Program Inspection to inter-connect the Operational and Behavioral Views for Program Comprehension

Mario M. Berón[1,3]
Pedro R. Henriques[1], Maria J. Varanda[2], Roberto Uzal[3]

[1]Department of Informatics
University of Minho, CCTC, Braga, Portugal
{marioberon|prh}@di.uminho.pt
[2]Department of Informatics
Polytechnical Institute of Bragança, Bragança, Portugal
mjoao@ipb.pt
[3]Informatics Department
National University of San Luis, San Luis, Argentina
ruzal@uolsinectis.com.ar
Academic Degree: Ph.D. in Computer Science

Abstract. Program Comprehension (PC) is a discipline of Software Engineering aimed at creating models, methods, techniques and tools, based on a *learning process* and *engineering process*, to help the engineer in getting a deeper knowledge about software systems.
The learning process is concerned with the mental process followed by the programmer when he needs to understand programs. This topic is explained in the context of *Cognitive Models*.
The engineering process includes the study of methods for: *Information Extraction from Programs* and *Software/Data Visualization*. The research in these areas allows to build Program Comprehension tools with quality. Moreover, it aids in to inter-connecting the problem domain (system behavior) and program domain (software components). This is one of the biggest challenge in PC.
In this paper, we present the background and results obtained in the topics described above during a Ph.D. thesis denominated: *Program Inspection to inter-connect the Operational and Behavioral Views for Program Comprehension*.
Key words: Cognitive Models, Information Extraction, Software Visualization.

1 Introduction

Program Comprehension (PC) [Sto05] is a discipline of Software Engineering aimed at helping the programmer to understand programs. PC is useful for Software Maintenance because it helps to reduce efforts and time during the software analysis and reverse-engineering phases.

Software Maintenance and Evolution is a critical activity that has three important task: *Perfective*, *Corrective* (also conceptualizad as extensibility and modifiability) and *Adaptive* (also known as portability). The first is related with the incorporation of new functionalities. The second covers the detection and elimination of bugs in the program. The third is concerned with the system adaptation to a new context either in the problem domain or in the program domain, such as a new file format, changes in its code organization, etc. These three main activities consume much time and many resources. To be more precise, Xi and Hassan in [XPH07] declare that 39% of the activities are perfective, 56.7% are corrective and 2.2% correspond to adaptative tasks. Furthermore, the maintenance and evolution history shows, from 1975 until 2005, a linear increment in the budget of SE project devoted to this activity [XPH07]. This information justifies the scientific community interest in finding aids to reduce those costs.

In the maintenance and evolution context, the programmer must understand large documents with different formalism and methodologies. Furthermore, he must match the information gathered with the system source code. Finally, he must comprehend how the system carry out its functionalities in a higher abstraction level. Program Comprehension (PC) aims at facilitating this task by providing methods, techniques and tools. In this way, it is possible to overcome the problem described in the precedent paragraphs. In order to create methods, techniques and tools, is important to study topics such us: Cognitive Models (CM), Software Visualization (SV) and Information Extraction (IE). In PC context, CM topics are very important, because PC involves a learning process that describes how the programmer understands programs. To know the CM theories is useful. It is because helps to decide how the PC system components must be composed. This composition is important because assists the programmer's mental process. SV allows to represent the system information in order to facilitate the system information comprehension. Finally, IE is needed because it provides the information to implement CM and to build SV strategies.

In this paper, we contextualize and give the motivation for Mario Berón's Ph.D. work, introduce the theme, and present the advances made in the first year and an half. The research topic is concerned with finding a way to combine static and dynamic analysis techniques, software visualizations approaches and cognitive models in order to propose strategies effective for PC; that target requires a deep study and understanding of each of the four components, and the ability to find out the commom denominator among them. In another words, this paper is intended to report on the way followed to elaborate PC strategies aimed at relating the problem and program domains to facilitate program understanding. It is important to remark that our work has not yet been validated with real test cases (we could not verify its effectiveness with end-users), because we need more time for integrating our proposal in one PC environment and to design an appropriated set of assessment tests. However, we believe that our partial results are promising to continue our work on PC.

This paper is organized as follow. Section 2 presents the researches made on Cognitive Models. Section 3 describes the main topics of Software Visualization.

Section 4 expounds one strategy to extract information from the systems. Section 5 shows different approaches to reach the behavioral-operational relation. Finally, section 6 presents the conclusion of this paper.

2 Cognitive Models

Cognitive Models (CM) [BHVU07b] [O'B03] [Xu05] in PC context describe how the programmers understand programs. CM are important because they allow to understand different mental processes used by the programmer for PC [VMV95]. Understanding the CM's theory makes it possible to develop strategies and tools with cognitive support.

After studying the state of the art of CM we can extract the following conclusions:

- Many authors say that the programmer understands a program when he can explain the problem and program domains. For each system behavior (problem domain), the programmer must find the piece of code (program domain) used to perform that behavior [O'B03] [Tie89] [BHVU07b].
- The CM concepts [RW02] are unclear. This characteristic makes difficult the interpretation and implementation of these concepts in one PC tool. For example, the *chunk* concept is often used to explain how the programmer extracts information from the systems. However, CM bibliography do not give a precise definition or method to extract *chunk*. For these reason, the programmer applies his our criteria to implement this concept. This activity produces conflicts in the interpretation.
- Analyzing different CM theories, we conclude that several approaches can be used to learn. For instance, on one hand, Brook [Bro78] says that the process used by the programmers to understand programs is top-down. On another hand, Schneiderman and Mayer hold that the learning follows a bottom-up process. Finally, other authors developed integrated models that use both learning approaches [NG84] [O'B03]. It is important to notice that all authors present empirical studies to prove their hypothesis.

The first conclusion was useful because it allows us to select an approach to build strategies and PC tools. The second makes possible to develop some contributions on this topic. Finally, the third confirms that the learning approach is selected by the programmer.

Our contributions are related with the specialization of the CM concepts. This task consists in re-define some CM concepts in order to make them more understandable for the programmer. In this way, we avoid the complications produced by the use of pedagogic vocabulary. More details about the concept systematization can be seen in [BHVU07b].

3 Software Visualization

In this context, the work was divided in two steps: to extract the main concepts of software visualization and to select useful views to be implemented in PC

tools. The next two sections are devoted to shortly describe the state of the art on each of them together with our preliminary results.

3.1 Software Visualization Concepts

Software Visualization (SV) [GH01] [Che06] is an area of Software Engineering aimed at visualizing the main properties of systems [LELP06]. In SV [PBS93] [Mye90], we can distinguish different concepts: algorithms, programs, and systems. In *Algorithm Visualization*, the focus is in teaching algorithms and data structures. This kind of SV is oriented to the student and it is good to be used in the context of learning. In *Program Visualization* [BA01], the focus is to discover the program functionalities and visualize the relation between its components. It is used to study the program's behavior. Finally, *System Visualization* is oriented to study big programs composed by different modules. For each module, techniques to visualize programs can be applied. However, it is necessary to incorporate another levels of visualization.

From the discussion above, we can establish the following inclusion relation between those visualizations:

$$Alg.\ Visualization \subset Prog.\ Visualization \subset Sys.\ Visualization \quad (1)$$

With the expression (1) (see [BHVU07b] for more details), we want to express the inclusion relation in the scope of SV's strategies. We declare that all techniques applied on one given subset is possible to use in its super-set. The producer of SV tools must analyze if the technique used on one level are useful on another (high or lower) level. On the other hand, SV is related with another programming concepts such as: *Visual Programming* (VP) and *Programming by Example*(PE). VP is a programming paradigm where the programs are specified using graphical objects. Commonly, these elements represent basic constructions of some programming language. PE is another kind of programming. The programmer specify the input and output of the program, and then the computer builds the system through an inference process.

This short discussion make clear distinction between the concepts involved in SV. *System Visualization*, *Program Visualization* and *Algorithm Visualization* are topics of SV concerned with represention of system's information; *Visual Programming* and *Programming by Example* use SV techniques but they are concerned with the programming task itself.

3.2 Views

One main approach to software understanding consists in the use of views. One view is a system perspective that shows the static and dynamic information of the system [MFM03] [PFVI01].

In order to create views is necessary to consider the structural and graphical representations of the data. The first one is necessary to have efficient access to the information needed to construct the graphical representation. The second one is related with the way to show the information to the user.

We select different views to carry out the system comprehension. For example: *module graph*, *module dependence graph*, *function graph*, *runtime function graph*, *type dependence graph*, *symbol table visualization*, *source and object codes*, etc.

For each view, we define: attributes to be shown, structural and graphical representations and strategies to overcome the problem of information overloading. Furthermore, we propose a different way to inter-connect these views using navigation functions. This allows the implementation of strategies to explain behavioral-operational relation. More details can be seen in [BHVU07b].

4 Method of Information Extraction

In order to extract dynamic information from the system we use *Code Instrumentation* [BHVU07b] [BHVU07a]. This technique consists in the insertion of useful statements in the source code. We select as check points the beginning and the end of each function. In these places, we insert inspection functions. These functions print the function name or some other information (name, parameters, etc.) needed by the programmer to build strategies or to implement tools for PC. The result obtained with this approach were not good because the system functions can be invoked many times (see [BHVU07b] for more information about evaluation). This situation appears when the functions are invoked inside the body of the iterations, or they are part of some complex algorithm.

In this case, the quantity of functions gathered by our approach is huge. For this reason, we need to instrument the source code to control the iteration statements. To reach this goal, this kind of statement was instrumented inserting statements before, inner and after loops.

The statement before the loop inserts in the stack the number of times that the functions inner the loop must be shown. The statement inner the loop decrements the top of the stack by one. Finally, the statement after the loops recovers the number of times that the functions in the previous loops must be shown.

This scheme allows to reduce the quantity of functions gathered by our instrumentation technique.

This first result was promising and let us to go further. Figure 1.a shows the scheme implemented for the function instrumentation. Figure 1.b shows the scheme to control loops.

Finally, the implementation of these approaches allowed us to recover the system static information. The information gathered in this way allows us to build the system's symbol table, module/functions level descriptions and to create a strategy to detect abstract data types. See [BHVU07b] for further reading on these topics.

5 Strategies to reach a Behavioral-Operational Relation

In this section, we describe two procedures to inter-connect the operational and the behavioral views [LF94]. The first one is a consequence of our code in-

```
void f (int a, int b)
  { INPUT_INSPECTOR("f");
    ........
    OUTPUT_INSPECTOR("f");
    return;
  }
```

(a)

```
push(stack,N)
for(i=0;i<SIZE;i++)
 { /* Loop actions */
  .....
  decTop(stack);
 }
pop(stack);
```

(b)

Fig. 1. Instrumentation of Functions and Loops

strumentation scheme. The second is a strategy that use dynamic and static information to recover this relation.

5.1 Side-Effect of the Code Instrumentation Scheme

This approach allows to execute in parallel the instrumented system and the *Inspection Function Manager* [BHVU07b] [BHVU07a]. This procedure allows to visualize, directly, the functions used by the system to build its output. The effect of this approach is similar to see a program trace. The main difference with the debugging tools is that this trace is made to function level. Basically, this result is a side-effect of our code instrumentation scheme.

5.2 BORS

BORS (Behavioral-Operational Relation Strategy) [BHVU07a] [BHVU07b] is a strategy aimed at explaining the functions used by the system to build its output. This task is carried out in two ways: i) showing the context where the functions are called, and ii) describing the activities made by the functions.

BORS is based in the following observation:

The system output is composed by problem domain objects. Normally, these objects are implemented by abstract data types, in the case of imperative languages, or classes, in the case of object oriented languages. The TDAs and Classes have data objects to store their state. Furthermore, they have an operation set that works with these data objects. For these reasons, it is possible to describe each problem domain object using TDAs or classes that implement them.

BORS consists the following steps:

To detect the functions related with each problem domain object: this task is semi-automatic. The user selects the ADTs that he wants to explain. For this task, he can use one heuristic to detect ADT described in [BHVU07b]. After that, all functions related with that types are ready to be explained and they are inserted in one list.

To build a function execution tree: with the output of our code instrumentation scheme we can build a fe-tree (**F**unction **E**xecution **T**ree) [BHVU07b] [BHVU07a]. This data structure describes how the system functions were executed.

To explain the functions found in the step 1 using the tree built in the step 2: The information built in the previous steps (function list and fe-tree) can be combined to explain the functions. This task is carried out showing: i) the context where each function was called (path from the fe-tree root until the function under analysis), and ii) the task that this function made for the system (sub-tree with root the function under study). To carry out this task, BORS apply a breath first traversal on the fe-tree. If the visited node is a member of the list of function to be explained, the strategy reports the path of this node until the root and its sub-tree.

6 Conclusion

In this paper we present the studies developed in the context of a Ph.D thesis. They are divided in the following areas: Cognitive Models, Software Visualization and Method to extract information. These topics allow us to extract the most important aspects to build strategies and tools for PC. The main contributions of this work are: *the concept systematization of Cognitive Models*, *the characterization of different system views*, *the creation of one scheme of code instrumentation* and *BORS strategy*. The first and second contributions are more theoretical while the others are more practical. The practical procedures are currently implemented in PICS (Program Inspection and Comprehension System) a system aimed at helping in the task of program understanding. PICS implements all the techniques and views described in this paper. Furthermore, it uses BORS and code instrumentation as strategies to relate the problem and program domains. Detailed description can be read in [BHVU07b] [BHVU07a]. The future work includes the following tasks: to complete the PICS's visualization system using the theoretical skills described in section 3, to optimize the code instrumentation scheme allowing the user to choose what instructions need to be instrumented, and to create another approaches to relate the program components with the system behavior. Moreover, we wish to implement techniques to create high level documentation using the conceptual systematization presented in section 2.

Finally, we want to make behavioral analysis with the goal to create another strategy behavioral-operational using a hybrid approach (top-down between the behavior and data dictionary and bottom-up between the code and the data dictionary).

References

[BA01] M. Ben-Ari. Program Visualization in Theory and Practice. *UPGRADE*, 2(2):8–11, 2001.

[BHVU07a] M. Beron, P. Henriques, M. Varanda, and R. Uzal. A System to understand Programs Written in C Language by Code Annotation. *European Joint Conference on Theory and Practice of Software, (2007)*, 2007.

[BHVU07b] M. Berón, P. Henriques, M. Varanda, and R. Uzal. Program Inspection to interconnect Behavioral and Operational Views for Program Comprehension. *Technical Report*, 2007.

[Bro78] R. Brook. Using a behavioral theory of program comprehension in software engineering. *Proceedings of the 3rd international conference on Software engineering*, pages 196–201, 1978.

[Che06] Chaomei Chen. *Information Visualization*. Springer Verlag, 2006.

[GH01] L. Gómez Henríques. Software Visualization: An Overview. *Informatique*, pages 4–7, 2001.

[LELP06] W. Lowe, M. Ericsson, J. Lundber, and T. Panas. Software Comprehension-Integrating Program Analysis and Software Visualization. *Technical Report*, 2006.

[LF94] H. Lieberman and C. Fry. Bridging the gulf between code and behavior in programming. In *ACM Conference on Computers and Human Interface*, Denver, Colorado, April 1994.

[MFM03] A. Marcus, L. Feng, and J.I. Maletic. 3D representations for software visualization. *Proceedings of the 2003 ACM symposium on Software visualization*, 2003.

[Mye90] B. Myers. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, 1990.

[NG84] J.D. Novak and D.B. Gowin. *Learning How to Learn*. Cambridge University Press, 1984.

[O'B03] Micheal P. O'Brien. Sofware Comprehension - A Review and Research Direction. *Technical Report*, 2003.

[PBS93] B.A. Price, R. Baecker, and I.S. Small. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.

[PFVI01] C. Pareja-Flores and J. A. Velásquez-Iturbide. Representing Abstractions. *UPGRADE*, 2(2):2–3, 2001.

[RW02] V. Rajlich and N. Wilde. The role of concepts in program comprehension. *Program Comprehension, 2002. Proceedings.*, pages 271–278, 2002.

[Sto05] M.A. Storey. Theories, methods and tools in program comprehension: past, present and future. *Proceedings of the 13th International Workshop on Program Comprehension*, pages 181–191, 2005.

[Tie89] Tim Tiemens. Cognitive Model of Program Comprehension. *Technical Report*, 1989.

[VMV95] A. Von Mayrhauser and AM Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[XPH07] Tao Xie, Jian Pei, and Ahmed E. Hassan. Mining software engineering data. In *Proc. 29th International Conference on Software Engineering (ICSE 2007), Companion Volume, Tutorial*, pages 172–173, May 2007.

[Xu05] S. Xu. A Cognitive Model for Program Comprehension. *Conference on Software Engineering Research, Management and Aplications.*, 2005.