Visualization/Animation of Programs in Alma: obtaining different results

Maria João Varanda Pereira *
Polytechnic Institute of Bragança — Portugal
mjoao@ipb.pt

Pedro Rangel Henriques
University of Minho — Portugal
prh@di.uminho.pt

Abstract

Alma, a system for program animation, receives as input a computer program and produces a sequence of visualizations that will describe its functionality. The system generates automatically program animations basing this process on the internal representation of those programs. The back-end of this system works over an execution tree (DAST - Decorated Abstract Syntax Tree), implementing the animation algorithm. This algorithm uses two bases of rules: visualizing rules (to associate graphical representation with program elements creating a visual description of the program state) and rewriting rules (to change the program state).
In this paper, the main goal will be to present the extensibility of the system in the sense of adding or modifying inputs and outputs. We also discuss the characteristics of Alma's architecture that make this possible.

1. Introduction

The purpose of an animation system is to help the programmer to inspect the data and control flow of a source program—static view of the algorithm implemented by the program—and to understand its behavior—dynamic view of the algorithm.
Convinced about the importance of *program visualization* and *algorithm animation* and after reviewing the existing systems [2], we decided to design and develop a new visualization environment, Alma, obeying the following design goals:build an integrated and easy to use environment; avoid the need for any kind of change in the source code; allow the selection of different views of the same program; create a system as generic as possible in order to be used by different source languages.
The Alma system was designed to become a new generic tool for program visualization and animation 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 comply with the requirements above, and based on our background on compiler specification and implementation, we conceived the architecture shown in figure 1.

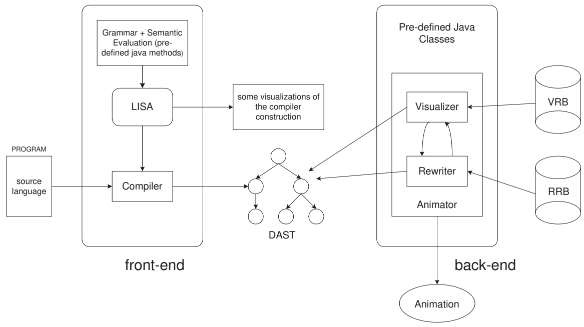

Figure 1. Architecture of Alma system

The system has a *front-end* specific for each language and a generic *back-end*, and uses a decorated abstract syntax tree (DAST) for the intermediate representation between them. The DAST represents the meaning of the program we intend to visualize; in that way, we isolate all the source language dependencies in the *front-end*, while keeping the generic animation engine in the *back-end*. The DAST is specified by an abstract grammar independent of the concrete source language. In some sense we can say that the abstract grammar models a virtual machine. So the DAST is intended to represent the program state in each moment, and not to reflect directly the source language syntax. The tree generated by the *front-end* (after program analysis) represents the input program. The program tree is static and is kept unchanged. Applying some rewriting rules to that tree, we will get an execution tree representing the first state of program execution. In this way we rewrite the DAST to describe different program states, simulating its execution; notice that we deal with a semantic transformation process, not only a syntactic rewrite.
A *Tree Walk Visualizer*, traversing the execution tree, creates visual representations of nodes, gluing figures in order

*The work of M. João is partially support by the Portuguese program PRODEP, acção 5.2 da medida 5 – doutoramentos

to get the program image on that moment. Then the DAST is rewritten (to obtain the next internal state), and redrawn, generating a set of images that will constitute the animation of the program.

2. Basic use of Alma System

A typical user only has to create the input of the system, editing the program he wants to animate. Then, he submits his program using a command line like:

```
> java Animate file_name.test
```

and the program will be animated. For each programming language that has an Alma *front-end* previously created, the programmer just uses the system (as exemplified above) without any additional specification or modification. This is the original Alma purpose and its main use mode.
The generality of the system can be an handicap to achieve output effects. We think that the system would be more useful if it allows the addition of new rules to support new concepts or generate different outputs. Alma system has a fixed part (visualization/animation algorithms; tree, nodes, identifier table and rules structure; rule bases interpretation, etc) and an extensible part (visualization and rewriting rules, nodes, etc). We conclude that the fixed part gives generality and the extensible part makes possible to obtain more adequate visualizations.
In the next section we will show how different kind of users can interact with the system in order to get the most appropriate animation.

3. Advanced use of Alma System

Table 1 shows how to customize Alma system. As we have said, the system is divided into two parts: fixed part and extensible part. Column 1 represents the desired effect and column 2 explains how to extend.

EXTENSIBLE PART	
To obtain	Has to modify
$\neq$ **languages**	new FE
$\neq$ **level of animation detail**	`shownow` variable
$\neq$ **visualization types** $\neq$ level of visualization detail new drawings $\neq$ level of abstraction	visualization rules + interpretation mechanism of drawings
$\neq$ **language paradigms**	new semantics (RR) + new nodes = new rewriting rules

Table 1. Extensions of Alma System

Different Languages
If the user wants to apply the system to a different source language, he only has to construct a *front-end* that defines the concrete syntax of the new language and maps its main concepts to Alma nodes. This *front-end* can be generated using LISA system [1].

Different level of animation detail
The advanced user can also modify the sampling frequency (number of state transformations (tree rewritings) before a visualization) or choose the set of nodes he wants to visualize, in order to get a different level of animation detail. An animation can have more or less visualizations depending on the desired detail level. The most detailed animation implies the visualization of the tree after each rewriting. The synchronization between these processes depends on a function called *shownow*. This function counts the rewritings and returns 0 or 1 depending on the desired frequency.
The visualization is obtained traversing a DAST that has associated drawings. If we decide to show only some nodes we will get a less detailed visual representation. There are nodes that are more important than the others and their visualization can explain all the functionality of the program. The user will access an interface where he can easily choose those nodes and watch the results.
We have to distinguish animation detail level from visualization detail level. On the first one, we do not have to change the drawings (it is concerned with process synchronization, number of visualizations), and on the second we have to redefine visualization rules in order to get different results, as we will discuss in the next paragraph.

Different types of visualizations
Alma system has two bases of rules that can be improved with new semantics or new drawings. The user may want to get different visualizations for the same language he used before or he may want to animate a very different language and he must define new visualizations for it. There are several possibilities to change visualizations: varying the level of visualization detail using a different mapping between nodes and drawings; choosing different drawings; or both in order to get a different abstraction level.
The generated visualizations are generated based on rules that map nodes to draws. If the user wants to change the draws in order to get a different visualization, he can modify rules or specify new ones. He can represent the same concepts with different drawings.
If we want to change the visualization detail we must associate the drawings to another level of nodes. In some cases, we can use the same drawings but when the concepts concerned to this level are different we must define another drawings too. Changing drawings and associated nodes, we can modify the abstraction level of the visual results. The idea is to create new visualization rules in order to associate more abstract drawings to higher level nodes.

Different type of paradigms
When we have a very different source language mainly if it implements a different programming paradigm, we have to verify which concepts are common and which are not. For the last ones, we have to specify new visualization rules, create new DAST nodes and specify new semantics with rewriting rules. In this section we will show an example: Prolog.
We take as example the following input program:

```
mother(julie,susan).
mother(susan,john).
father(peter,paul).
father(peter,susan).
parents(M,P,E):-mother(M,E),father(P,E).
```

We must have a *front-end* to map the Prolog concepts to Alma nodes. An extended LISA grammar associates facts and rules to `PROCDEF` node, because this node represents the definition of a code block that can be invoked from any place of the program. The execution tree for the query:

```
? - parents(M,P,susan).
```

can be seen on figure 2.

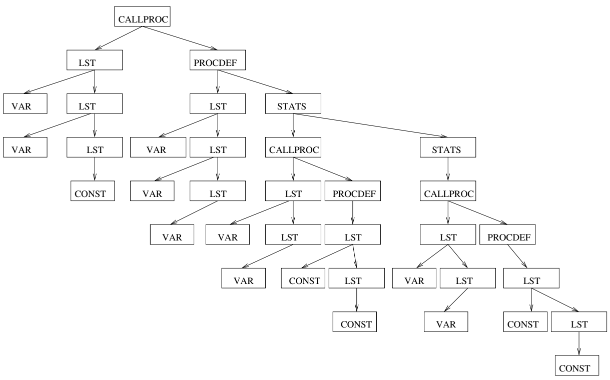

Figure 2. Execution tree created by Alma

A *query* is mapped to a `CALLPROC` node that has associated a set of parameters (`LST`) and a `PROCDEF` node which defines a fact or a rule. In the first case, the `CALLPROC` has the value `true` but, in the second case, it is necessary to verify the truth of every predicate on the rule body, replacing the inner `CALLPROC` nodes by the appropriate `PROCDEF` nodes. Each `PROCDEF` node has a local identifier table associated, whose variable values will be used (on PROCDEF exit) to update the outer table.
The animation of the execution tree (simulation of the proof process) uses the visualization rules already defined for other languages. In a similar way, we apply the same rewriting rules used to simulate procedure calls.
The effect can be seen in figure 3 that presents the less detailed version of the generated animation (the minimal number of steps are shown).

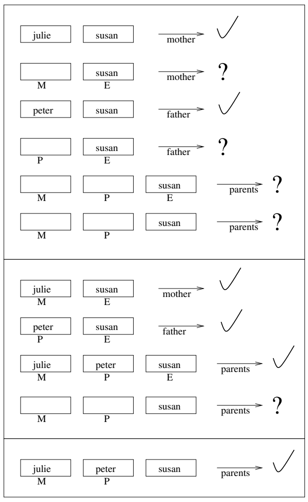

Figure 3. Alma **generated animation**

With this example, we have illustrated the possibility of reusing the visualization and rewriting rules, already defined in Alma for imperative languages, to animate declarative programs (proof processes).
But the system is also prepared to be extended with extra rules if it is necessary. For example, we can have several `PROCDEF` for each `CALLPROC` node and, in this case, we should use a backtracking stack. So, we have to define new rewriting rules to specify the management of this stack.

4. Conclusion

This paper introduced the architecture of Alma system that allows to achieve the proposed objective: systematize the program animation process making it automatic and algorithm/language independent. The main idea is to avoid an algorithm or language oriented approach, developing instead a new generic model based on the program semantics (the program **soul**, or *Alma* in Portuguese). We discussed the characteristics that enable us to customize the system - to adapt to new languages or produce different outputs.
As future work, we plan to develop an interface that helps the user to perform the desirable extensions.

References

[1] P. Henriques, M. J. Varanda, M. Mernik, and M. Lenic. Automatic generation of language-based tools. In *LDTA - Workshop on Language, Descriptions, Tools and Applications (ETAPS'02)*, April 2002.
[2] J. Stasko, J. Domingue, M. H. Brown, and B. A. Price. *Software Visualization - Programming as a Multimedia Experience*. The MIT Press, 1997.