Visualization/Animation of Programs based on Abstract Representations and Formal Mappings

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

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

Abstract

In the context of Alma *(a system for program visualization and algorithm animation), we use an internal representation—based on the concept of an attributed abstract syntax tree decorated with attribute values, a* DAST*—to associate (static) figures to grammar rules (productions) and to step over program dynamics executing state changes in order to perform its animation. We do not rely upon any source program annotations (visual/animation statements, or parameters), neither on any special visual data types.*
On account of such principle, the approach becomes source language independent. It means that we can apply the same visualizer and animator, that is the Alma*'s back-end, to different programming languages; all that we need is different front-ends to parse each program into the* DAST *we use.*
In this paper we discuss Alma *design goals and architecture, and we present the two mappings that associate to productions figures and rewriting rules to systematically draw a visual representation (exhibiting data and control flow) of a given source program and to animate its execution.*

1. Introduction

An algorithm animation is a dynamic visualization of the main abstractions of that algorithm. The animation is a natural way to represent behaviour and its purpose is to show the concepts involved in a program and how they evolve during execution.
The visual representation used in an animation helps on algorithm/program understanding. Program visualization belongs to a lower level of abstraction and algorithm visualization is at a higher level of abstraction. The Visual Programming is a new way to specify programs which turns this task easier to perform. The software visualization intends to turn programs easier to understand.
The Alma system aims at being a new program visualization system, based on the internal representation of each source program. This system avoids any kind of annotation of the source code with visual types or statements, and receives (as input) programs written on several kinds of languages. This is possible because the visualization is not based on the source program but on its universal internal representation. So, the system knows the grammar of the source text, and then the visualization is generated automatically.
In this paper we discuss Alma principles, and we present the two mappings that associate to productions figures and rewriting rules to systematically draw a visual representation (exhibiting data and control flow) of a given source program and to animate its execution.

The paper is divided into this introduction and seven more sections. The motivation for the work here discussed is presented in section 2, where we give an overview of the area of *algorithm animation systems*. Then we introduce Alma system, our approach to program visualization and algorithm animation, in section 3; the underlying principles and the architecture chosen to implement them are discussed. The way we specify and build the visual representation of programs is presented along section 4, while similar considerations about animation are dealt with in section 5. In section 6 we discuss an example of animation; using a Pascal program as input and introducing some rules, we show the visualization that will be produced. Some case studies are presented in section 7. Here we talk about the implementation of two *front-end's*, mapping two different languages on the Alma internal representation. As usual the paper ends with a synthesis, some words about the present state of Alma development and future work (section 8).

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

2. Algorithm Animation: approaches and tools

To support our proposal, we started studying as much visualization and animation systems as possible. This section is devoted to the review of *algorithm animation systems* developed and published so far.

We are mainly concerned with the different ways the programmer has to specify the animation, the various approaches to implement the process, and the generality of those systems; with that in mind we have looked for a classification of animation systems.

With that approach, we are able to compare our proposal against classes of existing animators, instead of doing that for each specific one.

2.1. Approaches to the Specification of the Animation

At a first glance over Animation Systems, we understood that several approaches were used in their implementation. Some of those approaches are based on: libraries of visualization functions; animation direct manipulation; algorithm annotation; self-animated data types; animation specific languages; and program semantics annotation.

The very first systems used libraries with visualization functions in order to insert (visual) function calls in the source program to produce the desired visualization. The animation process was not automatic neither systematic, and the visualization was generated in real time during program execution.

There are other systems where the visualization is totally controlled by the user. The user defines the mapping between variables and statements of the source program and their visual representation. Then the system applies that finite function to the program during its execution displaying all the generated drawings. LENS [MS93] uses this technique to construct animations.

Other systems focus on the definition of the most interesting points of the program to be visualized. For that purpose, the user annotates those points with visualizing procedures. In those cases, the animation is explicitly specified in the source code. BALSA [BS84] is a notable example of this kind of systems.

After them, some systems (like JELIOT [HPS+97]) were proposed with the goal of avoiding a full annotation of the source program; instead, special data types are used. Those data types are then translated by the system into animations procedures that produce the desired visualization.

Another family of systems (for very specific domains) require that the source text is written on a system dependent language whose syntax and semantics is specially defined for that purpose. Some examples are: JCAT [BNR97]; VIP [MM88]; ZSTEP [LF95].

We also found systems that use declarative visualization. Declarative visualization is a technique which provides the animator with the ability to construct complex visual representations for programs by defining abstract mathematical mappings from program states to graphical objects. This approach is used in PAVANE [CkCJ92], PROVIDE [Moh88], ALADDIN [HHR89] and ANIMUS [Dui98].

More recently, a new approach was introduced, defending a semantic directed annotation in opposition to the traditional structure oriented (algorithmic) annotation. The idea is to formalize the program semantics, and associate animation functions to concepts whose meaning is so far specified. The CENTAUR system [Ber91] uses this new approach.

2.2. Animation Systems

Searching for tools that use some kind of animation to explain an underlying reasoning, is an easy task because there are lots of program animators accessible via WWW; we also found many reports and articles devoted to this subject. As told above, we made some effort to classify all the applications related to algorithm animation and program visualization. As a start point, we looked for existing criteria.

Stasko in [SDBP97] proposed a classification for software visualization systems, defining six evaluation parameters: **scope** (specifies the range of programs that the animator is able to take as input for visualization); **content** (defines the subset of program information that is visualized); **form** (specifies the characteristics of the output produced by the animator); **method** (defines how the visualization is specified); **interaction** (defines how the user interacts and controls the animator); and **effectiveness** (concerning how the system presents the information to the user).

This classification criteria can be used on visualization systems which are language oriented: use specific source language, specific annotation language or specific data types. However we felt the need for a broader criteria that could couple with a larger set of systems. The classification system that we looked for should be more concerned with methodological parameters (as identified in the beginning of that section) and not so closed to technical perspectives.

In that sense, we arrived to a set of five types of visualisation/animation systems, as follows.
Most of the well known animators are not general purpose; instead they only work with a specific algorithm. Many applications in that family allow some interaction with the user (we classified them as Type II); while the rest just show the animation without any interaction with the user, who is limited to watch the explanation in a completely passive way (this group is classified as Type I).
The animation systems that came up along the last ten years, tend to be much more generic (applicable to a larger set of algorithms) while providing an interactive environment. Systems like BALSA [BS84] (the first in this class); TANGO [Sta90]; JCAT [BNR97]; VIP [MM88]; ZSTEP [LF95]; JELIOT [HPS+97]; PAVANE [CkCJ92] ; LENS [MS93] (already mentioned along the last section) belong to that class, that we call Type III, but use different approaches to generate the visualizations. The first three systems use algorithm annotations; the next two use a specific source language and automatic algorithm annotation; JELIOT uses special data types and precompilation; and, at last, LENS uses direct manipulation of the visualization.
Some programming environments, also source language dependant, provide important functionality for program visualizations. Examples of that are: PECAN [Rei85] which provides multiple views of the program syntax, semantics and execution, GARDEN [Rei87] and FIELD [Rei90]. In our opinion, those systems also should be classified as Type III.
We also found some compiler and programming environment generators that provide means to visualize some steps of the generation process, or even include some visual debugging capabilities in the code produced, or can generate environments with visual outputs. We decided to consider those generators —as LRC [Sar99], CENTAUR [Ber91] and LISA [MLAZ00], for instance — in the family of animator systems, classifying them as Type IV.
At last, there are some environments —like AGG [MRRT99], CPN [Jen96], and FORMS3 [CBC96]— for visual programming that simulate the behaviour of the system specified by those programs. We classified them as Type V.
Nowadays, there are systems that use special data types, automatic annotation, automatic generation of animations and other modern techniques that must be studied under the characterization system above. So, it would be useful to add new parameters to Stasko's classification proposal, such as: degree of automation of the visualization construction; number of views that can be created from the same program; degree of independence between the source language and animation system; degree of the source text alteration required by the animation process.

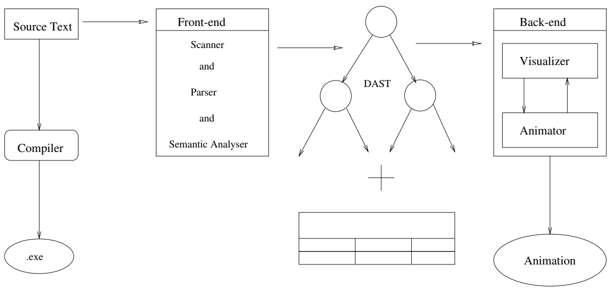

Figure 1. Architecture of Alma system

3 Architecture of Alma

Convinced about the importance of *program visualization* and *algorithm animation* and after reviewing the existing systems, 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.

To comply with the requirements above, and based on our background on compiler specification and implementation, we conceived the architecture shown in figure 1. The architecture proposed follows the compilers implementation method called *semantics directed translation*, by opposition to the older and well known *syntax directed translation*. That implementation model is influenced by the use of an *attribute grammar* to specify (formally) the syntax and semantics of a language. According to that model, the meaning of the source program (to process) is explicitly represented by a syntax tree decorated with attribute values, DAST, and a clear separation between the compiler's *front-end*—responsible for program analysis, and tree building and decoration—and the compiler's *back-end*—in charged of the translation—is maintained.

In Alma we also use a DAST as an internal representation for the meaning of the program (we intend to visualize), and we isolate all the source language dependencies in the *front-end*, while keeping the generic animation engine in the *back-end*.

This allows us to concentrate on both tasks—meaning recognition and representation, and visualization—separately, but most important, the approach gives generality to our system.

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. In this way we will 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.

Each node has a production identifier (`ProdId`) that is its key; a symbol identifier; a set of pairs (name,value) which represents the attributes associated with the symbol labelling that node; and a set of subtrees that are its children—the tree rooted in that node is equivalent to the grammar derivation rule `ProdId`.

A *Tree Walker Visualizer*, crossing the tree, creates visual representations of nodes, gluing figures in order to get the program image on that moment. Then the DAST is rewritten, and after that the process will be repeated generating a set of images (that represents the animation of the program).

4. Visualization in Alma

The visualization is achieved applying visualizing rules (VR) to DAST subtrees; the specification of those rules defines the mapping between trees and figures. Gluing those partial figures creates a visual representation for the program.

4.1. Visualizing Rules

The VRB (Visualizing Rule Base) is a mapping that associates with each attributed tree, defined by a grammar rule (or production), a set of pairs.

$$\text{VRB: DAST} \mapsto \text{set (cond} \times \text{dp)}$$

where each pair has a matching condition, `cond`, and a procedure, `dp`, which defines the tree visual representation. Each `cond` is a predicate, over attribute values associated with tree nodes, that constrains the use of the drawing procedure (dp), i.e., `cond` restricts the visualizing rule applicability.

The written form of each visualizing rule is as follows:

```
vis_rule(ProdId)= <tree-pattern>,
                  (condition),
                  {drawing procedure}

<tree-pattern> = <root, child_1, ..., child_n>
```

In this specification, `condition` is a boolean expression (by default, evaluates to true) and `drawing procedure` is a sequence of one or more calls to elementary drawing procedures.

A visualizing rule can be applied to all the trees that are instances of the production `ProdId`. A tree-pattern is specified using variables to represent each node. At least, each node has the attributes `value`, `name` and `type` that will be used on the rule specification, either to formulate the condition, or to pass to the drawing procedures as parameters.

Notice that, although each VR associates to a production a set of pairs, its written form, introduced above, only describes one pair, for the sake of simplicity; so it can happen to have more than one rule for the same production.

To illustrate the idea suppose that in Alma's abstract grammar a *relational operation*, `rel_oper`, is defined by the 13th production:

```
p13:   rel_oper : exp exp
```

where `exp` is defined as:

```
p14:   exp : CONST
p15:   exp : VAR
p16:   exp : oper
```

To build a visual representation for that relational operation we need to distinguish two cases: the first occurs when the value of operand expressions is unknown (the `value` attributes are not yet instantiated); the second occurs when the value of the operands is known (that means that the expressions have been evaluated). So, different semantic cases of production `p13` will be represented by different figures, as shown in the example of figure 2 where we assume that the first expression (left operand) is a variable and the second expression (right operand) is a constant: the drawing on the top describes the first case, and the drawing below the second one. The visualizing rules to specify that mapping are written below.

```
vis_rule(p13) =
      <opr,a,c>,
      ((a.value=NULL) AND (c.value=NULL) AND
       (a.type=VAR) AND (c.type=CONST)),
      {drawRect(a.name),drawRect(),put(opr.name),
       put('?')}
```

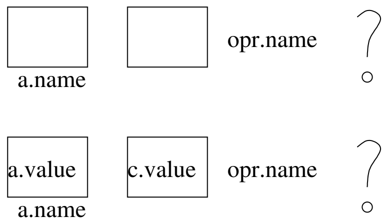

Figure 2. Visualization of a relational operation

```
vis_rule(p13) =
      <opr,a,c>,
      ((a.value!=NULL) AND (c.value!=NULL) AND
       (a.type=VAR) AND (c.type=CONST)),
      {drawRect(a.name,a.value),drawRect(c.value),
       put(opr.name),put('?')}
```

4.2. Visualization Algorithm

The visualization algorithm traverses the tree applying the visualizing rules to the sub-trees rooted in each node according to a bottom-up approach (postfix traversal). Using the production identifier of the root node, it obtains the set of possible representations; then a drawing procedure is selected depending on the constraint condition that is true.
The proposed algorithm is presented below.

```
visualize(tree){
     If not(empty(tree))
     then forall t in children(tree)
             do visualize(t);
          rules <- VRB[prodId(tree)];
          found <- false;
          while not(empty(rules)) and not(found)
             do r <- choice(rules);
                rules <- rules - r;
                found <- match(tree,r)
          If (found) then draw(tree,r);
}
```

5. Animation in Alma

Each rewriting rule (RR) specifies a state transition in the process of program execution; the results of applying the rule is a new DAST obtained by a semantic (may be also a syntactic) change of a sub-tree.

This systematic rewriting of the original DAST is interleaved with a sequence of visualizations producing an animation. A main function synchronizes the rewriting process with the visualization in a parameterized way, allowing for different views of the same source program.

5.1. Rewriting Rules

The RRB (Rewriting Rule Base) is a mapping that associates with each tree a set of tuples.

$$\text{RRB: DAST} \mapsto \text{set(cond} \times \text{newtree} \times \text{atribsEval)}$$

where each tuple has a matching condition, `cond`, a tree, `newtree`, which defines syntactic transformations and an attribute evaluation procedure, `atribsEval`, which defines the changes in the attribute values (semantic modifications).
The written form of each rewriting rule is as follows:

```
rule(ProdId)= <tree-pattern>,
              (condition),
              <NewProdId: newtree>,
              {attributes evaluation}

<tree-pattern> = <root, child_1, ..., child_n>
<newtree> = <root, child_1, ..., child_n>
```

In this specification, `condition` is a boolean expression (by default, evaluates to true) and `attributes evaluation` is a set of statements that defines the new attribute values (by default, evaluates to skip).
A rewriting rule can be applied to all the trees that are instances of the production `ProdId`. The new tree is also a production belonging to the same abstract grammar, so it will be specified by the new production identifier.
A `tree-pattern` associates variables to nodes in order to be used in the other fields of the rule specification: the matching condition, the new tree and the attribute evaluation. When a variable appears in both the `tree-pattern` (we call the left side of the RR) and the `newtree` (the so called right side of the RR), it means that all the information contained in that node, including its attributes will not be modified, i.e. the node is kept in the transformation as it is.
Notice that, although each RRB associates to a production a set of tuples, its written form, introduced above, only describes one tuple. So, it can happen to have more than one rule for the same production.
For instance, consider the following productions, belonging to Alma's abstract grammar, to define a conditional statement:

```
p8:     IF     : cond actions actions
p9:            | cond actions
```

The DAST will be modified using the following rules:

```
rule(p8) = <if,op,a,b>,
           (op.value=true),
```

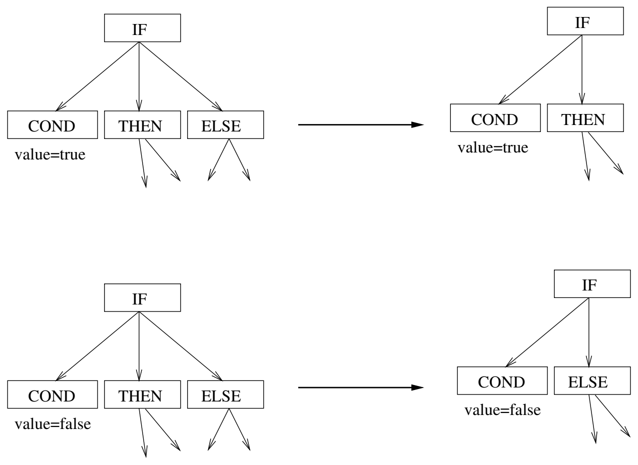

Figure 3. Conditional Statement Rewriting Rules

```
               <p9:if,op,a>,
               { }

   rule(p8) = <if,op,a,b>,
               (op.value=false),
               <p9:if,op,b>,
               { }
```

The graphical representation of these two RRs is shown in figure 3. The `THEN` and `ELSE` nodes represent the `actions` blocks.

5.2. Rewriting Algorithm

The rewriting process traverses the tree until a rewriting rule can be applied, or no more rules match the tree nodes (in that case, the transformation process stops). For each node, the algorithm determines the set of possible RR using its production identifier (`ProdId`) and evaluating the contextual condition associated with those rules. The DAST will be modified removing the node that matches the left side of the selected RR and replacing it by the new tree defined by the right side of that RR. This transformation can be just a semantic modification (only attribute values change), but it can also be a syntactic modification, (some nodes disappear or are replaced).
The rewriting algorithm follows:

```
DAST rewrite(tree){
   If not(empty(tree)) then
     rules <- RRB[prodId(tree)];
     found <- false;
     while not(empty(rules)) and not(found)
     do r <- choice(rules);
        rules <- rules - r;
        found <- match(tree,r)
     If (found)
     then tree <- change(tree,r)
     else a <- nextchild(tree)
          while not(empty(a)) and not(rewritten(a))
          do a <- nextchild(tree)
          If not(empty(a))
          then tree <- rebuild(tree,a,rewrite(a))
     return(tree)
}
bool rewritten(t){ return(t != rewrite(t))}

DAST rebuild(t,a,b){ t.a <- b; return( t ) }

DAST change(tree,<<maptree>,<cond>,
                 <newtree>,<eval>>){
     instantiate(maptree,tree)
     build(newtree,maptree)
     evaluate(newtree,eval)
     return(newtree)
}
```

5.3. Animation Algorithm

The main function defines the animation process, calling the visualizing and the rewriting processes repeatedly. The simplest way consists in redrawing the tree after each rewriting, but the sequence of images obtained can be very long and may be not the most interesting. So the grain of the tree redrawing is controlled by a function, called bellow `shownow()`, that after each tree syntactic-semantic transformation decides if it is necessary to visualize it again; the decision is made taking into account the internal state of the animator (that reflects the state of program execution) and the value of user-defined parameters.
The animation algorithm, that is the core of Alma's *back-end*, is as follows:

```
animator(tree){
        visualize(tree);
        Do     rewrite(tree);
               If shownow()
               then visualize(tree);
        until (tree==rewrite(tree))
 }
```

When no more rules can be applied, the output and input of the rewrite function are the same.

6. An example

To illustrate Alma system and the visualization and animation processes (mappings and algorithms) discussed in the previous subsections, consider the following extract of a Pascal program:

```
...
read(a);
read(b);
if(a>b) then a:=a-b
        else write(b/2)
...
```

Consider also the productions of Alma's internal abstract grammar bellow (we only show the subset that will be used in the example above):

```
p1:    statements      : stat statements
p2:                    | stat
p3:    stat            : IF
p4:                    | WHILE
p5:                    | ASSIGN
p6:                    | READ
p7:                    | WRITE
p8:    IF              : cond actions actions
p9:                    | cond actions
p10:   WHILE           : cond actions
p11:   cond            : rel_oper
p12:   actions         : statements
p13:   rel_oper        : exp exp
p14:   exp             : CONST
p15:                   | VAR
p16:                   | oper
p17:   oper            : exp exp
p18:   ASSIGN          : VAR exp
p19:   READ            : VAR
p20:   WRITE           : VAR
p21:                   | oper
```

Applying the visualization algorithm and the VRB with visualizing rules like the following:

```
vis_rule(p17) =
       <op,a,b>,
       ((a.value=NULL) AND (b.value=NULL) AND
        (a.type=VAR) AND (b.type=VAR)),
       {drawRect(a.name),drawRect(b.name),
        right_arrow(op.name)}
vis_rule(p17) =
       <op,a,b>,
       ((a.value!=NULL) AND (b.value!=NULL) AND
        (a.type=VAR) AND (b.type=VAR)),
       {drawRect(a.name,a.value),drawRect(b.name,
        b.value),right_arrow(op.name)}
```

to the DAST obtained from the given Pascal program and abstract grammar, and then rewriting the DAST using the proposed algorithm and a RRB containing rewriting rules as

```
rule(p17) = <op1,a,b>,
            ( ),
            <p17:op2,a,b>,
            {op2.value=op1.name(a.value,b.value)}
```

```
rule(p18) = <at,a1,b>,
            ( ),
            <p18:at,a2,b>,
            {a2.name=a1.name;
             a2.value=b.value}
```

we obtain an animation of that program; some of the tree pictures belonging to the animation can be seen in figure 4.

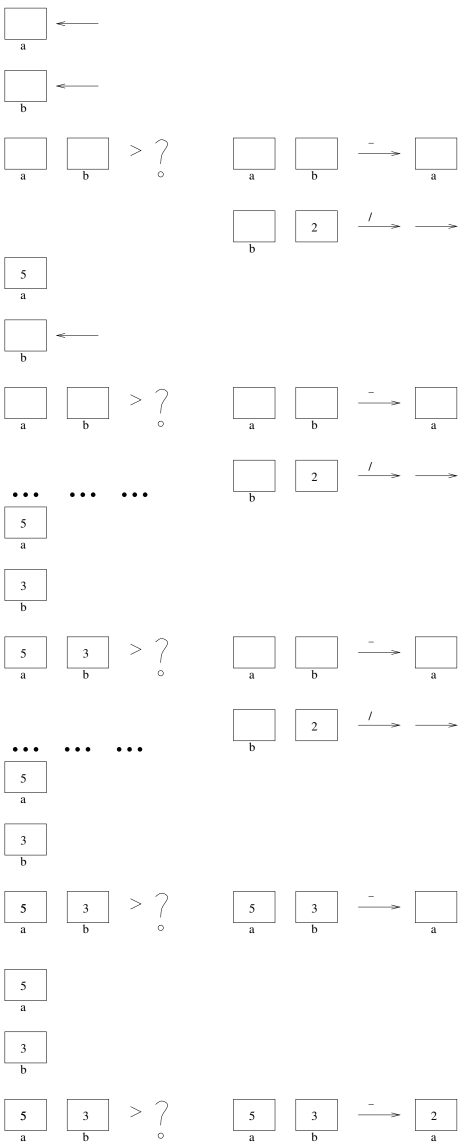

Figure 4. 5 Steps of the Animation of a Pascal Program.

7 Cases Studies

As a first case study, we chose a classical, but very short and simple, programming language that allows integer variables declaration and includes atribution and input/output statements (with integer expressions).
After the complete definition of Alma's internal abstract language (in the previous section we shown just a part of its grammar), we developed a *front-end* for it. Of course, as it is a procedural language, we did not find problems to map that language into the intermediate representation.
As a second case study, we have created a *front-end* for a domain specific language called *Timed State Charts* [AB96], [Var96] (TSC for short). Instead of a classical programming language, TSC is a specification language used to describe the behaviour of dynamic systems based on transition state machines.
The lesson learned from that experiment was that we could express the translation of TSC into Alma's intermediate representation without any problems. So, a source text in TSC will be translated into a DAST as a classical imperative language!

8. Conclusion

In this paper we discussed how to build a generic visualization mapping that associates figures to tree nodes in order to create a visual representation for a program. We also presented another mapping associating semantic rewriting rules to tree nodes to simulate program execution by state transitions. At last, we proposed a general way of combining both mappings and algorithms to animate a program, without need for extra annotation or use of visual data types.
At moment the architecture of the system is defined according to the design goals derived from the review of existing animators; the rewriting and drawing algorithms are created; and we have specified (in textual and graphic forms) both rules bases for procedural and alike languages.
To proceed into the implementation we decided to rely upon LISA tool. LISA system is a generic and interactive environment for programming language development. From a formal specification (attribute grammar) of a particular programming language, LISA produces a language specific environment that includes a language-oriented editor, a compiler/interpreter and other graphic debugging tools.
This generator receives an attribute grammar, and produces an internal representation from which generates a scanner, a parser and an attribute evaluator that are the components of a new compiler for the language described by the input grammar.
At a first step we used LISA to create (generate automatically) the *front-end*'s of Alma for 2 different languages, a classical imperative language and another one based on temporal state machines to specify dynamic systems. For each case, we tested the generation of the DAST for different source programs of the two ellected languages.
The next step (at moment, under development) will be the reuse of LISA to implement Alma's *back-end*, the module that must recognize the internal representation created by LISA generated *front-end*'s and produce the animation.
As LISA, and the generated compilers, are implemented in Java following an object-oriented approach, it was not difficult to find and understand the structures and functions used to process the given attribute grammar and the specific source programs. So the coding of structures and algorithms needed to implement the *back-end* seems to be straight forward.

8.1. Future Developments

In the near future we intend to conclude the implementation of the *back-end*. We are advised that we will have two major tasks to perform: add a large number of rewriting rules and visualization rules to the *back-end* in order to cover the procedural semantics (logical and functional semantics later); study the generation of several views for the same program, controlling the level of detail for each visualization.

References

[AB96] J. Armstrong and L. Barroca. Specification and verification of reactive system behaviour: The railroad crossing example. *Real Time Systems*, 1996.

[Ber91] Yves Bertot. Occurences in debugger specifications. In *PLDI91*, 1991.

[BNR97] M. H. Brown, M. A. Najork, and R. Raisamo. A java-based implementation of collaborative active textbooks. In *VL'97 - IEEE Symposium on Visual Languages*, pages 376–384. IEEE, September 1997.

[BS84] M. H. Brown and R. Sedgewick. A system for algorithm animation. In *SIGGRAPH'84*, volume 18, pages 177–186, Minneapolis, July 1984. ACM Computer Graphics.

[CBC96] Paul Carlson, Magaret Burnett, and Jonathan Cadiz. A seamless integration of algorithm animation into a visual programming language. In *AVI'96 - International Workshop on Advanced Visual Interfaces*. acm, May 1996.

[CkCJ92] Roman G. C., Cox k., Wilcox C., and Plun J. Pavane: A system for declarative visualization of concurrent computations. *Journal of Visual Languages and Computing*, 3(1):161–193, 1992.

[Dui98] R. A. Duisberg. Animation using temporal constraints: An overview of the animus system. *Human-Computer Interaction*, 3(3):275–307, August 1998.

[HHR89] E. Helttula, A. Hyrskykari, and K. Raiha. Graphical specifications of algorithm animations with aladdin. In *22nd Hawaii International Conference on System Sciences*, January 1989.

[HPS+97] J. Haajanen, M. Pesonius, E. Sutien, T. Terasvirta, P. Vanninen, and J. Tarhio. Animation of user algorithms in the web. In *VL'97 - IEEE Symposium on Visual Languages*, pages 360–368. IEEE, September 1997.

[Jen96] Kurt Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*, volume 1. Springer-verlag, 2 edition, 1996.

[LF95] H. Lieberman and C. Fry. Bridging the gap between code and behavior in programming. In *ACM Conference on Computers and Human Interface*, Denver, Colorado, April 1995.

[MLAZ00] Marjan Mernik, Mitja Lenic, Enis Avdicausevic, and Viljem Zumer. Compiler/interpreter generator system lisa. In *IEEE Proceedings of 33rd Hawaii International Conference on System Sciences*, 2000.

[MM88] A. J. Mendes and Teresa Mendes. Vip - a tool to visualize programming examples. *Education and Application of Computer Technology*, 1988.

[Moh88] T. G. Moher. Provide: A process visualization and debugging environment. In *IEEE Transactions on Software Engineering*, volume 14, pages 849–857, June 1988.

[MRRT99] Boris Melamed, Michael Rudolf, Olga Runge, and Gabriele Taentzer. The attributed graph grammar system - homepage. http://tfs.cs.tu-berlin.de/agg/, 1999.

[MS93] S. Mukherjea and J. T. Stasko. Applying algorithm animation techniques for program tracing, debugging, and understanding. In *15th International Conference on Software Engineering*, pages 456–465, Baltimore, May 1993.

[Rei85] Steven Reiss. Pecan: Program development systems that support multiple views. *IEEE Transactions on Software engineering*, 1985.

[Rei87] Steven Reiss. Working in the garden environment for conceptual programming. *IEEE Software*, 1987.

[Rei90] Steven Reiss. Interacting with the field environment. *Software Practice and Experience*, 1990.

[Sar99] João Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Utrecht University, 1999.

[SDBP97] John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price. *Software Visualization - Programming as a Multimedia Experience*. The MIT Press, 1997.

[Sta90] John T. Stasko. Simplifying algoritm animation with tango. In *IEEE Workshop on Visual Languages*. IEEE, October 1990.

[Var96] Maria João Varanda. Concepção, especificação de uma linguagem visual. Master's thesis, Universidade do Minho, 1996.