# VisualLISA
## Visual Attribute Grammars

Nuno Oliveira

University of Minho
Informatics Department
Braga,Portugal

**Abstract.** This document reports the work concerned with the development of an environment for visual attribute grammars, named VisualLISA. Its main purpose is to be used as a front-end for LISA(a compiler generator tool based on textual attribute grammars), in order to ease and enrich the way language engineers design their attribute grammars.
This environment is generated from the specification of a visual language, and ensures the possibility to draw, syntactically and semantically correct, attribute grammars, in an integrated editor. The visual specification of the attribute grammar is production-oriented and incremental. Semantic rules are drawn, together or separately, over the syntactic layout (in the form of a tree) of the respective production. Attribute declarations are collected and gathered from tree nodes. Moreover, the editor translates the drawn attribute grammar directly into LISA notation (generating LISA textual specifications) or alternatively into a universal XML representation designed to support Attribute Grammar specifications.
Special focus will be devoted to specification of visual languages and consequent automatic and systematic generation of visual programming environments. The visual programming environments generator, DEViL, will be introduced and its use explained.

**Key words:** Attribute Grammars, LISA, Compiler Generator, Visual Programming Environment Generation, Visual Languages, Code Generation, Intermediate Representation, DEViL, VisualLISA

## 1 Introduction

*Attribute Grammars* (AGs) [7] are a well-known and powerful artifact to create language processors. But AG definitions are not as easy as people would desire. The difficulties of choosing the appropriate attributes and conceiving the attribute evaluation rules are significant, but the effort required to write the complete specification is enormous. Normally is easier to sketch up on paper the complex dependencies among symbols, attributes and functions in an AG. This strategy allows the developers to think abstract and syntax-independent.

However, after being sketched the productions and the semantic dependencies between the attributes, they are not more than gibberish on paper. The person who draw it must go through the translation of the (sometimes imperceptible) pencil strokes into the concrete syntax of the compiler generator.

These problems make the developers avoid the usage of AGs and go through non systematic ways to achieve the same results. This fact lead gEPL team, from University of Minho, some time ago, to propose a *Visual Language* (VL) as a meta-language to write AGs.

VLs are not easy to define, because there is not a consensual definition. The notion of VL is deeply connected with the notion of *Visual Programming Language* (VPL). In fact, it can be established the following relation between them: $\mathtt{VPL} \subset \mathtt{VL}$

VLs or VPLs aim at offering the possibility of solving complex problems by describing their properties or their behavior through graphical/iconic definitions [3]. Icons are used to be composed in a space with two or more dimensions, defining sentences that are formally accepted by parsers, where shape, color and relative position of the icons are relevant issues. Thus, a good definition for VPL can be found in [12]:

> *A Visual Programming Language defines a set of sentences formed by the spatial disposition of graphical objects with a very well defined semantics.*

There are many types of VL. Examples cover a large range from Musical Scores, Traffic Signals, Modeling Languages (Entity-Relation Diagram, Class-Diagrams, Use Cases Diagrams, State-Machines) until programming environments like graph transformations, ETL tools, Grafcet for digital equipment control and robotics, Prograph, etc.

The literature about VL is also very large and distinct addressing several areas in this topic. Regarding these differences, to classify the various articles, other documents, projects and prototypes in VL area a classification system was developed.

A visual language implies the existence of a *Visual Programming Environment* (VPE) [6, 2], because its absence makes the language useless. Commonly, a visual programming environment consists in an editor, enriched by several tools for analyzing, processing and transforming the drawings resultant from the association of the VL's icons.

LISA [10, 8, 9] is a compiler generator based on attribute grammars, developed at University of Maribor. It generates a compiler from a textual AG specification, and also other graphical tools as can be seen in [5]. The fact of generating many graphical and visualization tools makes the textual specification of the AG very rudimental. Moreover, as a textual AG-based compiler generator, LISA pushes its users into the difficulties raised before. So that, under the bilateral work between Universities of Minho and Maribor, the members of the researching group intended to enhance the front-end of LISA by developing a VPE. The concretization of this work led to a project for the master's UCE-15, whose objectives are the development of a VPE, named VisualLISA, that assures the possibility of specifying visually AGs, and to translate it into LISA textual specifications or alternatively into a universal `XML` representation designed to support `AG` specifications. The main objective of this environment is to diminish the difficulties regarding the specification of `AG`s not only in `LISA` but also for other similar systems.

This paper is concerned with the development of `VisualLISA`, emphasizing the methodologies of work adopted for its development resorting to automatic `VPE` generators. Moreover it aims at the definition of what should be done and what is achieved as final outcome. For the sake of space, the detailed information about the specification and the implementation are not presented. That information can be found in the technical report that documents the entire project [11].

The remainder of this document has the following outline: in Sect. 2 the architecture and features of `VisualLISA` are exposed. In Sect. 3 the syntax for the new `VL` is defined along with its semantic constraints and dynamic semantics. In order to clearly produce the output textual notations, an overview of `LISA`'s syntax and the definition of a universal `XML` notation that abstractly support `AG`s structures will be specified. Then, in Sect. 4 is described the implementation of `VisualLISA`, using a systematic approach. In Sect. 5 the system is presented resorting to images. Finally, conclusions about the system and the work are drawn in Sect. 6.

## 2 `VisualLISA`, Problem Statement

Regarding the literature, there are no tools to specify `AG`s by means of visual composition of icons, that also generate code compliant with traditional compiler generators. So that, `VisualLISA` is a completely new approach for `AG` specification.

The arquitecture of `VisualLISA` complements the architecture of the associated `AG`-based compiler generator, since it is a front-end for such tools.

Figure 1 shows that the component labeled as *VisualLISA environment*, is composed by an editor and by mechanisms used to validate syntactic and semantically the model sketched up on the editor.

The syntax validation restricts some spatial combinations among the icons of the language's alphabet. This validation is a task to be performed at edition-time, originating a syntax-directed editor. The semantic validation deals with syntactic restrictions that can not be expressed by the productions, and covers a set of constraints concerning the `AG` definitions.

Besides that, `VisualLISA` generates code from the drawings. As told before, the target code will be `LISA` or `XML`. The generated `LISA` specification can then be passed straightforward to `LISA` system, and finally be used to create the compiler for the language defined with `VisualLISA`. With this approach, the programming environment emulates the two-steped behavior of the language designers referred before — *i*) sketch up the attribute dependencies w.r.t. the semantic rules and productions, and *ii*) mental effort to translate the drawings into `LISA` notation — into a single step effort.

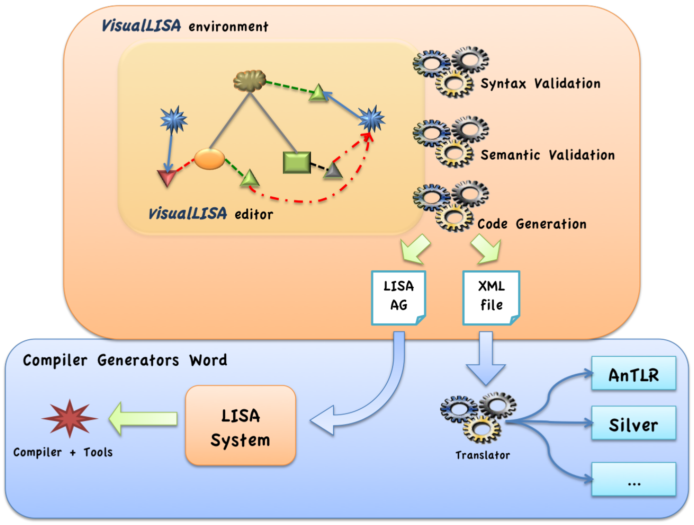


**Fig. 1.** Architecture of `VisualLISA`

The use of `XML` as the generated code gives to the system more versatility because it allows a functional separation between the visual environment and the compiler generator tool. Besides that separation of concerns, it transforms `VisualLISA` in a *VisualAG* — visual programming environment for `AG`s independent of the target compiler generator.

The development of `VisualLISA` must be systematic and based on the traditional compiler tasks [1]. This implies the use of an automatic environment generator tool.

Through out this document, references to the environment and to its underlying language will be used. Henceforth the following words will be used: *i*) `VisualLISA` to refer to the visual environment and *ii*) `VLISA` to identify the visual language underlying the environment.

## 3 `VisualLISA`, Formal Specification

The specification of `VisualLISA` lies on three main issues: *i*) the definition of the underlying language's syntax; *ii*) its semantics and *iii*) the description of the textual specifications into which the iconic compositions will be translated. This section addresses this topic, discussing separately each of the three issues.

### 3.1 Syntax

An `AG` can be seen as a decorated tree, and each production as sub-part of it, then the image required for the representation of a production is a tree, but, for the

VL, that representation must not be fixed. The terminals and nonterminals of the *Righ-Hand Side* (RHS) should be connected to the production's *Left-Hand Side* (LHS). Moreover the production should be decorated with attributes. Then connections between terminal or nonterminal symbols and attributes are mandatory to understand to which symbol the attribute belongs. At the end, the attributes should be associated to computation rules in order to define their values.

With this summary it was exposed what should be expected from the visual language in a visual point of view. But, besides it, syntactic constraints need to be defined. The following sentence defines a syntactic constraint concerning attributes and terminals: "*SC.1 - An intrinsic attribute can only be associated with a terminal symbol*". Other constraints can be seen in [11].

The *Picture Layout Grammar* (PLG) formalism [4], is an attribute grammar to formally specify visual languages. It assumes the existence of pre-defined terminal symbols like **text**, **circle**, **rectangle** and **line**, which are used to represent, abstractly, the icons which compose the VL. It uses a set of spatial relation operators which hide the computation of implicit spatial attributes. *over*(A, B) is an example of one operator; it relates two symbols vertically, saying that the second symbol must be above the first. More on this topic can be seen in [12].

This formalism is used to specify the syntax of the VLISA. Listings 1.1 and 1.2 present some parts of the VLISA specification. Notice how the syntactic constraint (**SC.1**) highlighted before was specified with PLG.

**Listing 1.1.** Syntax Specification - Part 1

```
AG → contains(VIEW, ROOT)

VIEW → labels(text, rectangle)

ROOT → left_to(PRODS, SPECS)

SPECS → contains(VIEW,
          over(LEXEMES, USER_FUNCS))

PRODS → group_of(SEMPROD)

SEMPROD → contains(VIEW, left_to(
    group_of(group_of(RULE_ELEM)),
    group_of(AG_ELEM)))

AG_ELEM → LEFT_SYMBOL
        | NON_TERMINAL
        | TERMINAL
        | SYNT_ATTRIBUTE
        | INH_ATTRIBUTE
        | TREE_BRANCH
        | INT_ATTRIBUTE
        | SYNT_CONNECTION
        | INH_CONNECTION
        | INT_CONNECTION
```

**Listing 1.2.** Syntax Specification - Part 2

```
RULE_ELEM → FUNCTION
          | IDENTITY
          | FUNCTION_ARG
          | FUNCTION_OUT

TERMINAL → labels(text, rectangle)

INT_ATTRIBUTE → labels(text,triangle)

SC.1:
INT_CONNECTION → points_from(
                   points_to(dash_line,
                    ~INT_ATTRIBUTE),
                   ~TERMINAL)

FUNCTION → over(rectangle, text)

FUNCTION_OUT → points_from(
                 points_to(arrow,
                  ~INH_ATTRIBUTE),
                 ~FUNCTION)
             | points_from(
                 points_to(arrow,
                  ~SYNT_ATTRIBUTE),
                 ~FUNCTION)
```

Some nonterminals were specified resorting to the PLG operators *labels* and *points_to/from*, which define final derivation rules. This means that those nonterminal symbols derive in a terminal, i.e., an icon of the VL. Figure 2 shows the concrete and connector icons used for VLISA specification.

*LeftSymbol* is the LHS of a production, while *NonTerminal* and *Terminal* are used to compose the RHS. The second line of icons present the several classes of attributes. *Function*, along with *Identity* are used to compute the attribute values. The other lines connect the concrete symbols rigging up the AG.

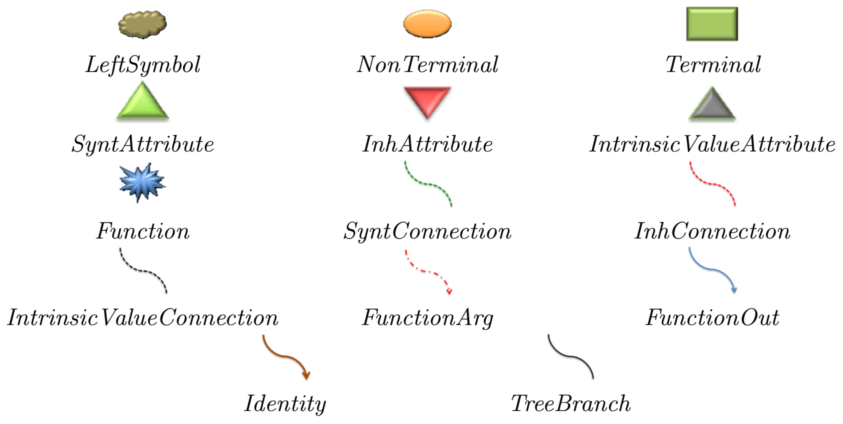


**Fig. 2.** The Icons for `VLISA`

## 3.2 Semantics

In order to correctly specify an `AG`, many semantic constraints must be held. The semantic constraints (or contextual conditions) of a programming language are directly related with attributes and their values. These values can be evaluated or inferred from the specific context in which symbols occur in a sentence.

For the sake of space, the set of attributes of each symbol $X$ of the grammar, $A(X)$ (the set of attributes of $X$), is not presented. However, the most important constraints concerning the semantic correctness of `VLISA`, and `AG`s in general, are listed and defined in natural language.

The constraints for `VLISA` can be separated into two major groups. One concerning the syntactic rules and another the respective computation rules. The former will be referred to as *Production Constraints* (PC), and the latter will be referred to as *Computation Rules Constraints* (CRC).

Apropos of the productions the following constraints were elicited:

**PC.1** The number of `LHS` symbols in a production must be one;
**PC.2** Every *NonTerminal* and *Terminal* symbol on a production must be connected only once to a the `LHS`by a *TreeBranch*;
**PC.3** Every *InhAttribute* or *SyntAttribute* on a production must be attached to a *NonTerminal* or to the `LHS` by a unique *InhConnection* or *SyntConnection*, respectively.
**PC.4** Every *IntrinsicValueAttribute* on a production must be attached to a *Terminal* by a unique *IntrinsicValueConnection*.
**PC.5** Every *NonTerminal* specified on the grammar must be root of one production.
**PC.6** One occurrence of $X.s$, where $s \in AS(X)$ (the set of synthesized attributes of symbol $X$)in a production must be coherent in the sense that $s \in AS(X)$ in any occurrence of $X.s$ in other productions.

**PC.7** One occurrence of $X.i$, where $i \in AI(X)$ (the set of inherited attributes of symbol $X$) in a production must be coherent in the sense that $i \in AI(X)$ in any occurrence of $X.i$ in other productions.

**PC.8** The data type of an attribute $X.a$ in a production, must be the same in any production that $X.a$ occurs.

In attribute grammars, computation rules are the assignment of values to the attributes of the production symbols. In order to fully understand some constraints below, it is needed to recall the formal definition of *In* and *Out* attributes of a production described in [11].

**CRC.1** Only *Out* attributes can be the target of an *Identity* or a *FunctionOut* connection;

**CRC.2** Only *In* attributes can be the source of an *Identity* or a *FunctionArg* connections;

**CRC.3** If there are some *out* attributes declared in a *Semprod*, then, at least one rule must exist for that production;

**CRC.4** The type of the target attribute and the return type of a function, when they are connected by a *FunctionOut* symbol, must match;

**CRC.5** The type of the target and the source attribute of an *Identity* connection, must match.

**CRC.6** A *Function* symbol must be the source of one and exactly one *FunctionOut* connection symbol;

**CRC.7** The number of arguments of a *Function* must match the number of arguments used on *Function*'s `operation`.

## 3.3 Translation

Besides the obvious objective of specifying visual `AG`s, `VisualLISA` has, as important objective, the translation of iconic `AG` into textual notations. `LISA` and `XML` are the target output code. The translation ($\mathcal{L}_s \rightarrow \tau \rightarrow \mathcal{L}_t$) is the transformation of a source language into a target language. $\tau$ is a mapping between the productions of the $\mathcal{L}_s$ (`VLISA`) and the fragments of $\mathcal{L}_t$ (`LISA` and `XML`). These fragments will be specified in this sub-section.

A *Context Free Grammar* (`CFG`) is a formal and robust way of representing `LISA` specifications' structure. Listing 1.3 presents that `CFG`, using `EBNF` notation.

**Listing 1.3.** `LISA` structure in a `CFG`.

```
p1:  LisaML         → language id { Body }
p2:  Body           → Lexicon Attributes Productions Methods
p3:  Lexicon        → lexicon { LexBody }
p4:  LexBody        → (regName regExp)⋆
p5:  Attributes     → attributes (type symbol . attName ;)⋆
p6:  Productions    → rule id { Derivation } ;
p7:  Derivation     → symbol ::= Symbs compute { SemOperations }
p8:  Symbs          → symbol+
p9:                  | epsilon
p10: SemOperations  → symbol . attName = Operation ;
p11: Operation      → ...
p12: Methods        → method id { javaDeclarations }
```

Reserved words, written in bold, indicate, in its majority, the beginning of important fragments. The fact of separating the structure in smaller chunks, makes the process of generating code easier and modular.

The second part of this section concerns with the explanation of an `XML` notation, universal enough, to support the generic and abstract structure of an `AG`.

The use of `XML` has been growing due to several reasons, but the most highlighted is its portability and readability. Therefore it is being used in several areas, from healthcare to astronomy, creating standard notations. Regarding the literature, there is not an `XML` standard notation for `AG`s.

So that, $\mathcal{X}$`AGra` was defined. Assembling all the knowledge about `AG`s with the one acquired from the study of `LISA` structure, the new dialect was defined by means of a schema. The whole structure of this schema can be separated into five big fragments: *i*) `symbols` — where the terminal, nonterminal and the start symbols are defined; *ii*) `attributesDecl` — where is stored information about the attributes and the symbols to which they are associated; *iii*) `semanticProds` — where the productions and the semantic rules are declared: in each production, is defined the `LHS`, the `RHS` and the attribute computations in a very modular way; *iv*) `importations` — where the modules or packages necessary to perform the computations are declared and *v*) `Functions` — is the element where the user declare necessary functions.

A more detailed explanation about these elements, its sub-elements and attributes can be seen in [11].

## 4 `VisualLISA`, Implementation

Usually, the development of `VPE`s is neither a systematic nor an automatic work. But underlying these environments always lies a `VL`. Then systematization can be attained by using traditional compiler development approach; and the automatization of the environment generation is accomplished by using tools for the effect.

### 4.1 Visual Programming Environment Generator

There are some tools that can help on the automatic generation of `VPE`s and underlying `VL`s.

Three of these tools (`TIGER`[1], `VLDesk`[2] and `DEViL`[3]) were experimented in order to choose the most featured and comfortable for `VisualLISA`'s development. For a fair decision, all of the tools were submitted to the development of a visual language and an environment for visually specify *Topic Maps* (`TM`), and to translate them into customizable textual code.

[1] `http://tfs.cs.tu-berlin.de/~tigerprj/`
[2] `http://www.scienzemfn.unisa.it/vldesk/`
[3] `http://devil.cs.upb.de/`

TIGER generates an editor (an Eclipse plugin) based on a formal graph-transformation visual language specification. The systematization on the process of the visual language development is possible, because it imposes a step-by-step definition of the language components. However this tool don't have any facility for translating the models into textual specifications.

VLDesk generates a VPE based on the eXtended Positional Grammars (XPG) formalism. The several steps of language definitions are supported, achieving the systematization. It provides tools for all of these tasks. Blends visual with YACC-based textual specifications, which have to be specified by the user. Moreover is only supported on Windows operating system.

DEViL generates languages and its environment from an Object-Oriented AG-based specification. The systematization can be achieved in a very modular and concern-separeted way. Is flexible, extensible and offers features for structure reuse. The code generation is a straightforward task. Besides that, it works in the most popular operating systems and generates stand-alone and very complete VPEs.

During the experiment, DEViL was the most convincing tool. Despite the initial difficulties, it revealed to be the most complete compared to the others: *i*) runs in the most important operating systems; *ii*) generates stand-alone and intuitive VPEs; *iii*) is extensible in the sense that is not limited to the base functionalities; *iv*) allows layout reutilization by coupling stuctures; *v*) the generation of customizable code is an easy AG-based translation task; *vi*) the specifications are easy to maintain and evolve; etc. For these reasons and several others, DEViL was the chosen tool.

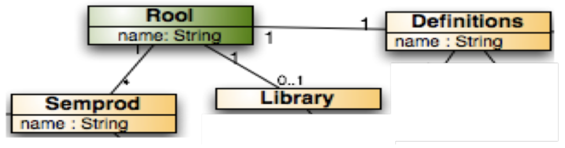
## 4.2 Step-by-Step Systematic Implementation

After having all the requirements formally specified (Sect. 3) and a VPE generator chosen, the implementation of VisualLISA is a straightforward work and can be systematized in four main steps: *i*) Abstract Syntax Specification; *ii*) Interaction and Layout Definition; *iii*) Semantics Implementation and *iv*) Code Generation.

**Abstract Syntax.** The specification of the abstract syntax of VLISA, in DEViL, must follow an object-oriented AG specific notation. This means that the nonterminal symbols of the grammar are defined modularly: the symbols can be seen as classes and the attributes of the symbols as class attributes.

The syntax of the visual language is determined by the relations among their symbols. Therefore, for a higher level representation of the language's syntax, a class diagram can be used. This diagram should meet the structure of the PLG model in Listings 1.1 and 1.2. The final specification for the language is then an easy process of converting the diagram into DEViL-compliant notation. Figure 3 shows a small example of the diagram and the resultant specification.

There are two types of classes in this notation: concrete and abstract. The concrete classes are used to produce an *Abstract Syntax Tree* (AST), which is manipulated in the other steps of the environment implementation. The abstract



```
CLASS Root {
  name: VAL VLString;
  semprods: SUB Semprod*;
  defs: SUB Definitions!;
  library: SUB Library?;
}
```

**Fig. 3.** Class Diagram and Respective `DEViL` Notation

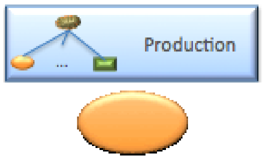
classes are used to group concrete classes with the purpose of defining robust syntactic constraints.

In order to make possible the specification of separated computation rules over the same syntactic layout of a production, was used the DERIVED constructor [14]. It couples the syntactic structure of a given symbol — for `VLISA` the symbol used was the one used to model a production: *Semprod*. In practice, it means that the layout defined for a production is replicated whenever a computation rule is defined, maintaining both models synchronized all the time.

**Interaction and Layout.** After the abstract structure of the language is defined, it is needed to give it a layout and make it usable. This concerns the second step of the `VPE` generation. This implementation, in `DEViL`, consists in the definition of views. A view can be seen as a window with a dock and a specification area, where a part or the whole language is used to specify the drawing.

In a first moment the buttons of the dock are defined. They are used to create the icons in the specifications area. In a second moment is defined the visual layout of the concrete symbols of the grammar. Figure 4 shows parts of view definitions and the respective results created in the editor.

```
VIEW rootView ROOT Root{
 BUTTON IMAGE "img::btnProd"
  INSERTS Semprod
   INFO "Production";
}
```



```
SYMBOL pview_NonTerminal
INHERITS VPForm
COMPUTE
 SYNT.drawing =
      ADDROF(ntDrawing);
END;
```

**Fig. 4.** Parts of View Definitions and Respective Visual Outcomes

The code on the left side of Fig. 4 is a chain of simple instructions used to declare a button and assign its behavior. The default behavior is the insertion of a symbol of the grammar in the specification area, but it can be extended. The bluish rectangular image represents the button resultant from that code.

Symbol *NonTerminal* is represented by the orange oval in Fig. 4. The code on the right reveals the semantic computation to define the shape of that symbol.

Shape and other visual aspects of the tree-grammar symbols are automatically defined associating them, by inheritance, visual patterns.

"*Visual patterns are reusable implementations of common representation concepts like lists, sets, line connections and forms*" [13]. Since these patterns are already implemented, the effort to define the layout of the visual language is the same of understanding all the patterns and know how and when to apply them.

**Semantics.** As long as `VLISA` is defined by an `AG`, the contextual conditions could be verified using the traditional approach. `DEViL` is very flexible and offers at least two different ways, besides the traditional one, for implementation of the semantic constraints.

One of these approaches is based on the events that are risen whenever an edition (like creation or deletion of a symbol) occurs. These events are always associated to a symbol, i.e., a context in the tree generated by `DEViL`. This makes possible the change of the normal behavior of the event's action, in order to verify, in the given context, several conditions. However, more complex verifications can not be implemented using this approach.

The other approach is completely focused on the contexts of the generated tree. `DEViL` offers a tree-walker, named *addCheck*, that traverses the tree-grammar and for a given context — a symbol of the tree — executes a verification code, returning an error whenever it occurs. With this approach is easy to define data-structures that help on the verification process.

The latter approach is the one advised to use. Using it may seem that semantics module is tool-dependent, but it is not. The approach is very similar to the generic `AG` approach, but instead attributes and semantic rules, it uses variables which are assigned by the result of queries on the tree of the model.

Listing 1.4 shows the code for the implementation of the constraint **PC.1** defined before.

**Listing 1.4.** Implementation of Constraint **PC.1**

```
checkutil::addCheck Semprod {
 set n [llength [c::getList {$obj.grammarElements.CHILDREN[LeftSymbol]}]]
 set symbName [c::get {$obj.name.VALUE}]
 if { $n == 0 } {
  return "Production '$symbName' must have one Root symbol!"
 } elseif {$n > 1} {
  return "Production '$symbName' must have only one Root symbol!"
 }
 return ""
}
```

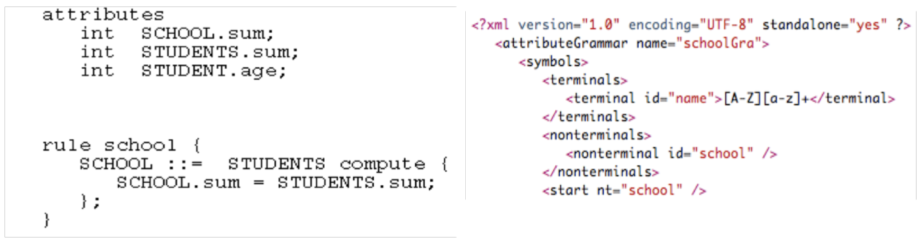
A great part of the constraints defined in Sect. 3.2 were verified resorting to Identifier Tables, which are very used for that purpose in language processing area.

**Code Generation.** The last step of implementation concerns with the translation of the visual `AG` into `LISA` or $\mathcal{X}$`AGra` notation. This task, as usually in language processing, can be done using the `AG` underlying the visual language. `DEViL` does not offer other ways, besides that, to attain it. Instead presents *i*) powerful mechanisms to ease the semantic rules definition; *ii*) facilities of extending the semantic rules by using functions and *iii*) template language incorporation to structure out the output code.

The use of templates is not mandatory. But as can be seen by the formal definition of `LISA` and $\mathcal{X}$`AGra` notation (Sect 3.3), both of them have static parts which do not vary from specification to specification. Hence templates are very handy here. But even with templates (which are converted into functions), the translation of the visual `AG` into text is not a pacific task. Some problems arise from the fact that there is not a notion of order in a visual specification.

For `VLISA`, is important to know the order of the `RHS` symbols. On one hand that order could be achieved by relying on the temporal order of the creation of symbols; on the other hand the same order could be retrieved from the position of these symbols, but only regarding their alignment over the $X$-axe. Both were possible to do, but time is abstract and invisible, then the order would not be perceptible in the drawing, only in the textual specification. Instead, the position is a visual aspect, therefore it could be apprehended from the drawing. Based on this approach, several other problems, like numbering repeated symbols in the production definition, were considered and solved.

To complete the translation it is needed to define the name of the processor — this name will be associated with a button in the final editor — and to specify the process. Figure 5 shows a small part of the code generated for `LISA` and for $\mathcal{X}$`AGra`.



```
attributes
   int   SCHOOL.sum;
   int   STUDENTS.sum;
   int   STUDENT.age;


rule school {
   SCHOOL ::=  STUDENTS compute {
      SCHOOL.sum = STUDENTS.sum;
   };
}
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
  <attributeGrammar name="schoolGra">
    <symbols>
      <terminals>
        <terminal id="name">[A-Z][a-z]+</terminal>
      </terminals>
      <nonterminals>
        <nonterminal id="school" />
      </nonterminals>
      <start nt="school" />
```

*a*) *b*)

**Fig. 5.** Example of small parts of the generation of *a*) `LISA` and *b*) $\mathcal{X}$`AGra`

## 5 `VisualLISA`, The Environment

With so short specifications distributed by several files, `DEViL` generates a very intuitive and complete programming environment. In Figure 6 can be seen the main window with three opened sub-windows. The main window is automatically generated. It offers functionalities like save and load, cut and paste, undo and redo, export to several types of files from HTML to PNG, zoom, and many others. Yet it can be extended to support other functionalities.

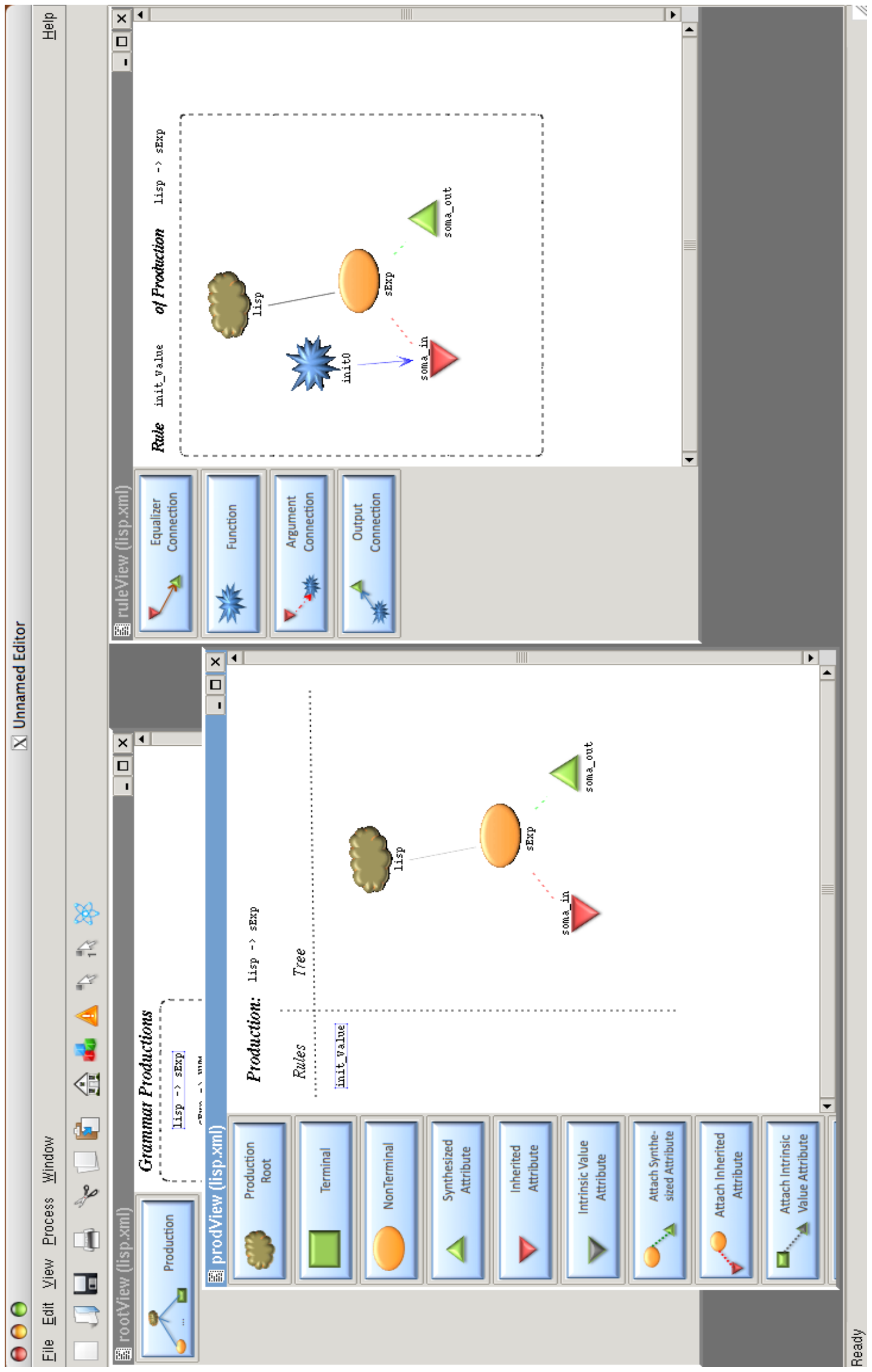


**Fig. 6.** `VisualLISA` Environment

The sub-windows define the views of VLISA. In each window lies a dock with buttons and the specification area, where some icons are already composed. Notice the replication of structures in the two topmost windows. Like specified in the requirements, the computation rules reuse the layout of the respective production, avoiding the necessity of recreating such structure and possible consistency problems. Figure 7 shows an example of an highlighted computation rule and Fig. 8 presents the global definitions area. In Fig. 7, to define the semantic rule it was taken the base layout of the production, and was incremented with the icons that syntactical and semantically define a semantic rule.

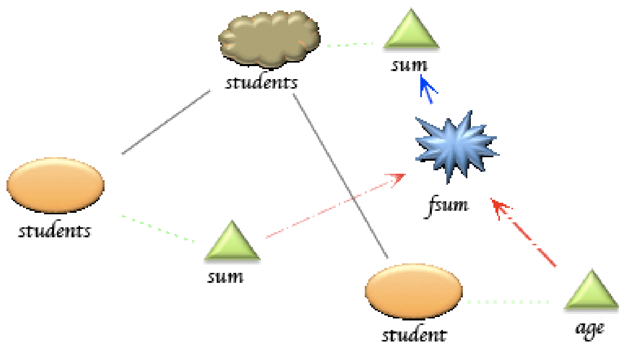
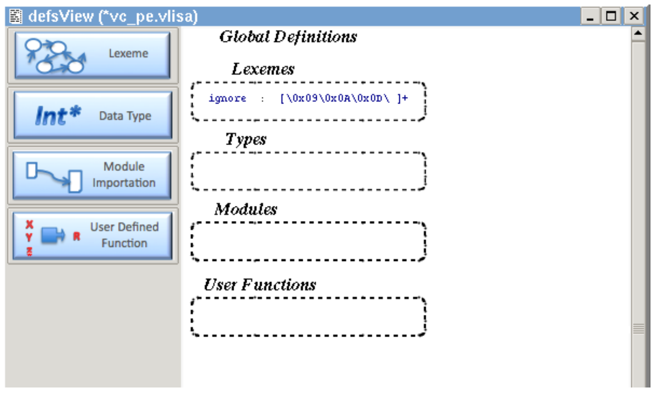


**Fig. 7.** Computation Rule



```
ignore  :  [\0x09\0x0A\0x0D\ ]+
```

**Fig. 8.** Global Definitions

The star-shaped blue symbol represents a function or operation that takes two arguments. The arguments are attribute values, and are represented as red dashed arrows in the figure. The operation assigns a value to an attribute by using the blue full arrow. The mathematical operation is set in the form that appears when clicking twice on the function symbol. That operation can use simple functions, operators, or even user defined functions. The latter must be defined in the global definitions area of the visual language shown in Fig. 8. Here it is possible to define new lexemes, data-types or even user-defined functions, and modules or packages to import.

## 6 Conclusion

In the project reported in this paper was developed a completely new concept on the specification of attribute grammars: a new visual language (`VLISA`) to attribute grammars specification was defined and a visual programming environment (`VisualLISA`) was generated taking advantage from the usage of the `DEViL` tool.

A new `XML` dialect, called $\mathcal{X}$`AGra`, was defined to make possible the translation of the visual `AG` specification into an abstract representation of an `AG`.

Moreover, from this work, some lessons were learnt. Firstly it was confirmed that using automatic `VPE` generator tools, a complete and usable visual environment can be developed, resorting to small and maintainable specifications separated by several files. Secondly, regarding the fact that developing a visual environment has always underlying the specification of a visual language, it is possible to resort to a systematic approach based on the compilers construction to specify and develop the complete environment. This approach was proposed and followed. It is characterized in four main steps: Abstract Syntax Specification; Interaction and Layout Definition; Semantics Implementation and Code Generation.

At the end, the environment development was completed meeting all the requirements elicited. `VisualLISA` allows the visual specification of attribute grammars and its translation into `LISA` textual notation. Optionally, it allows the translation into $\mathcal{X}$`AGra`, what opens, in different ways, the purposes of `VisualLISA`'s usage, and therefore originates new work around this tool.

During the development of `VisualLISA`, several talks about underlying issues to different audiences were given. An intermediate and simple usability test, resorting to a group of students, was made, in order to gather information to improve `VisualLISA`. Also a web-site[4] was created to spread widely the ideas and to distribute the software versions as well as the technical report of this work and other documentation published.

[4] `www.di.uminho.pt/~gepl/VisualLISA/`

### 6.1 Future Work

A complete usability test is lacking for this tool. In the future is important to submit `VisualLISA` through an usability test, to see how well it does with cognitive dimensions. Depending on the results of these tests, improvements should be made.

The fact of generating $\mathcal{X}$`AGra`, allows the use of `VisualLISA` to specify `AG`s for other compiler generators rather than `LISA`. This implies the creation of translators that take $\mathcal{X}$`AGra` as input, and transform it into the target compiler notation. The other way around is also a possibility. The latter makes of `VisualLISA` a visualization tool to help on `AG` comprehension.

## References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques and Tools.* aw, 1986.

[2] Gennaro Costagliola, Genoveffa Tortora, Sergio Orefice, and Andrea De Lucia. Automatic generation of visual programming environments. *Computer*, 28(3):56–66, 1995.

[3] Luis Miguel da Sivla Dias. Linguagens visuais de programação - paradigmas e ambientes. Master's thesis, Universidade do Minho, Escola de Engenharia, December 1996.

[4] Eric J. Golin. *A Method for the Specification and Parsing of Visual Languages.* PhD thesis, Brown University, Department of Computer Science, Providence, RI, USA, May 1991.

[5] Pedro Rangel Henriques, M. J. V. Pereira, Marjan. Mernik, Mitja. Lenič, J. Gray, and H. Wu. Automatic generation of language-based tools using the lisa system. *Software, IEE Proceedings* -, 152(2):54–69, 2005.

[6] Uwe Kastens and Carsten Schmidt. Vl-eli: A generator for visual languages - system demonstration. *Electr. Notes Theor. Comput. Sci.*, 65(3), 2002.

[7] Donald E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, June 1968.

[8] Marjan Mernik, Nikolaj Korbar, and Viljem Žumer. Lisa: a tool for automatic language implementation. *SIGPLAN Not.*, 30(4):71–79, 1995.

[9] Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer. A reusable object-oriented approach to formal specifications of programming languages. *L'Object*, 4:273–306, 1998.

[10] Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer. Lisa: An interactive environment for programming language development. *Compiler Construction*, pages 1–4, 2002.

[11] Nuno Oliveira, M. J. V. Pereira, Daniela da Cruz, and Pedro Rangel Henriques. Visuallisa. Technical report, Universidade do Minho, February 2009. (To be published).

[12] Jorge Gustavo Rocha. Especificação de linguagens visuais de programação. Master's thesis, Universidade do Minho, Departamento de Informática, June 1995.

[13] Carsten Schmidt, Uwe Kastens, and Bastian Cramer. Using devil for implementation of domain-specific visual languages, 2006.

[14] Carsten Schmidt, Uwe Kastens, and Bastian Cramer. Specifying coupled structures for implementation of visual languages, 2007.