Evaluation Criteria of Software Visualization Systems used for Program Comprehension

Mario M. Berón
National University of San Luis, UNSL
Ejército de los Andes 950, San Luis, Argentina
mberon@unsl.edu.ar

Daniela da Cruz
University of Minho
Campus de Gualtar, 4715-057, Braga, Portugal
danieladacruz@di.uminho.pt

Maria João Varanda Pereira
Polytechnic Institute of Bragança
Campus de Sta. Apolônia, Apartado 134 - 5301-857, Bragança, Portugal
mjoao@ipb.pt

Pedro Rangel Henriques
University of Minho
Campus de Gualtar, 4715-057, Braga, Portugal
pedrorangelhenriques@gmail.com

Roberto Uzal
National University of San Luis
Ejército de los Andes 950, San Luis, Argentina
ruzal@uolsinectis.com.ar

Abstract

The program understanding task is usually very time and effort consuming. In a traditional way the code is inspected line by line by the user without any kind of help. But this becomes impossible for larger systems.
Some software systems were created in order to generate automatically explanations, metrics, statistics and visualizations to describe the syntax and the semantics of programs. This kind of tools are called Program Comprehension Systems.
One of the most important feature used in this kind of tool is the software visualization. We feel that it would be very useful to define criteria for evaluating visualization systems that are used for program comprehension. The main objective of this paper is to present a set of parameters to characterize Program Comprehension-Oriented Software Visualization Systems. We also propose new parameters to improve the current taxonomies in order to cover the visualization of the Problem Domain.

Keywords

Program Comprehension, Software Visualization, Problem Domain, Program Domain, System Views, Evaluation Criteria

1 INTRODUCTION

Program Comprehension (PC) is a discipline of Software Engineering(SE) aimed at providing models, methods, techniques and tools, based on specific learning and engineering processes, in order to reach a deep knowledge about software system. The learning process involves the study of the Cognitive Science and the relation between its main concepts with SE. The engineering process includes the study of areas such as: *Software Visualization*, *Information Extraction*, *Information Management* for representing the system information in one way that emphasize its main aspects.

A Program Comprehension Tool (PC Tool) is a software system whose development requires the combination of: *Cognitive Models*, *Extraction and Management of Information*, *Software Visualization*, and *Strategies for interconnecting several domains*. The main claim of this kind of software systems is to make easier the Program Comprehension process.

The Software Visualization subsystem plays an important role in program understanding, because its duty consists in showing the information extracted from the system in a coherent and easy to understand manner.

Nowadays, Software Visualization systems present useful program views (Program Domain) but they do not contemplate another interesting view such as the system output (Problem Domain) and its relation with the program components. This problem gave rise to a new kind of Software Visualization: the *Program Comprehension-Oriented Software Visualization Systems* (PC-SVS).

This kind of systems (PC-SVS) has the same characteristics as the traditional Software Visualization Systems (SVS) but now they should incorporate special visualizations oriented for both the Problem and Program Domains and the relation between them.

This new conceptualization introduces problems with the current Software Visualization Systems taxonomies because they do not consider the special kind of visualizations referred above. Thus, the old taxonomies can not be used for assessing the quality and usefulness of the subsystem of Software Visualization for PC tools (PC-SVS).

This paper presents a contribution to overcome this drawback, by completing the most important taxonomy with characterizations of the Problem and Program Domains and the relations between them.

The organization of the paper is as follow. Section 2 presents the current state of the art of the Software Visualization Taxonomies. Section 3 describes the characteristics absent from all taxonomies. Section 4 expounds several new criteria for characterizing the Problem Domain and its relation with the Program Domain. Section 5 concludes the paper and sets forth the main contributions of the work.

2 TAXONOMIES FOR SOFTWARE VISUALIZATION SYSTEMS

Currently, there are too many SV taxonomies. They have been defined to overcome the problem of categorizing and assessing the approaches used by SV tools. Each taxonomy emphasizes different SV aspects, for instance specific software characteristics, human activities, etc.

In this section, an overview of SV taxonomies is presented. Then, this systematization is used to detect items missing in all the taxonomies, and to set down a proposal to overcome them.

Brown in [3] introduces a visualization approach focussed on animations. He describes his proposal using three main axes: *Content*, *Transformation* and *Persistence*. The first is concerned with the way followed for representing the program. This characteristic is divided in *Direct*, in which the source code is directly represented using graphical artifacts, and *Synthetic*, in this case a source code abstraction is graphically depicted. The second refers to the animation process used in the visualization. It can be *Discrete*, a series of snapshots are shown, or *Incremental*, a smooth technique is employed to produce the transition between snapshots. Finally, *Persistence* is related with the possibility of holding the process history.

Myers, a pioneer in SV, presents in [4] one simple taxonomy with two main axes: *The kind of object that the visualization system attempts to illustrate* and *The type of information shown*. The first component consists of code, algorithms or programs, and the second one is concerned with the static and dynamic information. The combination of these axes produces the following visualization sorts:

Code-static: visualizations such as flowcharts and the Seesoft technique [1] use this approach.

Code-dynamic: it is concerned with software animations and strategies to show the code segment used in one specific execution.

Data-static: it follows the same approach that for static code visualizations. The main idea consists in presenting in a suitable way the system data objects and their values.

Data-dynamic: at the same way that code-dynamic visualizations it consists of animations aimed at showing the variables and their values at runtime. Furthermore, this topic is concerned with the strategies elaboration for depicting how the data changes through the time.

Algorithm-static: it is interested in generating snapshots of algorithms.

Algorithm-dynamic: it consists in building algorithms animations for presenting an integral dynamic view of each component used at runtime.

Roman and Cox [7] propose to classify the SVS using five criteria based in the SV model shown in Figure 1.

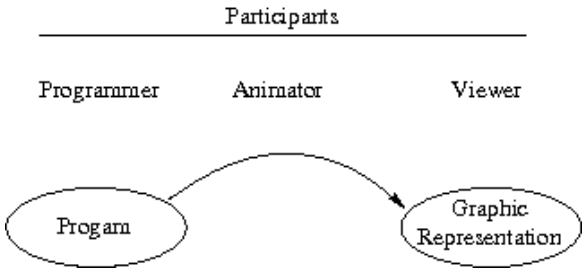

Figure 1. Roman and Cox model

Scope: this item covers the program aspects to visualize;

Abstraction level: describes the visualization sophistication degree;

Specification method: explains which are the mechanisms used by the animator for building the visualization;

Interface: expounds the facilities provided by the SV system for presenting and manipulating the visualization, and

Presentation: is concerned with the effectiveness of the information presentation. In another words, this item analyzes the SV performance to transmit information.

It is important to remark that each criterion is in detail explained through several sub-criteria that allow to improve the understanding of this classification. The reader interested in this taxonomy can read [7] for more details.

Price et. al. in [6] and [5] present a complete and multi-dimensional taxonomy of SV. This work describes SV system using six characteristics:

Scope: describes the general SV system features. For example, if the visualization is made for one example of for some kind of system, kind of programs, concurrences, etc. These features are explained by sub-characteristics mentioned in [6] and [5].

Content: makes references to the information proportionate by the visualization. For example, if it is used to visualize a program or algorithm, code or data, etc.

Form: characterizes the elements used in the visualization. For example, medium, graphical elements, colors, views, etc. One precise information about this topic is showed in [6] and [5].

Method: is concerned with the strategies used to specify the visualization. For example, fixed, customizable, specification style, etc. For more information see [6] and [5].

Interaction: delineates the techniques used to interact and control the visualization. For example, navigation, simplify the information, temporal control mapping, etc. See [6] and [5] for more details.

Effectiveness: gives the lineaments for assessing the visualization quality. This feature consider the following criteria: *Appropriateness and Clarity*, *Experimental Evaluation*, *Production Use* (see [6] and [5]).

Storey et. al. in [11] propose a classification, based in Price's taxonomy, but oriented to awareness of human activities. In this work, the authors present five main characteristics:

Intent: describes the visualization purpose through: *Role*, *Time*, *Cognitive Support*.

Information: makes reference to the information sources used for the tools. This item is presented delineating the following subcategory: *Change Management*, *Program Code*, *Documentation*, *Informal Communication*, *Derived Data*.

Presentation: describes how the information is presented the final user. It has the following divisions: *Kinds of views*, *Techniques*.

Interaction: characterizes the interaction facilities provided by the tool. It presents the following options: *Batch/Live*, *Customization*, *Query Mechanism*, *View Navigation*.

Effectiveness: explains the likelihood of implementing the tool. It is concerned with: *Status*, *Cost*, *Evaluation*.

The reader interesting in knowing the full details about each subcategories must read [11].

3. INCOMPLETENESS OF CURRENT TAXONOMIES

In subsection 2, the main Software Visualization System (SVS) taxonomies were presented. In general terms, these taxonomies describe the principal SVS characteristics.
The Program Comprehension process is based on cognitive models and when we want to comprehend a program we have two domains of knowledge: *Program Domain* and *Problem Domain*. The *Program Domain* is concerned with the technological issues like the programming language (statements, functions, modules) and how is the program executed to produce an output; while the *Problem Domain* is concerned with the effect of the program execution (the final result produced and the impact at the level of the problem to be solved).

The major drawback with current taxonomies is that *they are not based on Program Comprehension conception* because they only consider the Program Domain. For this reason, they lost some substantial aspects that determine the SVS quality.

Several authors declare that:

> *PC is simplified if both Problem and Program Domain are represented and interconnected.*

This sentence is the starting point for affirming that the current proposals forget some essential characteristics that a PC-oriented SVS must have. The SV research made in [2] shows that the SVS taxonomies describe very well the Program Domain, but not the Problem Domain, neither the Relation between Problem and Program domain. Questions as *How the Problem Domain can be characterized?*, *Which is the best way for describing the relation between Problem and Program Domains?*, *Why the Cognitive Factors are not mentioned in these taxonomies?*, *Are the SVS characteristics well organized?*, etc. motivated the elaboration of a SVS classification aimed at appropriately characterizing this kind of system.

4. NEW CRITERIA FOR IMPROVING THE ASSESSMENT OF PC-SVS

In this section some criteria for filling the gap presented by the current taxonomies are described. As was shown in section 3, they are concerned with the visualization of the Problem Domain and its relation with the Program Domain. The following subsections explain each one of them.

4.1 Problem Domain Visualization

Problem Domain is a characteristic absent in all, or almost all, SVS classifications. This aspect is relevant because it distinguishes the traditional SVS from the PC-oriented SVS, with explicit PC concerns. Problem Domain can be described through the following main categories: *Scope*, *Specification Method*, *Kind of Creation*, *Abstraction Level*, *Content*, *Interface* and *Cognitive Models*. The reader can observe that most of the criteria above are the same as those for Program Domain. However, their means are significantly different. This affirmation will be sustained in the following paragraphs.

Scope: is used to specify the Problem Domain characteristics will be visualized. In this case, it is possible to distinguish the following categories:

Stimulus/Response: the system is shown as a black box only its inputs and outpus are detailed. This task can be made for one example or for all the system.

Concepts/Relations: approaches such as conceptual maps described by Novak can be used to represent the Problem Domain.

Subsystem/Relations: if it is possible to identify some high level components then the relation between them is an attractive strategy to visualize the system behavior.

Behavior: refers to the system at execution time.

Specification Method: to specify the Problem Domain is a hard task. Yet more problematic, to find a specification easy of linking with the Program Domain components is a problem unsolved. This criterion is concerned with the analysis of several approaches to overcome this problem. The main idea is to assess:

1. The specification readability level.
2. The visualization type in sense that a more conceptual visualization turns smaller the gap between domains.
3. The standardization level of the specifications in sense that a standard specification turns easier the integration with other Software Engineering Projects.

Having present the topics described above, the following kind of specification have been found:

Ad-hoc: the specification method was created by the user or there is not method.

Rigorous: approaches with well defined procedures and notations are in this group. They are not considered formal because not all the mechanism have a mathematical demonstration. An example of this category is presented by the UML diagrams.

Formal: specifications made with formal languages or mathematical models are suitable to describe the problem characteristics. Nevertheless, they have the problem of making difficult the interconnection with the program components.

Kind of Creation: makes reference to the strategy used to create the Problem Domain visualization. It is subdivided in manual, semi-automatic and automatic.

Abstraction Level: expounds the level of detail employed to show the Problem Domain characteristics. It has the following subcategories: *Direct*, *Structural*, *Synthesized*.

Interface: it can be characterized through the following aspects:

Kind of interface: this item refers to the type of artifacts employed to show the information, they can be: *Textual*, *Graphical* or *Hybrid*.

Type of interaction: it is concerned with the strategy used to implement the human-computer interaction. Two interaction sorts are distinguished: *Classical* and *Innovative*. The first is associated with the traditional mechanism employed to simplify the interaction. For example: controls through keyboard or mouse, images, etc. The second is aimed at covering news interaction proposals.

Vocabulary: the visualization can be based in different vocabularies such as: *Textual*, *Iconic* or *Hybrid*. The important point is that the lexicon used must be appropriated for the problem under study and totally interpreted by the user. The first criterion is application dependent. The second allows to analyze if it is correctly defined studying the mapping between the visual objects and their means. In this context three kind of mapping are established:

No mapping: there are not explicit association between the visual artifacts and their means.

Partial Mapping: some visual artifacts have an explicit means.

Full Mapping: all the artifacts have defined a mean.

obviously the third kind of mapping is preferred because diminish the cognitive overhead.

Cognitive Models: in the criteria presented above, it is important consider other characteristic, frequently forgotten in the current classifications: *Cognitive Factors*. In this context, it was detected that there are useful criteria to evaluate PC tools [13] [9] [10] [8]. However some aspects are not considered. They are mentioned below:

CM Components: the CMs are composed by four main elements: *Internal and External Knowledge*, *Mental Model*, *Assimilation Process* and a *Dangling Unit*. Usually these elements are hidden in SVS and they are not used to help the user in the comprehension process. So, important conceptual information can be lost. A strategy to represent the internal knowledge must be used and a complete SVS taxonomy must consider if the tools contain representations for these four CM elements.

Learning Strategies: other important criterion is concerned with the learning strategies implanted inside SVS. In this ambit, three well-known approaches must be considered: *Top-down*,

Bottom-up and *Hybrid*. They are explained in [2] and a deeper analysis, from the point of view of exploration tools, of these criteria can be seen in [9].

Figure 2 shows how the criteria described above are organized.

4.2 Visualization of the Interconnection Between Problem and Program Domains

Another interesting characteristic to visualize is the relation between both Problem and Program Domains. In this context, the main claim is concerned with the visualization of the program components connected with the specific output components. In other words, the visualization must show which are the program elements used to build each part of the system output. The reader can observe that some Problem Domain criteria, such as Specification Method, Kind of Creation, Abstraction Level, etc. can be used to describe this relation. Nevertheless, there are some specific properties that deserve some discussion. In first place, the category Scope needs to be redefined because the relation is better explained using the following criteria:

Example: makes references to the kind of visualization that only exhibit a specific example. For instance, to visualize the functions employed for a particular system execution.

Partial: refers to the possibility of showing, for some or all Problem Domain components, some of the program elements related with them or vice versa. For example, if the program under study builds a graph then a particular partial visualization could be constructed to depict the functions used to build a graph, nodes and arcs. To be clearer, this kind of visualization links each Problem Domain component with the specific functions used to build it.

Total: the goal of this visualization is to show for all Problem Domain components all the Program Domain elements used.

Taking as example one system that builds graphs, one possible result of this kind of visualization is:

- for each graph the functions and data used to build it.
- for the node and arc the functions and data directly related with each one of them.

Another interesting characteristic to observe is how the strategy to interconnect domains gather the relation. In this context, two kind of approaches can be identified: *Alive* or *Post Mortem*. In [2] two systems are described. The first one, SVS, follows a strategy that uses the *Alive* approach. In the second one, BORS, the relation is established after the program execution; actually, BORS uses *Post Mortem* approach to interconnect both domains. Finally, it is relevant to indicate if the visualization displays: *Code*, *Data* or *Both*.

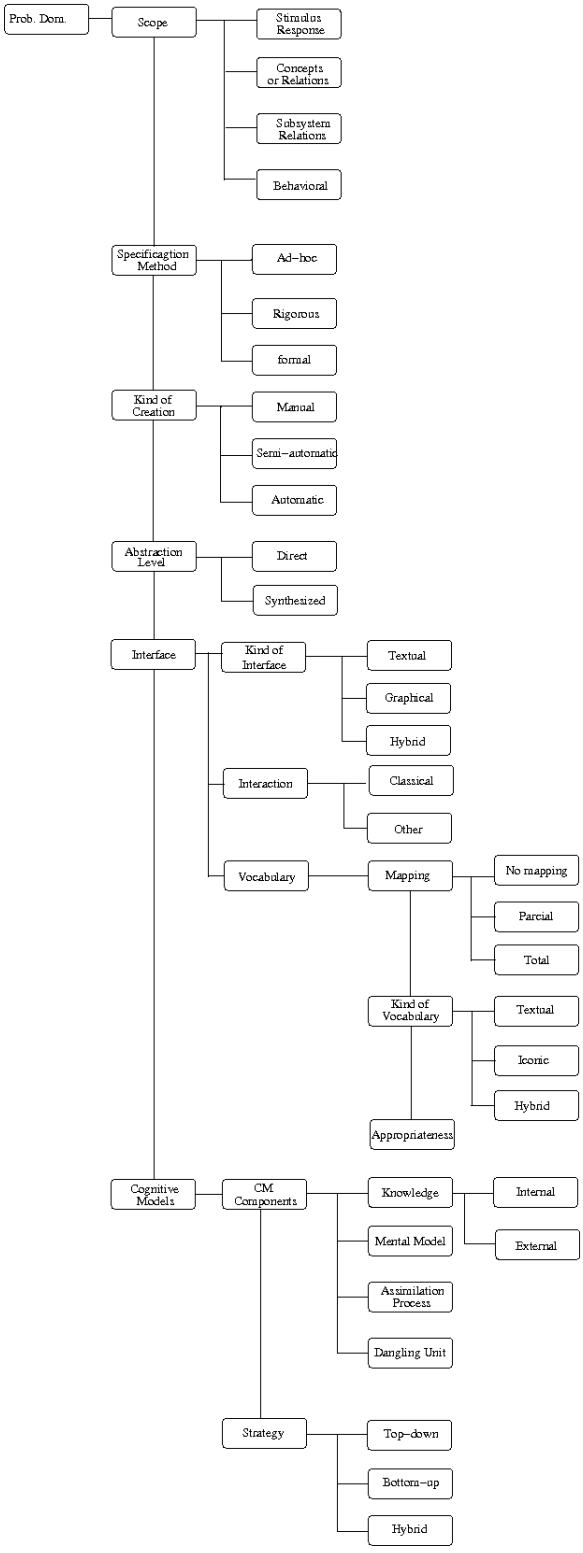

Figure 2. Problem Domain Criteria

5 CONCLUSION

The assessment of PC tools is not an easy task, because several criteria for analyzing each subsystems must be employed. Particularly, the evaluation of PC-oriented Software Visualization subsystem faces some problems because effective criteria to classify *the visualization of Problem and Program Domains and the relation between them* are missing in the current taxonomies. In this paper, the problem is overcome through the definition of several criteria for characterizing these important PC-SVS peculiarities. In this context, the Problem Domain is described using the following criteria: *Scope*, *Specification Method*, *Kind of Creation*, *Abstraction Level*, *Interface*, *Cognitive Models*. The interconnection between Problem and Program Domains is characterized using the same criteria used for the Problem Domain but the criterion Scope is redefined in order to explain clearly the relation between both domains.

This work brings a relevant contribution to the Program Comprehension area because: i) The combination of the criteria for evaluating the Program Domain and the criteria for assessing the Problem Domain and its relation with the program components make possible to obtain a more realistic evaluation of PC-SVS and PC tools; ii) Our criteria give guidelines for building useful PC-SVS and PC tools.

It is important to remark that the current PC tools do not have strategies for interconnecting the problem and program domains (see [2] for more details). For this reason it was not possible to use benchmarks in order to compare systems or approaches. In the context of PCVIA (Program Comprehension by Visual Inspection and Animation) project [12] there is a tool under development, named PICS, whose goal is to make easier the comprehension of C programs [2]. PICS has implemented several strategies for interconnecting the Problem and Program Domains based on dynamic and static information extracted from the system. These strategies help the programmer to only concentrate on the system aspect that he wants to study. After the identification of functions carried out through the techniques of interconnecting domains, the user is encouraged to inspect the functions reported by the domain interconnection strategies. In order to simplify this task, PICS provides a set of views automatically generated such as: *Function Graph*, *Module Graph*, *Static Information Visualizer*, *Function Execution Tree*, *Source Code Visualizer*, *Object Code Visualizer*, *Runtime Function Visualizer*, etc. These views have navigation functions, they allow the user to access to whatever system components. All the PICS' views mentioned in the precedent paragraph, and the approaches used for visualizing the interconnection between the Problem and Program Domains were designed and are being implemented to prove that the guidelines presented along the paper can be effectively used to build and evaluate PC tools.

References

[1] T. Ball and SG Eick. Software visualization in the large. *Computer*, 29(4):33–43, 1996.

[2] Mario Berón, Pedro Rangel Henriques, Maria J. Varanda Pereira, and Roberto Uzal. Program inspection to interconnect behavioral and operational views for program comprehension. In *York Doctoral Symposium*. University of York, UK, Oct 2007.

[3] M. H. Brown. Perspectives on algorithm animation. In *CHI '88: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 33–38, New York, NY, USA, 1988. ACM Press.

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

[5] 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.

[6] BA Price, IS Small, and RM Baecker. A taxonomy of software visualization. *System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on*, 2, 1992.

[7] G. C. Roman and K. C. Cox. A taxonomy of program visualization systems. *Computer*, 26(12):11–24, 1993.

[8] 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.

[9] Margaret A. Storey. *A Cognitive Framework for Describing and Evaluating Software Exploration Tools*. PhD thesis, Simon Fraser University, 1998.

[10] Margaret-Anne Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Control*, 14(3):187–208, 2006.

[11] Margaret-Anne D. Storey, Davor Čubranić, and Daniel M. German. On the use of visualization to support awareness of human activities in software development: a survey and a framework. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 193–202, New York, NY, USA, 2005. ACM Press.

[12] Maria J. Varanda and Pedro Henriques. Program comprehension by visual inspection and animation. http://wiki.di.uminho.pt/wiki/bin/view/PCVIA.

[13] Andrew Walenstein. Theory-based analysis of cognitive support in software comprehension tools. In *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*, page 75, Washington, DC, USA, 2002. IEEE Computer Society.