Automatic Generation of Language-based Tools using LISA system

Pedro Rangel Henriques [a,1] Maria João Varanda Pereira [b,2,*]
Marjan Mernik [c,3] Mitja Lenič [c]

[a] *University of Minho, Department of Informatics, Portugal*

[b] *Polytechnic Institute of Bragança, Portugal*

[c] *University of Maribor, Faculty of Electrical Engineering and Computer Science, Slovenia*

Abstract

Many tools have been built in the past years, based on the different formal methods and processing different parts of language specification, such as: scanner generators, parser generators and compiler generators. The automatic generation of a complete compiler was the primary goal of such systems. However, researchers soon recognized the possibility that many other language-based tools could be generated from formal language specifications. In the paper language-based tools generated automatically using LISA system are described. Examples of LISA generated tools among compilers/interpreters are editors, analyzers, and visualizers/animators. In the paper, special emphasis is given to the latter because, to our knowledge, none of the existing compiler generators are able to automatically generate visualizers/animators.

Key words: language theory, compiler generation, language-based tools, visualizers/animators

* Corresponding author.

Email addresses: prh@di.uminho.pt (Pedro Rangel Henriques), mjoao@ipb.pt (Maria João Varanda Pereira), marjan.mernik@uni-mb.si (Marjan Mernik), mitja.lenic@uni-mb.si (Mitja Lenič).

[1] The paper is an extension of already published paper at Workshop on Language Description, Tools and Applications, ENTCS 65, No. 3, 2003

[2] The work of M. João is partially supported by the Portuguese program PRODEP, acção 5.2 da medida 5 – doutoramentos

[3] The project was supported by Slovenian and Portugal governments under the contract SLO-P-11/01-04

1 Introduction

The advantages of formal specification of programming language semantics are well known. First, the meaning of a program is precisely and unambiguously defined; second, it offers a unique possibility for automatic generation of compilers or interpreters. Both these factors contribute to the improvement of programming language design and development. The programming languages that have been designed with one of the various formal methods have a better syntax and semantics, less exceptions and are easier to learn. Moreover, from formal language definitions many other language-based tools can be automatically generated, such as: pretty printers, syntax-directed editors, type checkers, dataflow analyzers, partial evaluators, debuggers, profilers, test case generators, visualizers, animators, documentation generators, etc. For a more complete list see [1]. In most of these cases the core language definitions have to be augmented with the tool-specific information. In other cases, just a part of formal language definitions is enough for automatic tool generation or implicit information must be extracted from the formal language definition in order to automatically generate a tool.

The goal of the paper is twofold. On one hand we discuss some of the tools in this last case, like editors to help in writing sentences of the language and various inspectors (such as automata visualizers, syntax tree visualizers, semantic evaluator animators) that are helpful for a better understanding of the language analysis process. Those examples have all been incorporated in the compiler generator system LISA [2]. On the other hand we present some extensions to the language definitions in a manner to make automatic generation of an algorithm animator and program visualizer possible.

Program visualizers/animators are very useful tools for deeper and clearer understanding of algorithms. As such, they are very valuable for programmers and students. Currently, algorithm animators and program visualizers are strongly language and algorithm-oriented, and they are not developed in a systematic or automatic way. In this paper we aim to show that animators could be also automatically generated from extended language definitions. We will briefly propose a specific solution for the development of such a tool, the Alma system, discussing its architecture and its implementation. The system has a front-end specific for each language and a generic back-end, and uses a decorated abstract syntax tree (DAST) as the intermediate representation between them. In the implementation of the Alma system the language development system LISA is used twice. It generates the front-end for each new language, and some parts of it (Java classes) are reused to build the back-end.

The organization of the paper is as follows. In section 5 related work is described. Language-based tools that are automatically generated by the LISA

system are described in section 3. The design and implementation of the Alma system are described in section 4. A synthesis and concluding remarks are presented in section 6.

2 Background

Let us start with some standard definitions about languages [3] that make automatic implementation of programming languages and language-based tools possible. An *alphabet* Σ is a finite nonempty set of symbols, which are assumed to be indivisible. A *string* over an alphabet Σ is a finite sequence of symbols of Σ. A set of all strings over an alphabet Σ is denoted Σ^* and a set of all nonempty strings over Σ is denoted Σ^+. For any alphabet Σ, a *language* over Σ is a subset of Σ^*.

A *context-free grammar* G is a quadruple (V, T, P, S), where V is a set of non-terminal symbols, T is a set of terminal symbols with $T \subseteq \Sigma^*$, $T \cap V = \emptyset$, the relation $P \subseteq V \times (V \cup T)^*$ is a finite set of production rules and S the start symbol with $S \in V$. The production of the form $A \rightarrow \alpha$ means A derives α, where $A \in V$ is a non-terminal symbol and $\alpha \in (V \cup T)^*$ a string of terminal and non-terminal symbols. A *sentential form* of G is any string of terminal and non-terminal symbols. Let γ_1 and γ_2 be two sentential forms of grammar G, then γ_1 *directly derives* γ_2, denoted $\gamma_1 \Rightarrow \gamma_2$, if $\gamma_1 = \sigma\alpha\tau$, and $\gamma_2 = \sigma\beta\tau$, and $\alpha \rightarrow \beta$ is a production in P. Let γ_1 and γ_2 be two sentential forms of a grammar G, then γ_1 *derives* γ_2, denoted $\gamma_1 \Rightarrow^* \gamma_2$, if there is a sequence of (zero or more) sentential forms $\sigma_1, \ldots, \sigma_n$ such that $\gamma_1 \Rightarrow \sigma_1 \ldots \Rightarrow \sigma_n \Rightarrow \gamma_2$.

The *context-free language* $L(G)$ produced from grammar G is the set of all strings consisting only of terminal symbols that can be derived from the start symbol S by sequential application of production rules, $L(G) = \{x | S \Rightarrow^* x, x \in T^*\}$. The membership problem, given a string $x \in T^*$ does it belong to $L(G)$, is decidable for any context-free grammar. This has very practical implications, since it enables us to check if the computer program is grammatically (syntactically) correct. To specify the semantics of programming languages, context-free grammars need to be extended. Attribute grammars [4] are a generalization of context-free grammars in which each symbol has an associated set of attributes that carry semantic information, and with each production a set of semantic rules with attribute computation is associated. An attribute grammar consists of:

- A context-free grammar G.
- A set of attributes $A(X)$ for each non-terminal symbol $X \in N$. $A(X)$ is divided into two mutually disjoint subsets, $I(X)$ of inherited attributes and $S(X)$ of synthesized attributes. Now set $A = \bigcup A(X)$.

- A set of semantic rules R. Semantic rules are defined within the scope of a single production. A production $p \in P, p : X_0 \rightarrow X_1 \ldots X_n$ $(n \geq 0)$ has an attribute occurrence $X_i.a$ if $a \in A(X_i)$, $0 \leq i \leq n$. A finite set of semantic rules R_p is associated with the production p with exactly one rule for each synthesized attribute occurrence $X_0.a$ and exactly one rule for each inherited attribute occurrence $X_i.a, 1 \leq i \leq n$. Thus R_p is a collection of rules of the form $X_i.a = f(y_1, \ldots, y_k), k \geq 0$, where $y_j, 1 \leq j \leq k$, is an attribute occurrence in p and f is a semantic function. In the rule $X_i.a = f(y_1, \ldots, y_k)$, the occurrence $X_i.a$ depends on each attribute occurrence $y_j, 1 \leq j \leq k$. Now set $R = \bigcup R_p$. For each production $p \in P, p : X_0 \rightarrow X_1 \ldots X_n$ $(n \geq 0)$ the set of defining occurrences of attributes is $DefAttr(p) = \{X_i.a | X_i.a = f(\ldots) \in R_p\}$. An attribute $X.a$ is called synthesized $(X.a \in S(X))$ if there exists a production $p : X \rightarrow X_1 \ldots X_n$ and $X.a \in DefAttr(p)$. It is called inherited $(X.a \in I(X))$ if there exists a production $q : Y \rightarrow X_1 \ldots X \ldots X_n$ and $X.a \in DefAttr(q)$.

Therefore, an attribute grammar is a triple $AG = (G, A, R)$ which consists of a context-free grammar G, a finite set of attributes A and a finite set of semantic rules R. Attribute grammars have proved to be very useful in specifying the semantics of programming languages, in automatic constructing of compilers/interpreters, in specifying and generating interactive programming environments and in many other areas [5].

3 Tools from language definitions generated by the LISA system

LISA is a compiler-compiler, or a system that generates automatically a compiler/interpreter from attribute grammar based language specifications. The syntax and semantics of LISA specifications and its special features ("templates" and "multiple attribute grammar inheritance") are described in more detail in [6]. The use of LISA in generating compilers for real programming languages (e.g. PLM, AspectCOOL and COOL, SODL) are reported in [7], [8], [9].

To illustrate LISA style, the specification of a toy language—*Simple Expression Language with Assignments, SELA*—is given below. From these descriptions LISA automatically generates a SELA compiler/interpreter.

```
language SELA {
   lexicon
         {
                  Number     [0-9]+
                  Identifier [a-z]+
                  Operator   \+ | :=
                  ignore     [\0x09\0x0A\0x0D\  ]+
         }
   attributes Hashtable *.inEnv, *.outEnv;
```

```
            int *.val;

   rule Start
         {     START ::= STMTS compute
                  {     STMTS.inEnv = new Hashtable();
                         START.outEnv = STMTS.outEnv;
                  };
         }
   rule Statements
         {     STMTS ::= STMT STMTS compute
                  {     STMT.inEnv = STMTS[0].inEnv;
                         STMTS[1].inEnv = STMT.outEnv;
                         STMTS[0].outEnv = STMTS[1].outEnv;
                  }
                | STMT  compute
                  {     STMT.inEnv = STMTS[0].inEnv;
                         STMTS[0].outEnv = STMT.outEnv;
                  };
         }
   rule Statement
         {     STMT ::= #Identifier \:= EXPR compute
                  {     EXPR.inEnv = STMT.inEnv;
                         STMT.outEnv = put(STMT.inEnv,
                         #Identifier.value(), EXPR.val);
                  };
         }
   rule Expression
         {     EXPR ::= EXPR + EXPR compute
                  {     EXPR[2].inEnv = EXPR[0].inEnv;
                         EXPR[1].inEnv = EXPR[0].inEnv;
                         EXPR[0].val = EXPR[1].val + EXPR[2].val;
                  };
         }
   rule Term1
         {     EXPR ::= #Number compute
                  {     EXPR.val = Integer.valueOf(
                                     #Number.value()).intValue();
                  };
         }
   rule Term2
         {     EXPR ::= #Identifier compute
                  {     EXPR.val = ((Integer)EXPR.inEnv.get(
                                    #Identifier.value())).intValue();
                  };
         }
   method Environment
         {
                import java.util.*;
                public Hashtable put(Hashtable env, String name, int val)
                  {
                         env = (Hashtable)env.clone();
                         env.put(name, new Integer(val));
                         return env;
                  }
         }
}
```

Besides that, LISA derives other tools (Figure 1. In the following subsections three families of such tools are described: *editors* to help the final users in the creation and maintenance of the sentences of the specified language, i.e., the *source texts* (or source programs) that he wants to process (compile/interpret); *inspectors* that are useful to understand the behavior or debug the generated language processor itself (compiler/interpreter); and *visualizers/animators*,

Generated tool	Purpose
Editors	Easier creation and maintenance of programs written in newly specified langauge
Inspectors	Useful to understand the behavior of generated language compiler/interpreter
Visualizers/Animators	Useful to understand the meaning of programs written in newly specified langauge

Table 1
LISA generated language-based tools

similar to inspectors, useful to understand the meaning of the source program that is being processed.

Automatic generation is possible whenever a tool can be built from a fixed part and a variable part; and also the variable part, language dependent, has to be systematically derivable from the language specifications. That part has a well defined internal representation that can be traversed by the algorithms of the fixed part. Table 2 summarizes those parts for some of the language-based tools generated by the LISA system. It is not the aim of this paper to describe all those algorithms (many of them are described in [10]), except the most interesting ones for program visualizations and animations. They are described in detail in section 4.

3.1 Editors

Two different LISA generated language oriented editors, that is editors that are sensitive to the language lexicon/syntax, are described in this section.

3.1.1 Language Knowledgeable Editors

LISA generates a language knowledgeable editor, which is a compromise between text editors and syntax-directed editors, from formal language specifications. The LISA generated language knowledgeable editor is aware of the regular definitions of the language lexicon (see table 2). Therefore, it can color the different parts of a program (comments, operators, reserved words, etc.) to enhance understandability and readability of programs. In Figure 1 operators in SELA programs are recognized while editing and displaying in a different colour.

Generated tool	**Formal specifications**	**Fixed part**	**Variable part**
Lexer	regular definitions	algorithm which interprets action table	action table: $State \times \Sigma \rightarrow State$
Parser (LR)	BNF	algorithm which interprets action table and goto table	action table: $State \times T \rightarrow Action$ goto table: $State \times (T \cup N) \rightarrow State$
Evaluator	Attribute Grammars (AG)	tree walk algorithm	semantic functions
Language knowledgeable editor	regular definitions (extracted from AG)	matching algorithm	same as lexer
Structure editor	BNF (extracted from AG)	incremental parsing algorithm	same as parser
FSA visualization	regular definitions (extracted from AG)	FSA layout algorithm	same as lexer
Syntax tree visualization	BNF (extracted from AG)	syntax tree layout algorithm	syntax tree
Dependency graph visualization	extracted from AG	DG layout algorithm	dependency graph
Semantic evaluator animation	extracted from AG	semantic tree layout algorithm	decorated syntax tree & semantic functions
Program visualization and animation (ALMA)	additional formal specifications	visual and rewrite rules & visualization, rewriting and animation algorithm	decorated abstract tree (DAST)

Table 2
Fixed and variable parts of LISA generated language-based tools

3.1.2 Syntax-directed Editors

Syntax-directed editors are editors which are aware of the language syntax of edited programs. They help users to write syntactically correct programs before they are actually compiled, exhibiting that structure and/or inserting directly the keywords at the right places (the user only has to fulfill the variable parts of his text). A Structure Editor is a kind of syntax-directed editor, where the syntax structure of written programs are explicitly seen while editing the program (see Figures 2 and 3).

Fig. 1. Language knowledgeable editor

Fig. 2. Structure editor

3.2 Inspectors for Language Processors

Four different LISA generated inspectors are introduced in this subsection. Inspectors are useful in better understanding how automatically generated language compiler/interpreter works.

3.2.1 Finite State Automaton Visualization

With the help of visual representation of directed graphs it becomes clear how complex automata can be specified with simple regular expressions and how some simple automata require a complex regular expression (like comments in C). It is also possible to determine the conflicts in specifications and resolve them using FSA visualization. In Figure 4 a finite state automaton of the

Fig. 3. Syntax tree view

Fig. 4. FSA view

SELA language is presented.

3.2.2 Syntax Tree Visualization

The basic understanding of a compiler is based on the understanding of the parsing procedure. This tool is a graphical browser for the syntax tree built

Fig. 5. Dependency graph view

by the LISA generated compiler after parsing a given source program. With this kind of visualizations it is possible to measure the impact of different grammar specifications on the shape and size of the tree, and assess their effect on compiler implementation; in that way it is possible to design a better syntax and thereby an easier and understandable semantics.

Figure 3 illustrates the output of this tool: a part of the syntax tree selected by the user while editing a program.

3.2.3 Dependency Graph Visualization

As attribute grammars are specified on the declarative level, the order of attribute evaluation is determined by the compiler construction tool. But that sequence is also important for language designer to understand the actual evaluation order.

Even more relevant is the detection of cycles on a grammar; if the attribute dependencies induce indirect cycles, they can be easily discovered with the aid of visual representations.

In Figure 5 an augmented dependency graph, drawn by the LISA generated tool for the 5th SELA production, is presented.

3.2.4 Semantic Evaluator Animation

In attribute grammars a set of attributes carrying semantic information is associated with each nonterminal. For example, attributes `inEnv` and `val` are associated with nonterminal `EXPR` (see SELA language specifications). In

Fig. 6. Semantic Evaluation view

the evaluation process the value of these attributes has to be computed. The semantic analysis is better understood by animating the visits to the nodes of the semantic tree, and the evaluation of attributes in these nodes; Figure 6 shows a snap-shot of the animation process. Therefore, the animation of the evaluation process is also very helpful in the debugging process. Users can also control the execution by single-stepping and setting the breakpoints.

3.3 Program Visualization and Animation

Another instance of tools that can be derived from formal language specification are *program visualizers/animators*. The purpose of such a family of tools is to help the programmer to inspect the data and control flow of a source program—static view of the algorithms realized by the program (visualization) —and to understand its behaviour—dynamic view of the algorithms' execution (animation). Such a tool can be obtained by the specialization of a generic visualizer/animator (a language-independent back-end) providing an extension to the LISA attribute grammar that specifies the language to be analyzed. The AG extension just defines the way the input sentence should be converted into the animator's internal representation (DAST)—see Figure 7.

Below is an example of such an extension for the SELA language, introduced in the previous subsections.

```
import  "AlmaLib.lisa";
```

Fig. 7. Animator generation from LISA specification

```
language  AlmaSELA  extends SELA,  AlmaBase  {
   rule extends  Start
         {
                START ::= STMTS compute
                  {     ALMA_ROOT<START,STMTS>
                         ALMA_TAB<START,STMTS>
                  };
         }
   rule  extends Statements
         {     STMTS ::= STMT STMTS compute
                  {     ALMA_STATS<STMTS, STMT,  STMTS[1]>
                  }
                | STMT  compute
                  {     ALMA_IDENT<STMTS, STMT>
                  };
         }
   rule  extends Statement
         {     STMT ::= #Identifier := EXPR compute
                  {
                         ALMA_ASSIGN<STMT,ALMA_VAR(#Identifier), EXPR>
                  };
         }
   rule extends  Expression
         {     EXPR ::= EXPR + EXPR compute
                  {
                         ALMA_OPER<EXPR[0],EXPR[1], EXPR[2], "+">
                  };
         }
   rule  extends Term1
         {     EXPR ::= #Number compute
                  {
                         ALMA_CONST<EXPR,#Number>
                  };
         }
   rule extends  Term2
         {     EXPR ::= #Identifier compute
                  {
                         ALMA_VAR<EXPR,#Identifier>
                  };
         }
}
```

The extension shown above illustrates the use of *templates*, and *multiple attribute grammar inheritance* that are both standard LISA features [6]. It is used to specify the attribute evaluation related to the DAST construction; it

Fig. 8. Visualization generated by the animator

assumes the syntax and semantics specified in SELA attribute grammar and adds new computing statements just to build the internal representation used by Alma. To write in a clear and concise way those statements, we use templates. Each template is specified as:

```
template<attributes X_in,Y_in>
compute ALMA_ROOT
{   X_in.dast=new Alma.CRoot(Y_in.tree);
}
```

The method CRoot is one of pre-defined methods inherited from AlmaBase and is used to construct the DAST nodes. The attribute tree is used to collect the nodes of the DAST and then all the tree will be in dast attribute.

From this specification we generate a parser and a translator that converts each input text into an abstract representation used by the animator, common to all different source languages. That processor, we call it the animator's *front-end*, is the language dependent component of the tool. In this case, its fixed part is more complex than in the cases studied in previous subsections 3.1 and 3.2: it is not just a standard algorithm (we use three language independent algorithms), but it requires also two standard data structures (a visual rule base, and a rewriting rule base).
NEWmarjan
In the DAST used in Alma each node is related with concepts involved in the source program. The visualization of these concepts will allow to understand the program. The DAST is language-independent. The Alma project is an independent work and its approach can be implemented in any compiler compiler tools. In this case it is developed to work over LISA system.
NONEWmarjan

So, taking a source program in SELA:

```
a:=2+5
b:=a+3
c:=a+b
```

these algorithms can generate a visualization like the one that can be seen in Figure 8. Take the picture as an example because the final layout (drawings used) can be modified by the Alma designer. Drawing procedures called by the visualizing rules can easily be changed.

NEWmarjan
The system allows the visualization of data structures and it can also cope with

procedures and objects.
NONEWmarjan

The system, to be discussed in the next section 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. A general overview of Alma's specifications and structures, and its relation to a LISA generated compiler is provided in Figure 7.

4 Alma Implementation

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. NEWmarjan
The users of Alma system are people who want to quickly visualize a program without been worried about the semantics of the program. By other hand,a person that has a program to be animated can not be worried about where the program must be annotated, which is the syntax of those annotations or what can be watched in this program in order to understand it. Many concepts are common to many languages. So, using some rewriting and visualizing rules written for the main concepts involved, we can automatically generate animation of many many programs. We only have to identify the concepts to be visualized.
The rewriting and visualizing rules are already constructed for those concepts, what allows to automatic generate animations without modifying the source text neither specify the semantics of the program.
NONEWmarjan

4.1 Alma Architecture

To comply with the requirements above, we conceived the architecture shown in Figure 9.

In Alma we also use a DAST as an internal representation for 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. We include in appendix A that grammar to formalize the type (structure, and attributes) of DAST nodes. 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

Fig. 9. Architecture of Alma system

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 tree, creates visual representations of nodes, gluing figures in order 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.
NEW
Different visualizations can be generated from the same DAST depending on the visualization rules. Different *front-end* (one for each source language) generate nodes from the internal representation of Alma. These nodes will constitute the DAST of the source program.
NONEW

4.1.1 Visualization in Alma

The visualization is achieved applying visualizing rules (VR) to DAST subtrees; those rules define a mapping between trees and figures. When the partial figures corresponding to the nodes of a given tree are assembled together, we obtain a visual representation for the respective program.

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 template, `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 VRB 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
```

A visual representation for that relational operation could be as the one shown in figure 10.

Fig. 10. Visualization of a relational operation

The visualizing rule to specify that mapping are written below.

```
vis_rule(p13) =
      <opr,a,c>,
      ((a.type=exp) AND (c.type=exp)),
```

```
{drawRect(a.name,a.value),drawRect(c.name,c.value),
 put(opr.name),put('?')}
```

NEW
This visualization rule is applied to three node tree: an operator and two operands. Each operand can be a `CONST`, `VAR` or `oper` (another operation). For each operand a rectangle is constructed with its value inside and its name as a label of the rectangle. Then the operation is drawed and a `?` which says that is a relational operation.
NONEW

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 (post-fix 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 first constraint condition that is true.
The 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); }
```

NEWmarjan
A program animation could not be code visualization: it depends on the granularity of the visualization rules. The DAST is an abstract representation of the source code and we can apply a visualization rule to each node of the tree (getting a more detailed visualization, usually an operational view like a debugger) or we can apply a visualization rule to a set of nodes or even to the root. In this last case, the animation can be very different from a debugger output, it's more abstract.
We assume that Alma system can have the same problems as other systems that use visual languages. We can have scalability problems and we must care about the drawings been used in order to get a visualization that really helps program understanding.
NONEWmarjan

4.1.2 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.

Rewriting Rules

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

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

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>,
              {attribute evaluation}
```

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

In this template, `condition` is a boolean expression (by default, evaluates to true) and `attribute 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`. 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` (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),
           <p9:if,op,a>,
           { }


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

Rewriting Algorithm

The rewriting algorithm is also a tree-walker that 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:

```
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) }
```

Fig. 11. Architecture of LISA system

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 not be the most interesting. So the grain of the tree redrawing is controlled by a function, called below `shownow()`, that after each tree's 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:

```
animate(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.

4.2 Reusing LISA in Alma implementation

LISA itself, and the generated compilers, are implemented, in the programming language Java, following an object-oriented approach—see Figure 11 to get a general picture of LISA architecture integrated with all its generated tools. It was very easy to identify and understand the data structures and functions used by LISA system and tools to process a given attribute grammar specification or a source programs—they are properly encapsulated in classes, as attributes and methods. Therefore, the coding of data structures and algorithms needed to implement Alma became straightforward, due to the reuse of some of the referred classes.
A global view of Alma implementation is provided in Figure 12. To build the DAST—that is the output of Alma's *front-end* (generated by LISA), and the input of Alma's

Fig. 12. Connection between LISA system and Alma system

back-end—the developer of the AG specification shall call some specific methods (provided in Alma's standard library) to create a new tree node for each symbol of the DAST abstract language and to collect the trees associated with its children. To implement those methods, we just reused the Java classes `CTreeNode`, `CSyntaxTree`, `CParseSymbol`, used by LISA to create its internal tree representations.
As an immediate consequence, all the facilities provided in the LISA environment to manipulate those trees became available to process the DAST.
Those classes were reused once again to build the maps VRB and RRB necessary for the implementation of the *back-end* (visualization and animation) algorithms, as described in subsections 4.1.1 and 4.1.2—remember that every (visualizing or rewriting) rule in both maps is defined in terms of tree-patterns.

The *back-end* itself is another Java class, specially developed for that purpose, that reuses the data structures and implements directly the algorithms shown in the previous subsections. To code that class, we kept the OO approach followed in LISA development.
To code the main class of Alma, it should contain the necessary methods to call and synchronize the animator's *front-end* and *back-end* functions, we simply reused and adapted the standard Java class, `Compile.java`, made available in LISA library as the main class for its generated compilers.

4.3 Alma animation examples

In this section, some Alma animation examples will be presented. `SELA` language, already introduced in section 3 and used in all examples of this paper, will be improved in order to show the animation of conditional and repetitive statements.
Now, each statement can be an assignment as in the first version of SELA language, but it can also be a conditional or repetitive statement, or a reading/writing statement. In this new language we can have several kinds of arithmetic and relational

Fig. 13. Example source program

operators.
Two program examples written in this language will be presented. For each one some images of the generated animation will be shown and also the visualization of the DAST generated by the front-end of Alma.

4.3.1 An example

The example presented in figure 13 has an assignment, a repetitive statement, a reading and a writing statement. The figure 14 gives a visualization of the DAST generated by the front-end of Alma. The figures 15, 16, 17, and 18 show several visualizations belonging to Alma animation. The first one represents the initial state of the program; the second and third one shows the first and second iteration of the cycle; in the fourth one the cycle is finished and the last one represents the final state of the program.
Notice that symbol `--->` represents an assignment or an operation (if an arithmetic symbol is under the arrow); the symbol `#` represents a conditional statement; the symbol `@` represents a repetitive statement; the symbol `=~=>>` represents a write and `<<=~=` a read statement.
NEW
The litle window under each visualization its an overview of the execution trace and the circled part corresponds to the visualization that is shown.
NONEW

Fig. 14. Generated DAST for the example

Fig. 15. 1st visualization of the example

4.4 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 Compile 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 a 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 a variable part

Fig. 16. 7th visualization of the example

Fig. 17. 11th visualization of the example

(visualization and rewriting rules, nodes, etc). We conclude that the fixed part gives generality and the variable 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.

Fig. 18. 13th of the example

4.5 Other Alma features

NEW
Alma System can also cope with different languages, different level of animation detail, different types of visualizations and different types of paradigms.
NONEW

Different Languages
If we want to apply the system to a different source language, we only have 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.

Different level of animation detail
We 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. We will also access an interface where we 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. We 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 based on rules that map nodes to draws. If we want to change the draws in order to get a different visualization, we can modify rules or specify new ones. We 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 in 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 19.

Fig. 19. 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 20 that presents the less detailed version of the generated animation (the minimal number of steps are shown).
NEW
In figure 21 we show another kind of visualization for the same program. We get this visualization using other visualizing rules.
NONEW
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.
NEW
Although these examples, we assume that Alma will be used in small and domain specific languages for which there are no debugging tools neither any

Fig. 20. Alma generated animation

Fig. 21. Another Alma generated animation

kind of visualizers. Alma produces graphical representations that usually have problems of scalability and it's also very dificult to chose the appropriate drawings for better understanding. So, we are not discussing here the output quality or the system performance, we are just defending that we can automaticaly visualize different concepts and different languages using the same old approach: a DAST-based approach.
NONEW

5 Related Work

The development of the first compilers in the late fifties without adequate tools was a very complicated and time consuming task. For instance, the implementation of the compiler for the programming language FORTRAN took about 18 human years. Later on, formal methods, such as operational seman-

tics, attribute grammars, denotational semantics, action semantics, algebraic semantics, and abstract state machines, were developed. They made the implementation of programming languages easier and finally contributed to the automatic generation of compilers/interpreters. Many tools have been built in the past years, based on the different formal methods and processing different parts of language specification, such as: scanner generators, parser generators and compiler generators. The automatic generation of a complete compiler was the primary goal of such systems. However, researchers soon recognized the possibility that many other language-based tools could be generated from formal language specifications. Therefore, many tools not only automatically generate a compiler but also complete language-based environments. Such automatically generated language-based environments include editors, type checkers, debuggers, various analyzers, etc. For example, the FNC-2 [11] is an attribute grammar system that generates a scanner/parser, an incremental attribute evaluator, a pretty printer, a dependency graph visualizer, etc. The CENTAUR system [12] is a generic interactive environment which produces a language specific environment from formal specifications written in Natural Semantics, a kind of operational semantics. The generated environment includes a scanner/parser, a pretty printer, a syntax-directed editor, a type checker, an interpreter and other graphic tools. The SmartTools system [13], a successor of the CENTAUR system, is a development environment generator that provides a compiler/interpreter, a structured editor and other XML related tools. The ASF+SDF environment [14] generates a scanner/parser, a pretty printer, a syntax-directed editor, a type checker, an interpreter, a debugger, etc, from algebraic specifications. In the Gem-Mex system [15] the formal language is specified with abstract state machines. The generated environment includes a scanner/parser, a type checker, an interpreter, a debugger, etc. The LRC system [16] from high-order attribute grammar specifications generates incremental scanner/parser and attribute evaluators, syntax-directed editor, multiple views of the abstract semantic tree (unparsing windows), windows-based interfaces, etc. From the above description of various well known compiler/interpreter generators can be noticed that editors, pretty printers, and type checkers are almost standard tools in such automatically generated environments. To our knowledge none of the existing compiler generators automatically generate visualizers and animators for programs written in a newly specified language.

Searching for tools that produce some kind of animation in order to explain the semantics underlying a given program (to help the programmers reasoning about it), we found several interesting animators and visualization systems — for instance: BALSA [17]; TANGO [18]; JCAT [19]; ZSTEP [20]; JELIOT [21]; PAVANE [22] or LENS [23]. However, most of the well known animation systems are not general purpose. They just animate a specific algorithm (allowing or not the choice of some configuration parameters) or, if they accept a larger set of algorithms, the programs must be written in a specific language. Usually in that case, the programmer shall use special data types or procedures (visual

annotations) on the source code, which means that, the source program must be modified in order to be animated. The animators described in the literature are not constructed automatically from language specifications.

6 Conclusion

Many applications today are written in well-understood domains. One trend in programming is to provide software tools designed specifically to handle the development of such domain-specific applications in order to greatly simplify their construction. These tools take a high-level description of the specific task and generate a complete application.
One such well established domain is compiler construction, because there is a long tradition of producing compilers by hand, and because the underlying theory (supporting all the analysis phases, and even code generation and optimization processes) is well understood. At present, there exist many generators which automatically produce compilers or interpreters from programming language specifications.
As shown in this paper, not just standard compilers/interpreters can be generated automatically. Formal language specifications contain a lot of information from which many language-based tools, such as editors, type checkers, debuggers, visualizers, animators, can be generated. Sometimes the implicit information is enough, but in some other cases some extra data must be added. Concrete examples of both types, produced by the generator system LISA, were introduced and discussed along the article. We do not intend to discuss in this paper the actual performance of LISA, because our aim is to enhance the capabilities of the attribute grammar specifications; however our experiments allow us to say that execution time of the generated tools is completely acceptable when compared with similar programs.
The benefits of this approach to software development are many fold. On one hand, the developer just writes formal descriptions that are more concise and clear—they are faster to produce and easier to understand and maintain, because they are shorter; it is well known that thousands of lines of complex code are automatically obtained from a grammar definition written in just some dozens of lines. On the other hand, different tools can be obtained from the same specification, which is obviously a major advantage. Last but not least, a very good code (developed by experts in language processing methods and algorithms) is reused in the automatic generation process.
Discussing in detail the architecture and implementation of one particular tool, the program visualizer/animator Alma, we also proved that: (1) a grammatical approach to software engineering, supported by generators, is a nice way to develop applications (we systematically created a general animator instead of an algorithm or language dependent one); and (2) a modular, object-oriented,

way of programming is valuable for effective reuse of the code (we used some LISA classes in Alma), saving in fact development time and effort.

References

[1] J. Heering, P. Klint, Semantics of programming languages: A tool-oriented approach, ACM Sigplan Notices 35 (3) (2000) 39–48.

[2] M. Mernik, M. Lenič, E. Avdičaušević, V. Žumer, Compiler/interpreter generator system LISA, in: IEEE CD ROM Proceedings of 33rd Hawaii International Conference on System Sciences, 2000.

[3] J. Hopcroft, J. Ullman, Introduction to Automata Theory, Languages and Computation, Addison-Wesley, Reading, MA, 1979.

[4] D. Knuth, Semantics of contex-free languages, Math. Syst. Theory 2 (2) (1968) 127–145.

[5] J. Paakki, Attribute grammar paradigms - a high-level methodology in language implementation, ACM Computing Surveys 27 (2) (1995) 196–255.

[6] M. Mernik, M. Lenič, Enis Avdičaušević, V. Žumer, Multiple Attribute Grammar Inheritance, Informatica 24 (3) (2000) 319–328.

[7] M. Mernik, M. Lenič, E. Avdičaušević, V. Žumer, A reusable object-oriented approach to formal specifications of programming languages, L'Objet 4 (3) (1998) 273–306.

[8] E. Avdičaušević, M. Lenič, M. Mernik, V. Žumer, AspectCOOL: An experiment in design and implementation of aspect-oriented language, ACM SIGPLAN Notices 36 (12) (2001) 84–94.

[9] M. Mernik, U. Novak, E. Avdičaušević, M. Lenič, V. Žumer, Design and implementation of simple object description language, in: ACM Symposium on Applied Computing, SAC'2001, 2001, pp. 590–594.

[10] J. D. U. A. V. Aho, R. Sethi, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986.

[11] M. Jourdan, D. Parigot, The FNC-2 system user's guide and reference manual, release 1.19, Tech. rep., INRIA Rocquencourt (1997).

[12] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, CENTAUR: The system, ACM SIGPLAN Notices 24 (2) (1989) 14–24.

[13] I. Attali, C. Courbis, P. Degenne, A. Fau, D. Parigot, C. Pasquier, SmartTools: A generator of interactive environments tools, in: 10th International Conference on Compiler Construction, Vol. 2027, Lecture Notes in Computer Science, Springer-Verlag, 2001, pp. 355–360.

[14] M. van den Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Oliver, J. Scheerder, J. Vinju, E. Visser, J. Visser, The ASF+SDF Meta-environment: A component-based language development environment, in: 10th International Conference on Compiler Construction, Vol. 2027, Lecture Notes in Computer Science, Springer-Verlag, 2001, pp. 365–370.

[15] M. Anlauff, P. Kutter, A. Pierantonio, Formal aspects and development environments for Montages, in: 2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97), Electronic Workshops in Computing, Springer/British Computer Society, 1997.

[16] J. Saraiva, M. Kuiper, Lrc - a generator for incremental language-oriented tools, in: 7th International Conference on Compiler Construction, Vol. 1383, Lecture Notes in Computer Science, Springer-Verlag, 1998.

[17] M. H. Brown, R. Sedgewick, A system for algorithm animation, in: SIGGRAPH'84, Vol. 18, ACM Computer Graphics, Minneapolis, 1984, pp. 177–186.

[18] J. T. Stasko, Simplifying algoritm animation with TANGO, in: IEEE Workshop on Visual Languages, IEEE, 1990.

[19] M. H. Brown, M. A. Najork, R. Raisamo, A Java-based implementation of collaborative active textbooks, in: VL'97 - IEEE Symposium on Visual Languages, IEEE, 1997, pp. 376–384.

[20] H. Lieberman, C. Fry, ZStep 95: A reversible, animated source code stepper, in: ACM Conference on Computers and Human Interface, Denver, Colorado, 1995.

[21] J. Haajanen, M. Pesonius, E. Sutien, T. Terasvirta, P. Vanninen, J. Tarhio, Animation of user algorithms in the web, in: VL'97 - IEEE Symposium on Visual Languages, IEEE, 1997, pp. 360–368.

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

[23] S. Mukherjea, J. T. Stasko, Applying algorithm animation techniques for program tracing, debugging, and understanding, in: 15th International Conference on Software Engineering, Baltimore, 1993, pp. 456–465.

A Alma Abstract Grammar

```
pprog:     prog          : STATS
pstats:    STATS         : stat STATS
pstat:                   | stat
pstat1:    stat          : IF
```

```
pstat2:                  | WHILE
pstat3:                  | ASSIGN
pstat4:                  | READ
pstat5:                  | WRITE
pstat6:                  | CALLPROC
pstat7:                  | PROCDEF
pstat8:                  | RETURN
pifelse:   IF            : cond actions actions
pifthen:                 | cond actions
pwhile:    WHILE         : cond actions
passign:   ASSIGN        : VAR exp
pread:     READ          : VAR
pwrite1:   WRITE         : VAR
pwrite2:                 | CONST
pcall1:    CALLPROC      : LST
pcall2:                  | LST PROCDEF
pcond:     cond          : RELOPER
pactions:  actions       : STATS
popr:      RELOPER       : exp LST
pop:       OPER          : exp LST
plst:      LST           : exp LST
pnull:                   | epsilon
pexp1:     exp           : CONST
pexp2:                   | VAR
pexp3:                   | OPER
pexp4:                   | RELOPER
pexp5:                   | CALLPROC
pconst:    CONST         : num
pret:      RETURN        : exp
ppdef:     PROCDEF       : LST STATS
pvar1:     VAR           : id
```