Problem Domain Oriented Approach for Program Comprehension

Maria João Varanda Pereira[1], **Mario Berón**[2], **Daniela da Cruz**[3], **Nuno Oliveira**[3], **and Pedro Rangel Henriques**[3]

1 **Polytechnic Institute of Bragança**
Bragança, Portugal
`mjp@ipb.pt`

2 **National University of San Luis**
San Luis, Argentina
`mberon@unsl.edu.ar`

3 **Universidade do Minho**
Braga, Portugal
`{danieladacruz,nunooliveira,prh}@di.uminho.pt`

Abstract

This paper is concerned with an ontology driven approach for Program Comprehension that starts picking up concepts from the problem domain ontology, analyzing source code and, after locating problem concepts in the code, goes up and links them to the programming language ontology.

Different location techniques are used to search for concepts embedded in comments, in the code (identifier names and execution traces), and in string-literals associated with I/O statements. The expected result is a mapping between problem domain concepts and code slices. This mapping can be visualized using graph-based approaches like, for instance, navigation facilities through a System Dependency Graph.

The paper also describes a PCTool suite, Quixote, that implements the approach proposed.

1998 ACM Subject Classification I.2.2 Automatic Programming

Keywords and phrases Program Comprehension, Ontology-based SW development, Problem and Program domain mapping, Code Analysis. Software Visualization

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.91

1 Introduction

Software maintenance is known to be the most time-consuming and expensive phase on the software life cycle [4, 15]. It is incepted by the emergence of new requirements and entails a first phase to comprehend the program and a second to evolve it according to the requirements. Software evolution (once it is comprehended) is a fast task because there is not much expertise involved besides the programming basics. Program comprehension, however, requires more advanced skills where source code analysis techniques play an important role.

Program comprehension theories sprang from cognitive and psychological sciences [36, 34]. Such theories state that cognition is achieved by the construction of a mental model as a structured way of gathering knowledge about the program under analysis. Mental models are constructed either (*i*) top-down [5], i.e., from the knowledge on the problem domain to the knowledge embodied in the program domain, (*ii*) bottom-up [47], i.e., from the knowledge on the program domain to an abstraction capable of being mapped into the problem domain or (*iii*) hybrid [28, 31, 51], i.e., combining the other two, interchangeably. Nevertheless, the majority of these theories agree that a complete understanding of a program is reached when the analyst can relate the program domain—how statements are executed (operational semantics)—with the problem domain—what are the effects caused by the execution of those statements (logical semantics).

Program and problem domains are two sets of knowledge. The former aggregates concepts related with the program, the action of programming and the programming language. The latter gathers discourse-level concepts which are close to the humans' perception because of being fruit of empirical knowledge or expertise in the area where these concepts appear. The knowledge in both domains may be represented in ontologies providing a systematic way of mapping concepts of both domains, and therefore bring them closer.

At the beginning of a program comprehension activity, the problem domain is usually known, i.e., it is possible to identify the main concept in which the program is centered. Related concepts may come from the analyst's empirical knowledge or from the description of a requirement or a task for software maintenance. These descriptions are (usually) short natural language statements performed as a discourse at the problem level. This way, the identification of problem domain concepts is easy and enriches the analyst's knowledge on the problem domain. In a task-oriented approach for software maintenance, such descriptions and the terms involved may be used to focus the analyst's work. The number of concepts involved per task is within what the human brain can handle. Therefore, it is possible to search for them in the source code in desirable time. However these approaches are not always easy to follow and may require more complex solutions. Moreover, tool support for program comprehension is a requirement for automatize, systematize and make effortless the cognitive process.

In this paper we propose an approach and a PCTool suite implementing it, that would allow for a full comprehension of the program slice to maintain and its dependencies. It takes advantage of both ontologies to knowledge representation and task-oriented approach for concept location. We rely on the assumption that the problem domain is known and the comprehension is achieved when the concepts of both program and problem domains are mapped.

The main novelty of our approach is, therefore, the use of ontologies to formally describe the problem and program domains and drive the comprehension by creating a systematic way of mapping the concepts on both domains.

In abstract, we start by collecting the concepts from task or requirement descriptions (henceforth referred to as maintenance statements) using the problem domain ontology. Meanwhile the source code is analyzed to extract information considered important to understand the meaning of each part of the program. Information retrieval techniques (applied upon such information) are used to collect program blocks and program identifiers and associate them with the concepts involved in the maintenance statement. Later, the program ontology concepts are mapped into the previous associations allowing for interpreting the program elements involved in the maintenance task. Finally, the approach is endowed with traditional software visualization techniques for exploration of the provided results. The system is developed as it will be described along the paper.

Section 2 surveys related work, to support the appropriateness of our proposal and some of the choices and decisions taken along the subjacent project (that will be discussed along the paper). A detailed description of the approach is presented in section 3 in a very concise style. In section 4, three developed tools are described. The first two tools perform the static and dynamic source code analysis needed to implement some parts of the proposed approach. Moreover a third tool for comment analysis is also described in this section. The second part of our program comprehension system is concerned with software visualization and it is described in section 5, right before the conclusion of the paper in section 6.

2 Related Work

The use of ontologies is spread into literature and it appears strongly related with the web semantic area. In our case, we study the use of ontologies in program comprehension.

An ontology is informally described by several authors but the most known is Gruber [21] with the following definition: *An ontology is an explicit formal specification of a shared conceptualization of a domain of interest.* In order to use ontologies in Computer Science it is required to be machine readable, accepted by a community and restricted to a given domain. So, the formal definition of an ontology is based on a set of classes, a set of instances (objects), a set of relations, a set of instances of relations and attributes (object properties) [13]. Several representation languages can be used to describe an ontology: RDF, RDF-schema, OWL.

Ontology learning (automatic, or semi-automatic support in ontology development) and ontology population (the process of defining and instantiating a knowledge base) are the main activities concerned about the use of ontologies [7, 6]. Following Jinsoo Park [25], data resources may be textual data, dictionaries, knowledge bases, semi-structured schemata or relational schemata. As information extraction techniques, the same authors refer natural language processing, statistics and information retrieval. There are also some authors that infer ontologies based on the comparison of other ontologies from the same domain [9]. So, there are lots of research work on learning ontologies from texts, where text mining (e.g. text categorization, text clustering, concept/entity extraction, production of granular taxonomies, sentiment analysis, document summarization, entity relation modeling) is the basic technique [25]. All these activities consist on collecting, selecting, grouping, classifying the words extracted from the data resources, mapping them to concepts and analyzing a set of possible relations. There are several tools for ontology extraction [25] like: OntoLT, Text2Onto, OntoBuilder, Doddle-owl, asium, soat, vetlan, Mok Workbench. Most of these tools require the hands of a domain expert, operating, then, as semi-automatic.

In our case and at a first phase we consider that the problem domain ontology is already constructed and we want to locate these concepts on source code trying to map them to the program domain ontology. So, we need to explore deeply concept location techniques. The most important work in this area is presented by Vaclav [46, 32]: *The important task is then to understand where and how the relevant concepts are implemented in the code.* The techniques can be based on a top-down process—it consists on analyzing the domain to discover its concepts and then trying to match parts of the code with these concepts—or in a bottom-up process—it consists on analyzing the code and trying to cluster the parts that are most closely related according to a certain criteria. However, the most common is a combination of both [37]. The techniques presented by Vaclac are based on string pattern matching (grep), searching through the static code following call graphs and software reconnaissance using code instrumentation in order to discover the parts of the program that are related with each concept. Other technique is concerned with use of an information retrieval system: for instance, the LSI (Latent Semantic Indexing) [33, 41, 12] method that is based on queries to map external documentation to code. The work of Freitas described in [17] uses this kind of techniques to identify the comments associated to each problem domain concept.

Other authors systematically transform program identifiers into fragments of natural language sentences and then check whether the sentences are meaningful for humans using Google web search engine [10].

Fry [19] concludes that there are many natural language clues in program literals identifiers and comments. Natural language analysis of source code complements traditional program analysis. So, the idea is to use algorithms to automatically extract verb information from program source code comments and methods signature using concept location techniques.

The works described along this section were considered and influenced the design and decisions behind our development proposal. Full program comprehension tools—such as Alma [8], Alma2 [40], CodeCrawler [27], DA4Java [44], JIRiSS [45], Cerberus [14], Rigi [35], SHriMP [49, 48], SHriMP with Creole [30]—described in our previous works [3] or [39]) were also taken again in consideration, but the proposed ontology-based philosophy turns the present approach into a novelty.

3 An approach for Program Comprehension

The Program Comprehension (PC) approach we advocate in this paper is:

- **Problem Domain oriented**, because the PC process always starts with the terms involved in the description of the maintenance task to accomplish (a natural language statement, written as a discourse at the problem level); the idea is to locate these terms in the program, actually in its source code.
- **NL supported**, because maintenance task terms, before being located in the code, are first located in the *natural language sentences* (`NL-strings`) embedded in the source code, this is, in the *comments* and in the *text-messages* that are included in the input/output (`I/O`) statements in the form of string literal.
- **Ontology driven**, because ontologies are used to represent the knowledge that characterize both the Problem domain and the Program domain. This means that the maintenance task terms, we want to look for, should be formalized as *concepts* or *relations* belonging to the Problem Ontology (PrbO); when located in the program source code, these terms should be interpreted according to the *concepts* and *relations* in the Program Ontology (PrgO), an extension of the Programming Language Ontology (PLO).

After this short characterization, we can describe our proposal more precisely decomposing the approach into the following steps:

- read the maintenance task statement (a natural language sentence) and identify in the PrbO the terms (concepts and relations) that rigorously describe that task (sketch in Figure 1);
- pick up these terms from the PrbO and use an Information Retrieval (`IR`) engine to search for all `NL-strings` (*comments* or *`I/O` string literals* where the terms, or similar terms[1], appear more frequently (sketch in Figure 2);
- go through the retrieved `NL-strings` read them and choose the code chunks associated with the `NL-strings` that best fit the maintenance needs; use static or dynamic graphs (like the System Dependency Graph, or Execution Trace Graphs) to help on that choice (sketch in Figure 3);
- select the identifiers contained in the retrieved code chunks that are more similar to the terms used in the initial search (sketch in Figure 4);

[1] Dictionaries or Thesaurus shall be used to look for synonymous or translations.

- use the PrgO to interpret these identifiers, recognizing in this way the program elements involved in the maintenance task (sketch in Figure 5);
- use traditional PC techniques—like code visualizations and animation tools—to help in a deeper analysis of those code chunks to get a better understanding of them (sketch in Figure 6).

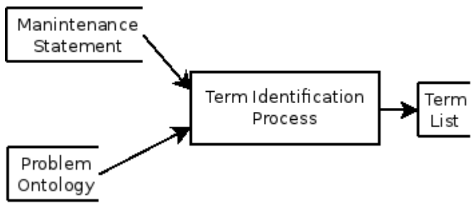

Figure 1 Term Process Identification Sub-system.

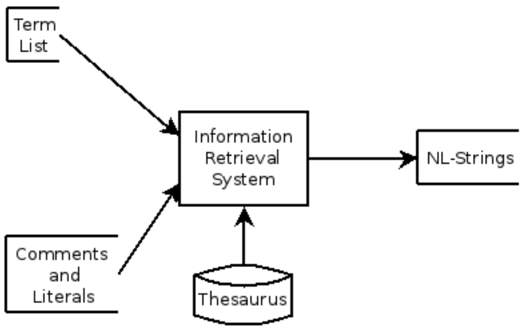

Figure 2 Information Retrieval Sub-system.

Figure 7 depicts the PC approach proposed summarizing the detailed list of steps above described.

To start the implementation of Quixote, we developed some tools combining several approaches and techniques for source code analysis and for concept location using ontologies. These tools will be described in the next sections, identifying the step where each one should be integrated.

4 Source code analysis tools

Obviously the *source code* is the first information resource to be considered for understanding a program.

In this section, a set of tools for analyzing `Java` source code is described. Each tool extracts static or dynamic information from the source code and applies different strategies to enable both the inspection and the comprehension of the software system under study.

Usually, the techniques employed in this context are centered in analyzing declarations and statements to identify data structures and control-flow, building representations such as: data/control-flow graphs, function-call graph, module dependency graph, etc. Strategies for analyzing natural language information sources, for instance comments and literals associated with I/O statements, are not commonly considered. However these program elements

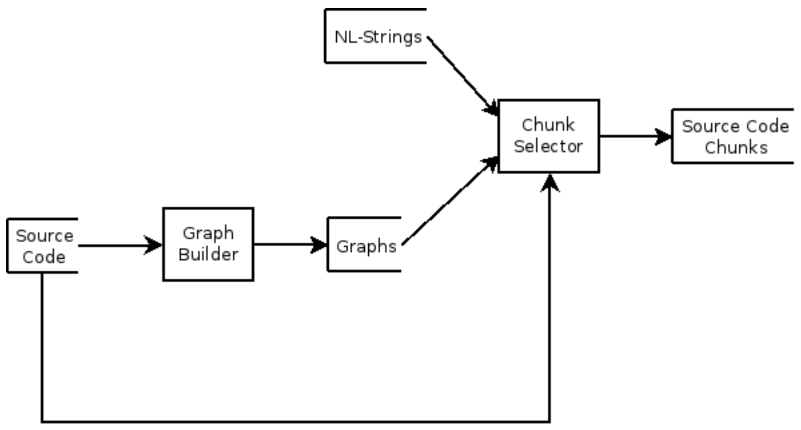

Figure 3 Chunk Selector Sub-system.

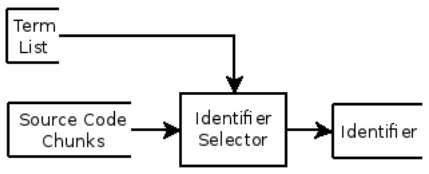

Figure 4 Identifier Selector Sub-system.

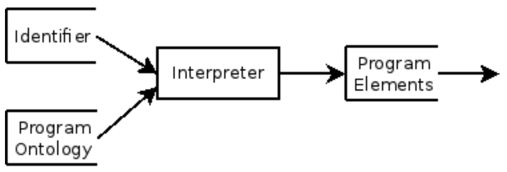

Figure 5 Interpreter Sub-system.

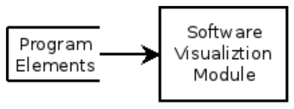

Figure 6 Visualization Sub-system.

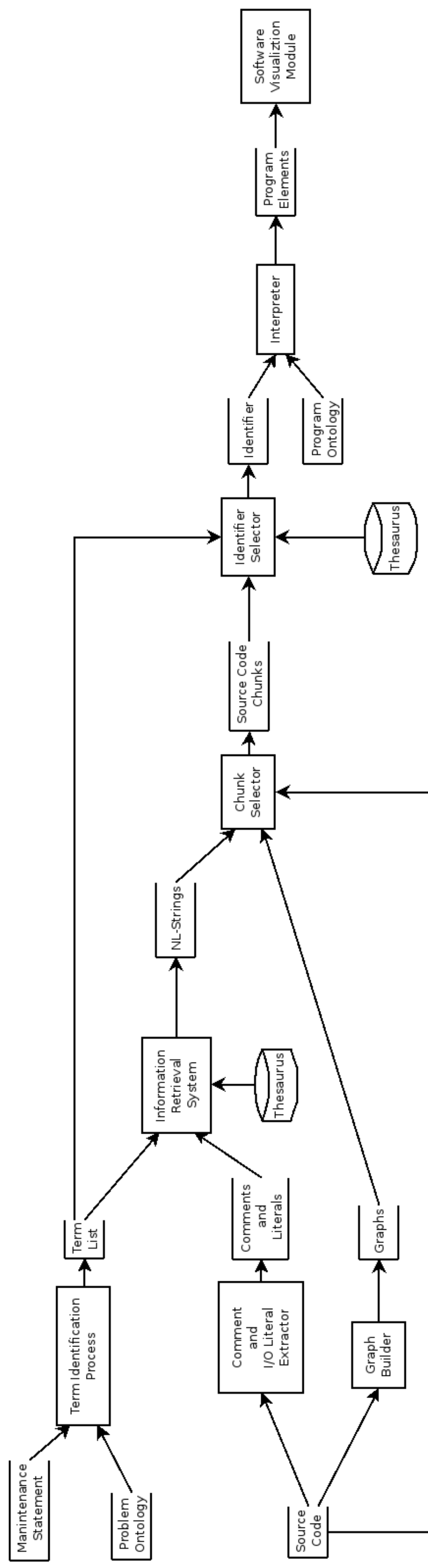

Figure 7 Program Comprehension Approach.

are important because they can provide information related with the problem domain, and it is well known that discover the concepts and relations of that domain is a hard task.

The tools that will be described in the following subsections were developed considering the arguments above mentioned.

4.1 Static Source Code Analysis

The first tool implements an innovative approach to work with static system information.

This approach builds as usual a parser tree and a rich symbol table where all the identifiers extracted from the code are associated with their context (scope and code block) and with their role or class according to the concepts defined in the Programming Language Ontology (PLO); in this way a Program Ontology (PrgO) is built. The PrgO is a useful internal representation of the knowledge inferable from the source code at a programming level (program domain). It allows the user to navigate through the programming language concepts down to their instances (identifiers) in the code, or from the instances up to the programming concepts.

In parallel we are working over the set of detected identifiers exploring them lexically and morphologically in order to enable the discovery of similarities with the concepts of the problem domain. For that purpose, several techniques are used: splitting identifiers into terms; expanding terms; and looking for translations, synonyms or Word_net meanings.

In this case, `AnTLR` [42], a well known compiler construction tool, was used to parse the code and to extract program identifiers (names of classes, interfaces, methods, variables, etc), but other tools like `LISA` [23], `Eli` [20], `JavaCC` [26], `CoCo/R` [24], `lex/yacc` [29] could have been used for the same purpose.

This tool will be used in steps, *Identifier Selector Sub-system* and *Interpreter Sub-system*, shown in Figures 4 and 5, in order to aid in the interpretation of the identifiers according to the PrgO. Till the moment this part of the system copes with C and Java programs.

4.2 Dynamic Source Code Analysis

The second tool is based on dynamic information [22]. Code Instrumentation (`CI`) is a strategy for gathering this kind of information. `CI` consists of inserting *specific sequences of statements* (*inspectors*) in strategic locations of the source code with the goal of capturing the system behavior. This task is performed automatically by parsing the source program. In [3] a tool called PICS was developed to inspect C programs using exactly the same methodology. Now we are interested to cope with object-oriented code.

JDIE (Java Dynamic Information Extractor) is a tool aimed at extracting dynamic information from `Java` programs using `CI`. In order to carry out this task, two factors must be taken into consideration:

1. The information to extract;
2. The source code places where this information can be recovered.

Considering the first factor, JDIE extracts information related with *methods*. This information is useful because it allows to detect which were the methods actually employed for a specific scenario. Concerning the second factor, the places selected are: the *beginning* and the *end* of each system method. With the observations previously mentioned, the instrumentation was carried out by inserting the *inspectors* at the beginning and at the end of each system method. These *inspectors* record the name of the method when its execution starts and finishes. This simple approach, that is effective for imperative languages like *C*, rise some limitations when applied to an object-oriented language like `Java`. Methods can finish with *return* statements that contain invocations to other methods inside them. In this case, the strategy does not work because the execution traces recorded are not correct. The solution to this problem is carried out by applying the following transformation to each *return* statement:

1. Create a local variable, *rvalue*, with the following characteristics:
   a. The type of *rvalue* is the method type (i.e., the type of the value returned by that method);
   b. The value of *rvalue* is determined by assigning to it the expression found inside the *return* statement.
2. Modify the statement *return* replacing the expression inside its body by the variable *rvalue* defined in previous step.

This approach works well. However, several problems appear when the methods have loops. Observe, the source code shown below.

```
for (i=0; i<30.000; i++) {
  g(x);
  h(x);
}
```

The instrumentation strategy will inform 30.000 times that function g begins and ends its execution. Similar information will be recorded 30.000 times for function h. In other words, a huge amount of information will be produced. One possible way for solving the problem presented by loops consists in the use of a stack. This stack contains in its top a value that indicates the number of times that the functions used inside the loop shall be registered. This stack is necessary because the iterations can be nested. The value is decremented at the end of each iteration. When it is zero, no more information concerned with this method will be registered. When the loop finishes its execution, the top of the stack is deleted.

To sum up, the instrumentation strategy can be fully described by the following algorithm: For each method M found in the source code, do:

1. Insert an extraction statement at the beginning of M.
2. For each statement *return* found in M, do:
   a. Create a local variable, *rvalue*, with the following characteristics:
      i. The type of *rvalue* is the method type (i.e., the type of the value returned by that method);
      ii. The value of *rvalue* is determined by assigning to it the expression found inside the *return* statement.
   b. Modify the statement *return* replacing the expression inside its body by the variable *rvalue* defined in previous step.
3. Insert an extraction statement at the end of M.

The reader can see [1] for more details about the instrumentation scheme described above.

The result of this instrumentation work is a sequence of function calls. Usually trace summarization techniques must be used in order to reduce the amount of extracted information. Over that summarized information several analysis are performed—graph based analysis (for instance, to discover sequence of calls) and identifier analysis—aimed at understanding the behavior of the code under study.

Besides the classic use of this instrumentation tool, there is the possibility of relating the traces extracted (graphs) with the identifiers in the PrgO, upgrading both. Moreover, the dynamic traces can be used in conjunction with the module for concept location in strings (comments and literals), aiding the user choosing the sequence of strings to read and relate to code.

This tool will be used in the step of our proposal that is described in Figure 3, Chunk Selector Sub-system, in order to construct the graphs. Additionally static and dynamic analysis can help to infer the actual types of input/output data in generic programs that will also be useful in program comprehension tasks.

4.3 Comment Analysis

Comments are interspersed by the Programmer among code lines, at software development phase. They are scattered all over the source code, sometimes wrapping a block of code (placed at the beginning or at its end), other times complementing a single statement. It is important to remember that comments are inserted by a programmer with two main purposes: to help himself during the development phase (and in this case they are not too much useful); to help other programmers, at the maintenance phase, in understanding his ideas. In that case, comments will contain, for sure, concepts associated with the problem domain, and they will be very relevant for PC tools.

The third tool we want to introduce, Darius, is responsible for locating automatically concepts in comments extracted from source code. The approach adopted is aimed at finding a relevant code chunk[2] using information retrieval techniques to locate problem domain concepts within comments. Darius extracts all the comments and classifies them per type (inline, block or JavaDoc comment), keeping their context, i.e., the code lines before/after the comments.

Picking up concepts from the ontology that describes the problem, it is possible to find all the comments that contain that concept (similar words) and rate them. Reading comments from the retrieved list, the programmer can select those that seem to him meaningful and dive directly into the associated chunk. Our idea, building this tool, is not to analyze comments to understand the associated statements. In the other way around, we propose to locate problem domain concepts on comments, and then identify the relevant code chunks associated with them. This approach is based on the ontology that describes the problem domain.

Darius [17] is precisely a tool developed to corroborate the previous statement.

In order to address the *concept assignment problem*, Darius follows standard IR strategies many times referred in the literature. To accomplish its task, we built it up from the following components:

- the document database builder that constructs the logical view of the documents (that in this case are the source program comments) and stores all of the relevant information associated with the comments;
- the IR engine (actually two models are available) that explore the information of the document database, and retrieve documents according to the queries the users provide;
- the graphical interface that provides all the interaction with the system and displays the results of finding Problem Domain concepts using comment information.

All the technical details regarding the development of each one of these components can be seen in [17][3].

[2] The code block where the programmer should focus the attention for some software maintenance purpose.

[3] This thesis is available at the URL `http://www3.di.uminho.pt/~gepl/QUIXOTE/FreitasJL2011thesis.pdf`.

Moreover, Darius provides some features to study the frequency of comment occurrences in the source files of a given project (for more details, see [18]). The data extracted and the measurements performed allow the user to compute some statistical information that is useful to verify if the source code contains enough comments that make this PC approach worthwhile[4]. Darius identifies and extracts *inline*, *block* and *javadoc comments* and provides some metrics useful to appraise the commentary policy followed by the program authors.

Clearly, this tool will be incorporated in step, *Information Retrieval Sub-system*, shown in Figure 2, in order to aid in locating problem concepts in the comments.

5 Visualization

To build *S*oftware Visualization components (for the sake of simplicity represented in Figure 6 as just one module in the last step), visualization techniques—concerned with scaling, drawings, coloring and processing speed—for graphs (flowcharts, function graph, module graph, system dependency graph, etc.), for trees (tree function, treemaps, and so on), for textual representations (source code structure or emphasized blocks, program metrics), for graphics, and animations will be fully exploited. Easy to use and advanced navigation features enabling the interconnection among all the above described visual artifacts and the interaction with the user, will be included to aid in the location of the code chunks and in the mapping between domains.

To produce these different visualizations, and address all the described features (actually not only concerned with the final results and restricted to the last process phase, but spread out all over the system), we rely on our strong background in this field. Theoretical works like the thesis [43, 11, 3, 38, 8] gave the support to the applications developed under the program comprehension project PCVIA [50]. Tools like Alma [8], Alma2 [40], PICS [2] and WebAppViewer [16] provide graphical features to expose information of both program and problem domains.

In Quixote PCTool suite, we plan to create visualizations for the three ontologies — PrbO, PLO, PrgO— as well as for the program source code, and the system dependency graph. Adequate navigation techniques among them will allow the visualization of the envisaged mapping.

6 Conclusion

It is well known that a programmer understands programs when he can relate the problem and program domains. Nevertheless, it is recognized that such relation is difficult to reach and build. Having present this inconvenient, and with the purpose of providing a new solution to this challenge, several approaches in the literature were analyzed. From that study it came out that basically all of the approaches rely on the use of static and dynamic information to build program representations and to infer problem concepts.

Strategies, based on knowledge representation techniques enabling semantic directed manipulation for precise definition of the concepts used in both the problem and program domains, are rare in the literature.

In this paper a novel ontology-based approach for Program Comprehension was presented. Here, ontologies allow for a precise description of a domain in terms of its concepts and relations.

[4] Notice that this approach has no meaning if source code is not interleaved with comments.

The approach mentioned above uses four inputs for mapping problem domain concepts into system source code. The first one is the Problem Domain Ontology. It describes the problem concepts and the relations between them. The second one, Maintenance Statement, is used for identifying the problem domain concepts involved in a maintenance task. The third one, Source Code, is an important information resource employed by several tools concerned with the extraction of identifiers and concepts location. The fourth one, Program Ontology, is used to take advantage of the semantic provided by the relation between the program elements linked with programming language ontology.

The program domain ontology (PrgO) is built automatically and is based on the programming language ontology (PLO) and on the source code. The PLO is previously developed and by default incorporated in our PCTools suite.

The information provided by the inputs previously mentioned are processed for: (*i*) linking the problem concepts with chunks of source code that implement them and (*ii*) providing several views for helping the programmer to understand there behavior. The processing referenced before implies: (*i*) extraction of static and dynamic information and (*ii*) interpretation of the information gathered, in order to understand how the system works.

On the one hand, the information extraction is carried out by applying several techniques of static and dynamic analysis and strategies of natural language processing. On the other hand, the interpretation phase is accomplished using the program ontology and identifier list appropriately filtered. At the end of the processing phase, it is possible to visualize, from several perspectives, the chunks of code that implement the problem domain concepts involved in the maintenance task. As a final remark, the approach described makes a deeper exploration of information of both the problem and program domains for reaching clear and robust relations between them.

References

1 Hernan Bernardis, Daniel Riesco, Carlos Salgado, Mario Beron, and Pedro Rangel Henriques. Analisis dinamico para la creacion de estrategias de comprension de programas. In *WICC 2012 - XIV Workshop de Investigadores en Ciencias de la Computacion, Misiones, Argentina*, April 2012.

2 M. Berón, P. Henriques, M. Varanda, and R. Uzal. PICS un sistema de comprensión e inspección de programas. *Congreso Argentino de Ciencias de la Computación CACIC 2007*, 13:462–473, 2007.

3 Mario Marcelo Berón. *Program Inspection to interconnect the Behavioral and Operational Views for Program Comprehension.* PhD thesis, National University of San Luis & University of Minho, Nov. 2009.

4 Barry W. Boehm. *Software Engineering Economics.* Prentice Hall PTR, 1981.

5 Ruven E. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, Nov. 1983.

6 P. Cimiano, A. Hotho, and S. Staab. Learning concept hierarchies from text corpora using formal concept analysis. *Journal of Artificial Intelligence Research*, (24):305–339, 2005.

7 Philipp Cimiano. *Ontology Learning and Population from Text.* Springer-Verlag New York Inc., 2010.

8 Daniela da Cruz, Pedro Rangel Henriques, and Maria João Varanda Pereira. Constructing program animations using a pattern-based approach. *ComSIS – Computer Science an Information Systems Journal, Special Issue on Advances in Programming Languages*, 4(2):97–114, 2007.

9 Jan Jurjens Daniel Ratiu, Martin Feilkas. Extracting domain ontologies from domain specific. 2008.

10 Lars Heinemann Daniel Ratiu. Utilizing web search engines for program analysis. *18th IEEE International Conference on Program Comprehension*, 2010.

11 Eva Ferreira de Oliveira. Caracteristicas de um sistema de visualiza cão para compreensão de programas web. Master's thesis, University of Minho, Sep. 2006.

12 S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

13 Lucas Drumond and Rosario Girardi. Extracting ontology concept hierarchies from text using markov logic. In *SAC*, pages 1354–1358, 2010.

14 Marc Eaddy, Alfred V. Aho, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. CERBERUS: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, pages 53–62, Washington, DC, USA, 2008. IEEE Computer Society.

15 R. K. Fjeldstad and W. T. Hamlen. Application program maintenance study: Report to our respondents. In *Proceedings GUIDE 48*, Apr. 1983.

16 Ruben Fonseca, Daniela da Cruz, Pedro Henriques, and Maria Jo ao Varanda Pereira. How to interconnect operational and behavioral views of web applications. In IEEE, editor, *ICPC'08 - 16th International Conference on Program Comprehension*. Amsterdam, Holanda, June 2008.

17 José Luís Freitas. Comments Analysis for Program Comprehension. Master's thesis, Dec 2011.

18 José Luís Freitas, Daniela da Cruz, and Pedro Rangel Henriques. The role of Comments on Program Comprehension. In Luis Caires and Raul Barbosa, editors, *INForum'11 — Simpósio de Informática (CoRTA'11 track)*, Coimbra, Portugal, September 2011. Universidade de Coimbra.

19 Z. P. Fry, D. Shepherd, E. Hill, L. Pollock, and K. Vijay-Shanker. Analysing source code: looking for useful verb-direct object pairs in all the right places. *Natural Language in Software Engineering, IET Software*, 2(1):27–36, 2008.

20 R. Gray, V. Heuring, S. Kram, A. Sloam, and W. Waite. Eli: A complete, flexible compiler construction system. Research report, Univ. of Colorado at Boulder, Oct. 1990.

21 T. Gruber. Towards principles for the design of ontologies used for knowledge sharing. *L. of Human and Computer Studies*, 43:907–928, 1994.

22 Bernardis H. Instrumentacion de programas escritos en java para interconectar los dominios del problema y del programa. In Universidad Tecnologica Nacional, editor, *40 Jornadas Argentinas de Informatica e Investigacion Operativa. 40 JAIIO. Concurso Estudiantil. EST 2011.*, volume 40, 2011.

23 Pedro Henriques, Maria Joao Varanda, Marjan Mernik, Mitja Lenic, Jeff Gray, and Hui Wu. Automatic generation of language-based tools using lisa system. *IEE Software Journal*, 152(2):54–70, April 2005.

24 Pedro R. Henriques and Jose Joao Almeida. O Gerador de Compiladores COCO. Relatório de instalação, G.D. Ciências da Computação, D.I./ Univ. Minho, Mar. 1990.

25 Sangkyu Rho Jinsoo Park, Worchin Cho. Evaluating ontology extraction tools using a comprehensive evaluation framework. *Data&Knowledge Engineering*, 69:1034–1061, 2010.

26 Viswanathan Kodaganallur. Incorporating language processing into java applications: A javacc tutorial. *IEEE Software*, 21:70–77, 2004.

27 Michele Lanza and Stéphane Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proceedings of OOPSLA 2001*, pages 300–311, 2001.

28 S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *Software, IEEE*, 3(3):41–49, 1986.

29 J.R. Levine, T. Mason, and D. Brown. *Lex & Yacc.* Ed. Dale Dougherty. O'Reilly & Associates Inc., 1992.

30 Rob Lintern, Jeff Michaud, Margaret-Anne Storey, and Xiaomin Wu. Plugging-in visualization: experiences integrating a visualization tool with eclipse. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 47–56, New York, NY, USA, 2003. ACM.

31 David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. *J. Syst. Softw.*, 7(4):341–355, 1987.

32 Radu Vanciu Maksym Petrenko, Vaclav Rajlich. Partial domain comprehension in software evolution and maintenance. *16th IEEE International Conference on Program Comprehension*, 2008.

33 Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan Maletic. An information retrieval approach to concept location in source code. *WCRE 2004 - 11th IEEE Working Conference on Reverse Engineering*, 2004.

34 Thomas P. Moran and Stuart K. Card. Applying cognitive psychology to computer systems: A graduate seminar in psychology. In *Proceedings of the 1982 conference on Human factors in computing systems*, pages 295–298, New York, NY, USA, 1982. ACM.

35 H. A. Müller and K. Klashinsky. Rigi-a system for programming-in-the-large. In *ICSE '88: Proceedings of the 10th international conference on Software engineering*, pages 80–86, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

36 Ulric Neisser. *Cognitive Psychology.* Appleton-Century-Crofts, New York, 1967.

37 Timothy Lethbridge Nicolas Anquetil. Extracting concepts from filenames. 1998.

38 Nuno Oliveira. Improving program comprehension tools for domain specific languages. Master's thesis, University of Minho, Braga, Portugal, October 2009.

39 Nuno Oliveira, Pedro Rangel Henriques, Daniela da Cruz, Maria João Varanda Pereira, Marjan Mernik, Tomaž Kosar, and Matej Črepinšek. Applying program comprehension techniques to karel robot programs. In *Proceedings of the International Multiconference on Computer Science and Information Technology - 2nd Workshop on Advances in Programming Languages (WAPL'2009)*, pages 697–704, Mragowo, Poland, October 2009. IEEE Computer Society Press.

40 Nuno Oliveira, Maria João Varanda Pereira, Pedro Rangel Henriques, and Daniela da Cruz. Visualization of domain-specific program's behavior. In *Proceedings of VISSOFT 2009, 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 37–40, Edmonton, Alberta, Canada, September 2009. IEEE Computer Society.

41 Pocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. On the equivalence of information retrieval methods for automated traceability link recovery. *18th IEEE International Conference on Program Comprehension*, 2010.

42 Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages.* The Pragmatic Bookshelf, Raleigh, 2007.

43 Maria João Varanda Pereira. *Sistematização da Animação de Programas.* PhD thesis, University of Minho, Nov. 2003.

44 M. Pinzger, K. Grafenhain, P. Knab, and H. C. Gall. A tool for visual understanding of source code dependencies. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 254–259, 2008.

45 Denys Poshyvanyk, Andrian Marcus, and Yubo Dong. Jiriss - an eclipse plug-in for source code exploration. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 252–255, Washington, DC, USA, 2006. IEEE Computer Society.

46 Vaclav Rajlich and Norman Wilde. The role of concepts in program comprehension. *IWPC-10th IEEE International Workshop on Program Comprehension*, 2002.

47 Ben Shneiderman and Richard Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Parallel Programming*, 8(3):219–238, Jun. 1979.

48 Margaret-Anne Storey. Designing a software exploration tool using a cognitive framework of design elements. In Kang Zhang, editor, *Software Visualization: From Theory to Practice*, pages 113–148. Springer, 2003.

49 Margaret-Anne Storey, Casey Best, Jeff Michaud, Derek Rayside, Marin Litoiu, and Mark Musen. SHriMP views: an interactive environment for information visualization and navigation. In *CHI '02: CHI '02 extended abstracts on Human factors in computing systems*, pages 520–521, New York, NY, USA, 2002. ACM.

50 Maria João Varanda and Pedro Henriques. Program comprehension by visual inspection and animation.

51 A. von Mayrhauser and A. M. Vans. From program comprehension to tool requirements for an industrial environment. In *In Proceedings of IEEE Workshop on Program Comprehension*, pages 78–86, 1993.