UDC 004.4’426

Constructing program animations using a pattern-based approach

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

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

Abstract. The aim of this paper is to discuss how our pattern-based strategy for the visualization of data and control flow can effectively be used to animate the program and exhibit its behavior. That result allows us to propose its use for Program Comprehension. The animator uses well known compiler techniques to inspect the source code in order to extract the necessary information to visualize it and understand program execution. We convert the source program into an internal decorated (or attributed) abstract syntax tree and then we visualize the structure by traversing it, and applying visualization rules at each node according to a pre-defined rule-base. In order to calculate the next step in the program execution, a set of rewriting rules are applied to the tree. The visualization of this new tree is shown and the program animation is constructed using an iterative process. No changes are made in the source code, and the execution is simulated step by step. Several examples of visualization are shown to illustrate the approach and support our idea of applying it in the context of a Program Comprehension environment.

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 single programs instead of the complete set of programming units (the application), 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**. We are developing a tool 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. In the Section 2 of this paper, we are going to discuss this approach and the generated visualizations. To attain such an objective, we parse the source program and build a decorated syntax tree and a symbol table that are kept in memory to support all the other subsequence operations (visualization and animation), while the program itself is discarded and is not compiled (it will not be executed in the target machine).

The animation will be generated gluing all the visualizations of the execution steps. In order to calculate a new execution step a set of rewriting rules will be applied to the internal tree.

Following this approach we implemented **Alma**, a program animation system.

We show some visualizations created by **Alma**. **Alma** facilitates the representation of abstract program concepts and can be useful in circumstances where there are not specific tools to visualize programs written in the language. The produced visual representation contains information about instructions and data that will allow the user to get the perception of the program's execution behavior and the changes in the value of variables.

1.1. Related work

During our study of the state of the art we found several software handling tools: classic Program Comprehension (**PC**) 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 these 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. These parsers are used to transform the source code in order to instrument it with inspection functions or special data types. They can be also used to construct 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 of the tool are: Moose [1], CANTO [2] or Bauhaus [3]. 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 almost all 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 an 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. The 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 — well-known language processing techniques are applied to visualization and animation. **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.

A first difference is that **Alma** can be easily prepared to cope with a new language; it is simply a matter of building a map between language concepts and **Alma** nodes, and nothing more is needed. Another difference is that **Alma** is not a tool to analyze an application (a set of modules) and extract information for its comprehension. Instead of that, Alma aims at aiding to understand a program by visual inspection of its structure and by animation.

Of course, one of its handicaps is that more complementary tools are needed to comprehend an application. Another disadvantage is that the visual representations can be not so beautiful as those produced by tools dedicated to a language or a problem class; but, on the another hand the system is more generic.

TKSee [4] or SeeSoft [5] are some examples of tools that collect statistic information about the source program and then this information is shown in a structured way without changing the source code. TKSee permits users to 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 purpose and static and dynamic analysis) and shows the information using different coloured lines.

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 to this task, 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 (a *static view* of the algorithm realized by the program —visualization), and to understand its *behavior* (a *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 an 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; our purpose is just to discuss the *information we need to extract* from the source program, *how we do it*, the format under which this *information is represented*, and *how is it visualized* to help the user to understand the program.

3. 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 in that direction, it is essential to find out a set of program patterns that we know how to deal with (display and rewrite). That 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, allows us to state that all of them have common entities, like: *literal values* and *variables* (atomic or structured), *assignments*, *loops* and *conditional statements*, *write/read statements and functions/procedures*.

After the common entities have been identified, 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.

In this paper, we consider that a **pattern** is a tree that represents an abstraction of a programming concept; it is composed by two parts: a structural component (given by a grammar production) and a semantic component (given by a set of attribute occurrences affected to the symbol that labels each tree-node).

These patterns capture the abstract syntax of each entity (value or operation) in order to preserve and keep, via attributes, the necessary information to express its static semantics.

This set of patterns can be compared with the instruction set of a machine. At compile time, the statements of a program, in the source language, are mapped into the proper instructions of the target machine (translation from high-level to low-level, or machine-level). In the same way, with our approach, the statements of a program are translated into patterns (in this case, from high-level to an abstract-level). For example, in an high-level language, the *reference to a variable* in an expression means the *access to its stored value*. The corresponding machine instructions have to load the variable value into the stack or accumulator; in the assembly language of a stack machine these instructions are something like: **PUSH var_address**, followed by **LOAD**. Similarly, in our approach, this operation will be mapped, in our internal representation, to the pattern that matches a variable; similar to assembly language, we get the value of this variable from the attribute where it was stored at parsing time.

4. Program representation: Pattern Tree

Once we decided the information that we need to extract from a source language, we must now find a way to represent it. The internal representation chosen to store these patterns is a **DAST** [6][7] that describes the meaning of the program we intend to represent and visualize, being separated from any particularity of a source language. This **DAST** is specified by an abstract grammar independent of a concrete source language. This **DAST** is intended to represent the program in each execution point.

Consider the following program in some imperative source language:

a=2;

write((a+10)*2);

Clearly, we have two different statements: an assignment and an I/O statement (write).

One possible representation for it could be the syntax tree shown in Fig. 1(a).

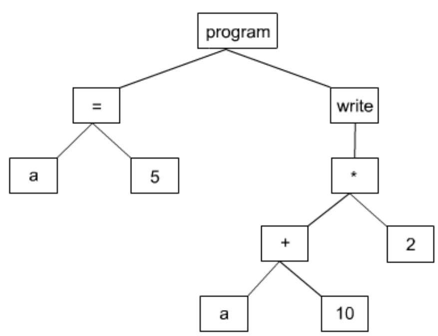

Fig. 1. (a) Syntax Tree representation of the program

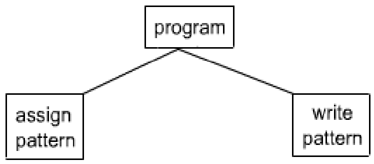

Fig. 1. (b) Pattern Tree representation of the program

Fig. 1(b) 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.

Looking at Fig. 1(a), we see that assignment node has two children—a *variable name*, and a *value*—and an implicit *type*. So, the corresponding **DAST** pattern will use three attributes: *name*, *value* and *type*. Bellow, we list the patterns considered in our approach, as well as some of the attributes used in each one.

- **Constants** — *value, type*;
- **Variables** — *name, value, type*;
- **Assignments** — left-side: *variable name*, right-side: *expression*;
- **Arrays** — particular case of variable, have one more attribute: *dimension*;
- **Conditional If/Then** — *boolean expression* (1) to evaluate, *set of statements* to execute in case (1) is true;
- **Conditional If/Then/Else** — *boolean expression* (2) to evaluate, *set of statements* to execute in case (2) is true, *set of statements* to execute in case (2) is false;
- **Loops** — *boolean expressi*on (3) to evaluate, *set of statements* to execute in case (3) is true;
- **Read** — (variable) *name, value, type*;
- **Write** — *expression*;
- **Functions/Procedures** — *table* for local variables, *arguments*, *set of statements*, and *return value* (for functions).

The visualization and animation are internally supported by trees. At first, the *program tree* is constructed, representing a static visualization of the entire source program. Then, an execution tree is constructed representing the dynamic facet of the program. The rewriting and visualizing processes are applied precisely to this second tree; the first one will be only used as a repository of nodes. For example, when an instruction is executed three times, three instances of the corresponding nodes will be copied from the *program tree* to the *execution tree*.

In order to simulate the execution, all the pattern instances have one common attribute: **isEvaluated**. This attribute is mainly used to control the *rewrite process* (necessary for program animation) — it indicates if the tree had *already* been evaluated or *not yet*.

In next section, we show how we implement the patterns and how the DAST is built.

5. Pattern Extraction and Implementation

To extract information from a concrete source program its is necessary to parse it. This operation will be responsible for 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. Our choice was LISA [8,9]. LISA is implemented in the programming language Java, following an object-oriented approach either in its internal implementation or in the attribute grammar it accepts. To generate a *front-end* for a specific source language, we use the syntax and static semantics of that language specified by its grammar, and then we add to each production new attribute evaluation rules (**computing statements** in **LISA**'s metalanguage) to build the internal representation of the corresponding **DAST** pattern.

For example, consider a grammar derivation rule (production) to define the assignment statement in some imperative language. Its definition in **LISA**'s metalanguage using attribute evaluation *templates* is:

```
rule extends Assign {
          ASSIGN ::= DESIGNATOR \= EXPR \; compute {
             ALMA_ASSIGN_VAR<ASSIGN, DESIGNATOR.name,
                               EXPR.value, DESIGNATOR.type>
          };
}
```

ALMA ASSIGN VAR is a *template* and has the three attributes mentioned in subsection 4 — the variable name, value and type.

```
Each template is previously defined in an Alma library
and has the generic form shown below:
template<attributes X_in, Y_in, ...> compute NODETYPE {
          X_in.dast = new Node(Y_in, Z_in, ...);}
```

Notice that **NODETYPE** identifies the type of the **DAST** node to be built corresponding to the pattern found (one of those listed in page 3).

As we are using the **LISA** tool to automatically produce the extractor, we also decided to implement the patterns (subsection 3) reusing some **LISA** classes. It was very easy to identify and understand the data structures and methods used by the **LISA** system to process a given attribute grammar specification or a source program — they are properly encapsulated in classes with attributes and methods. So the coding of patterns became straightforward, due to the reuse of **CSyntaxTree** and **CTreeNode** classes to build the internal tree representation. As an immediate consequence, all the facilities provided by **LISA** to manipulate the attributed tree became available to process the **DAST**. This consequence made the development of the **Alma** *back-end* (another **Java** class that implements the visualizer and animator) much easier and faster; to code that class, we kept the object oriented approach followed in **LISA**.

6. Pattern Visualization

At this point, we had already decided which information to extract, how to extract it, how to represent it, thus making how to visualize it is a natural consequence of the previous decisions.

Once we have a pattern tree as the intermediate representation between the *front-end* and the *back-end*, the **DAST** will be used to construct a visual representation for the source program. Each pattern will be extended with one additional attribute: *vr*, that contains the corresponding visualization rule. Thus, the visualization of a program is obtained by making a *top-down* traversal over the **DAST**, applying the specified rule to each node instance. The first traversal produces a picture of the entire program before execution. So, the animation of a program will be done by multiple *top-down* traversals to the **DAST**, until program is totally rewritten.

The objectives of our approach are twofold: 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 purpose 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 (and, of course, of the algorithm). **Alma** has also included some features in order to cope with scalability problems and can also be adapted to other kind of views (different abstract levels) depending on the purpose of its user.

For the first program listed below, the visualization obtained is shown in Fig. 1 (using the pattern tree corresponding to the source program shown in the bottom-left sub-window).

Fig. 1 and Fig. 6 show the animation of a **LISS** program. **LISS** [10] is a language where all variables are initialized at declaration time (with explicit values or default ones). Fig. 7, 8 and 9 (subsection 6.2) are related with **C** language.

```
program Integer {
    declarations
        a = 7, c, d = 2 -> integer;
    statements
        c = 3 + d*(15-a);
        write(d);
}
```

As can be seen in Fig. 2, the user interface of **Alma** system is split into 4 sub-windows: the definition's table (at left, top corner); the source program (at left, bottom corner); the program tree (at right, the main and biggest sub-window); and the button's to control the animation steps.

To better understand the way how works **Alma**, let us consider the expression c = 3 + d*(15-a) extracted from the source program above. In **Alma** system this expression is represented as in Fig. 3.

After on step over this expression is executed, the sub-tree corresponding to sub-expression 15-a is reduced to the root of this sub-tree, having as attribute the result of the operation (Fig. 4). In this way, after executing all operations, only the node to respect with the final result is leaved, see Fig. 5.

To makes easier follow the animation process, there are used 2 colours: the red colour points to the next operation to execute (in the definition's table point to the variable that will change its value; in the source program the line of code; and in the execution tree, the sub-tree that match with the operation to execute); and the green colour points to the last operation executed.

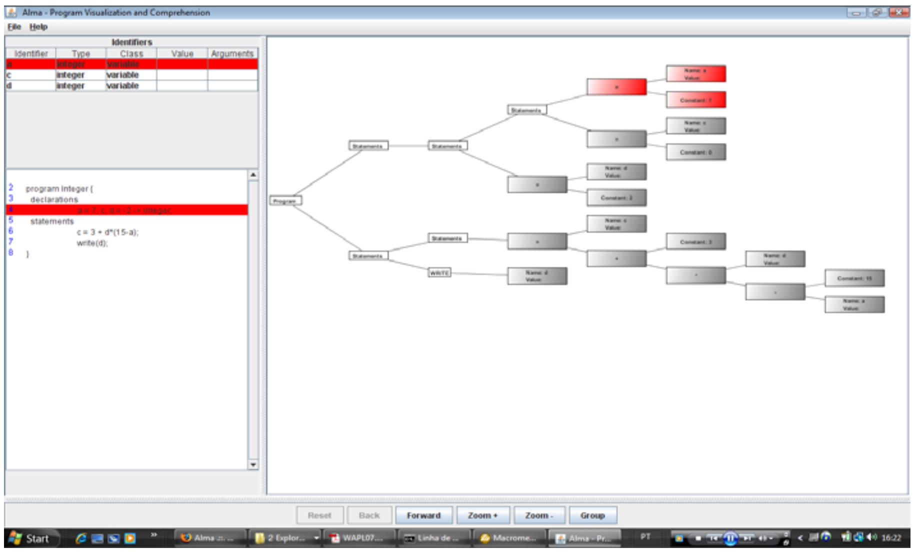

Fig. 2. Global visualization of the source program

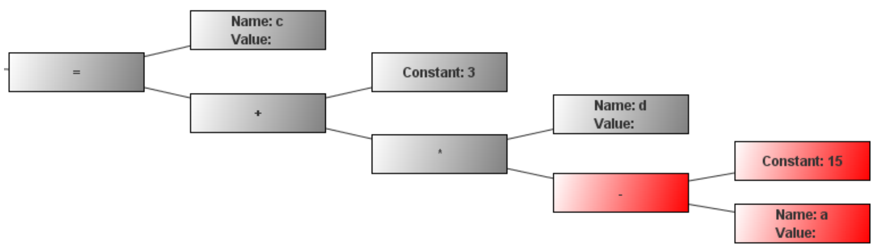

Fig. 3. Sub-tree to the expression c = 3 + d*(15-a)

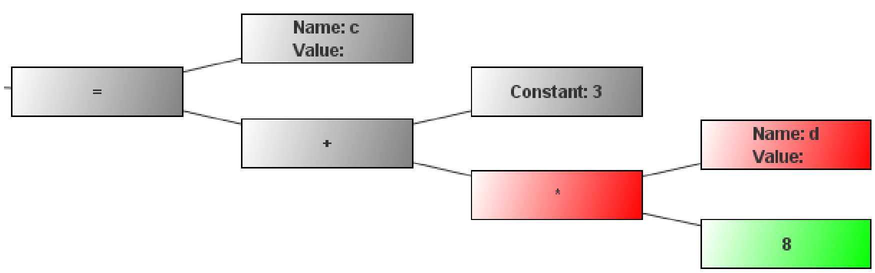

Fig. 4. Sub-tree to the expression c = 3 + d*(15-a) (after computing (15-a))

Fig. 5. Final state of the sub-tree after doing the assignment c = 3 + d*(15-a)

6.1. Arrays and Structures

For arrays and structures we chose a different way to show their initialization. The initialization of variables of this kind is usually more difficult to understand; when a variable of one of those data types is initialized, a new sub-window is shown, giving the user the opportunity to see in detail the attribution of values to each component or to skip all the steps at once.

To the program below, the Fig. 6 shows the sub-window to initialize a structured variable. After a variable is initialized in the respective row of the identifiers table, will appear a link "*See table*" to the current value of this variable. In the case of an array, will appear the values at each index (Fig. 6). In the case of a structured data type will also appear a local identifiers table.

```
program StructTest {
    declarations
       first -> struct {
         def -> integer;
         vec -> array size 3;
         b -> boolean;
       };
       def -> integer;
   statements
      first.def = 5;
      first.vec[def+1]  = first.def;
}
```

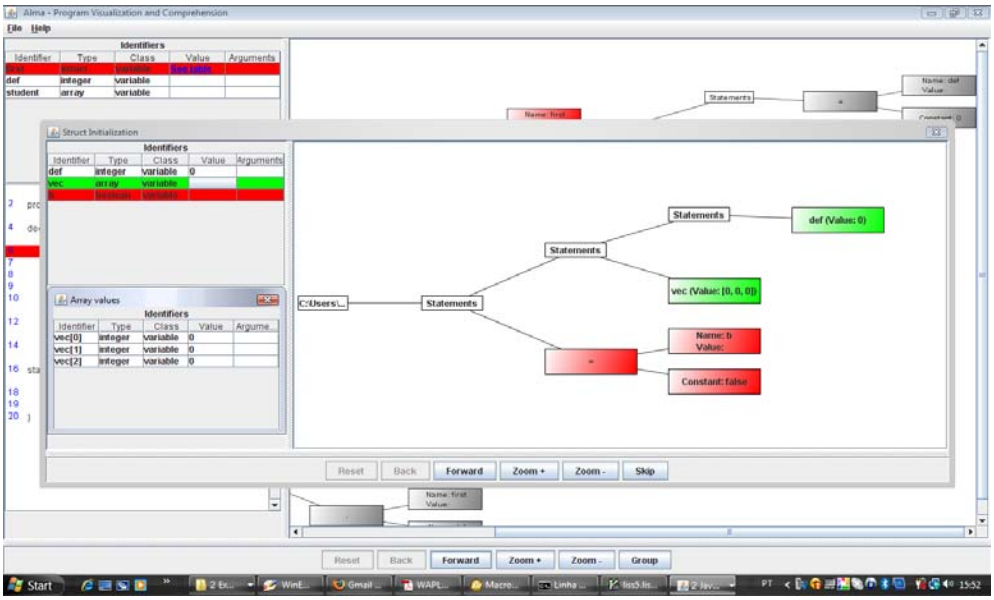

Fig. 6. Struct initialization and local table with values of an array variable

6.2. Functions and Procedures

To animate functions or procedures, a new window is opened each time the subprogram is invoked. This window is divided into parts: the first one is used to animate the parameters passing process; and the second one, to animate the execution of the function/procedure body. In case of subprograms without parameters, the first window (related with parameters passing) is omitted.

To illustrate the *function call mechanism* — suspending the execution of the invoker, evaluating and passing actual values to the function formal parameters, executing the function body, returning a value and resuming the invoker execution — we include Fig. 7, 8 and 9 that are concerned with the animation of a classic program written in **C** language listed below (program below).

```
int factorial(int n) {
    int res = 1;
    if (a != 0) {   res = n * factorial(n-1);    }
    else  {}
    return res;
```

```
}
int main() {
    int f, r;
    scanf("%d",&f);
    r = factorial(f);
    printf("%d",r);
    return 0;
}
```

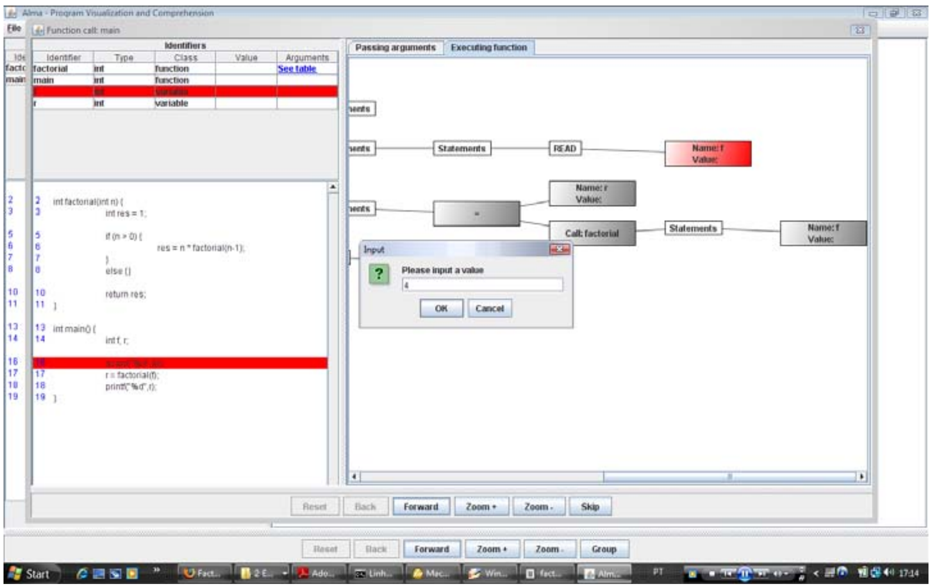

Fig. 7. Invoking the main function

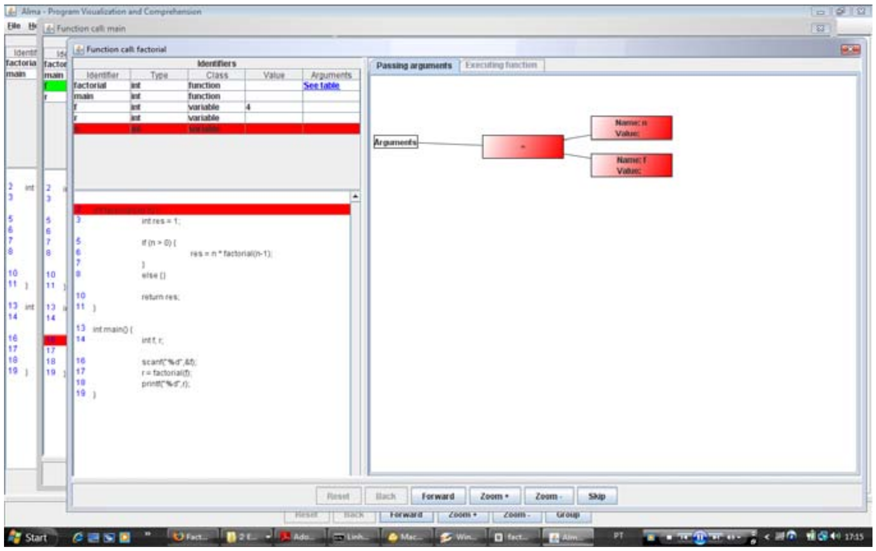

Fig. 8. Parameters passing to factorial function

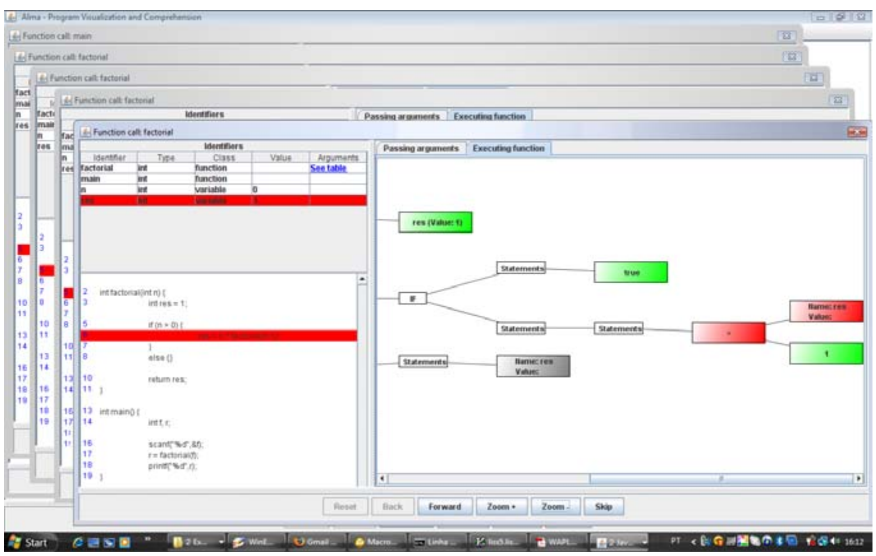

Fig. 9. Calling recursively the factorial function – Executing the function

The program under consideration in this example is composed by two functions: *main()* that prints the factorial of a given integer, invoking a function to do the computation; and *factorial()* that receives a parameter and computes recursively its factorial.

The first screen displayed by **Alma** for this example, corresponds directly to the program tree, and shows that program global structure.

Automatically the Animator invokes function *main(),* opening a second window to show inside it the *main()* execution. Fig. 7 is a screen-shot of the *main()* state corresponding to the execution of the read statement; notice the input window that appears in the middle of the screen to get a new value from the user. This picture also illustrates the mapping of a concrete C instruction, *scanf(%d,&f)*, into the **Alma**'s abstract pattern read. The next two figures (Fig. 8 and Fig. 9) illustrate the invocation of *factorial()* function. The first describes the parameter passing (immediately after executing the call statement, a new window is opened and the animation of the evaluation and assignment of the actual parameter is displayed). The second (Fig. 9) corresponds to the execution of the third recursive invocation of *factorial()*. Notice that a new window is opened for each function invocation; a new identifier table and a new tree are displayed in order to animate that new execution process.

For each function, a local identifier table is created, and it is possible to map each row of this table to the visualization of the function body execution. When the function execution ends, the local table disappears, and the return value is transferred to the previous identifier table.

The execution tree for a huge program will be very large and its visualization will be difficult. However, this approach — the association of a new main window to each function (that opens when it is called, and closes when it returns)—has an important side effect: it solves scalability problems. Nowadays most of the huge programs are split into a large number of subprograms (functions or procedures). In that case (the most probable) the maximum size of the execution tree to visualize corresponds to the size of the biggest subprogram. Many windows will be opened, but the size of the tree inside each one is reasonable and manageable! This allows us to say that our solution for the visualization of a source program as a forest of the tree-patterns will scale-up without problem.

7. 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 from it *static data* — concerned with variable/type declarations and statement structure — and *dynamic data* — concerned with the data and control flows.

The objectives of our approach are two-fold: 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 purpose 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 (or definition table) and an abstract syntax tree are enough to store this information. The visualization process is then performed by a systematic tree traversal, applying straightforward rules to each tree pattern, and to the correspondent row in the symbol table and line in the source text.

In our approach, we no longer need the source program; furthermore, after extracting information and building the **DAST**, we are able to give visual details helpful to get easily an *operational view* of the program. This approach

does not modify the source program, and relies upon a visualization/animation engine (the *Back-End* of the tool) that is independent of the source language (and, of course, of the algorithm); thus, tuning the tool to analyze programs in different languages is not an hard task.

In order to use **Alma** for a new language we just have to construct a *front-end* for that language. This *front-end* will map each source language concept to an **Alma** pattern. To start **Alma** development, we have created a front-end for **LISS** language, enabling us to begin the tests. Recently, we have followed a similar systematic process to construct another front-end, this time for **C** language. Using an attribute grammar (based on a public **CFG** for **C**) and **LISA** generator, this *front-end* was developed very fast.

We also believe that **Alma** can be very useful to visualize more declarative languages, like functional/logic programming languages, or specification languages, but we will work out this point as future work.

Hence **Alma** patterns correspond to the Turing machine basic operations, we argue they suffice to deal with the common imperative programming languages.

To cope with other paradigms, as referred above, possibly it will be necessary to upgrade the patterns library to include some others that can contribute to a more clear understanding of their specificities.

Alma system can be particularly useful for domain specific languages and other special languages that don't have any kind of **PC** tool implemented. For these languages a new PC tool would be constructed from the scratch.

As we already said using **Alma** a specific **PC** tool can be easily prepared.

We are convinced that present **Alma** animations really help on program understanding — they show a program execution simulation with data and control flow information. However, this statement will be measured in the near

future, via usability tests.

8. Refences

1. Ducasse, S., Gˆırba, T., Lanza, M.: Moose: an agile reengineering environment. In: ESEC-FSE'05, Lisbon, Portugal (2005)
2. 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 (1997)
3. Raza, A., Vogel, G., Plodereder, E.: Bauhaus - a tool suite for program analysis and reverse engineering. Technical report, Department of Programming Languages, Institute for Software Technology, University of Stuttgart (2006)

4. Herrera, F.: A usability study of the tksee software exploration tool. Master's thesis, University of Ottava (1999)
5. Eick, S., Steffen, J., Jr., E.S.: Seesoft - a tool for visualizing line oriented software statistics. IEEE Transactions on Software Engineering 18 (1992) 957–968
6. Moher, T.G.: PROVIDE: A process visualization and debugging environment. In: IEEE Transactions on Software Engineering. Volume 14. (1988) 849–857
7. Reiss, S.: PECAN: Program development systems that support multiple views. IEEE Transactions on Software engineering (1985)
8. Mernik, M., Zumer, V., Lenic, M., Avdicausevic, E.: Implementation of multiple attribute grammar inheritance in the tool LISA. ACM SIGPLAN not. 34 (1999) 68–75
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. da Cruz, D., Henriques, P.R.: Liss — The Language and the Compiler. In: Proceedings of the 1st Conference on Compiler Related Technologies and Applications, CoRTA'07—Universidade da Beira Interior, Portugal. (2007)

Daniela Carneiro da Cruz got 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 is 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, and she is now enrolled in the development of similar tools for XML, UML and so on.

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 a prática" book, publish by FCA in 2002. Pedro Rangel Henriques has supervised M.Sc. (13) and Ph.D. (12) thesis, and more than 50 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 Tinoco Varanda Pereira got a degree in "Systems and Informatics Engineering", at University of Minho in 1994, finished a master degree in Computer Science in 1996 and finished a Ph.D. thesis in "Program Animation Systematization" at University of Minho in 2003. In 1995 she joined the Informatics and Communications Department of Polytechnic Institute of Bragança, where she is adjunct-professor since 1999. She is researcher in the "Language Processing Group" at University of Minho since 1995, where she is working in Program Comprehension, Program Animation, Visual Languages, Automatic Generation of Compilers and Software Engineering.

She teaches several courses in programming area: Programming Languages and Paradigms, Language Processing and Programming Techniques.

Maria João Tinoco Varanda Pereira has supervised a M.Sc. during 2005/2006, about 10 graduating projects and is responsible for PCVIA (Program Comprehension by Visual Inspection and Animation), a FCT funded national research project. From 2003 until 2005 she was head of the Department and, at the moment, she is vice-president of Technologic and Management School of the Polytechnic Institute of Bragança.