View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Biblioteca Digital do IPB

# Strategies for Program Inspection and Visualization

Daniela da Cruz[1], Mario Béron[1], Pedro Rangel Henriques[1], and Maria João Varanda Pereira[2]

[1] University of Minho - Department of Computer Science,
Campus de Gualtar, 4715-057, Braga, Portugal
{danieladacruz,prh}@di.uminho.pt
[2] Polytechnic Institute of Bragança
Campus de Sta. Apolónia, Apartado 134 - 5301-857, Bragança, Portugal
mjoao@ipb.pt

**Abstract.** The aim of this paper is to show the strategies 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.
On one hand 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.
On the other hand, we traverse the source program and instrument it with inspection functions. Those 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.

## 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 one 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 system 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 DAST Approach

In this section, we discuss the approach to program inspection and visualization followed in the context of Alma, one of the PCVIA tools under development. 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**).
The core of such tool is language independent; it is similar to a compiler's *back-end* that takes as input an abstract representation, and implements the visualizer and the animator components in a systematic way. To process a concrete programming language, the tool is specialized providing a dedicated *front-end* that converts the input programs into that internal abstract representation. As intermediate representation, between the *front-end* and the *back-end*, we chose a DAST— Decorated Abstract Syntax Tree.
In this paper we do not want to introduce or explain Alma in detail neither discuss the quality of the generated visualizations; our purpose is just to 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 [9][10] that describes the meaning 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 Figure 1:

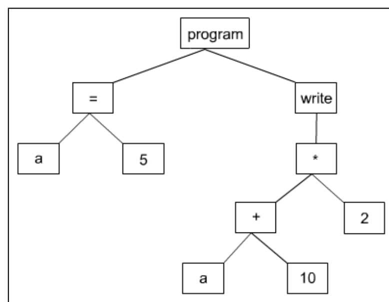


**Fig. 1.** Syntax Tree representation of the program

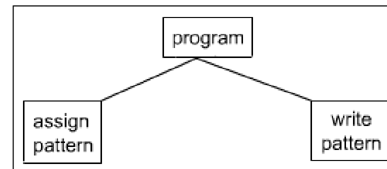


**Fig. 2.** Pattern Tree representation of the program

However, Figure 2 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.

### 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. Figures 3 and 4 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.

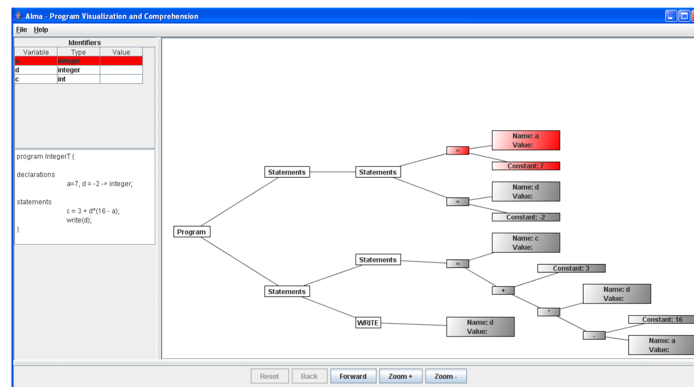


**Fig. 3.** Global visualization of the source program

## 3 Code Instrumentation Approach

In this section we discuss a strategy that aims at extracting information from programs [11][12] using code instrumentation. With this technique we insert inspection functions (or *inspectors*) in strategic places of a program to capture its execution flow [13]. The information extracted along this inspection will be used to show different views to help understanding its behavior.
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 annotation strategy;
- Tracing summarization strategy;

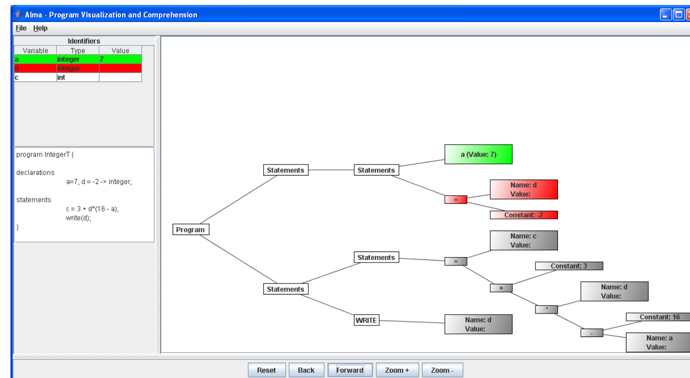


**Fig. 4.** Program state after one step of animation

– 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 annotate the program with the 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

```
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 inspection functions

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 [14] 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.

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.

Figure 5 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.

### 3.3 Visualization of the Information

In this section we present the approach used to visualize [15] 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;

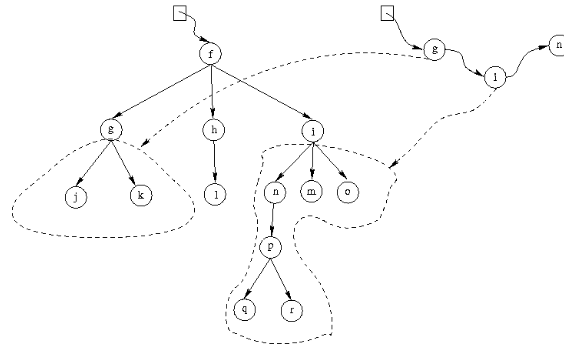


**Fig. 5.** Strategy to explain system aspects

5. Module — represents the recovered static information at module level;
6. Behavioral — concerns the system output.

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.
The two first levels 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's 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 is to relate the 5 to the 6 level because it implies to *interconnect the program domain with the problem domain*.
As a case study and aiming at test 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 [16]. 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, we extended EAR functionalities with: (1) Object and source code and runtime function inspection; (2) MCG and FCG visualization. The first extension (1) implement the levels 1, 2, 3 of the visualization architecture, and the second extension (2) implements levels 4 and 5.

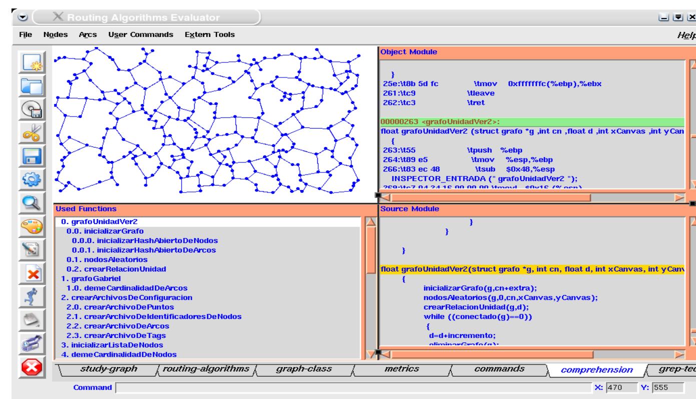


**Fig. 6.** System views

The implementation of the visualization system allowed us to determinate the usefulness of the abstraction levels. In this context, we can observe 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 and a single window because it facilitate the inspection and debugging. Furthermore, to have an integrated view allows us to relate problem and program domains, which is indeed a relevant aid to program understanding.

## 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 of our 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 to 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 annotation technique was used to trace program behavior during execution. 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 will be 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 in PCVIA, to another domains, namely to XML documents, and modeling. Mainly the non-invasive approach is being considered.

Concerning the first case, we have proposed in [17] a system called eXVisXML to 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 precisely, Class Diagrams extended with OCL constraints)and sets of tests. A prototype is being developed and a

paper was submitted to an international conference.
At last, a Ph.D. work is starting to study the adaptation of the referred techniques (usually defined to care of source code) to deal with intermediate or object code.

## 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 incerconnect 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
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. Moher, T.G.: Provide: A process visualization and debugging environment. In: IEEE Transactions on Software Engineering. Volume 14. (June 1988) 849–857
10. Reiss, S.: PECAN: Program development systems that support multiple views. IEEE Transactions on Software engineering (1985)
11. 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
12. Béron, M.M., Henriques, P., Varanda, M.J., Uzal, R., Montejano, G.: Language processing tool for program comprehension. In: XII Argentine Congress on Computer Science. (2006)
13. Yuying, W., Qingshan, L., Ping, C., Chunde, R.: Dynamic fan-in and fan-out metrics for program comprehension. In: PCODA: Program Comprehension through Dynamic Analysis. (2005) 38–42
14. Hamou-Lhadj, A.: The concept of trace summarization. In: PCODA: Program Comprehension through Dynamic Analysis. (2005) 38–42
15. Balmas, F., Werts, H., Chaabane, R.: Ddgraph: a tool to visualize dynamic dependences. In: PCODA: Program Comprehension through Dynamic Analysis. (2005) 22–27
16. Berón, M.M., Henriques, P., Varanda, M.J.: Understand routing algorithms. In: Interacção'06. (2006)
17. da Cruz, D., Henriques, P.R., Pereira, M.J.V.: Exploring and visualizing the "alma" of xml documents. In: XATA 2008 - XML: Aplicações e Tecnologias Associadas, Universidade de Évora. (2008)