be chosen properly. To achieve this, we need a convention for naming identifiers. In programming languages, naming conventions are rules for naming identifiers and other entities in the program and documentation [24].

2.4.1 Potential Benefits for using Naming Conventions

Below, we list some of the potential benefits for using naming conventions:

1. **Better readability and understandability**: The use of naming conventions reduces the effort needed to read and understand a program, and it provides better understanding if the program is modified after a long time [15]. For example, although the code `a=b*c;` has a correct syntax, its purpose is not clear. In contrast to `weeklyPay = hoursWorked * payRate;` which carries the meaning of code.

2. **Avoid name collisions**: Adopting a convention prevents collision between names that might occur when the work products of different organizations are combined. In such situations, attaching the name of the organization or the component (a component is a directory in the system) to the identifiers could be a solution.

3. **Mistake avoidance**: We can easily avoid grammatical and semantic mistakes. For example, suppose that a developer wanted to add a new method called "updateMyAccount" to the Student class, and he accidentally wrote "updateMeAccount". Because of the use of naming convention, it is easy to determine the grammatical mistake. If naming convention was not used, then it is more difficult to find the mistake in the following representation "updatemeaccount".

4. **Platform independence**: Any reference to the operating system, programming language and hardware should be avoided within names.

5. **Better usability of tools**: Allow the use of automated refactoring or search and replace tools with minimum potential for error.
Thus, following naming conventions allows developers to understand source codes, and to easily modify existing ones. Moreover, naming conventions do not only save time and effort, but also prevent collisions between names. Furthermore, they reduce semantic and grammatical mistakes.

2.4.2 Approaches Used for Naming Conventions

Below, we present two well known approaches used for naming convention, which they play an important role in establishing a new convention.

**Letter-Case separated words:** also known as camelCase and other names. Word boundaries are indicated by capital letters, because the words that form an identifier are separated by uppercase. We used this approach in the source code presented in Listing 2.1. For example, we presented the variable "grade point average" as "gradePointAverage". This convention is commonly used in Java and C#.

**Delimiter-separated words:** This approach separates words of an identifier with a non-alphanumeric character. The most two common characters used for this purpose are the hyphen (-) and the underscore (_). For example, the three words "grade point average" would be represented as "grade_point_average". In C and Pascal languages, the hyphen character is reserved, and underscore is therefore used instead.

The two common conventions presented above do not always meet the requirements of companies. That is, some companies have established their own set of conventions, which are based on previous conventions, but with additional features (e.g., the use of abbreviations). For example, the naming convention used at Bosch is based on camelCase with a considerable modification, where the individual words that form an identifier are abbreviated, and both abbreviations and their expansions are stored in a dictionary. The variable name "grade point average" would be represented as DirNam_nuGrdPntAvg. The value of this variable is a real number. That is, the abbreviation "nu" -stands for number- has been attached to the identifier. The component name (DirNam) should also be attached to prevent collision between variables.

The work of this thesis has been carried out at Bosch. Thus, we focus on Bosch convention for naming identifiers. In the next chapter, we discuss in detail, the concrete common convention used at Bosch.
In the previous chapter, we introduced all the necessary background knowledge to understand this thesis. In this chapter, we propose a semi-automatic approach to allow experts and developers to recover domain knowledge from C code. In Section 3.1, we present an overview of our approach to allow the reader getting a high-level understanding of it. Since the identifiers are the key to recover domain knowledge, we extract and resolve them in Section 3.2. Because domain concepts are the most important part of domain knowledge, we generate them using the identifiers in Section 3.3. Finally, we create in Section 3.4 the relationships between the generated concepts. That is, creating relationships is the complementary part of domain knowledge recovery.

3.1 Overview

We start with an overview of our approach (Figure 3.1), which consists of three parts:

**Part 1:** Extract and resolve identifiers from source code (Section 3.2).

**Part 2:** Generate set of domain concepts (Section 3.3).

**Part 3:** Create relationships between domain concepts (Section 3.4).

In Part 1, we extract identifiers from the source-code files, because knowledge contained in a program is manifested in identifiers. Then, we resolve them according to a naming convention. The first part consists of the following steps:

**Step 1:** Extract all identifiers conforming to the used naming convention (Section 3.2.2).
**Step 2:** Expand abbreviations of each identifier (Section 3.2.3).

In Part 2, we generate the set of domain concepts, because they are the most important part of domain knowledge. This part consists of the following steps:

**Step 1:** Prepare initial set of candidates (list of nouns) (Section 3.3.1).

**Step 2:** Introduce candidates that consist of multiple terms (Section 3.3.2).

**Step 3:** Unify all candidates with identical stem (Section 3.3.3).

**Step 4:** Remove wrong candidates (Section 3.3.4).

In Part 3, we create relationships between domain concepts. It consists of the following step:

**Step:** Find relevant concepts, and extract their relationships from the intermediate results (Section 3.4.3).

In Figure 3.1, we present an overview of our method for extracting domain knowledge from the source code.

![Diagram of method used to extract domain knowledge](image-url)

**Figure 3.1:** Method used to extract domain knowledge
3.2 Part 1: Extracting and Resolving Identifiers from Source Code

Part 1 of our approach can be divided into two main steps which are: extracting and resolving the identifiers. Throughout this section, we explain in detail how we chose, extracted, and resolved the identifiers.

3.2.1 Identifiers in the Present Work

In contrast to objected-oriented code, our C code has limited structure. It is merely a set of functions and variables. There are no further advanced elements, such as classes and interfaces. Although the \texttt{struct} element is very common in the C language, we did not encounter its use in the source code.

In the present work, function labels should follow a unified structure \((\text{componentname}_\text{Proc}())\). \texttt{Proc} stands for process, and it is fixed for all functions. In contrast to this, the \texttt{componentname} value changes from one function to another. However, the \texttt{componentname} is included in the variable labels of the same function. That is, function labels do not carry additional information about domain knowledge. Therefore, they are not good candidates for being identifiers. As a result, the relevant information is presented only in variable names. Variables in the present work have several uses. For example, variables are used by functions to exchange information, and store intermediate values.

In the present work, an identifier denotes a variable label and follows a naming convention, which consists of the following rules:

1. An identifier must contain at least one underscore character.

2. An identifier must consist of at least three parts: component name, unit, and description.

3. The “component name” and “description” parts must start with uppercase and consist of at least one abbreviation. In contrast, the “unit” part must start with lowercase and consists of exactly one abbreviation (abbreviations expansions are stored apart in a dictionary).

4. The abbreviations of the same part must be separated by uppercase. That is, the camelCase approach is used at the level of abbreviations.

The use of the component name is to avoid collision between labels of variables and to ensure their uniqueness in the domain namespace. This permits the use of the same variable within different components in the system. The
“unit” part holds information about variable type. That is, whether it is physical (e.g., voltage, acceleration, and pressure) or logical (e.g., counter, number, and bit). The physical type is used to specify physical elements, whereas the logical type is used whenever physical units are not appropriate, or they cannot be specified. Finally, the description part holds a description about the variable.

Example of an identifier is: \texttt{EnvP.pHiRes}

Component name: \texttt{EnvP} = Environmental pressure

Unit: \texttt{p} = pressure

Description: \texttt{HiRes} = High resolution

### 3.2.2 Extracting Identifiers

The input for our approach is twofold: On the one hand, source-code files and on the other hand, dictionaries. The source-code files are the only resource for identifiers. In the present work, we used source-code files that are related to engine-control software. However, the user can use any software. These files are distributed over a set of components (a component is a directory in the system). The dictionaries are divided into two main groups: The first group contains abbreviations that are used to form the identifiers and their expansions. The second group contains a considerable number of identifiers along with their descriptions. The input files have been processed. That is, all variable labels that conform to the naming convention have been retrieved and stored in a new file. In particular, we extracted 16 210 different variables, of which 16 018 conform to the used naming convention. Thus, only a few variables -in comparison to total number- have been excluded (Table 3.1). In Table 3.2, we show a sample of the extracted identifiers that conform to the naming convention.

<table>
<thead>
<tr>
<th>Total number of variables</th>
<th>20 560</th>
</tr>
</thead>
<tbody>
<tr>
<td>Different variables</td>
<td>16 210</td>
</tr>
<tr>
<td>Variables without underscore</td>
<td>184</td>
</tr>
<tr>
<td>Variables without component name</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 3.1: Statistics about the extracted variables
Resolving Identifiers

After having extracted the identifiers, we started resolving them. We call this the intermediate results. The process of resolving consists of the following steps:

1. **Map identifiers to source-code files**: To achieve this, we need to store the absolute paths of the exact locations where an identifier occurs. This step is necessary for locating concepts.

2. **Resolve the “component name” part**: This part consists of at least one abbreviation. These abbreviations are separated by uppercase. Based on this fact, we have split up this part into individual abbreviations, and expanded each one of them.

3. **Resolve the “unit” part**: This part consists only of one abbreviation, so resolving it requires substituting the abbreviation by its expansion. That is, there is no need for further processing.

4. **Resolve the “descriptive part”**: This part is similar to the “component name” part. That is, we have treated it in the same way.

5. **Add identifiers descriptions**: In many cases, a variable is associated with a description. A description provides information about the variable in natural English language. Although the main source for the description is a dictionary, the source-code files themselves might contain further descriptions. That is, we have collected description from both the source-code files and the dictionary.
In Listing 3.1, we show the result of the process of resolving an identifier named "DewDet_stHtgLSULstTrp_mp". The absolute paths to all files that contain this identifier are listed in the <URLs> element. The content of the <DESC> element represents abbreviations that form the identifier along with their expansions, and the collected description.

<IDENTIFIER>

<SHORT-NAME>DewDet_stHtgLSULstTrp_mp</SHORT-NAME>

<URLs>
  C:\... DewDet_HtrRls\dewdet_htrrls_in1_lavast.xml
  C:\... DewDet_HtrRls\dewdet_htrrls_in2_lavast.xml
  C:\... EDewDet_HtrRls\dewdet_htrrls_pavast.xml
</URLs>

<DESC>
  DewDet: dew detect
  st: state
  LSU: lambda sensor unit
  Htg: heating
  Lst: last
  Trp: trip
  State of dew point detection release of last driving cycle
</DESC>

</IDENTIFIER>

Listing 3.1: Resolving an identifier

3.2.4 Issues with the Identifiers

In practice, it turned out that there are some problems associated with resolving identifiers. Below, we list the encountered problems, and how we have addressed them.

Incomplete dictionary: Some of the abbreviations along with their expansions were not in the dictionary. The main two reasons for that are: Either a programmer has forgotten to add some pairs of abbreviations and their expansions to the dictionary, or some pairs have been lost under other circumstances (e.g., during updating the dictionary, a programmer might have accidentally deleted several records from the dictionary). We have overcome this problem by manually adding the missing abbreviations to the
dictionary. That is, we have consulted domain experts to get the expansions of the missing abbreviations.

**Conflict between abbreviations:** Sometimes, the same abbreviation had several meanings (expansions). For example, “Pos” has been used to represent both positive and position, although it stands only for positive. Solving this problem manually means we have to traverse through the results of resolving all the extracted identifiers, and check each one separately. That is, we need to traverse over 16000 identifiers. Thus, fulfilling this task manually is not only time consuming (in the order of months), but also a tedious task (repeating the same action to each identifier). Automating this task is impossible, because we need an expert to say, where the abbreviation “Pos” stands for position and where it stands for positive. Therefore, this problem is unavoidable. As a consequence, we need to validate the set of domain concepts, whenever we are done with generating them.

**Spelling mistakes:** The expansion of an abbreviation or the description of an identifier might contain typing errors. For example, the term "temparature" has been used a few times in the descriptions instead of "temperature". These errors have severe effects on defining concepts. Thus, such terms must be excluded or replaced before generating the domain concepts. As a consequence of their continuous presence, wrong concepts would have been introduced into the domain knowledge. Fixing this problem manually requires too much time and effort, because each of the two dictionaries contains several thousand entries. To address this problem, we have used a stemmer (Section 3.3.3), and exploited the low frequency of such typos to exclude them (Section 3.3.4.3).

**Uppercase characters:** Abbreviations are sometimes written all uppercase or partly uppercase. For example, in Listing 3.1, the three letters L, S and U are not three different abbreviations, but one abbreviation which stands for “lambda sensor unit”. Considering each letter separately as an abbreviation, changes the whole meaning (e.g., L stands for litter, U stands for voltage). Not conforming to naming conventions makes camelCase splitting difficult. To overcome this problem, we have used a greedy algorithm. The main steps for this algorithm are presented below:

1. Split each part of the identifier into individual abbreviations.

2. Identify and expand the longest sequence of abbreviations that belong to the same part, and which can be found in the abbreviation dictionary.

3. Repeat Step 1 for the remaining abbreviations of the identifier, un-
til all abbreviations have either been assigned, or until only a single abbreviation remains.

For example, after we had applied the first step in this algorithm on the identifier represented in Listing 3.1, we got the following abbreviations for each part:

First part: Dew and Det
Second Part: st
Third part: Htg, L, S, U, Lst, and Trp

The longest possible sequence that could be formed from the abbreviations of the first part, and which could be found in the dictionary is DewDet. That is, we have considered DewDet as one abbreviation, rather than two. The second part always contains one abbreviation. Therefore, there is no need to fetch the longest sequence of abbreviations; we only need to resolve st. The longest possible sequence that could be formed from the abbreviations of the third part, and which could be found in the dictionary is LSU. That is, we have considered LSU as one abbreviation, rather than three. The rest of the abbreviations have exactly the same length and there is no dictionary entry for HtgLst, LstTrp, or HtgTrp. Therefore, we can pick any single abbreviation to continue.

**Inadequate descriptions:** Some descriptions are inadequate. Below, we list some of the cases.

1. Few descriptions are overly long and contain irrelevant information.
2. Few descriptions are written in a language other than English (e.g., German, French).
3. A considerable part of sentences that form descriptions does not follow the main structure for a sentence in English language (i.e., subject-verb agreement). For example, the following three sentences do not contain verbs: "rate of change of torque", "heat change of the tube", and "wall temperature of the cooler".
4. Some descriptions contain abbreviations.

We have fixed the first two cases manually. The third case is unavoidable. It has no influence on generating the domain concepts, but on creating the relationships among concepts. That is, creating relationships based on such sentences leads to a considerable number of meaningless relationships. For the last case, we have created an algorithm which automatically substitutes abbreviations by their expansions using a dictionary.
The dictionary used for expanding abbreviations consists of several thousand entries, of which about 1500 have been used in the variable names of our system. We have adjusted the dictionary manually by adding about 100 entries and improved the expansions of about 600 existing abbreviations. This adjustment resulted in a considerable improvement: The number of completely correct expanded identifiers nearly doubled from 44% to 87%, and the average number of incorrectly expanded individual parts dropped from 20% down to 3%. This evaluation was done by manual assessment on a random sample of about 1% of the identifiers. In Listing 3.2, we show an example of an identifier that was partially resolved incorrectly. That is, the abbreviation “Pos” has been resolved into positive, although in this case, it stands for position.

<IDENTIFIER>
  <SHORT-NAME>TrbCh_rAdapPos_mp</SHORT-NAME>
  <URLs>
    C:\...\trbchapos_vd_pavast.xml
  </URLs>
  <DESC>
    r: resistance
    Adap: adaptation
    Pos: positive
    Position of zero point.
    TrbCh: turbo charger
  </DESC>
</IDENTIFIER>

Listing 3.2: An identifier that was partially resolved incorrectly

3.3 Part 2: Generating Domain Concepts

Domain concepts are a main component of domain knowledge. In this work, domain concepts do not have a programming orientation, but rather a human one, because programming concepts do not contribute to the understanding of domain knowledge [1]. A domain concept may consist of a single term or a compound of terms. The main sources for identifying domain con-
cepts are identifier labels and descriptions.

In this section, we show how to use identifier labels and descriptions to generate domain concepts.

3.3.1 List of Nouns

Some researchers on domain knowledge have restricted domain concepts to be nouns, and have ignored terms that belong to other lexical categories (e.g., verbs and propositions) [1]. Thus, the majority of domain concepts are nouns, whereas terms that belong to other lexical categories have either a negligible (e.g., verbs and propositions) or a small chance (e.g., adjectives) to be domain concepts. Based on the foregoing, we decided to keep only the nouns and to discard all the other terms. To achieve this, a Part Of Speech (POS) tagger has been used. A POS tagger is software that reads a text and outputs the POS of each word in the text.

Using Stanford POS tagger\(^1\), we have collected all nouns from the intermediate results. Then, we have mapped each noun to the components that contain it by using the information stored in the \(<\text{URLs}>\) element. After that, we have gathered for each component, all the nouns that are mapped to it. In Table 3.3, we show a set of 50 nouns that are mapped to a component named “AirDvp”. Although the involved POS tagger has high precision, it has wrongly classified a few terms as nouns. These terms are either abbreviations (Trbnus, Ptrbnus, and ttrbnus) or misspelled terms (e.g., speeed and convertr). Throughout this section, we show how to get rid from these terms.

3.3.2 Introduce Candidates with Multiple Words

After we had applied the first step, we got a list of candidates (nouns) that are all individual terms, as shown in Table 3.3. In many cases, the meaning of several words that occur together cannot be reflected by combing the meaning of each word separately. Thus, it is necessary to introduce candidates that consist of compound words (from now on, we call such candidates “groups”). In contrast to individual candidates, the terms that form a group can belong to any lexical category, because a group is usually a part of a sentence. To achieve this goal, we have used two methods: a greedy algorithm and considering abbreviations.

\(^1\)http://nlp.stanford.edu/software/tagger.shtml (last accessed on Feb. 20, 2014 )
### Table 3.3: Set of nouns that are mapped to “AirDvp”

<table>
<thead>
<tr>
<th>Noun</th>
<th>Noun</th>
<th>Noun</th>
<th>Noun</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pressure</td>
<td>Coolant</td>
<td>Plausibility</td>
<td>Slope</td>
</tr>
<tr>
<td>Air</td>
<td>Time</td>
<td>resolution</td>
<td>number</td>
</tr>
<tr>
<td>Value</td>
<td>Status</td>
<td>release</td>
<td>synchronous</td>
</tr>
<tr>
<td>Turbine</td>
<td>State</td>
<td>pulse</td>
<td>schedule</td>
</tr>
<tr>
<td>Exhaust</td>
<td>Valve</td>
<td>manifold</td>
<td>Controller</td>
</tr>
<tr>
<td>Trbnus</td>
<td>Engine</td>
<td>module</td>
<td>check</td>
</tr>
<tr>
<td>Intake</td>
<td>Ptrbnus</td>
<td>Limitation</td>
<td>analog</td>
</tr>
<tr>
<td>Voltage</td>
<td>Error</td>
<td>segment</td>
<td>convertr</td>
</tr>
<tr>
<td>Environment</td>
<td>Filter</td>
<td>management</td>
<td>trbnus</td>
</tr>
<tr>
<td>Velocity</td>
<td>Charge</td>
<td>crankshaft</td>
<td>part</td>
</tr>
<tr>
<td>Bank</td>
<td>Point</td>
<td>condition</td>
<td>segments</td>
</tr>
<tr>
<td>Sensor</td>
<td>Position</td>
<td>network</td>
<td>synchronization</td>
</tr>
<tr>
<td>Speced</td>
<td>Cylinder</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 3.3.2.1 First Method: a Greedy Algorithm

The input for this algorithm is the intermediate results (Section 3.2), and it consists of the following steps.

1. Split the intermediate results into individual sentences. A sentence is either an expansion of an abbreviation (e.g., in Listing 3.1, we considered “lambda sensor unit” as a sentence), or a description of an identifier (e.g., in Listing 3.1, we consider “State of dew point detection release of last driving cycle” as a sentence).

2. Create several groups from each sentence.

3. Count the number of times each group appears in the intermediate results.

4. Consider groups that appear most often as domain concepts.

While creating groups from a sentence, two restrictions need to be fulfilled:

1. Words that form a group must be consecutive in the sentence.

2. The maximum size of a group is five words.

For example, in Table 3.4, we show all the possible groups obtained from applying this algorithm on "Raw value of injection mass". The
first column in Table 3.5 shows a sample of candidates obtained by applying this algorithm on the intermediate results.

<table>
<thead>
<tr>
<th>Two Word Groups</th>
<th>Three Word Groups</th>
<th>Four Word Groups</th>
<th>Five Word Groups</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raw value</td>
<td>Raw value of</td>
<td>Raw value of</td>
<td>Raw value of</td>
</tr>
<tr>
<td>Value of</td>
<td>Value of injection</td>
<td>Value of</td>
<td>Raw value of</td>
</tr>
<tr>
<td>Of injection</td>
<td>Of injection</td>
<td>Value of</td>
<td>injection mass</td>
</tr>
<tr>
<td>Injection mass</td>
<td></td>
<td>Value of injection</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.4: Groups obtained from the sentence “Raw value of injection mass”

3.3.2.2 Second Method: Considering Abbreviations

We have considered abbreviation expansions that have been used in the descriptions as groups. In Table 3.5, the second column shows a sample of abbreviations (e.g., EGR, LSU, ECU, NSC, and OBD) and their expansions that we considered as candidates.

<table>
<thead>
<tr>
<th>First method</th>
<th>Second method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Air System</td>
<td>EGR: Exhaust gas recirculation</td>
</tr>
<tr>
<td>Turbo charger</td>
<td>NSC: Nox storage catalyst</td>
</tr>
<tr>
<td>Engine speed</td>
<td>ECU: electronic control unit</td>
</tr>
<tr>
<td>Air temperature</td>
<td>LSU: Lambda sensor unit</td>
</tr>
<tr>
<td>Air control</td>
<td>OBD: Onboard diagnosis</td>
</tr>
</tbody>
</table>

Table 3.5: Sample of concepts created by both methods

As a result for applying this step, three groups have been added to the set of candidates presented in Table 3.3 which are: "analog digital converter", "Onboard diagnosis" and "Engine speed". Thus, the total number of candidates increased from 50 to 53. The majority of candidates presented in the new set are individual terms. That is, only few candidates consist of compound words. The next question was how to remove as many wrong candidates (i.e., candidates that do not carry domain knowledge, such as speed, converter, status, state, and
value) as possible from the set of candidates.

3.3.3 Stemming

Some of the obtained candidates from the previous step vary morphologically, but share the same stem. For example, we have noticed the presence of "cooler", "cooling" and "coolant" in the same component. Such candidates refer to the same concept and need to undergo unification. That is, each group of candidates that share the same stem must be replaced by one candidate. Otherwise, we would have redundant information. To achieve this, we have used Lancaster stemmer\(^2\) to identify groups of candidates that share the same stem.

After that, we have replaced each group by a unified candidate and we computed its frequency as the sum of all individual frequencies of candidates in the same group. The easiest way to choose the unified candidate is by considering the stem itself (in the previous example, the stem was "cool"). Unfortunately, we found that in the majority of cases, the stem has either an ambiguous meaning, or is not an English word. Thus, we had to consider another strategy for unifying candidates. That is, we have chosen the candidate with the highest frequency in the group to be the unified candidate.

The process of unifying candidates that share the same stem has the following advantages:

1. Redundancy avoidance. The candidates that share the same stem have almost the same meaning. That is, their continuous presence would lead to a kind of redundancy. Thus, we get poor domain concepts.

2. The unified candidate has a higher frequency. Thus, it has more chances to be considered as a domain concept.

3. It helps partially to overcome spelling mistakes. For example, if a developer accidently wrote "temperature" instead of "temperature", then the result of unification would be "temperature", because both terms share the same stem. That is, the term "temperature" should have a higher frequency, because a programmer is less likely to commit the same spelling error more than a few times, and certainly not hundreds of times.

In the set of candidates that are presented in Table 3.3, there are 3 pairs of candidates that share the same stem: \{segment, segments\}, \{status, state\},

\(^2\)http://www.comp.lancs.ac.uk/computing/research/stemming/ (last accessed on Feb. 20, 2014)
and \{synchronization, synchronize\}. Each pair underwent unification. The produced unified candidates are segment, status, synchronization. As a consequence, the number of candidates relevant to “AirDvp” decreased to 50.

3.3.4 Refinement

The obtained set of candidates contains a mixture of domain concepts, general terms (e.g., programming concepts) and meaningless terms (e.g., mis-spelled words and terms that do not exist in the English language). The next challenge was to separate domain concepts from the rest. For this purpose, we have prepared and applied several techniques; some of them have failed, while others succeeded. In the following, we explain the main features of each technique and its result.

3.3.4.1 TFIDF Technique

TFIDF is a numerical statistic which reflects how important a word is to a document in a collection or corpus [21]. TFIDFs value is proportional to the frequency of a word in a document, but is offset by the frequency of the word in a corpus. Below, we show the formula used to calculate TFIDFs value for a certain term \(t\) in a document \(d\) and corpus \(D\).

\[
Tfidf(t, d, D) = tf(t, d) \times \log \frac{|D|}{|\{d \in D : t \in d\}|}
\]

The value of \(tf(t, d)\) is estimated by dividing the raw frequency of \(t\) by the total number of terms in document \(d\). \(|D|\) is the number of documents in corpus \(D\), whereas \(|\{d \in D : t \in d\}|\) is the number of documents where the term \(t\) appears.

If a term is distributed over all documents, the ratio inside the logarithm approaches 1, bringing the idf and tfidf closer to 0. Thus, general terms tend to have low tfidf values.

We have regarded components as documents and candidates as terms. Our goal was to exclude the general candidates from the set of candidates. After we had applied this technique, we took a closer look at the excluded candidates. It turned out that we have removed a lot of domain-specific terms. Furthermore, a considerable amount of general terms still existed in the remaining set.

The TFIDF technique had a poor result. Thus, we decided to apply other techniques. We chose DF and TF techniques, because they have been used in similar works [1]. In the next section, we explain these two techniques in detail.
3.3.4.2 DF Technique

The DF technique is used widely in the information-retrieval domain [1]. The main idea behind it is counting, for each candidate, in how many documents it is contained. If the result is above a certain threshold, then this candidate is considered a general term. Thus, it holds no useful information about the domain knowledge, and should be excluded. The opposite does not always hold: A term may occur only in few documents, but it could be classified as a general term. Thus, this technique is used to partially remove general terms. That is, only those terms that occur in several documents would be excluded.

Setting the threshold is not problematic under the condition that frequencies of candidates are sorted in descending order. A user can simply go through the ordered set of candidates, and fix the threshold to be exactly the same as the frequency of the first encountered domain concept (first non-general term).

To apply this technique, we have considered a component as a document, and a term as a candidate. Then, we have collected all different candidates from all components, and counted their frequencies (a frequency of a candidate reflects how many components contain this candidate). After that, based on the frequencies, we sorted the candidates in descending order. Then, we traversed through the sorted list of candidates. The first 68 candidates were general concepts. That is, we have set the threshold to be exactly the frequency of the candidate that has rank 69, because it was the first encountered non-general term. Finally, we have eliminated each candidate that has a frequency higher than the threshold from all the components.

In Table 3.6, we show a sample of terms that have been excluded. Each term is associated with a frequency, which shows the number of components that contain it.

After we had applied the DF technique, the number of candidates relevant to the component “AirDvp” decreased to 32. In Table 3.7, we show the set of remaining candidates.

3.3.4.3 TF Technique

Similar to the DF technique, the TF technique is popular in the information-retrieval domain. The main idea for it is counting how often a candidate occurs in a document. If the result is below a certain threshold, then this term is either meaningless (e.g., a term that resulted from a wrong expansion of an abbreviation, misspelled term, and non-English term) or general (e.g., programming concepts). Thus, it should be excluded.
<table>
<thead>
<tr>
<th>Candidate</th>
<th>Frequency</th>
<th>Candidate</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>60</td>
<td>Maximum</td>
<td>42</td>
</tr>
<tr>
<td>Time</td>
<td>56</td>
<td>Start</td>
<td>42</td>
</tr>
<tr>
<td>Value</td>
<td>54</td>
<td>Signature</td>
<td>40</td>
</tr>
<tr>
<td>Position</td>
<td>48</td>
<td>Output</td>
<td>40</td>
</tr>
<tr>
<td>Control</td>
<td>47</td>
<td>Input</td>
<td>38</td>
</tr>
<tr>
<td>Pointer</td>
<td>47</td>
<td>Minimum</td>
<td>38</td>
</tr>
<tr>
<td>Number</td>
<td>44</td>
<td>Set</td>
<td>38</td>
</tr>
<tr>
<td>Error</td>
<td>43</td>
<td>Switch</td>
<td>38</td>
</tr>
<tr>
<td>Limitation</td>
<td>43</td>
<td>Differential</td>
<td>37</td>
</tr>
</tbody>
</table>

Table 3.6: Sample of candidates excluded by DF technique

<table>
<thead>
<tr>
<th>Candidate</th>
<th>Candidate</th>
<th>Candidate</th>
<th>Candidate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Air</td>
<td>analog digital converter</td>
<td>Coolant</td>
<td>engine speed</td>
</tr>
<tr>
<td>Turbine</td>
<td>onboard diagnosis</td>
<td>resolution</td>
<td>Analog</td>
</tr>
<tr>
<td>Exhaust</td>
<td>Bank</td>
<td>pulse</td>
<td>Convertr</td>
</tr>
<tr>
<td>Trbnus</td>
<td>Valves</td>
<td>manifold</td>
<td>Ttrbnus</td>
</tr>
<tr>
<td>Intake</td>
<td>Ptrbnus</td>
<td>crankshaft</td>
<td>Cylinder</td>
</tr>
<tr>
<td>Voltage</td>
<td>Charge</td>
<td>network</td>
<td>Speed</td>
</tr>
<tr>
<td>environment</td>
<td>Engine</td>
<td>slope</td>
<td>Release</td>
</tr>
</tbody>
</table>

Table 3.7: Set of candidates for the component “AirDvp” after applying DF technique

We have applied the TF technique at the level of each component. That is, we have traversed through the set of candidates for each component, and counted how many times each one of them occurs in the component itself. Then, we have identified the candidate with the highest frequency, and automatically set a threshold based on its frequency. That is, we have divided the highest frequency by a constant. To define a global constant (for all components), we have used a few components as a training set. Finally, we have removed from the list every candidate that has a frequency below the threshold.

Setting the threshold proportional to the highest frequency adds more reliability to our approach, because size varies from one component to an-
other. That is, there is a disparity between the highest frequencies of different components. Thus, we cannot set a fixed threshold for all components.

After we had applied the TF technique, the number of candidates for the component “AirDvp” decreased to 11. In Table 3.8, we show the set of remaining candidates. If we compare Table 3.7 and Table 3.8, we notice that, the misspelled words (e.g., speeed, convertr, and ttrbnus) and some of the general terms (e.g., release, network, and manifold) have been excluded. Furthermore, some of the excluded candidates are neither meaningless nor general (e.g., engine speed, crankshaft, and pulse). However, removing such candidates does not have severe effects, because they have a low frequency in comparison to the set of remaining candidates.

<table>
<thead>
<tr>
<th>Candidate</th>
<th>Candidate</th>
<th>Candidate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pressure</td>
<td>Bank</td>
<td>Velocity</td>
</tr>
<tr>
<td>Air</td>
<td>Intake</td>
<td>analog digital converter</td>
</tr>
<tr>
<td>Turbine</td>
<td>Voltage</td>
<td>Coolant</td>
</tr>
<tr>
<td>Exhaust</td>
<td>Environment</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.8: Relevant candidates to AirDvp component

Based on the way that we used to estimate the threshold (dividing the highest frequency by a constant), the threshold has never exceeded 10. As a consequence, some general candidates still existed. Thus, we have decided to manually traverse through the rest of candidates and remove non-domain-specific terms.

3.3.4.4 Manual Step

The manual step involves the user, who must go through the list of automatically identified candidates tagging each candidate as being a general term or not, then removing the general ones. Usually, such user intervention requires small effort. That is, classification of each candidate can be done very quickly (in the order of a few seconds). Thus, the overall process can be accomplished in short time. In contrast, the same process cannot be applied to the initial list of candidates directly, because of the high number of candidates (4740). In Table 3.9, we show a sample of candidates that have been removed manually. In Table 3.10, we show how the number of candidates decreased throughout the different steps.

After we had applied the TF technique, the number of candidates relevant to “AirDvp” remained the same. That is, none of the candidates
represented in Table 3.8 is a general term. Therefore, the candidates shown in Table 3.8 are the final set of domain concepts that belong to the “AirDvp” component.

The good thing about our approach is that we need to apply the manual step only on a relatively small number of candidates. For example, after we had applied the TF technique, we got only 462 candidates (gathered from all components), which is very small in comparison to the initial number of candidates (4740). That is, using our approach, we assume that a user would save both time and effort. In Table 3.11, we show a sample of the domain concepts gathered from several components.

<table>
<thead>
<tr>
<th>Candidate</th>
<th>Candidate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Byte</td>
</tr>
<tr>
<td>Default</td>
<td>Stop</td>
</tr>
<tr>
<td>Fraction</td>
<td>Level</td>
</tr>
<tr>
<td>Coefficient</td>
<td>Success</td>
</tr>
<tr>
<td>Transfer</td>
<td>Loop</td>
</tr>
<tr>
<td>Total</td>
<td>Reason</td>
</tr>
<tr>
<td>Interface</td>
<td>Unit</td>
</tr>
</tbody>
</table>

Table 3.9: Sample of manually excluded candidates

<table>
<thead>
<tr>
<th>Step</th>
<th>Number of candidates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial candidates</td>
<td>4740</td>
</tr>
<tr>
<td>Only nouns</td>
<td>2030</td>
</tr>
<tr>
<td>Stemming</td>
<td>1563</td>
</tr>
<tr>
<td>DF Technique</td>
<td>1494</td>
</tr>
<tr>
<td>TF Technique</td>
<td>462</td>
</tr>
<tr>
<td>Manual Step</td>
<td>279</td>
</tr>
</tbody>
</table>

Table 3.10: Reduction of concept-candidate list throughout steps
3.4 Part 3: Relationship Creation

To create the relationships between concepts, we have tried different approaches. We first start with approaches that have been used in similar work (language-defined relations and ontology). Since the methods were not applicable, we tried a new method as shown below.

3.4.1 Language-Defined Relations

Creating relationships based on the language-defined relations is not possible, because our C files have a limited structure. That is, some of the relations (e.g., memberOfClass, memberOfPackage, and subTypeOf) cannot be matched to the input files (Section 2.2.2.1). Thus, we had to try a different approach.

3.4.2 Ontology

To create relationships based on ontology, we either have to use a pre-prepared ontology that is specific to the source-code files, or we have to use a general existing ontology such as WordNet 3.0.

In our situation, the source-code files are related to engine-control software. Unfortunately, no ontology has been created in advance for this domain. Creating a new ontology requires a lot of time and effort. Thus, this option is not applicable.

<table>
<thead>
<tr>
<th>domain concept</th>
<th>domain concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pressure</td>
<td>Exhaust</td>
</tr>
<tr>
<td>Turbine</td>
<td>Coolant</td>
</tr>
<tr>
<td>Analog digital converter</td>
<td>Engine speed</td>
</tr>
<tr>
<td>Exhaust gas recirculation</td>
<td>Torque</td>
</tr>
<tr>
<td>Throttle valve</td>
<td>Air System</td>
</tr>
<tr>
<td>Charger</td>
<td>Voltage</td>
</tr>
<tr>
<td>Lambda sensor unit</td>
<td>Nox storage catalyst</td>
</tr>
<tr>
<td>Battery</td>
<td>Zero fuel calibration</td>
</tr>
<tr>
<td>Rail pressure Electrode</td>
<td>Acceleration</td>
</tr>
<tr>
<td>Electrode</td>
<td>Fuel</td>
</tr>
</tbody>
</table>

Table 3.11: Sample of domain concepts
The other option -using a general existing ontology- is also not applicable, because of the polysemy problem. Below, we show a few examples.

Example 1: One of the generated concepts is “Tooth”. This concept has been used to refer to a small part of the engine. That is, the concept “Tooth” is not related to one of the main parts of the human mouth. Therefore, if we create relations among concepts based on a general ontology, we would fail in creating a relationship between “Tooth” and “Engine”, although “Tooth” is a part of “Engine”.

Example 2: Another generated concept is “Bank”. Contrary to popular usage, the experts of engine domain do not use the term “Bank” to refer to transactions and money.

Thus, creating relationships based on ontology in not suitable, although using a domain-specific ontology is promising for future work.

3.4.3 Creating Relationships Based on Identifier Descriptions

We have tried a further method to create the relationships among concepts, which consists of the following steps.

1. Find related pairs of domain concepts: To achieve this goal, we have analyzed the result obtained from resolving the identifiers (intermediate results). That is, if two domain concepts occur in the result of resolving the same identifier, then they are most likely related. The more often they occur together, the more relevant their relationship is.

2. Collect a set of sentences: To this end, we have collected the description associated with each identifier from the intermediate results. For example, we have collected “state of dew point detection release of last driving cycle” from Listing 3.1, and we ignored the rest (abbreviations and their expansions), because an abbreviation expansion cannot be a complete sentence. In contrast to an abbreviation expansion, a description may be a complete sentence. We focused only on complete sentences, because they contain verbs. That is, verbs are helpful to generate relations [1].

3. Extract the relations from the collected set of sentences: For each two related concepts, we traversed through the set of collected sentences. If both concepts exist in a sentence, then the words that are separating the two concepts form the relation. As a consequence, we
got for each pair of related concepts several relations having different intensities. The intensity shows how often a relation occurs in different sentences.

We have created an application that shows the relationships between a given pair of concepts. The user can browse through the relationships starting from the most intensive relationships reaching the weaker ones. Unfortunately, a significant number of the recovered relationships was meaningless. In Figure 3.2, we show relationships along with their intensities between the two domain concepts “nox” and “lambda”.

![Figure 3.2: Relationships between the two concepts “nox” and “lambda”](image)

The main reason for getting a considerable number of meaningless relationships is the low quality of the description. That is, the majority of sentences that form descriptions do not follow the main structure for a sentence in the English language. In particular, they do not contain verbs. However, we believe that if the programmers have followed a standard during writing a description, then we would get much more meaningful relationships.

Although we got a considerable number of meaningless relationships, the recovered knowledge has several uses. In the next section, we illustrate four use cases, which show what the recovered knowledge can be used for.

As a conclusion, we implemented an approach to recover domain knowledge that was once built into the code. In contrast to similar works, which
dealt mainly with object-oriented code, our approach works best for C code. Our approach consists of three parts. In the first part, we extracted and resolved identifiers. In the second part, we generated domain concepts. Finally, in the third part, we created relationships between the generated concepts.
Chapter 4

Use Cases

In the previous chapter, we introduced our approach to re-construct the main components of domain knowledge from C code: Generating domain concepts and creating their relationships. In this chapter, we discuss several use cases for the extracted concept and relationship information that are relevant in practice. In particular, we show their use for concept distribution/logical-cohesion analysis (Section 4.1), reflecting the relevance of a concept to the components that contain it (Section 4.2), finding the shared concepts between several components (Section 4.3), and finding the related concepts to a given concept (Section 4.4). In Section 4.5, we implement a GUI, which supports experts and developers to visualize each use case.

4.1 Concept Distribution

Using our approach, we have collected for each component a set of the relevant domain concepts. Such information can be used to investigate how the domain concepts are distributed over components. That is, whether a domain concept occurs only in a few components, or whether it is distributed over many components. Analyzing the distribution of concepts over components (previous work refers to it as logical-cohesion analysis) is helpful for experts to evaluate the software design, because:

1. If a domain concept is concentrated in one component or in a few components, then it either means that the concept is very specific, thus it is not an important aspect, or it means that the concept has been encapsulated in a perfect way. Hence, it is an indicator for the good design of the software.

2. If a domain concept is distributed in many components, then it either means that, the concept is a very central for the software, or it means that the software has a bad design (e.g., missing separation of concerns and missing logical structure).
The two phrases “many components” and “few components” are abstract. That is, based on current input, our approach cannot specify the exact values for both phrases; only experts can do that. Furthermore, only experts can decide between alternatives (e.g., whether a concept is very central, or whether the software has a bad design). However, the contribution of our approach so far is preparing and visualizing the required information for the experts to decide whether the software has a bad or a good design. That is, the experts do not need to look all over the software to evaluate its design, but only at the indicated concepts. In Figure 4.1, we show the distribution of concepts over the components that form the engine-control software. In particular, we show how many concepts occurred in k components ($k \in [1, 33]$). For example, 45 concepts occurred in exactly one component, and only one concept occurred in 33 different components.

![Figure 4.1: Distribution of domain concepts over the components](http://www.cs.umd.edu/hcil/treemap/) (last accessed on Feb. 20, 2014)

We have also visualized the information that shows where each concept is located by using a tree map. That is, using the tree map software\(^1\), an expert can choose a concept from the list of all concepts (as shown in Figure 4.3) and request information about it. Then, he receives all the information about where the given concept is located. Thus, an expert can get a quick overview of whether the given concept is located in a few components, or many components. Furthermore, the expert can get the absolute frequency (number of occurrences) for the given concept in each component. In Fig-

---

\(^1\)http://www.cs.umd.edu/hcil/treemap/ (last accessed on Feb. 20, 2014)
Figure 4.2, we show an example. The given concept is “pressure”. The tree map shows that this concept is located in 31 components, such as “BstSet”, “BstCtl”, and “ChrCtl” and its absolute frequencies in these components are 1363, 518, and 340, respectively. The tree map software offers a further feature, which is encoding metrics by colors. That is, the tree map software assigns a specific color to a component based on how often the given concept occurred in the component (i.e., the metric). In Figure 4.2, the more the frequency decreases below one hundred -the expert can choose a different value-, the darker the green color assigned to the component. The more the frequency increases above one hundred, the lighter the green color assigned to the component. The tree map software assigns black color to the components that do not contain the concept “pressure” and yellow color the component that contains the concept “pressure” most often.

![Tree map for usage intensity of the concept “pressure” in the different components of the system](image)
4.2 Relevance of a Concept to each Component

For each component, our approach has generated a list of concepts, which are relevant to the component, along with their frequencies (how often a concept occurred in the component). This information can now be used to reflect the relevance of a concept to each component that contains it. For this purpose, we have created a metric. The higher the value of the metric is, the more relevant the concept is to the component.

The metric is based on two criteria. That is, to compute the relevance of a concept \( C \) to a component \( P \) in a set of components \( S \) that all contain \( C \), the following two criteria have been considered:

1. The frequency of \( C \) in \( P \) divided by the highest frequency of \( C \) in \( S \).
2. The rank of \( C \) (with respect to frequency) in the list of concepts that occurred in \( P \).

We combined both criteria by using the two weights: \( w_1 \in [0,1] \) and \( w_2 = 1 - w_1 \). To set these weights, one should use a few components as a training set. Below, we show the metric used to compute the relevance of \( C \).

\[
\text{Relevance}(C) = w_1 \times \frac{\text{Freq}(C)_{inP}}{\text{highestFreq}(C)} + w_2 \times \frac{1.0}{\text{Rank}(C)}
\]

Reflecting the relevance of a concept to each component is very useful. We can use it to find out, for a considerable number of concepts, where the concept has been implemented, and where it has been used only. That is, if a concept is extremely relevant (the value of the metric approaches 1) to exactly one component, then it is implemented in this component, or if the concept is only of little relevance (the value of the metric is low) to a component, then it is only used in this component. However, the gathered information is not sufficient to answer cases where the same concept
is (almost) equally relevant to several components. Thus, we need to pass further information to our approach (e.g., the largest possible number of components that can implement the same concept) in the future work.

In Figure 4.4, we show the list of concepts that are relevant to the component “BstSet”. In Figure 4.5, we show a list of components to which the concept “pressure” is relevant. For each component, we show how often it contains the concept “pressure” and how relevant the concept “pressure” is to it (metric value). Apparently, the concept pressure is most relevant to the “BstCt”, and second most relevant to “BstSet” component. For example, for the concept “pressure” and the component “BstSet”, we have the following values:

\[
\begin{align*}
C &= \text{pressure} \\
W_1 &= 0.4 \\
\text{Freq}(C) &= 518 \ (\text{see Figure 4.4}) \\
P &= \text{BstSet} \\
\text{HighestFreq} &= 1363 \ (\text{see Figure 4.5}) \\
W_2 &= 0.6 \\
\text{Rank}(C) &= 1 \ (\text{see Figure 4.4})
\end{align*}
\]

Thus, the relevance calculates to \(0.4 \times \frac{518}{1363} + 0.6 \times \frac{1.0}{1} = 0.75\), indicating that the concept “pressure” is very relevant to BstSet.

![Concepts that belong to BstSet](Figure 4.4: List of the domain concepts relevant to “BstSet” component)
4.3 Shared Domain Concepts

Since each component is associated with a list of domain concepts, we can easily find the common concepts between several components. Such information may help experts, along with their experience, to conceive how components are related.

In Figure 4.6, we show the list of common domain concepts (e.g., “pressure”, “engine speed”, “coordinator”, “temperature”, “engine temperature” and “torque”) between the three components: “LamAd”, “PFlt”, and “ThS”. In addition to their experience, the experts can now conceive exactly the relation between the three selected components.
4.4 Related Concepts

Although we could not create many meaningful relationships between pairs of related concepts, we succeeded in determining for each concept a set of the related concepts (Section 3.4.3). The members of the same set are not equally strong to the given concept. That is, some of them are more related than others.

Below, we give two examples for how developers can use this information.

1. If a developer wants to implement a new function that deals with a certain concept, he can use this information to determine all the related concepts that he may have to take into consideration.

2. If a developer wants to remove a function that implements a certain concept, and this concept is in existential relation with another concept (e.g., two concepts are in existential relation if they are 100% related), then the function that implements the other concept should be removed as well.

In Figure 4.7, we show all related concepts for the concept “torque”, along with their intensities. The intensity associated with each concept shows how related this concept is for the concept “torque”. That is, the most related concept for “torque” is “air” with intensity 25.46. This means that 25.46% of the identifiers that contain the concept “torque” contain also the concept “air”. Based on this information, a developer who wants to implement a new function that deals with the concept “torque” can now find out what the other concepts (e.g., “air”, “temperature”, and “engine”) that he may take into consideration are.
4.5 Graphical User Interface

To support developers and experts, we implemented a GUI to visualize the generated concepts and their interrelations. Moreover, we visualized for each concept how relevant it is to each component, and what its related concepts are. Furthermore, we visualized the common concepts between components, which can be selected by the user.

In Figure 4.8, we show the start page of the GUI. It contains all the generated concepts. A user (a developer or an expert) can choose and double-click on a concept. As a result, a new page similar to the one in Figure 4.9 will open, containing three parts.

1. **The first part is a table.** Each entry in the table shows a component to which the given concept is relevant. In addition to that, it shows the degree of relevance between the given concept and the component.

2. **The second part is a text area.** It displays the relationships between the given concept and a related concept. To choose the related concept, the user has to double-click on one of the entries in the table that is represented in the third part.

3. **The third part is a table.** Each entry in the table shows a concept to which the given concept is related and the intensity of the relationship between both concepts.

To find out the common concepts between a set of components, the user has to select these components, and then press the enter button. A new page will start containing the common concepts (see Figure 4.10).
<table>
<thead>
<tr>
<th>Concepts</th>
<th># of Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>temperature</td>
<td>33</td>
</tr>
<tr>
<td>pressure</td>
<td>31</td>
</tr>
<tr>
<td>engine speed</td>
<td>31</td>
</tr>
<tr>
<td>resistance</td>
<td>29</td>
</tr>
<tr>
<td>engine temperature</td>
<td>23</td>
</tr>
<tr>
<td>injection quantity</td>
<td>23</td>
</tr>
<tr>
<td>torque</td>
<td>22</td>
</tr>
<tr>
<td>voltage</td>
<td>21</td>
</tr>
<tr>
<td>monitoring</td>
<td>19</td>
</tr>
<tr>
<td>charger</td>
<td>19</td>
</tr>
<tr>
<td>engine operation mode</td>
<td>18</td>
</tr>
<tr>
<td>exhaust</td>
<td>17</td>
</tr>
<tr>
<td>coordinator</td>
<td>17</td>
</tr>
<tr>
<td>flow</td>
<td>17</td>
</tr>
<tr>
<td>air system</td>
<td>17</td>
</tr>
<tr>
<td>gas</td>
<td>17</td>
</tr>
<tr>
<td>bank</td>
<td>16</td>
</tr>
<tr>
<td>cylinder</td>
<td>16</td>
</tr>
<tr>
<td>air temperature</td>
<td>16</td>
</tr>
<tr>
<td>environment</td>
<td>15</td>
</tr>
<tr>
<td>field</td>
<td>15</td>
</tr>
<tr>
<td>ramps</td>
<td>15</td>
</tr>
<tr>
<td>mass</td>
<td>14</td>
</tr>
<tr>
<td>air mass</td>
<td>14</td>
</tr>
<tr>
<td>engine</td>
<td>14</td>
</tr>
<tr>
<td>exhaust gas recirculation</td>
<td>13</td>
</tr>
<tr>
<td>fuel</td>
<td>13</td>
</tr>
<tr>
<td>volume</td>
<td>13</td>
</tr>
<tr>
<td>vehicle</td>
<td>13</td>
</tr>
<tr>
<td>turbine</td>
<td>12</td>
</tr>
<tr>
<td>governor</td>
<td>12</td>
</tr>
<tr>
<td>power</td>
<td>12</td>
</tr>
<tr>
<td>lambda</td>
<td>12</td>
</tr>
<tr>
<td>diagnosis</td>
<td>12</td>
</tr>
<tr>
<td>heating</td>
<td>12</td>
</tr>
<tr>
<td>base</td>
<td>12</td>
</tr>
<tr>
<td>velocity</td>
<td>11</td>
</tr>
<tr>
<td>intervention</td>
<td>11</td>
</tr>
<tr>
<td>liter</td>
<td>10</td>
</tr>
<tr>
<td>lambda sensor unit</td>
<td>10</td>
</tr>
<tr>
<td>injection</td>
<td>10</td>
</tr>
<tr>
<td>oxygen</td>
<td>10</td>
</tr>
<tr>
<td>speed</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 4.8: The start page of the GUI
Figure 4.9: Detailed information about the given concept

Figure 4.10: Common concepts between the selected components
Chapter 5

Experiments

In the previous chapters, we introduced our approach to recover domain knowledge from C code. We used the files that are related to engine-control software as input to our approach. In particular, we applied our approach to 1493 source-code files. These files are distributed over 61 components. A component is a directory in the system. We applied the first part of our approach. That is, we extracted 16018 different identifiers and we resolved them. After that, we applied the second part. That is, generating domain concepts. In Table 5.1, we show the variation of the number of candidates throughout each step in the second part. The initial number of different candidates is 4740. After we had applied the TF technique, the number decreased to 462 different candidates. After we had applied the manual step, which is the last step in Part 2, we got 279 different domain concepts that are distributed over 61 components. As a result, each component has its set of generated concepts. In Table 5.2, we show several components. Each component is associated with a frequency that represents how many concepts are relevant to it. In Table 5.3, we show the generated concepts for the “AirDvp” component.

<table>
<thead>
<tr>
<th>Step</th>
<th>Number of candidates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial candidates</td>
<td>4740</td>
</tr>
<tr>
<td>Only nouns</td>
<td>2030</td>
</tr>
<tr>
<td>Stemming</td>
<td>1563</td>
</tr>
<tr>
<td>DF Technique</td>
<td>1494</td>
</tr>
<tr>
<td>TF Technique</td>
<td>462</td>
</tr>
<tr>
<td>Manual Step</td>
<td>279</td>
</tr>
</tbody>
</table>

Table 5.1: Reduction of concept-candidate list throughout steps
<table>
<thead>
<tr>
<th>Component</th>
<th>Frequency</th>
<th>Component</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPM</td>
<td>27</td>
<td>TS</td>
<td>17</td>
</tr>
<tr>
<td>Mo</td>
<td>54</td>
<td>ESC</td>
<td>2</td>
</tr>
<tr>
<td>VehC</td>
<td>14</td>
<td>CSW</td>
<td>2</td>
</tr>
<tr>
<td>PtDev</td>
<td>13</td>
<td>VehMot</td>
<td>22</td>
</tr>
<tr>
<td>NSC</td>
<td>82</td>
<td>PwrTrn</td>
<td>29</td>
</tr>
<tr>
<td>LamAd</td>
<td>42</td>
<td>VehDev</td>
<td>14</td>
</tr>
<tr>
<td>ThDev</td>
<td>8</td>
<td>ELDev</td>
<td>4</td>
</tr>
<tr>
<td>ESS</td>
<td>8</td>
<td>AirDvp</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 5.2: Sample of Components along with their frequencies

<table>
<thead>
<tr>
<th>Domain concept</th>
<th>Domain concept</th>
<th>Domain concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pressure</td>
<td>Bank</td>
<td>Velocity</td>
</tr>
<tr>
<td>Air</td>
<td>Intake</td>
<td>analog digital converter</td>
</tr>
<tr>
<td>Turbine</td>
<td>Voltage</td>
<td>Coolant</td>
</tr>
<tr>
<td>Exhaust</td>
<td>Environment</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3: Generated domain concepts for the “AirDvp” component

While we were resolving identifiers, we failed to address the conflict problem between abbreviations (Section 3.2.4). Thus, it is necessary to validate the obtained result of our approach.

Usually, validating the whole result is not possible except for small-size software, because of the lack of resources and tools. In our case, the input is a big-size software. Therefore, we cannot validate the whole result. The alternative option is to partially validate the result. To achieve this, we used the fact that domain concepts are distributed over components. That is, we validated the set of generated concepts for several components, instead of all components. For this purpose, we conducted two experiments, in which we evaluated the generated concepts of 5 and 10 components, respectively. Therefore, in total, we have validated about 25% of the whole result. In the first experiment, a domain expert has been involved to evaluate the result. In the second experiment, the documentation has been used to validate the result. In the following, we explain the two experiments in detail.
5.1 First Experiment

5.1.1 Experimental Setup

Throughout this section, we show the goals of the experiment (Section 5.1.1.1), the involved persons in the experiment (Section 5.1.1.2), the performed task by each subject (Section 5.1.1.3), and finally the execution of each task (Section 5.1.1.4).

5.1.1.1 Objective

For each component, we have generated a set of domain concepts. Thus, the generated concepts are supposed to be relevant to the component. From practical point of view, a concept is relevant to a component if it has been either implemented or used in that component. In this experiment, we want to investigate whether the generated concepts for five components are relevant to them or not, and what are the missing concepts.

5.1.1.2 Subjects

In this experiment, only one subject, a domain expert, has been involved. Usually, a domain expert is professional with few components, but not with all. The involved expert is familiar with only five components: “NSC”, “PtDev”, “VehC”, “EPM”, and “Mo”. We tried to contact other experts so that we cover more components, but no one was available.

5.1.1.3 Tasks

The expert received a list of generated concepts for five components. These concepts have been generated using our approach. For each component, the domain expert is asked to judge whether a given concept is relevant or irrelevant to it. In addition to that, he is also asked to state the missing-relevant concepts to it. The judgment has been rendered based on the experts knowledge about the components. That is, no further tools have been used in this experiment.

5.1.1.4 Execution

We have used the same plan for all components that underwent validation. That is, for each of the five components:

1. The expert traversed through the concepts that have been generated using our approach and tagged each concept as being relevant (True Positive (TP)) or irrelevant (False Positive (FP)).

2. After that, the expert stated the missing domain concepts (False Negative (FN))
5.1.2 Threats to validity

The task of stating the missing domain concepts is very difficult for experts. As a consequence, we cannot guarantee that the domain expert has stated all the missing concepts. Thus, the result of this experiment may be biased. That is, the number of irrelevant concepts, for each component, may be less than the actual number.

We cannot avoid this threat, but our results are still valuable, because this threat has a small effect on the results. That is, the expert might forget to state a few concepts only, which would not have severe effect on the results.

5.1.3 Results

In Table 5.4, we show the result of validating 5 components. For each component, we can see the number of TP concepts, FP concepts, and FN concepts. Furthermore, we compute the precision \( P = \frac{TP}{TP+FP} \), recall \( R = \frac{TP}{TP+FN} \) and F-measure \( F = \frac{2PR}{P+R} \) metrics to measure the effectiveness of our approach.

The precision indicates how many of the generated concepts are relevant, while the recall shows the percentage of the generated domain concepts out of all concepts that could ideally be generated (based on the input). The harmonic mean of the precision and recall, F-measure, is used to aggregate the values of precision and recall to a single value. That is, using the aggregate value, we can compare the performance of our approach on different components.

To evaluate the goals of our experiment, we answer two questions:

1. **Question 1:** How many domain concepts are relevant to each of the involved components?

   **Answer:** In Table 5.4, we show for each component the percentage of the relevant concepts. For example, 90.74% of the generated concepts for the component “Mo” are relevant, whereas 9.24% of the generated concepts are irrelevant.

2. **Question 2:** How many domain concepts are missing in each of the involved components?

   **Answer:** In Table 5.4, we show for each component the percentage of successfully generated domain concepts. For example, the percent-
age of successfully generated domain concepts for “Mo” component is 94.23%. Thus, the percentage of missing concepts is 5.77%.

<table>
<thead>
<tr>
<th>Component name</th>
<th>TP</th>
<th>FP</th>
<th>FN</th>
<th>Precision</th>
<th>Recall</th>
<th>F-measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPM</td>
<td>24</td>
<td>3</td>
<td>3</td>
<td>88.9%</td>
<td>88.9%</td>
<td>88.9%</td>
</tr>
<tr>
<td>Mo</td>
<td>49</td>
<td>5</td>
<td>3</td>
<td>90.74%</td>
<td>94.23%</td>
<td>92.45%</td>
</tr>
<tr>
<td>VehC</td>
<td>13</td>
<td>1</td>
<td>1</td>
<td>92.86%</td>
<td>92.86%</td>
<td>92.86%</td>
</tr>
<tr>
<td>PtDev</td>
<td>12</td>
<td>1</td>
<td>2</td>
<td>92.31%</td>
<td>85.71%</td>
<td>88.89%</td>
</tr>
<tr>
<td>NSC</td>
<td>77</td>
<td>5</td>
<td>1</td>
<td>93.9%</td>
<td>98.72%</td>
<td>96.25%</td>
</tr>
</tbody>
</table>

Table 5.4: The result of validating five components

5.1.4 Interpretation

Table 5.4 shows that the lowest valued obtained from computing the F-measure metric is 88%. The component that has the lowest value is PtDev. Thus, for each component, our approach has missed only few relevant concepts, and rarely generated irrelevant concepts. Therefore, our approach performed very well regarding these components.

After we had looked closely at the irrelevant concepts (FP) for each of the five components, we noticed that the concept “resistance” has been classified as irrelevant in four components: “NSC”, “PtDev”, “EPM” and “Mo”. The main reason for that is the conflict problem between abbreviations (Section 3.2.4). That is, the abbreviation “r” has been always resolved to “resistance”, although sometimes it should have been resolved to other terms, such as “ratio”.

In the first experiment, we have evaluated the generated concepts for only 5 components. To increase validity, we decided to conduct another experiment to evaluate more components. In the next section, we discuss the second experiment.

5.2 Second Experiment

5.2.1 Experiment Setup

In this section, we show the goals of the experiment (Section 5.2.1.1), the used materials (Section 5.2.1.2), the involved persons in the experiment
(Section 5.2.1.3), the performed task by each subject (Section 5.2.1.4), and finally the execution of each task (Section 5.2.1.5).

5.2.1.1 Objective

Similar to previous experiment, the goal of this experiment is to investigate whether the generated concepts for ten components are relevant to them or not, and what are the missing concepts.

5.2.1.2 Materials

The used materials in this experiment are two documents that contain descriptions of ten components. Some of these components are associated with rich descriptions, whereas others are not. That is, some of the descriptions did not cover all the functions of a certain component.

5.2.1.3 Subjects

In this experiment, three subjects have been involved. The first subject has access to the documents that are supplied with engine-control software. The second subject has a general knowledge about engine-control software. In contrast to the involved expert in the previous experiment, this subject does not need to know all details about components, such as the missing concepts, because his task does not need a deep understanding about engine-control software. The third subject does need to have knowledge at all about engine-control software, because he needs only to do a simple comparison, which does not require any background knowledge about engine-control software.

5.2.1.4 Tasks

The first subject has collected two documents that are used for documentation. That is, they contain written comments, graphical illustrations, flowcharts, etc. They are supplied with engine-control software.

The second subject has manually extracted for each component a set of domain concepts using the collected set of documents.

For each component, the third subject has compared the manually generated concepts and the concepts that are generated using our approach. Based on this comparison, he can judge whether a given concept is relevant or irrelevant to the component. In addition to that, he can state the missing-relevant concepts to the component.
5.2.1.5 Execution

After the second subject had extracted the description for each component from the collected documents, he manually extracted, for each component, a list of relevant concepts. Then, the third subject compared, for each component, the list of concepts that are generated using our approach (A) and the list of manually extracted concepts (M).

1. If a concept appears in both lists A and M, then we classify it as TP (thus, it is a relevant concept).
2. If a concept appears in M and not in A, then we classify it as FN (thus, it is a missing-relevant concept).
3. If a concept appears in A and not in M, then we classify it as FP (thus, it is irrelevant concept).

5.2.2 Threats to validity

We cannot guarantee that the manual description is complete. That is, most often, the manual description does not cover all the functionalities of a module (e.g., class, method, and component). Thus, the result of this experiment may be biased. That is, the number of manually generated concepts, for each component, may be less than the actual number of the relevant concepts. As a consequence, the number of irrelevant concepts (false positive) may be relatively high (higher than the actual number of irrelevant concepts).

We cannot avoid this threat, but our results are still valuable, because on one hand, this threat has no influence on the number of missing concepts (false negative) which form important part of the goal of this experiment, and on the other hand, we expect that this threat has effect only on few components.

5.2.3 Results

In Table 5.5, we show the result of validating 10 components. For each component, we can see the number of the TP concepts, FP concepts, and FN concepts. Furthermore, we compute the precision, recall, and F-measure values for each component.
### Component TP FP FN Precision Recall F-measure

<table>
<thead>
<tr>
<th>Component name</th>
<th>TP</th>
<th>FP</th>
<th>FN</th>
<th>Precision</th>
<th>Recall</th>
<th>F-measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>LamAd</td>
<td>33</td>
<td>9</td>
<td>3</td>
<td>78.57%</td>
<td>91.67%</td>
<td>84.62%</td>
</tr>
<tr>
<td>ThDev</td>
<td>8</td>
<td>0</td>
<td>3</td>
<td>100%</td>
<td>72.73%</td>
<td>84.21%</td>
</tr>
<tr>
<td>ESS</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>87.5%</td>
<td>77.78%</td>
<td>82.35%</td>
</tr>
<tr>
<td>TS</td>
<td>15</td>
<td>2</td>
<td>5</td>
<td>88.24%</td>
<td>75%</td>
<td>81.08%</td>
</tr>
<tr>
<td>ESC</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>100%</td>
<td>66.67%</td>
<td>80%</td>
</tr>
<tr>
<td>CSW</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>100%</td>
<td>66.67%</td>
<td>80%</td>
</tr>
<tr>
<td>VehMot</td>
<td>15</td>
<td>7</td>
<td>5</td>
<td>68.18%</td>
<td>75%</td>
<td>71.43%</td>
</tr>
<tr>
<td>PwrTrn</td>
<td>18</td>
<td>11</td>
<td>4</td>
<td>62.07%</td>
<td>81.82%</td>
<td>70.59%</td>
</tr>
<tr>
<td>ELDev</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>50%</td>
<td>50%</td>
<td>50%</td>
</tr>
<tr>
<td>VehDev</td>
<td>8</td>
<td>6</td>
<td>2</td>
<td>57.14%</td>
<td>80%</td>
<td>66.66%</td>
</tr>
</tbody>
</table>

Table 5.5: The result of validating 10 components

5.2.4 Interpretation

Table 5.5 shows that the lowest value obtained from computing the F-measure metric is 50%. The component that has the lowest value is “ElDev”. However, the overall average is 75%. Thus, for the majority of component, our approach has missed a few relevant concepts, and generated a few irrelevant concepts. Therefore, regarding these components, the performance of our approach is good.

As apparent from the result, the number of irrelevant concepts (FP) for some components, such as “LamAd”, “VehMot”, “PwrTrn” and “VehDev” is higher than the number of irrelevant concepts for the rest of components. The main reason is the description associated with each of these components. That is, the description did not cover all the functions implemented in the component. Thus, the number of manually generated concepts was smaller than the actual number of the relevant concepts.

We assume that if the component descriptions cover all the functions that are implemented in these components, then the F-measure values obtained in this experiment would be similar to the F-measure values obtained in previous experiment. Thus, the performance of our approach, regarding these components, would be much better.

As a conclusion, the two experiments showed that for the majority of components, our approach has missed only a few relevant concepts, and
rarely generated irrelevant concepts. Thus, the performance of our approach is good.
Chapter 6

Related Work

Having presented our work in the previous chapters, in this chapter, we provide an overview of existing approaches that extract domain concepts and their interrelations from source code, and we also discuss how they are related and different from our approach.

Abebe and Tonella [1] have developed an approach to extract domain concepts and their interrelations from source code, and represent them in domain ontology. They have first created an ontology using the identifiers. However, the entities that form the obtained ontology include domain concepts as well as implementation concepts. Thus, the next challenge was how to filter out the implementation concepts from the ontology. They addressed this problem by applying Natural Language Processing (NLP) techniques.

Using NLP techniques, the authors have generated a list of keywords (the list is not supposed to contain implementation concepts). After that, they removed all the implementation concepts from the ontology using the generated list. That is, a concept is considered as a domain concept and thus kept in the ontology, if all the terms in the concept name present in the list. Otherwise, it is considered as an implementation concept and filtered out. Regarding relations, a relation is kept in the ontology if both its source and target concepts have been considered as domain concepts.

The authors have tried several NLP techniques: TF, Term Weight (TW), TFIDF, Latent Dirichlet Allocation (LDA), and Probabilistic Latent Semantic Indexing (PLSI). For each technique, they have generated three lists with different lengths: 15, 50 and 100 words, and tried each list separately. The input for the involved techniques is information from either the source code (e.g., LDA and PLSI) or the documentation (e.g., TF, TW, and TFIDF). To ensure that a list of keywords does not contain implementation concepts, the authors have manually revised it.
In the following, we list the advantages and disadvantages of using this approach:

**Advantages:**

1. This approach is partially-automated, but the human intervention requires a small effort. That is, the user needs to traverse through a list that can contain at most 100 concepts tagging each concept as being an implementation concept or not.

2. The highest value obtained for the F-measure metric is 73% (when the TF Technique has been used and the length of list was 100). Thus, the performance of this approach is good.

**Disadvantages:**

1. The list of the generated keywords is, most often, incomplete. That is, not all the domain concepts may be present in the list, especially if the list has been created from the documentation, because most often the documentation does not cover all the functions that are implemented in a class, a method, etc. As a consequence, a considerable number of domain concepts will be filtered out from the ontology.

2. This approach is appropriate for small and medium-size software. That is, we cannot apply this approach to huge software, such as engine-control software, because the maximum length of the list is 100 different-individual concepts, whereas we have obtained about 250 different-individual concepts for engine-control software.

3. It is not guaranteed that the ontology can contain all the domain concepts, because it is automatically created. That is, if a domain concept is not presented in the ontology, then it is impossible to generate it.

4. Three different values for the length of the list have been used. The results show that the higher the length is, the more recall but less precision values we get. Thus, setting a threshold is very problematic.

5. Based on this approach, a concept is either a noun or a noun phrase. Thus, the concepts that consist of multiple words, which have different lexical categories, cannot be generated (e.g., analog digital converter, electronic diesel control, and on board diagnosis).

Abebe and Tonellas approach is related to ours, in that we have used NLP techniques (DF and TF) to filter out the non domain concepts. However, we have not used these techniques to filter out implementation concepts from a pre-prepared ontology, but from a list of candidates produced after
running a POS tagger and a stemmer on the resolved identifiers. A further difference is that our approach can be used to all size software, so our approach needs less input and can be applied to big-size software.

Ratiu and Deissenboeck [22] have developed a semi-automated approach to extract the domain concepts and their interrelations from the source code using ontologies that target the same domain as the source code. To achieve this, the authors have established a mapping between source code and the involved ontologies. However, source code (programs) and ontologies have different syntactic representation. Thus, the authors decided to represent both of them as graphs.

To model a program as a graph, the authors used the identifiers and the relations between their corresponding program elements. That is, the nodes of the graph represent the names of the identifiers and the edges of the graph represent relations implied from Java programming language, such as `memberOfPackage`, `memberOfClass`, and `subTypeOf` (Section 2.2.2.1). We will refer to this graph by the letter P.

To model ontology as a graph, the authors used the entities that form the ontology. That is, they are themselves the nodes of the graph. If there is a relation between two entities in the ontology, then in the graph, the authors have created an edge between the nodes that denote the two entities and associated it with the same name of the relation in the ontology. We will refer to this graph by the letter O.

After the authors had created the convenient representations, they mapped the graph P to graph O. For this purpose, two criteria have been used. That is, to map a pair of nodes in graph P to a pair of nodes in graph O, the following two conditions need to be fulfilled:

1. The names of the nodes should be compatible. The compatibility relation between two names is checked with respect to a specified criterion.

2. The names of the edge (relation) that connects the nodes which form the pair should be compatible. The compatibility among relation names is checked with respect to a specified criterion.

By making explicit the mapping between the code and the ontology, a new representation of the program has been produced, which is centered on the identified ontological concepts. That is, the new representation contains the pairs, from the involved ontologies, that have been mapped to pairs in the graph P. However, not all the obtained concepts (nodes) are relevant. That
is, some of them are irrelevant. Thus, a human intervention is required to exclude such concepts.

In the following, we list the advantages and disadvantages of using this approach:

**Advantages:**

1. Due to the polysemy, a certain word can represent several concepts. This approach solves the polysemy problem by taking into consideration not only concepts in isolation (compatibility between concepts), but also how these concepts are interrelated (compatibility between relation names).

2. By establishing a mapping from the program to ontology, the authors identify the concepts that have been expressed in the program. That is, the extracted concepts are made explicit and shareable. Thus, it helps to reduce comprehension effort.

**Disadvantages:**

1. This approach heavily depends on the existence of ontologies that target the same domain as the source code. Unfortunately, in many cases, such ontologies do not exist.

2. Although imposing compatibility between relations solves the polysemy problem, it has a drawback. That is, there is little chance to find two pairs that have compatible names and compatible relations. Thus, this approach will miss a considerable number of domain concepts.

3. It is not guaranteed that the involved ontology contains all the domain concepts. That is, if a domain concept is not presented in the ontology, it is impossible to generate it.

4. This approach is specific to programs that are written in advanced languages, such as Java and C#. Thus, if a program is written in a low-level language, such as C, then the task of modeling the program as a graph is very difficult, because many relations that are implied from Java programming language cannot be implied from C language.

5. The authors have intentionally not considered the descriptions that are supplied with identifiers. As a consequence, this approach will miss some domain concepts.

This approach is related to ours, in that we have followed some-common steps while generating the domain concepts (e.g., stemming and running a POS tagger). However, our approach does not depend on the existence
of ontologies that target the same domain as the source code. A further difference is the target code. That is, our approach is designed not for object-oriented programs, but C programs.

The work of Falleri, Huchard, Lafourcade, Nebut, and Prince [10] proposes a fully automated approach that extracts the main concepts of a program from the identifier names, and organizes them into a lexical view making explicit their interrelations. For this purpose, the techniques from the NLP field have been used. That is, to create the lexical view, the approach performs the following steps:

1. **Tokenization:** Split each identifier into a list of individual words.

2. **POS tagging:** Assign to each term in the previously computed lists a lexical category (e.g., noun, verb or adjective).

3. **Dependency sorting:** The terms that form the same list are sorted by the dominance order. The dependency sorting is driven by an ordered set of rules, specifically designed for a list of terms that are associated with their lexical categories.

4. **Generate implicit concepts:** Implicit concepts are generated using the sorted lists of tagged terms.

5. **Lexical view computation:** The initial identifiers together with the generated-implicit concepts are organized in a lexical view.

In the following, we list the advantages and disadvantages of using this approach:

**Advantages:**

1. It is fully-automated. That is, no human intervention is required.

2. Both explicit and implicit concepts are generated using the identifiers. As a consequence, the number of missing domain concepts is much less than other approaches.

3. The words that form a concept are strongly related. That is, they are semantically related, because they belong to the same program element.

**Disadvantages:**

1. The authors have limited the relations between the nodes of lexical view to hyperonymy (more general than) and hyponymy (more specific than). That is, other types of relations are not represented in the lexical view, such as relations that are specific to the domain (Section 2.2.2).
2. The authors have considered the identifiers as nodes. That is, each identifier has been represented by a node (concept) in the lexical view. If the identifiers are overly long (just as in the case of engine-control software), then we may get ambiguous and poor concepts. Furthermore, a concept may represent several real domain concepts, instead of exactly one concept.

3. The authors have not considered the descriptions that are supplied with the identifiers. As a consequence, this approach will fail in retrieving few domain concepts.

This approach is related to ours, in that it uses the identifiers to generate domain concepts and create their interrelations. However, our approach does not treat the whole identifier as one concept. In addition to that, our approach does not sort the terms that are produced from tokenizing the same identifier.

The work of Carvalho [5] creates a bidirectional mapping between domain concepts and source code to enhance program comprehension. For this purpose, the author creates three ontologies: The first ontology, named program ontology, contains information that is extracted from source code by using software analysis, NLP, and information retrieval techniques. The second ontology contains, if possible, information about program execution. The last ontology, named problem ontology, is created by using a set of current available problem-domain analysis techniques to gather information about the application domain from documentation and comments found in the source code. After that, the author created semantic bridges between concepts in different ontologies by using a concept mapper. Once all maps between different ontologies have been created, a reasoning layer over ontologies has been used to provide useful information to the developers and experts.

The author proposed several general steps to create the bidirectional map. However, the required tools are still under implementation. For instance, it is not clear how the concept mapper will create bridges between different ontologies, and whether it is able to solve the polysemy problem or not. This approach is related to ours, in that we have used NLP techniques to generate domain concept. However, we have not considered information about program execution while generating domain concepts.
Chapter 7

Summary and Future Work

In this section, we summarize the outcome of our thesis. First, we recapitulate the motivation and the goal of our thesis. Then, we summarize our contribution. Finally, we conclude and present future work.

Empirical studies show that software maintainers spend most of their time understanding the software being involved. The software maintainers have to browse through a huge amount of source code to re-establish the link between their domain knowledge and its counterpart in source code, before any modification of the source code take place. Thus, domain knowledge is the soul of software systems.

The domain experts deliver knowledge about a certain domain to the developers, and the developers encode this knowledge in a program. During the process of encoding, a developer uses currently available programming languages. Unfortunately, most of these languages are not domain specific, but too general. As a consequence, the knowledge is weakly defined, distributed all over the program, and combined with knowledge coming from different domains. Thus, the domain experts can no longer capture the knowledge contained in source code. Therefore, it would be very helpful to recover the domain knowledge that was once built into the system from the code. In this thesis, we showed how identifiers can be used to recover domain knowledge from C code and what the recovered knowledge can be used for. Thereby, we concentrated our research on how to accomplish the following tasks:

**Task 1:** Extract and resolve identifiers.

In the first part of our approach (Section 3.2), we showed how to extract identifiers and resolve them. In addition to that, we presented the problems associated with resolving the identifiers - despite the existence of naming
conventions and a domain dictionary - and we introduced applicable solutions.

**Task 2:** Generate domain concepts.

In the second part of our approach (Section 3.3), we showed how to generate domain concepts. To this end, we have applied natural-language processing techniques, such as document frequency and term frequency. The process of generating domain concepts is semi-automated, but the human intervention requires only small effort. That is, the user needs to traverse through a short list of candidates, in comparison to the initial list, tagging each candidate as being a domain concept or not. In contrast to other approaches, we have not restricted domain concepts to individual words. That is, some of the generated concepts consist of compound words.

**Task 3:** Create relationships between domain concepts.

In the third part of our approach (Section 3.4), we proposed a method to create relationships among the generated concepts. We also created an application that shows the relationships between a given pair of concepts. The user can browse through the relationships starting from the most intensive relationships reaching the weaker ones. Unfortunately, a significant number of the recovered relationships was meaningless. However, using this method, we have succeeded in determining for each concept a list of related concepts.

**Task 4:** Identify several use cases.

To show what domain knowledge can be used for, we first applied our approach to engine-control software, and then, we identified four use cases. In particular, we showed their uses for:

1. **Concept distribution analysis:** That is, whether the concept is distributed everywhere in the software, or whether is concentrated in few components. Gathering such information is useful to evaluate the design of the involved software.

2. **Concept relevance:** It is important to know how much relevant a concept is to each component that contains it. Gathering such information is useful to know where a given concept has been implemented, and where it has been used only.

3. **Common concepts between components:** Some of the components share the same concepts. Finding out these concepts helps experts, along with their experience, to conceive how components are related.
4. **Related concepts:** We determined for each concept what its most related concepts are. Gathering such information helps developers to re-structure the source code.

7.1 Contribution

The contribution presented in this thesis is a semi-automated approach that reconstructs domain knowledge from C code. The core of our approach is the identifiers. Most of the previous works have mainly dealt with object-oriented software. However, a considerable amount of currently existing software is written in low-level languages, such as C. For this purpose, our work reports from the first step towards recovering domain knowledge from C code.

In the following, we explicitly list our goals, and how we completed them.

**Goal 1:** Recover domain knowledge from C code. To this end, we first extracted and resolved identifiers. After that, we used a part of speech tagger, a stemmer, and natural-language processing techniques to generate domain concepts. Finally, we created relationships between the generated concepts based on the identifier descriptions.

**Goal 2:** Exploit knowledge once it is gained. To show what domain knowledge can be used for, we identified four use cases. For each use case, we explained how experts and developers can use the gathered information.

While our approach reported from the first steps towards recovering domain knowledge from C code, it could not create many meaningful relations between relevant concepts, and it is not fully automatic. Thus, there are several points for future work.

7.2 Future Work

The contribution of our thesis is an important step towards recovering domain knowledge from C code. With our approach, we showed potentials for domain-knowledge recovery. We use our approach for future work and improve it.

Our attempts to create concrete relationships between related concepts, based on identifier description, have only partially succeeded. Thus, a big future task is to address this issue. To achieve this, the following two ideas may be useful:
**First idea:** Transform the expanded abbreviations to complete sentences. That is, we can expand abbreviations to single terms. Then, using a set of rules, we can construct complete and meaningful sentences from the splitted terms. The complete sentences contain verbs, which are useful to create relationships.

**Second idea:** Build a domain-specific ontology. Using a domain-specific ontology, we can create relationships between the generated concepts. That is, if two concepts are existed and related in the ontology, then the relationship between them is exactly the same as in the ontology.

Another future research task is to further reduce the human interaction. For this purpose, more sophisticated techniques are required for filtering. Applying sophisticated techniques could help in filtering out more irrelevant concepts. Thus, it reduces the effort required for human intervention.

Moreover, we can consider not only nouns, but also adjectives, so that we can reduce the number of missing concepts.

Furthermore, the input of our approach should include additional information, so that we can specify exactly where each concept has been implemented, and where it has been used only. So far, our relevance metric shows the relevance of a given concept to the components that contain it. This metric can be refined or complemented with other metrics to avoid the cases where the same concept was (almost) equally relevant to several components.

Addressing these future research tasks will reduce the human intervention or make it obsolete, so that our approach can be fully automated. Furthermore, we can reduce the number of irrelevant and missed concepts.
Bibliography


Declaration of Originality

Hiermit erkläre ich, dass ich die vorliegende Arbeit verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 27. Januar 2014