CODE INSPECTION APPROACHES FOR PROGRAM VISUALIZATION

Daniela da CRUZ*, Mario BÉRON**, Pedro Rangel HENRIQUES*, Maria João Varanda PEREIRA***
*Universidade do Minho - Departamento de Informática, Campus de Gualtar, 4715-057, Braga, Portugal,
E-mail: {danieladacruz,prh}@di.uminho.pt
**Universidade Nacional de San Luis - Departamento de Informática, San Luis, Argentina, E-mail: mberon@unsl.edu.ar
***Instituto Politécnico de Bragança, Campus de Sta. Apolónia, Apartado 134 - 5301-857, Bragança, Portugal,
E-mail: mjoao@ipb.pt

ABSTRACT

The aim of this paper is to show the approaches involved in the implementation of two tools of PCVIA project that can be used for Program Comprehension. Both tools use known compiler techniques to inspect code in order to visualize and understand programs' execution but one of them modifies the source code and the other not.

In the non-invasive approach, we convert the source program into an internal decorated (or attributed) abstract syntax tree and then we visualize the structure traversing it, and applying visualization rules at each node according to a pre-defined rule-base. No changes are made in the source code, and the execution is simulated.

In the invasive approach, we traverse the source program and instrument it with inspection functions. Those functions, also known as inspectors, provide information about the function-call flow and data usage at runtime (during the actual program execution). This information is collected and gathered in an information repository that is then displayed in a suitable form for navigation.

These two different approaches are used respectively by Alma (generic program animation system) and CEAR (C Rooting Algorithm Visualization tool). For each tool several examples of visualization are shown in order to discuss the information that is included in the visualizations, visualization types and the use of Program Animation for Program Comprehension.

Keywords: *Code inspection, Code analysis, Visualization, Program Comprehension.*

1. INTRODUCTION

PCVIA, Program Comprehension by Visual Inspection and Animation, is a research project looking for techniques and tools to help the software engineer in the analysis and comprehension of (traditional or web-oriented) computer applications in order to maintain, reuse, and re-engineer software systems.

To build up a Program Comprehension environment we need tools to cope with the overall system, identifying its components (program and data files) and their relationships; complementary to those, other kind of tools is also necessary in order to explore individual components. These tools—that are our concern along the paper—deal with programs instead of applications (set of programs), and their purpose is to extract and display static or dynamic data about a program to help the analyst to understand its structure and behavior.

Depending on the actual program facet we want to explore, different approaches to inspection and visualization can be followed. We are experiencing that in the context of PCVIA. On one hand, we want to develop a tool (Alma [1]) that does not modify the source program and uses abstract interpretation techniques, aiming at an easy and systematic adaptation to cope with different programming languages (see section 2). On the other hand we are working on a tool specific for the C programming language (CEAR [2]) that modifies the source code to be able to collect dynamic information at runtime (see section 3). In this paper we are going to discuss these two approaches and the generated visualizations.

Project goals, team, technical descriptions, and achievements can be found at the URL `http://www.di.uminho.pt/~gepl/PCVIA`.

1.1. Related work

During our study of the state of the art we found several software handling tools: classic program comprehension tools; software visualization tools that can be also seen as program understanding tools; development environments that use visual or textual representation to help the programmer; tools that are used just in some specific tasks of software maintenance; graph visualization tools that can be used for some program visualization tasks; and teaching tools.

Almost all of those tools were constructed for some specific language and are totally dependent of that language. Most of them use parsers automatically generated, and compiler techniques to process information. Those parsers transform the source code in order to instrument it with inspection functions or special data types. They can also build an internal representation of the program. This representation can be then systematically used to generate explanations, statistics, structured information, visualization or animation of programs.

Some examples of tools that create and use internal representation as the core: Moose [3], CANTO [4] or Bauhaus [5]. In Moose (a reengineering tool) the information is transformed from the source code into a source code model. Moose supports multiple languages via the FAMIX languages independent meta-model. In most cases a parser is constructed to directly extract information to generate the appropriate model. The CANTO environment has a front-end (for C) which parses the source code and creates a intermediate file with structural, flow and pointer information. Then a flow analysis tool is used to compute flow analysis on the code. The front-end also creates an abstract syntax tree that is used by an architectural recovery tool which recognizes architectural patterns. Bauhaus sys-

tem has tools that use compiler techniques which produce rich syntactic and semantic information creating a low level representation of programs. Alma follows this kind of approach. Alma uses a parser to construct an internal representation of the program and then uses a set of pattern based rules to inspect the code.

TKSee [6] or SeeSoft [7] are tools that collect statistical information about the source program and then this information is shown in a structured way without changing the source code. TKSee search the whole system for files, routines or identifiers whose name or lines match a certain pattern and build hierarchies to organize the information. SeeSoft also extracts statistical information from a variety of sources (like version control systems, programmer and purpose of the code and static and dynamic analysis) and shows the information using different colored lines. Our second approach goes in that direction.

Like CEAR, some tools do code instrumentation. ISVis [8] does instrumentation of the source code to track interesting events and analyzes the event traces in several scenarios using graphical views. PAVI [8] uses tags to annotate source code to specify target variables or pointers to be visualized as three-dimensional objects and to define scopes and styles for visualization.

2. NON-INVASIVE APPROACH

In this section, we discuss the approach to program inspection and visualization followed in the context of Alma, one of the PCVIA developed tools. Although not a classic tool for program comprehension, we believe that it can truly contribute for it, at the program understanding level (as argued in the Introduction).

Alma is a system for program visualization and animation. The purpose of such a family of tools is to help the programmer to inspect *data* and *control flow* for a given program (*static view* of the algorithm realized by the program — **visualization**), and to understand its *behavior* (*dynamic view* of the algorithm — **animation**).

Alma, as a generic tool for program visualization and animation, is based on the internal representation of the input program in order to avoid any kind of annotation of the source code (with visual types or statements), and to be able to cope with different programming languages.

To fulfill such requirements, we had been inspired in the classic structure of a compiler and we conceived an architecture that separates the source program recognition from its animation, using a decorated abstract syntax tree (DAST) as internal representation (see Fig. 1).

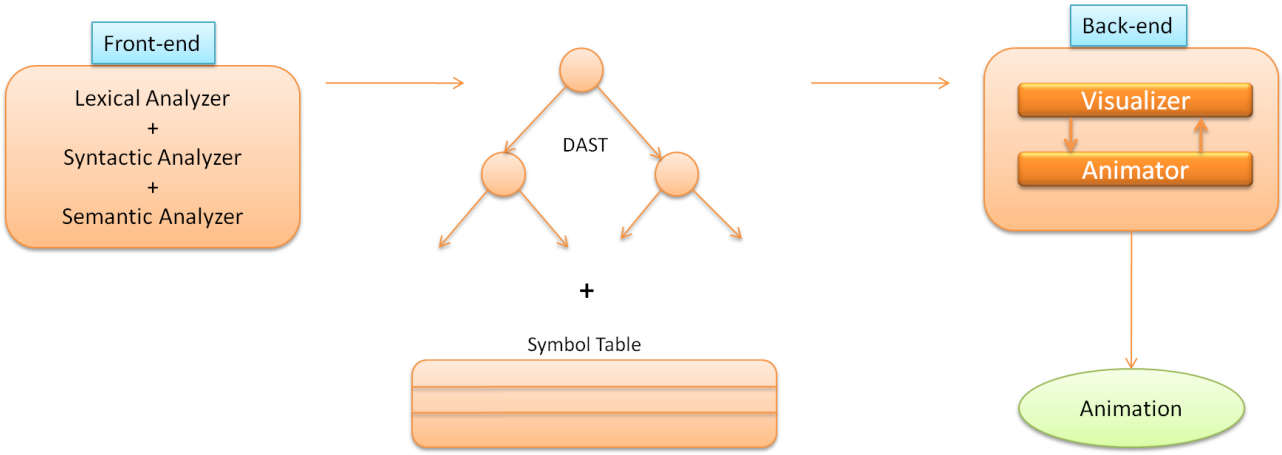

Fig. 1 Architecture of Alma system

Alma is implemented in Java, using and reusing the compiler generator system LISA [9], as specified and described in [10–12].

Alma system has a front-end (*front-end*) specific for each language and a generic back-end (*back-end*). It uses a decorated abstract syntax tree (DAST) for the internal representation of the program's meaning; it is the connection between the *front-end* and the *back-end*. Using a DAST as an internal representation, we isolate all the source language dependencies in the *front-end*, while keeping the generic animation engine in the *back-end*. The DAST is built using a set of pattern rules as will be described in the next subsection.

Applying specific rewrite rules (which are used according to the pattern-tree found in the DAST) to the *execution tree*, we obtain a description of the different program states, simulating its execution.

A *Tree-Walker Visualizer*, traversing the *execution tree* and applying visual rules create a representation for the nodes generating a visualization of the *program tree* in that moment. Then the DAST is rewritten (to obtain the next internal state), and redrawn, generating a new visualization which reflects the new state of the program.

This approach, using a DAST as internal representation and a set of pattern rules allows us to easily construct different abstraction levels of the same program from the operational view till the behavioral view. For that, it suffices to associate a new set of rewrite and visualize rules to the DAST patterns. This system is based on the concepts involved in a program and not directly in the source code.

In the remainder of the section, we discuss *the information we need to extract* from the source program, *how do*

we do it, the format under which this *information is represented*, and *how is it visualized* to help the user to understand the program.

2.1. Patterns: the information to extract

In contrast to the most common animators, we are looking forward to building a more generic system, in the sense that it can animate any algorithm and that it can be easily adapted to work with different programming languages. To go on that direction, it is essential to find out a set of program patterns that we know how to deal with (display and rewrite). This is, we need to discover the information, common to a set of programming languages, that describes the structure and semantics of the program, and that we know how to store and to display (we intend to create a set of rules to systematically visualize those patterns).

An analysis of the programming languages, belonging to the universe we want to deal with, allow us to state that all of them have in common entities, like: *literal values* and *variables* (atomic or structured), *assignments*, *loops* and *conditional statements*, *write/read statements*, *functions/procedures*.

Identified the common entities, we must find a way to describe them at an abstract level, in order to establish a generic set of rules to handle them in a language independent way. The solution is a set of *elementary programming patterns*. These patterns capture the abstract syntax of each entity (value or operation) to preserve and keep, via attributes, the necessary information to express its static semantics.

2.2. Program representation: Pattern Tree

After deciding the information we need to extract from a source language, we should define a way to represent it. The internal representation chosen to store those patterns is a DAST that describes the structure of the program we intend to represent and visualize, being separated from any particularity of a source language. The DAST is specified by an abstract grammar, independent of concrete source language, and is intended to represent the program in each moment.

Given a source program, one possible representation for it is the concrete syntax tree, shown in Fig. 2.

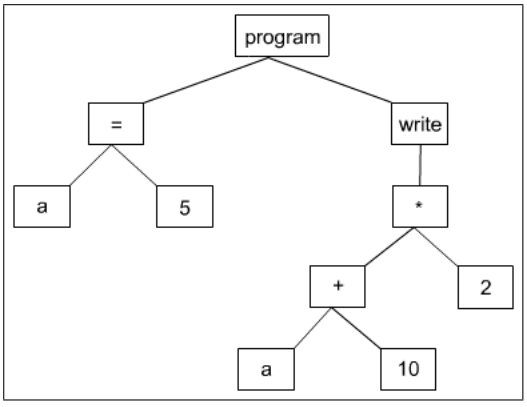

Fig. 2 Syntax Tree representation of the program

However, Fig. 3 shows the pattern tree (DAST) chosen in our approach. Each node in a concrete DAST will match and instantiate a specific pattern. These tree nodes are implemented with attributes, whose values are obtained during the information extraction phase, and describe the characteristics of the source program to preserve.

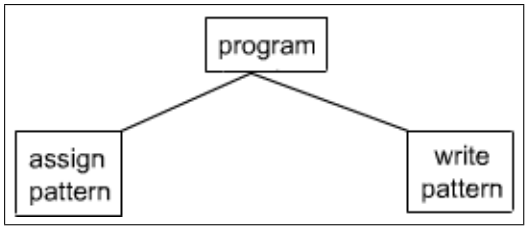

Fig. 3 Pattern Tree representation of the program

2.3. Pattern extraction and pattern visualization

To extract information from a concrete source program its is necessary to parse it. This operation will be carried out by a *front-end* built specifically for the concrete language under consideration. The *front-end* will be in charged of identifying the source language constructs and map them to the DAST patterns. To develop such a *front-end* we will use a compiler generator based on an attribute grammar.

As we have decided which information we need to extract, the way to do that extraction, and how to represent it, the visualization schema comes out as a natural consequence of the previous decisions.

As we have a pattern tree as the intermediate representation between the *front-end* and the *back-end* (the DAST), that tree will be used straight forward to build a visual representation for the source program. Each pattern will be extended with one more attribute: `vr`, that contains the corresponding *visualization rule*. Thus, the visualization of a program is obtained just making a *top-down* traversal over the DAST, applying that rule to each node instance. The first traversal produces an overall picture of the program before execution; successive traversals depict the program state after the execution of each statement. Fig. 4 and 5 show the visualization of a program. The animation of a program will be attained by multiple *top-down* traversals to the DAST, until program is totally rewritten.

As can be seen in Fig. 4 and 5, Alma's interface is split in four windows—3 main windows and 1 with the buttons for navigation—with the following content:

- The identifier table (on the upper left corner);
- The source text of the program to be animated (on the bottom left corner);
- The program tree (to the center, occupying most of the display);
- The interaction buttons, on the bottom of the screen, that allow the user to control the system. Using `back` and `forward` controls the user will be able to navigate through the program animation, step by step.

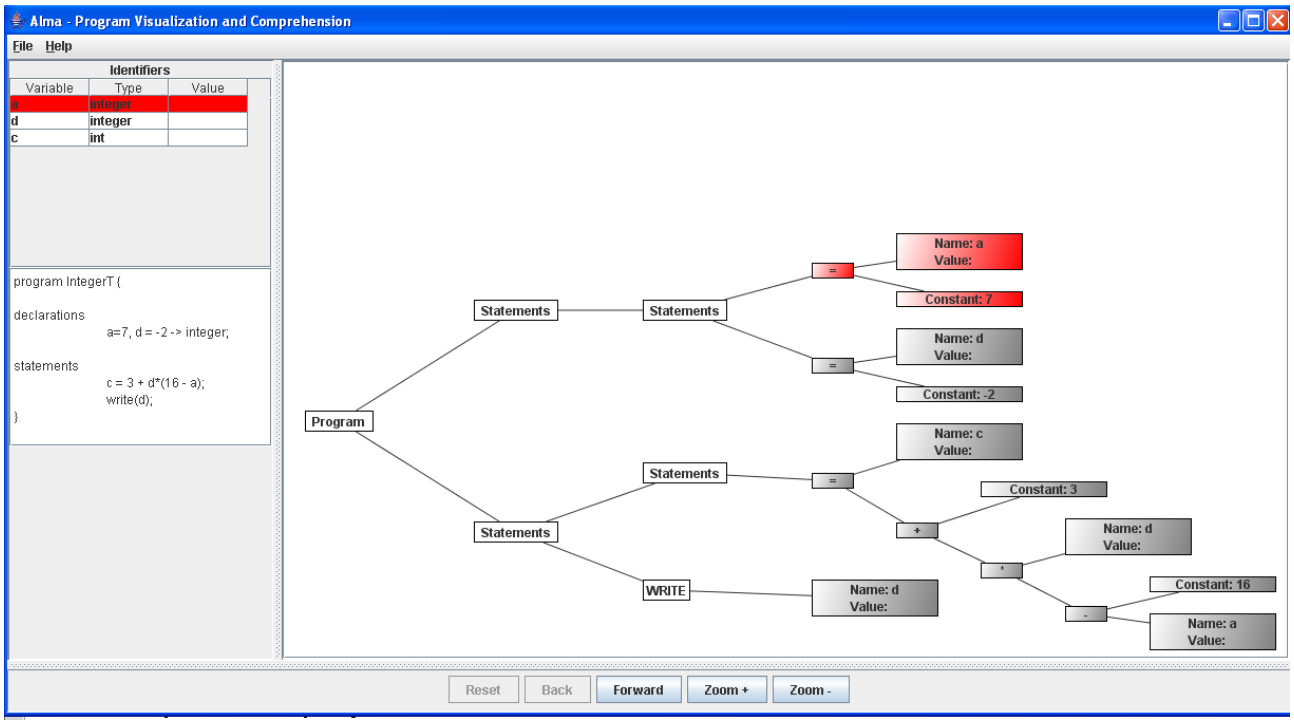

Fig. 4 Global visualization of the source program

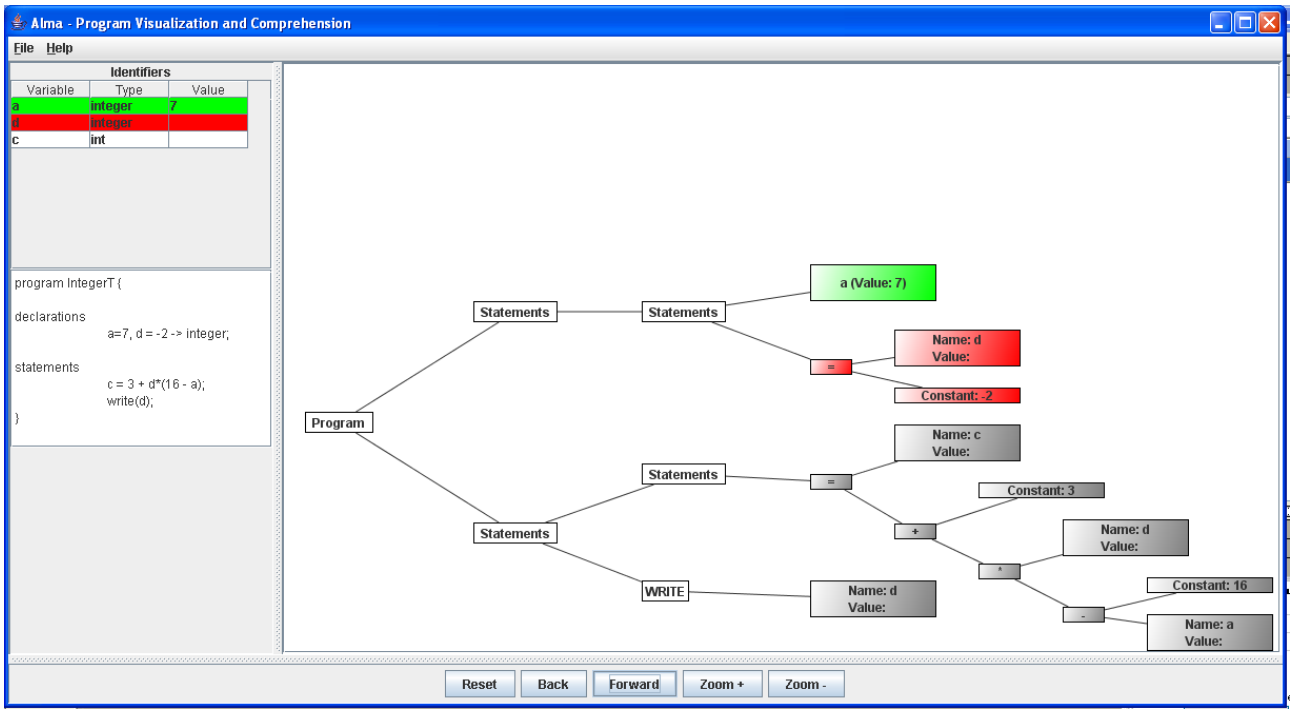

Fig. 5 Program state after one step of animation

Colors are used to make easier to follow the animation of a program: Red color indicates the identifier / subtree that will be evaluated in the next execution step (the execution of the program-statement also colored in red); Green color indicates the new status of the program (subtree /identifier) after the last evaluation.

To conclude, Alma allowed us to prove the usability of our non-invasive approach to construct program comprehension tools. Several *front-end*'s were constructed in order to animate programs in different programming languages.

3. INVASIVE APPROACH

In this section we discuss a strategy that aims at extracting information from programs using code instrumentation [13, 14]. To implement this technique we insert inspection functions (or *inspectors*) in strategic places of a program to capture its execution flow. The information extracted along this inspection will be used to show different views to help understanding program behavior.

This approach was applied to CEAR, a C Rooting Algorithm Visualization Tool, that will be described at the end of this section. To apply our code instrumentation technique we need to build a parser for the source language and select the places where the new code will be inserted. Besides capturing execution flow, we use the recovered static information to build different program views, corresponding to different abstraction levels.

Our approach consists of three tasks clearly defined:

- Code instrumentation;
- Trace summarization;
- Visualization.

In the next subsections we describe these three different tasks.

3.1. Code Instrumentation

To define a strategy to annotate the source code we have to know: *which information we need to extract*; and *what are the strategic points in the source code*.

To answer these questions we conceptualize a system as a state machine (SM). The input values represent the initial state and the final state can be represented by the variable values after execution. The intermediate states are represented by the variable values reached during the system execution. The transition between states is carried out through the system functions (build in or defined by the user). We believe that is useful to show only the states used by the system to build its output. In our case, we just keep track of global variables and those defined in the main function.

We use as inspection units the program functions because they produce system state transitions and it is useful to know the effects of their execution.

So, we think that the beginning and the end of the functions are convenient *check points*, i.e., are the right places to locate the inspectors.

To implement this strategy we need to build a parser for the source language extended with semantic actions. These actions insert into the program new statements that will allow to trace the state and the transitions. These statements are inspectors that show the global variables and the name of the functions called by the program. Table 1(a) shows a C function pattern and Table 1(b) illustrates the annotated version of the same function.

The problem with this approach is that the recovered information is huge, once the functions can contain loops and inside loops we can have other functions calls. Some special functions are then used to control the number of inspected iterations and a stack is used to store historical information.

Unfortunately, the recovered information is yet huge. For this reason we need to apply other strategy to reduce it. One possibility is to inspect smaller parts at each time. In other words, the programmer could want to see only some aspects of the system. For this reason we create and implement one strategy aimed at tracing summarization that uses the dynamic information recovered by our annotation schema.

3.2. Program Trace Summarization

Trace summarization [15] is a synthesis of the program flow just containing the execution main points. In this approach, we remove the details doing the selection and generalization of the program main aspects. An approach for carrying this task consists in the instrumentation the program iteration.

In this context, we distinguish three main points (indicated in Table 2 on the right side). In 1 and 3, we insert a control function *showFunction*(*value*). In 2, we insert another control function called *dec*(). The function *showFunction*(*value*) enables the inspector functions to show the called functions during the iteration. The parameter *value* and the *dec* function will be described after explaining the control scheme. Besides these functions we also need a stack. It is so because the program can have nested iteration sentences and the user may want to see the functions invoked during the iterations a given number of times. To better understand this situation, please consider the code segment of Table 3.a and assume that the user wants to see, once or several times, the functions invoked during the iteration. The code transformation can be seen in Table 3.b.

```
int f(int x, int y)
       {float z, y;
        /*more declarations*/
        ..........
        /*actions*/
        return value
}
```

(a)

```
int f(int x, int y)
       {float z,y;
        /*more declarations*/
        INPUT_FUNCTION("f")
        ...........
        /*actions*/
        OUTPUT_INSPECTOR("f");
        return value;
}
```

(b)

Table 1 Insertion of Inspectors (inspection functions)

```
for(init.; cond.; action)
   actions;
```

```
{
 1
 for(init.; cond.; action)
  {actions;
   2;
  }
 3
}
```

Table 2 Iteration Control: Check Points

```
for(i=0;i<10;i++)
 {for(j=0;j<10;j++) f(j);
  g(j);
 }
```

(a)

```
{showFunction(value);
 for(i=0;i<10;i++)
  {
   {showFunction(value);
    for(j=0;j<10;j++)
     {f(j);
      dec(value);
     }
    showFunction(value);
   }
   g(i);
   dec(value);
  }
 showFunction(value);
}
```

(b)

Table 3 Iteration Control Scheme: Example

The parameter *value* is used to indicate the number of times that the functions within the loop will be showed. The function dec decrements the parameter *value* each time that it is executed. When the parameter *value* is zero the inspection function does not show the function name.

This strategy uses an *fe-Tree* (**F**unction **E**xecution **T**ree) to inspect only the important/interesting aspects. An *fe-Tree* is a tree with arity *r* where: The root is the first function executed by the system (normally called main); for each node (function) *n*, its children are the functions called directly by *n* at execution time.

With the *fe-Tree* we can explain any function in the system. Furthermore, we can know the different context where the functions were invoked. For this reason, we can use the *fe-Tree* to inspect only the aspects chosen by the user.

Fig. 6 shows an example that illustrates the procedure we followed to describe just a partial aspect. On the left is the hypothetical system *fe-Tree* and on the right is the list that contains the functions selected by the user. In this figure, the reader can see the context and explanation for each function.

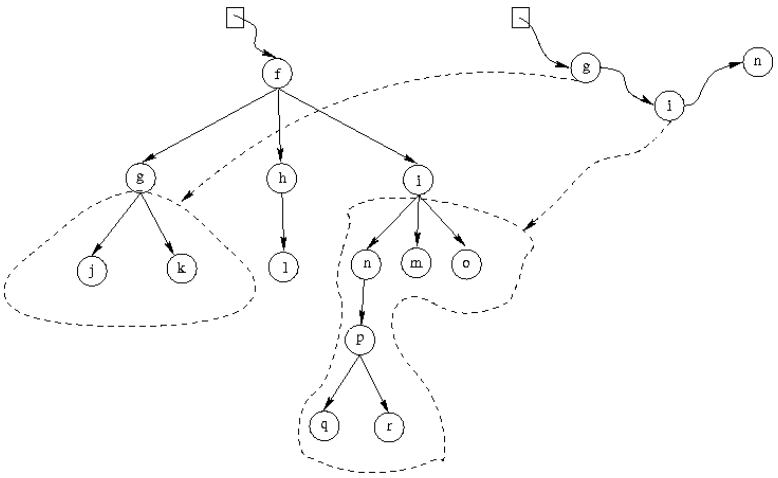

Fig. 6 Strategy to explain system aspects

3.3. Visualization of the Information

In this section we present the approach used to visualize [16] the static and dynamic information recovered by the application of our code annotation strategy. We think that it is a good idea to present the information in different abstraction levels. We distinguished the following abstraction levels:

1. Machine — describes the assembly code used to implement the system functions;
2. Program — describes the source code;
3. Function and data used in runtime — symbolize the recovered dynamic information;
4. Function — symbolizes the recovered static information at function level;
5. Module — represents the recovered static information at module level;
6. Behavioral — concerns the system output.

As can be seen in Fig. 7, we conceptualize the first five levels as *Program Domain Levels* and the last level as *Problem Domain Level*. Each level acquires importance depending of the program inspection state. For this reason, and to facilitate this task, we think that an important feature is to allow the navigation between levels.

Levels 1 and 2 are represented naturally by the *assembly and source code*. The third level can be represented by a *function list* or using an *fe-Tree*. We think the *fe-Tree* representation is better because it allows the user to know the relation called-caller clearer. The level 4 and 5 are represented by two graphs: The *Module Communication Graph* (MCG) and *Function Call Graph* (FCG). We intend to display these graphs as *layered directed graphs*. It is because the relation between the different component (functions or modules) is normally hierarchical. Therefore a graph with these characteristics is adequate to represent it.

The visualization system uses an information repository, that contains:

1. Runtime functions: Name, Module, Place;
2. System Module: Name, Directory, Functions and data defined in the module, FCG for this module;
3. System functions: Parameters, Local variables, Module where the function is defined;
4. The MCG;
5. The FCG.

It is possible to relate level 2 and 3 using code instrumentation. The other levels can be related to each other using the information stored in the repository.

Our big challenge was to relate levels 5 and 6, because it requires to *interconnect the program domain with the problem domain*.

To relate the operational (program domain) and behavioral (problem domain) views, we use the program execution flow. Our strategy to capture the execution flow returns us the functions used to build the output (program trace summarization in Sec. 6). On the other hand, we know the problem domain objects (produced by the program), because we can observe the system output.

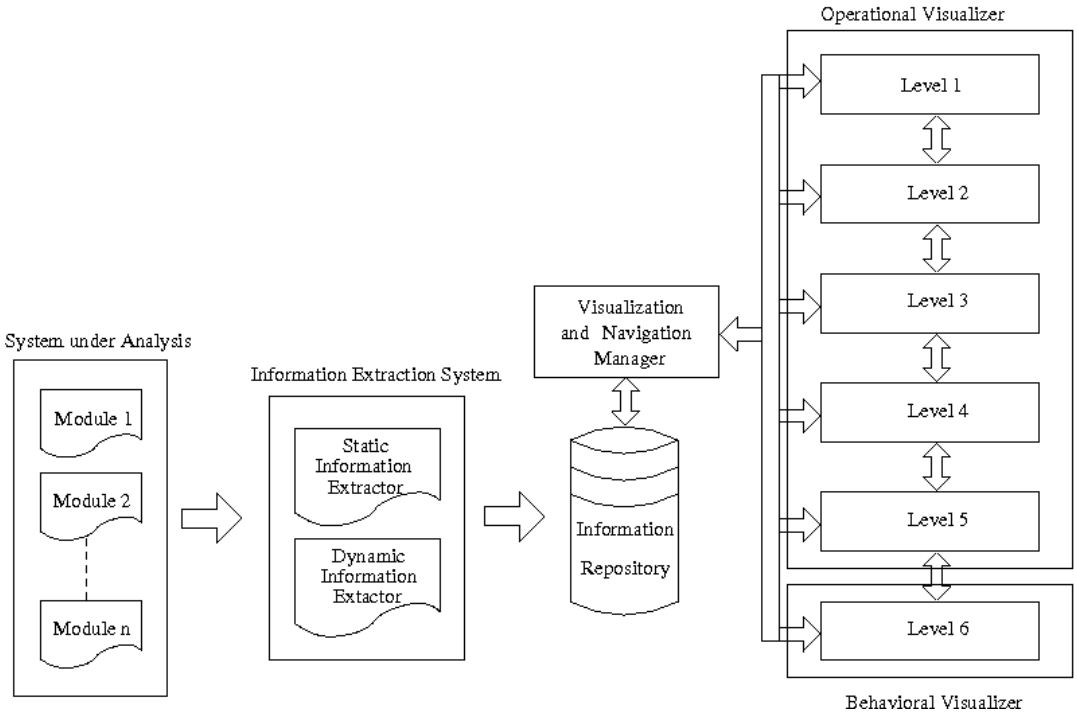

Fig. 7 Program Comprehension Tool Architecture

So, our behavioral-operational relation strategy (known as BORS) has three steps clearly defined:

1. Detect the functions related with each Problem Domain Object;
2. Build a fe-Tree with the function used at runtime;
3. Explain the functions found in step 1 using the tree built in step 2.

The first step is carry out recovering the Abstract Data Type interface. The second step consist in using the runtime information. The third step is implemented applying a breath-first traversal to visit each fe-Tree node. When the name of a visited node match some name of the function selected to be described we report the corresponding subtree.

3.4. The Program Comprehension tool CEAR

As a case study and aiming at testing the applicability of our approach as well as the reduction of the recovered information, we applied our strategies to EAR (**E**valuador de **A**lgoritmos de **R**uteo), an environment to experiment and assess routing algorithms. This tool has two main functionalities: *visualization* of routing schema; and *evaluation* of routing algorithms. The description of these tasks can be red in [17]. EAR has more than 4000 lines of C code that implement algorithms to build planar graph and routing algorithms and metric evaluations.

To carried out our task, the PC tool CEAR was implemented by extending EAR functionalities with: (1) Object and source code and runtime function inspection; (2) MCG and FCG visualization. The first extension (1) implements levels 1, 2, 3 of the visualization architecture, and the second extension (2) implements levels 4 and 5.

The implementation and the usage of CEAR allowed us to determinate the usefulness of the abstraction levels. In this context, we learned that:

1. The MCG is a useful view because allows us to have a clear insight over the system without information overload. FCG presents an important view when the program is small but when it is too big this representation give us few information. For this reason it is better to build the FCGs for each module instead to build it for the complete system. We can say this when observing the *MCG* and *FCG* graph. The first is more condensed and present a clearer view. The second is big and it is very difficult to understand.

2. It is a good idea to integrate the level 6 and 1, 2, 3 in a single window (see Fig. 8) because it facilitates the inspection and debugging.

3. The integrated view allows us to relate problem and program domains, which is indeed a relevant aid to program understanding. As can be seen in Fig. 9, it is possible to display, on the output window, the list of functions responsible for each step of the routing algorithm.

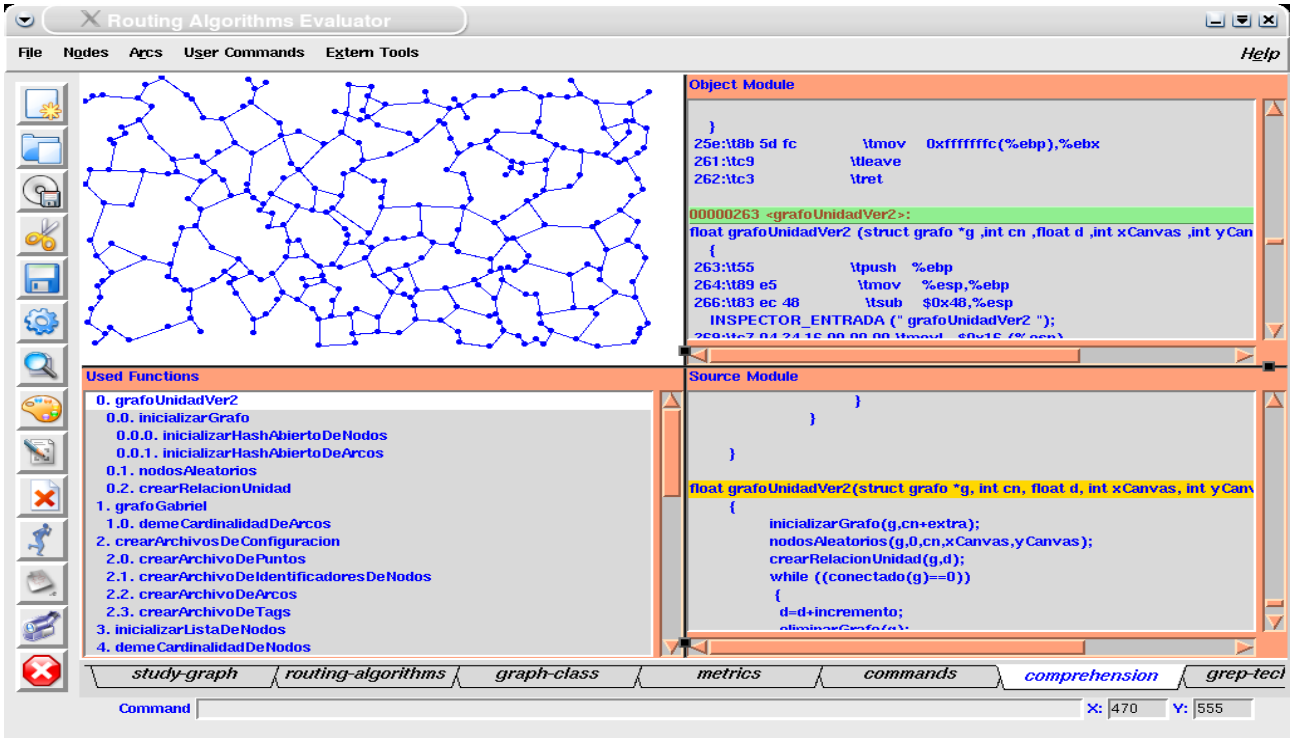

Fig. 8 System views

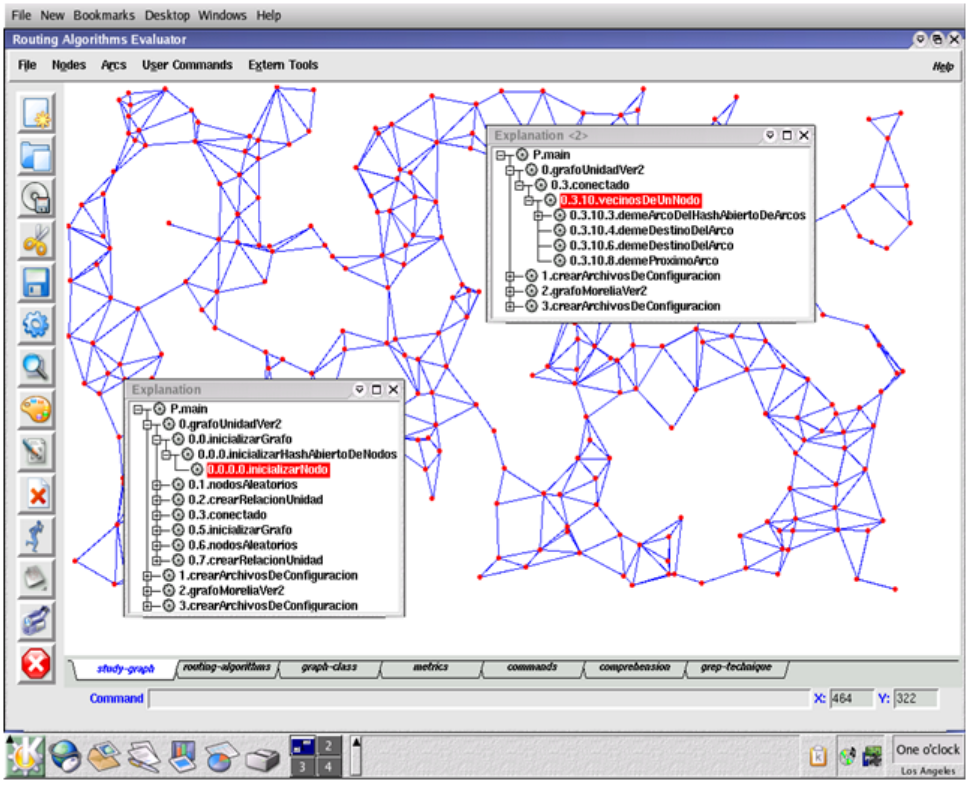

Fig. 9 Behavioral-Operational relation strategy in CEAR

4. CONCLUSION

To help the software engineer to understand the behavior of a given program (in the context of program comprehension environments), it is necessary to extract and collect *static data*—concerned with variables/types declaration and statements structure—and *dynamic data*—concerned with the actual data and control flows.

It was our intention, along the paper, to report on some of the lessons learned during the live research project PCVIA. More specifically in the paper we focussed on the invasive/non-invasive approaches to program analysis and visualization in order to show that for a similar purpose (program understanding) different techniques should be implemented, according to the characteristics of the information we want to exhibit and the interaction we want to provide.

In the first case, the objectives are to show the program structure (the hierarchy of the statements),and t o illustrate the execution flow and how it affects the program state. For that, Alma system shows an animation of an *abstraction* of the program. It does not work directly on the code and it generates visual representations that allow to understand what the program does. In this case, the user does not care about the syntax of the programming language neither with lexical or syntactic errors that the program may have. This approach is totally language independent. The user interaction with the system is not crucial because the system provides an animation of all the program. We just have to parse the source program in order to collect the information that defines its state (values and variables) and to find out its structure. A symbol table and an abstract syntax tree is enough to store this information. The visualization process is then performed by a systematic tree traversal, applying straightforward rules to each tree node, and to each symbol table row. We do not need any more the source program and we are able to give visual details helpful for the user to get easily an *operational view* of it. This approach does not modify the source program, and is relies upon a visualization/animation engine (the Back-End of the tool) that is independent of the source language; thus, tuning the tool to analyze program in different languages is not a hard task.

In the second case, a code instrumentation technique was used to trace program beha An actual flow graph can be built and displayed at different abstraction levels. A specific function can be selected from the sequence of functions called and some querying operations can be offered; for instance, one can see the source code of that function, or its object code. Moreover, we believe that we are able to relate that runtime operational view (at the *program domain* level) with the behavioral view, or output computed by the program (at the problem domain level). This approach, that obviously changes the source program, extracts much more information and enables us to provide another kind of debugging navigation and a richer interaction; however, it is language dependent, and the inspectors' weaver needs to be recoded for a different source language.

At present we are applying the analysis and visualization techniques, so far explored to another domains, namely to XML documents, modeling and domain specific languages. Mainly the non-invasive approach is being considered.

Concerning the first case, we have proposed in [18] a system called eXVisXMLto analyze and visualize XML documents and the underlying DTD or XML-Schema in order to evaluate a set of metrics and allow a qualitative/quantitative study of both. An initial prototype was developed, and a more elaborated one is under development.

We also extended that approach to study UML models (more extended with OCL constraints and sets of tests. A prototype is being developed and a paper was submitted to an international conference.

A master thesis, under development, has as main objective to study the use of our non-invasive approach to visualize the problem domain and map both domain context of DSLs (Domain Specific Languages).

At last, a Ph.D. work is starting to study the adaptation of the referred techniques (usually developed to work on sou

REFERENCES

[1] da Cruz, D., Henriques, P.R., Pereira, M.J.V.: Constructing program animations using a pattern-based approach. ComSIS – Computer Science an Information Systems Journal, Special Issue on Advances in Programming Languages **4**(2) (Dec 2007) 97–114 ISSN: 1820-0214.

[2] Berón, M., Henriques, P.R., Pereira, M.J.V., Uzal, R.: Program inspection to interconnect behavioral and operational view for program comprehension. In: York Doctoral Symposium, 2007, University of York, UK (Oct 2007)

[3] Ducasse, S., Girba, T., Lanza, M.: Moose: an agile reengineering environment. In: ESEC-FSE'05, Lisbon - Portugal (September 2005)

[4] Antoniol, G., Fiutem, R., Lutteri, G., Tonella, P., Zanfei, S., Merlo, E.: Program understanding and maintenance with the canto environment. In: IEEE International Conference on Software Maintenance (ICSM'97), Bari, Italy (October 1997)

[5] Raza, A., Vogel, G., Plodereder, E.: Bauhaus - a tool suite for program analysis and reverse engineering. Lecture Notes in Computer Science **4006/2006** (May 2006) 71–82

[6] Herrera, F.: A usability study of the tksee software exploration tool. Master's thesis, University of Ottava (1999)

[7] Eick, S., Steffen, J., Jr., E.S.: Seesoft - a tool for visualizing line oriented software statistics. IEEE Transactions on Software Engineering **18**(11) (November 1992) 957–968

[8] Jerding, D., Rugeber, S.: Using visualization for architectural localization and extraction. In: Fourth Working Conference on Reverse Engineering (WCRE'97), Amsterdam, The Netherlands (October 1997)

[9] Mernik, M., Lenic, M., Avdicausevic, E., Zumer, V.: Compiler/interpreter generator system LISA. In: IEEE Proceedings of 33rd Hawaii International Conference on System Sciences. (2000)

[10] Varanda, M.J., Henriques, P.: Visualization / animation of programs based on abstract representations and formal mappings. In IEEE, ed.: HCC'01 - 2001 IEEE Symposia on Human-Centric Computing Languages and Environments. (September 2001)

[11] Varanda, M.J., Henriques, P.: Visualization / animation of programs in alma: obtaining different results. In: VMSE2003 - Symposium on Visual and Multimedia Software Engineering (HCC'03), New Zealand. (October 2003)

[12] da Cruz, D., Henriques, P.R., Pereira, M.J.V.: Alma versus ddd. ComSIS – Computer Science an Information Systems Journal, Special Issue on Compilers, Related Technologies and Applications **5**(2) (Dec 2008) 119–136

[13] Zaidman, A., Adams, B., Schutter., K.: Applying dynamic analysis in a legacy context: An industrial experience. In: PCODA: Program Comprehension through Dynamic Analysis. (2005) 6–10

[14] Béron, M., Henriques, P.R., Pereira, M.J.V., Uzal, R., Montejano, G.: A language processing tool for program comprehension. In: CACIC'06 - XII Argentine Congress on Computer Science, Universidad Nacional de San Luis, Argentina. (2006)

[15] Hamou-Lhadj, A.: The concept of trace summarization. In: PCODA: Program Comprehension through Dynamic Analysis. (2005) 38–42

[16] Balmas, F., Werts, H., Chaabane, R.: Ddgraph: a tool to visualize dynamic dependences. In: PCODA: Program Comprehension through Dynamic Analysis. (2005) 22–27

[17] Béron, M., Henriques, P.R., Pereira, M.J.V.: A system for evaluate and understand routing algorithms. In: Interacção'06-Conferência Nacional em Interacção Pessoa-Máquina, Universidade do Minho. (2006)

[18] D. da Cruz, P. R. Henriques, Pereira, M.J.V.: Exploring and visualizing the "alma" of XML documents. In *XATA - XML: Aplicações e Tecnologias Associadas, Portalegre - Portugal*, Fev 2008.

Received April 24, 2009, accepted July 21, 2009

BIOGRAPHIES

Daniela da Cruz received a degree in "Mathematics and Computer Science", at University of Minho), and now she is starting a Ph.D. degree in "Computer Science" also at University of Minho, under the MAPi doctoral program. She joined the research and teaching team of "gEPL, the Language Processing group" in 2005. She is teaching assistant in different courses in the area of Compilers and Formal Development of Language Processors; and Programming Languages and Paradigms (Procedural, Logic, and OO). As a researcher of gEPL, Daniela is working with the development of compilers based on attribute grammars

and automatic generation tools. She developed a completed compiler and a virtual machine for the LISS language (an imperative and powerful programming language conceived at UM). She was also involved in the PCVIA (Program Comprehension by Visual Inspection and Animation), a FCT funded national research project; in that context, Daniela worked in the implementation of "Alma", a program visualizer and animator tool for program understanding. She is now enrolled in a new bilateral cooperation project with Slovenia under the subject "Program Comprehension for Domain Specific Languages".

Mário Marcelo Béron has four Computer Science degree, obtained at the National University of San Luis (UNSL), Argentina: "Programador Superior" in 1995; "Computer Science Lecturer" in 1996; "Bachelor in Computer Science" in 2002; and "Master in Software Engineering" in 2005. Currently, he is preparing his Ph.D thesis on Program Comprehension, under a Alfa LERNet EC-contract, at University of Minho, UM (Braga, Portugal), and National University of San Luis (Argentina). He works as assistant professor at the Informatics Department of National University of San Luis and he is a research member of the following projects: PCVIA, Program Comprehension by Visual Inspection and Animation (UM); and Ingeniería de Software: Conceptos, Métodos y Herramientas en un Contexto de Software en Evolución (UNSL). His research interests include Compilers, Program Comprehension, Programming Languages, Domain Specific Languages, Domain Engineering, Systems Specification using Rigorous and Formal Methods.

Pedro Rangel Henriques got a degree in "Electrotechnical/Electronics Engineering", at FEUP (Oporto University), and finished a Ph.D. thesis in "Formal Languages and Attribute Grammars" at University of Minho. In 1981 he joined the Computer Science Department of University of Minho, where he is a teacher/researcher. Since 1995 he is the coordinator of the "Language Processing group". He teaches many different courses under the broader area of programming: Programming Languages and Paradigms (Procedural, Logic, Functional and OO); Compilers and Formal Development of Language Processors; etc. He is co-author of the "XML & XSL: da teoria á prática" book, publish by FCA in 2002. Pedro Rangel Henriques has supervised M.Sc. (16) and Ph.D. (14) thesis, and more than 100 graduating trainingships/projects, in the areas of: language processing (textual and visual), and structured document processing; program animation and program comprehension; knowledge discovery from databases, data-mining, and data-cleaning. He also was responsible for several applicational projects (in the interface university/external-community, industry), mainly in the area of information systems (databases and web oriented). From 2002 until 2004 he was the Head of the Department, and at moment he is the President of APPIA, the Portuguese Association for Artificial Intelligence.

Maria João Varanda Pereira received the M.Sc. and Ph.D. degrees in computer science from the University of Minho in 1996 and 2003 respectively. She is a member of the Language Processing group in the Computer Science and Technology Center , at the University of Minho. She is currently an adjunct professor at the Technology and Management School of the Polytechnic Institute of Bragança, on the Informatics and Communications Department and vice-president of the same school. She usually teaches courses under the broader area of programming: programming languages, algorithms and language processing. But also some courses about project management.
As a researcher of gEPL, she is working with the development of compilers based on attribute grammars, automatic generation tools, visual languages and program understanding. She was also responsible for PCVIA project (Program Comprehension by Visual Inspection and Animation), a FCT funded national research project; She was involved in several bilateral cooperation projects with University of Maribor (Slovenia) and, at the moment, a new one is under development about the subject *Program Comprehension for Domain Specific Languages'*.