UDC 004.439, DOI: 10.2298/CSIS1002265O

VisualLISA: A Visual Environment to Develop Attribute Grammars

Nuno Oliveira[1], Maria João Varanda Pereira[2], Pedro Rangel Henriques[1], Daniela da Cruz[1], and Bastian Cramer[3]

[1] University of Minho - Department of Computer Science,
Campus de Gualtar, 4715-057, Braga, Portugal
{nunooliveira,prh,danieladacruz}@di.uminho.pt
[2] Polytechnic Institute of Bragança
Campus de Sta. Apolónia, Apartado 134 - 5301-857, Bragança, Portugal
mjoao@ipb.pt
[3] University of Paderborn - Department of Informatics
Fürstenallee 11, 33102, Paderborn, Germany
bcramer@upb.de

Abstract. The focus of this paper is on crafting a new visual language for attribute grammars (AGs), and on the development of the associated programming environment. We present a solution for rapid development of VisualLISA editor using DEViL. DEViL uses traditional attribute grammars, to specify the language's syntax and semantics, extended by visual representations to be associated with grammar symbols. From these specifications a visual programming environment is automatically generated. In our case, the environment allows us to edit a visual description of an AG that is automatically translated into textual notations, including an XML-based representation for attribute grammars ($\mathcal{X}$AGra), and is intended to be helpful for beginners and rapid development of small AGs. $\mathcal{X}$AGra allows us to use VisualLISA with other compiler-compiler tools.

Keywords: Attribute Grammar, Visual Languages, XML Dialect, DEViL, VisualLISA, XAGra.

1. Introduction

An AG can be formally defined as the following tuple: $AG = (G, A, R, C)$, where G is a context-free grammar, A is the set of attributes, R is the set of evaluation rules, and C is the set of contextual conditions. Each attribute has a type, and represents a specific property of a symbol X; we write $X.a$ to indicate that attribute a is an element of the set of attributes of X, denoted by $A(X)$. For each X (terminal or non-terminal), $A(X)$ is divided into two disjoint sets: the *inherited* and the *synthesized* attributes. Each R is a set of formulas, like $X.a = func(..., Y.b, ...)$, that define how to compute, in the precise context of a production, the value of each attribute. Each C is a set of predicates, $pred(..., X.a, ...)$, describing the requirements that must be satisfied in the precise context of a production.

As can be deduced from this complex definition of AGs they are not as easy to specify as people would desire because there is a gap between the problem solution (the desired output) and the source language that must be interpreted. The user must take care on choosing the appropriate attributes and their evaluation rules. Since the beginning, the literature related with compilers presents AGs using syntax trees decorated with attributes. So it is usual to sketch up on paper trees with attributes representing an AG. This strategy allows the developers to imagine a global solution of the problem (in a higher abstraction level) and to detect complex dependencies between attributes, symbols and functions, avoiding spending time with syntax details. However, such informal drawings require the designer to translate them manually into the input notation of a compiler generator. The person who drew it must go through the translation of the pencil strokes into the concrete syntax of the compiler generator. These inconveniences make the developers avoid the usage of AGs and go through non systematic ways to implement the languages and supporting tools. So, in this paper, we develop a *Visual Language* (VL), as a meta-language to write AGs, based on a previous conceptualization that we have proposed in [1]. The idea of this VL is not only about having a nice visual depiction and then to translate it into a target notation, but also about syntactic and semantic consistency checks.

VLs and consequently the *Visual Programming Languages* (VPLs) aim at offering the possibility to solve complex problems by describing their properties or their behavior through graphical/iconic definitions [2]. 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. A visual programming language implies the existence of a *Visual Programming Environment* (VPE) [3, 4], because its absence makes the language useless. Commonly, a visual programming environment consists of an editor, enriched by several tools to analyze, to process and to transform the drawings.

The main idea of this work is the development of a VPE, named VisualLISA, that assures the possibility of specifying AGs visually, and to translate them into plain text specifications or, alternatively, into a universal XML representation designed to support generic AG specifications. The original objective of this environment is to be used as front-end for LISA [5] system, diminishing the difficulties regarding the specification of AGs in LISA. However, the generality of the environment enables its use with systems other than LISA.

The visual programming environment is automatically generated by DEViL, our choice among many other tools studied; so, in this paper, the system is introduced and its use explained. However, our objective in this paper is not concerned with the discussion of compiler development tools, but show the befits of using an effective one.

In section 2, related work is presented. In Section 3 and 4, VisualLISA language and editor are informally described. In Section 5 the language is formally specified, defining syntactic rules, semantic constraints and a valid translation scheme for both LISA (the first target), and $\mathcal{X}$AGra notations. In Section 6, the DEViL generator framework, used for the automatic generation of the visual editor, will be presented. In Section 7, following the informal conception and its formalization, using DEViL, the visual language and the editor implementation is shown. An overview on how to use the editor to describe an AG, is given in Section 8.

In Section 9, $\mathcal{X}$AGra dialect is formally presented. Its main idea is to generalize the output of AG editing tools; instead of generating a description for a specific compiler generator, the editor under development can produce this general purpose dialect. Then to use this editor as a *Front End* (FE) for a specific generator, it is only necessary to resort to a simple translator to convert the XML description into the specific notation of that CG. This approach raises the usefulness of the editor, as it can be used as a FE for a larger range of grammar-based generators. However, as its applicability does not end here, we introduce, in Appendix A, $\mathcal{X}$AGraAl, a tool that, based on $\mathcal{X}$AGra specifications, performs grammar analysis and transformations.

The paper is concluded in Section 10.

2. Related Work

Despite of existing many other applications for AGs, they are commonly associated with the development of computer languages and related tools like parsers, translators, compilers, and others. In this context, the language engineers, started to develop tools to systematize and automatize the process of defining AGs. So, several works on this area may be cited.

LISA [5, 6] is a compiler generator based on attribute grammars, developed at University of Maribor at Slovenia. Its main objective is to generate a compiler for a language. The compiler is created by the specification of a textual attribute grammar. It automatically generates graphical and visualization tools [7] to inspect the written grammar, but it always need a textual specification of the AG.

In the same way, AnTLR [8], a powerful compiler generator, requires textual specifications for the language grammar. This system provides *online* visualization of the grammar productions but it does not provide any visualization about the attributes neither the semantic rules of each production.

Other similar compiler generators like UltraGram [9] or ProGrammar [10] also produce graphical tools to ease the understanding of the grammar. But still, the input for these compiler generators is always a text-based specification.

The same happens in the visual languages generation area. DEViL [11], a generator of visual programming languages and editors, takes advantage of AGs to define these visual outcomes. But, despite of providing excellent and usable results, the engineer needs to grasp a whole new syntax to define the AG used to produce the visual language.

In [12], *Ikezoe et al.* present a systematic debugger for attribute grammars integrated in a visual tool, and it provides visualizations of the grammar showing the dependency between the symbols and the attributes. Although this is a useful tool, it is only used after constructing the AG, not being a good help to map the mental construction of an AG into its specifications.

There are, indeed, several tools to support the specification and development of AGs and their associated tools, however, and according to our knowledge, acquired through years of research work on the area, there are no tools that allow the specification of AGs using a visual notation.

3. `VisualLISA` - A Domain Specific Visual Language

For many years we have been thinking about and working with AGs. Inevitably we created an abstract mental representation of how it can be regarded and then sketched, for an easier comprehension and use. So we decided to implement a framework that follows that representation. The conception of that framework is described in this section.

3.1. The Language Conception

VisualLISA, as a new *Domain Specific Visual Language* (DSVL) for attribute grammar specification, shall have an attractive and comprehensible layout, besides the easiness of specifying the grammar model.

We think that a desirable way to draw an AG is to make it production oriented, and from there design each production as a tree. The *Right-Hand Side* (RHS) symbols should be connected, by means of a visible line, to the *Left-Hand Side* (LHS) symbol. The attributes should be connected to the respective symbols, using a connection line different from the one referred before, as both have different purposes (see Figure 6). The rules to compute the values of each attribute should exhibit the shape of a function with its arguments (input attributes) and its results (the output attributes). Two kinds of functions should be represented: the identity function (when we just want to copy values) or a generic function (for other kind of computations). Often a production has a considerable number of attributes and nontrivial computations. Therefore we think that for visualization purposes, the layout of each production should work as a reusable template to draw several computation rules. Hence, the rules are drawn separated from each other, but associated to a production.

All these features can be seen in the following example, which gives a big picture of how things get easier when dealing with the visual notation on a real language. For this illustration we resort to LISS [13], which is a programming language allowing the operation with atomic or structured integers values. Moreover, it is fully specified using an AG, from where two semantic productions are shown in Listing 1. For illustration purposes, some semantic rules from the actual productions were dismissed.

Figure 3.1 shows the visual specification of production P1, taking advantage of the *production layout reuse* feature, for rapid development and clarity. This means that the user doesn't have to draw the production each time he needs to specify the computation of another attribute.

Listing 1. One Production form LISS

```
P1: Expr → Expr RelOp SingExp {
  (...)
  Expr[1].inRow  = Expr[0].inRow;
  Expr[1].inCol  = Expr[0].inCol;
  SingExp.inRow  = Expr[0].inRow;
  (..)
  Expr[0].out = Expr[1].out + RelOp.out + SingExp.out;
}

P2: SingExp → Term {
   SingExp.out = Term.out;
   (...)
}
```

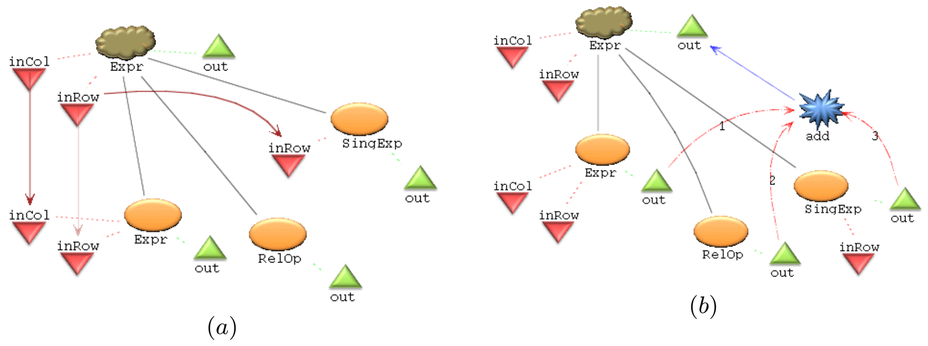


Fig. 1. LISS production on `VisualLISA`, with associated semantic rules

This way, Figure 3.1 (a) we copy the values from the inherited attributes of the `LHS` symbol to the symbols at `RHS`; and on another computation associated with the same production (c.f. Figure 3.1 (b)), we assign a value to the `LHS`'s *out* attribute using a function and the values of other attributes implied in the production.

4. VisualLISA: The Environment

`VisualLISA` editor should be compliant with the idea of offering a nice and non error-prone way of sketching the `AG`, as a first step; and an easy translation of the model into a target language, as a second step. So, three main features are highlighted: (i) syntax validation, (ii) semantics verification and (iii) code generation. The syntax validation restricts some spatial combinations among the icons of the language. In order to avoid syntactic mistakes, the edition should be syntax-directed. The semantics verification copes with the static and dynamic semantics of the language. Finally, the code generation feature generates code from the drawings sketched up. The target code would be `LISAsl` or $\mathcal{X}$`AGra`. `LISAsl` specification generated is intended to be passed to `LISA` system in a straightforward step. $\mathcal{X}$`AGra` specification generated is intended to give the system more versatility and further usage perspectives.

5. Specification of `VisualLISA`

The specification of `VisualLISA` bases on three main issues: $i)$ the definition of the underlying language's syntax; $ii)$ the language semantics and $iii)$ the description of the textual specifications into which the iconic compositions will be translated.

5.1. Syntax

The *Picture Layout Grammar* (`PLG`) formalism [14], is an attribute grammar to formally specify visual languages. It assumes the existence of pre-defined terminal symbols and a set of spatial relation operators. Our acquaintance with `PLG` formalism, from previous works, led us to use it to specify the syntax of `VisualLISA`. Listings 2 present some rules of the language specification. For the sake of space we only present the key rules of the specification; the missing productions are comparable to those shown.

Figure 2 shows the concrete and connector icons used for `VisualLISA` specifications. *LeftSymbol* is the `LHS` of a production, while *NonTerminal* and *Terminal* are used to compose the `RHS`. The second line of icons in Figure 2 presents the several classes of attributes. *Function* and *Identity*, both representing operations, are used to compute the attribute values. The other icons connect the concrete symbols with each other, to rig up the `AG`.

Listing 2. `VisualLISA` Partial Syntax Definition.

```
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
```

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

TERMINAL → labels(text, rectangle)

INT_ATTRIBUTE → labels(text, triangle)

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)
```

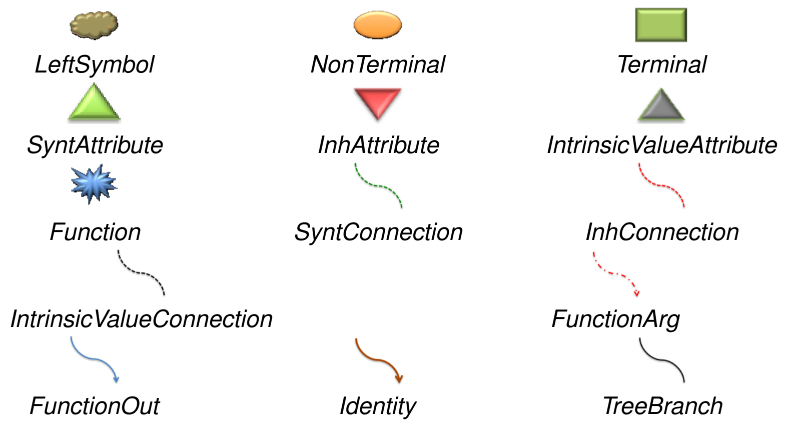


Fig. 2. The Icons of `VisualLISA`

5.2. Semantics

In order to correctly specify an AG, many semantic constraints must hold. These constraints are related with the attribute values that depend on the context in which the associated symbols occur in a sentence. We separated these constraints into two major groups. One concerning the syntactic rules, *Production Constraints* (PC), and another the respective computation rules, *Computation Rules Constraints* (CRC).

The following statements are representative constraints of VisualLISA's semantic correctness, concerning the two groups identified before:

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

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

The complete set of constraints can be seen in [9].

5.3. Translation

The translation ($\mathcal{L}_s \rightarrow \tau \rightarrow \mathcal{L}_t$) is the transformation of a source language into a target language. τ is a mapping between the productions of the $\mathcal{L}_s$ (VisualLISA) and the fragments of $\mathcal{L}_t$ (LISAsl $\cup$ $\mathcal{X}$AGra). 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 3 presents that high-level CFG.

Listing 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, enhance the main fragments in a LISA sentence, making it more readable. The definition of smaller chunks, introduced by each keyword, enables a more modular processing (code generation. . .)

Regarding the literature, there is not an XML standard notation for AGs. So that, $\mathcal{X}$AGra was defined using a schema. The whole structure of this schema can be seen in detail in Section 9.

6. DEViL - A Tool for Automatic Generation of Visual Programming Environments

We searched for VPE generators like MetaEdit+ [15], but their commercial nature was not viable for an academic research. Also, we experimented VLDesk [16], Tiger [17], Atom^3 [18] and other similar tools, however none of them gave us the flexibility that DEViL offered, as described below.

The DEViL system generates editors for visual languages from high-level specifications. DEViL (*Development Environment for Visual Languages*) has been developed at the University of Paderborn in Germany and is used in many nameable industrial and educational projects.

The editors generated by DEViL offer syntax-directed editing and all features of commonly used editors like multi-document environment, copy-and-paste, printing, save and load of examples. Usability of the generated editors and DEViL itself can be found in [11]. DEViL is based on the compiler generator framework Eli [19], hence all of Eli's features can be used as well. Specially the semantic analysis module can be used to verify a visual language instance and to produce a source-to-source translation.

To specify an editor in DEViL we have to define the semantic model of the visual language at first. It is defined by the domain specific language *DEViL Structure Specification Language* (DSSL) which is inspired by object-oriented languages and offers classes, inheritance, aggregation and the definition of attributes. The next specification step is to define a concrete graphical representation for the visual language. It is done by attaching so called visual patterns to the semantic model of the VL specified in DSSL. Classes and attributes of DSSL inherit from these visual patterns. Visual patterns [20] describe in what way parts of the syntax tree of the VL are represented graphically, e.g. we can model that some part should be represented as a “set” or as a “matrix”. DEViL offers a huge library of precoined patterns like formulae, lists, tables or image primitives. All visual patterns can be adapted through control attributes. E.g. we can define paddings or colors of all graphical primitives. Technically visual patterns are decorated to the syntax tree by specifying some easy inheritance rules in a DSL called LIDO.

To analyse the visual language, DEViL offers several ways. The first one results from the fact that editors generated by DEViL are syntax directed. Hence, the user cannot construct *wrong* instances of the VL It is limited by its syntax and cardinalities expressed in DSSL. Another way is to define check rules e.g. to check the range of an integer attribute or to do a simple name analysis on a name attribute. To navigate through the structure tree of the VL, DEViL offers so called path expressions which are inspired by XPath. They can be used in a small simple DSL to reach every node in the tree. After analysis, DEViL can generate code from the VL instance. This is done with the help of Eli which offers unparsers, template mechanism (*Pattern-based Text Generator* — PTG) and the well-known attribute evaluators from compiler construction.

7. Implementation of VisualLISA

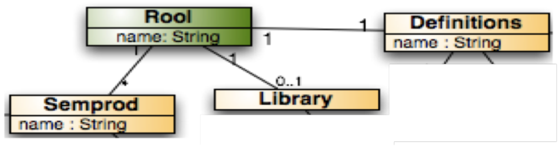
To implement VisualLISA, we could have followed a non systematic way, resorting to usual software development methods. But our know-how on compiler construction led us to reuse the systematic generative approach followed in that area. In this case, the implementation process will be supported by the formal specification made in Section 5, and automated by the VPE chosen, DEViL. Adopting the standard compiler construction process to the DEViL usage peculiarities, we will follow a four-step process: i) Abstract Syntax Specification; ii) Interaction and Layout Definition; iii) Semantics Implementation; and iv) Code Generation.

7.1. Abstract Syntax

The specification of the abstract syntax of VisualLISA, in DEViL, follows an object-oriented notation, as referred previously. 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 an high level representation of the language's syntax, a class diagram can be used. This diagram should meet the structure of the PLG model in Figure 2. The final specification for the language is then an easy manual process of converting the diagram into DSSL. 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 a syntax tree, which is manipulated in the other steps of the environment implementation. The abstract classes, besides the normal inheritance properties can be used to define syntactic constraints. These classes generate the syntax-directed editor.



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

Fig. 3. Class Diagram and Respective DEViL Notation

In order to make possible the specification of separated computation rules reusing the same layout of a production, we used DEViL's concept of coupled structures [21]. It couples the syntactic structure of two structure tree — for VisualLISA we used the structure of symbol *Semprod*, which is used to model a production. 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.

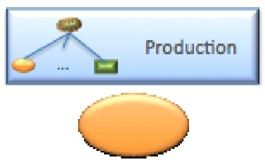
7.2. Interaction and Layout

The implementation of this part, in DEViL, consists of the definition of views. A view can be seen as a window with a dock and an editing area where the language icons are used to specify the drawing.

VisualLISA Editor is based on four views: *rootView*, to create a list of productions; *prodsView*, to model the production layout; *rulesView*, to specify the semantic rules reusing the production layout and *defsView*, to declare global definitions of the grammar.

At first the buttons of the dock, used to drag structure-objects into the edition area, are defined. Then the visual shape of the symbols of the grammar for the respective view are defined. Figure 4 shows parts of view definitions and the respective results in the editor. The code on the left side of Figure 4 defines the view, the buttons and the behavior of the buttons. The default action is the insertion of a symbol in the editing area. The bluish rectangular image represents the button resultant from that code.

```
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

Symbol *NonTerminal* is represented by the orange oval in Figure 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, by inheritance, visual patterns.

7.3. Semantics

As long as VisualLISA is defined by an AG, the contextual conditions could be checked using the traditional approach. DEViL is very flexible and offers some other ways to implement this verification module. The approach used to develop VisualLISA, is completely focused on the contexts of the generated syntax tree. DEViL offers a tree-walker, that traverses the tree and for a given context — a symbol of that tree — executes a verification code (callback-functions), returning an error whenever it occurs. With this approach it is easy to define data-structures helping the verification process. This approach is very similar to the generic AG approach, but instead of attributes and semantic rules, it uses variables which are assigned by the result of queries on the tree of the model.

Listing 4 shows the code for the implementation of a constraint defined in [22].

Listing 4. Implementation of Constraint: "Every *NonTerminal* specified in the grammar must be root of one production"

```
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 considerable amount of the constraints defined in Section 5.2 were verified resorting to the Identifier Table, which is a well known strategy in language processing for that purpose.

7.4. Code Generation

The last step of the implementation, concerning the translation of the visual AG into LISA or $\mathcal{X}$AGra, can be done using the AG underlying the visual language (as usual in language processing). For this task, DEViL supports $i)$ powerful mechanisms to ease the semantic rules definition; $ii)$ facilities to extend the semantic rules by using functions and $iii)$ a template language (PTG of Eli system) incorporation to structure out the output code.

The use of patterns (templates) is not mandatory. But, as seen in the formal definition of LISA and $\mathcal{X}$AGra notation (Section 5.3), both of them have static parts which do not vary from specification to specification. Hence templates are very handy here. Even with templates, the translation of the visual AG into text is not an easy task. Some problems arise from the fact that there is not a notion of order in a visual specification. We used auxiliary functions to sort the RHS symbols by regarding their disposition over an imaginary X-axe. Based on this approach we also solved issues like the numbering of repeated symbols in the production definition.

The templates (invoked like functions) and the auxiliary functions, together with other specific entities, were assembled into semantic rules in order to define the translation module. One module was defined for each target notation. New translation modules can be added, to support new target notations.

8. AG Specification in VisualLISA

Figure 5 shows the editor look and feel, presenting the four views of our editor.

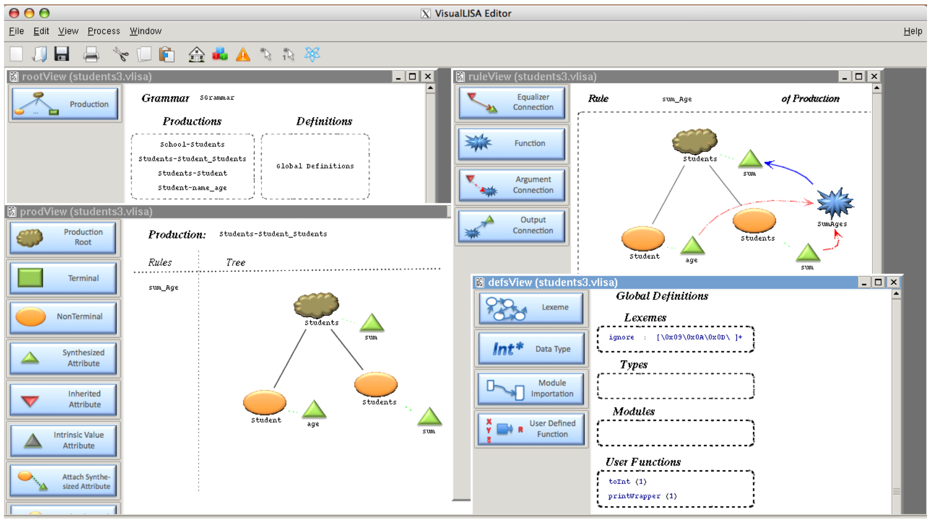


Fig. 5. VisualLISA Editor Environment

To specify an attribute grammar the user starts by declaring the productions (in *rootView*) and rigging them up by dragging the symbols from the dock to the editing area (in *prodsView*), as commonly done in VPEs. The combination of the symbols is almost automatic, since the editing is syntax-directed. When the production is specified, and the attributes are already attached to the symbols, the next step is to define the computation rules. Once again, the user drags the symbols from the dock, in *rulesView*, to the editing area, and compounds the computations by linking attributes to each other using functions. Sometimes it is necessary to resort to user-defined functions that should be described in *defsView*. In addition, he can import packages, define new data-types or define global lexemes.

As example we present a simple AG , called Students Grammar, used to process a list of students, described by their names and ages. The objective of this AG is to sum the ages of all the students. This grammar can be textually defined as shown in Listing 5.

Listing 5. Students Grammar

```
P1: Students → Student Students {Students0.sum = Student.age + Students1.sum}
P2: Students → Student          {Students.sum = Student.age}
P3: Student → name age          {Student.age = age.value}
```

Figures 6 and 7 show the three productions that constitute the grammar.

In Figure 6, the attributes are associated with the symbols of the production. Moreover, the production has a semantic rule that computes the value of the LHS's attribute, *sum*, by adding the value of the attributes in the RHS symbols, *sum* and *age*, using an *inline* function named *SumAges*

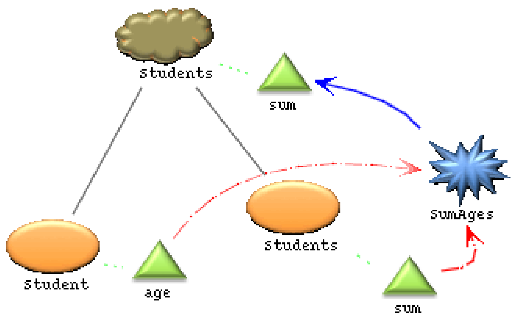


Fig. 6. Specification of production P1 with associated semantics

In Figure 7 (a), the identity function is used to copy the value of the attribute *age* to the attribute *sum*. In Figure 7 (b), the third production, makes use of terminal symbols and associated intrinsic values. The computation rule, in this production, is based on the conversion of the textual value of the age into an integer.

When the grammar is completely specified and semantically correct, code can be generated. Figure 8 shows, in LISA and $\mathcal{X}$AGra notations, the code generated for production P1 in Figure 6.

9. $\mathcal{X}$AGra- An XML dialect for Attribute Grammars

In this section is defined an XML dialect to cope with attribute grammars. We called it $\mathcal{X}$AGra, which stands for ***X****ML dialect for* ***A****ttribute* ***Gra****mmars*.

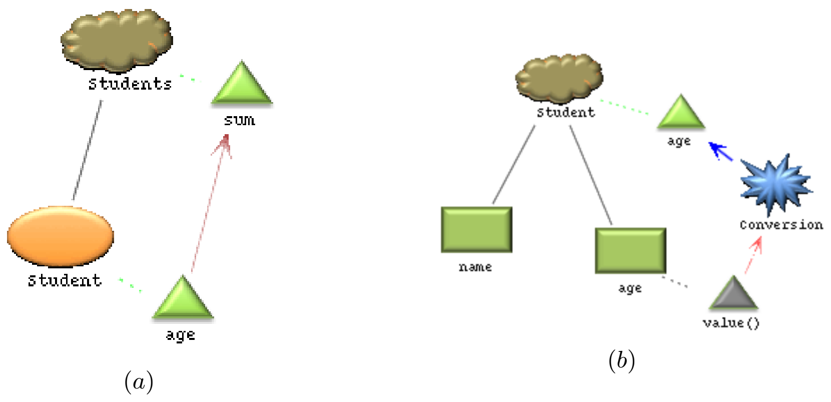
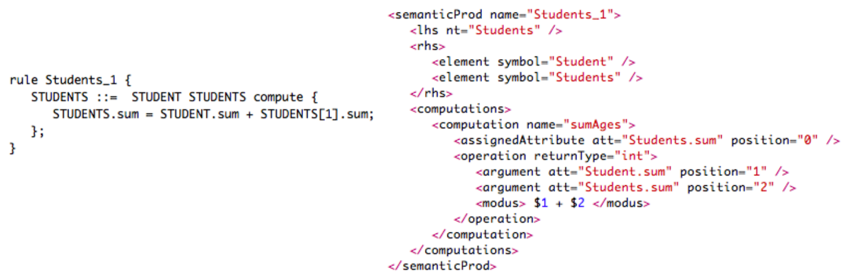


Fig. 7. Specification of Productions P2 and P3, (a) and (b) respectively, with associated semantic rules.



```
rule Students_1 {
   STUDENTS ::=  STUDENT STUDENTS compute {
      STUDENTS.sum = STUDENT.sum + STUDENTS[1].sum;
   };
}
```

```
<semanticProd name="Students_1">
   <lhs nt="Students" />
   <rhs>
      <element symbol="Student" />
      <element symbol="Students" />
   </rhs>
   <computations>
      <computation name="sumAges">
         <assignedAttribute att="Students.sum" position="0" />
         <operation returnType="int">
            <argument att="Student.sum" position="1" />
            <argument att="Students.sum" position="2" />
            <modus> $1 + $2 </modus>
         </operation>
      </computation>
   </computations>
</semanticProd>
```

(a) (b)

Fig. 8. Code Generated for LISA (a) and XAGra (b) specifications.

$\mathcal{X}$AGra denotes the abstract representation of an AG. The notation defined here, is mainly based on the definition of AG presented in the Introduction, but it also borrows parts from the notations inherent to various AG-based compiler generator tools.

One of the standardized ways to define a new XML dialect is the creation of a schema, using the standard XML Schema Definition (XSD) language. For the sake of space, the integral textual definition of $\mathcal{X}$AGra's schema is not presented, and for reasons of visibility and readability, the complete drawing of the schema is broken into several important sub-parts. Figures 9 to 13 are used to support the explanation of the dialect.

$\mathcal{X}$AGra's root element was defined as `attributeGrammar`. This element has a single attribute, `name`, whose objective is to store the name of the grammar, or the language that the grammar defines; and is a sequence of several elements. These elements represent components of the formal definition of an AG, incremented with extra parts related to the usage of AG-based compiler generators.

Table 1 defines a relation of inclusion between the $\mathcal{X}$AGra notation elements and the components that constitute the formal definition of an AG, which is recovered next:

$$AG = (T, N, S, P, A, R, C, \mathcal{T})$$

Table 1. Derivation of $\mathcal{X}$AGra Notation From the Formal Definition of AG

$\mathcal{X}$AGra Element $\supseteq$	AG Components
symbols	T, N, S
attributesDecl	A
semanticProds	$P, R, C, \mathcal{T}$
importations	$\emptyset$
functions	$\emptyset$

The relations depicted in Table 1 give an overview about the information that each element of $\mathcal{X}$AGra notation will store. The following sections will describe with more detail such elements and the information they store.

Listing 6 presents a fragment of a grammar that computes the age of a set of students. This example is used to compare the concrete notation of a compiler generator to the XML fragments that are shown in the sequent figures.

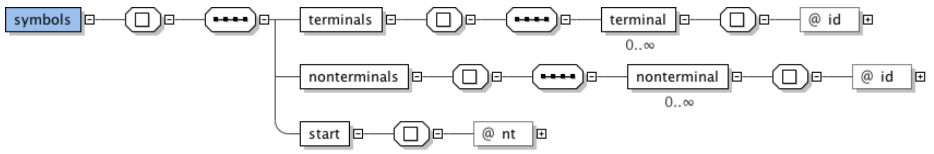
Next sections present a complete description of the elements of $\mathcal{X}$AGra scheme. However, the *importations* and *functions* elements are skipped, because their structure is simple and similar to the other parts shown.

9.1. Element *symbols*

Figure 9 presents the schema for the element `symbols`. As the name suggests, this element contains the declaration of the grammar's vocabulary.

Listing 6. Example of Students Grammar

```
language StudentsGra {
   lexicon{
      Name      [A–Z][a–z]+
   ...
   }
   attributes
      int   STUDENTS.sum;
   ...
   rule Students_1 {
      STUDENTS ::=  STUDENT STUDENTS compute {
         STUDENTS.sum = STUDENTS[1].sum + STUDENT.age;
      };
   }
   ...
   method user_Definitions {
      import java.util.ArrayList
      public int sum(int x, int y){
         return x+y;
      }
   }
}
```



```
<symbols>
    <terminals>
        <terminal id="name">[A–Z][a–z]+</terminal>
    </terminals>
    <nonterminals>
        <nonterminal id="students" />
    </nonterminals>
    <start nt="students" />
</symbols>
```

Fig. 9. $\mathcal{X}$AGra Schema – Element **Symbols**: definition and example

It is composed of a sequence of three elements: `terminals`, `nonterminals` and `start`.

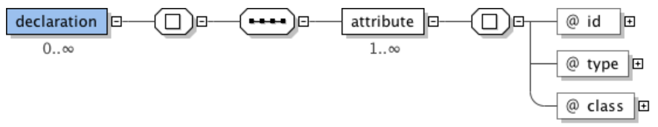
The element `terminals` is a sequence of zero or more elements named `terminal`, which, in its turn, has one attribute, `id`, used to store the name of a terminal symbol. This attribute is an identifier, hence any instance of it, must be different from the others, and must be always instantiated. Besides the information kept on the attribute, this element has a textual content where the respective *Regular Expression* (RE) can be declared.

The element `nonterminals` has similar structure. The difference lays on the fact that it represents a sequence of zero or more elements `nonterminal` which have no textual content. The attribute `id` has the same purpose as the attribute with the same name in the element `terminal`.

Finally, the element `start` has a single attribute named `nt`. This attribute is used to refer the nonterminal (already defined in the $\mathcal{X}$AGra specification), correspondent to the start symbol (or Axiom) of the AG.

9.2. Element *attributesDecl*

This element is composed of a sequence of zero or more elements `declaration`. For the sake of readability, Figure 10 only depicts the structure of the element `declaration`, which is a sequence of one or more elements `attribute`. This one has three mandatory attributes: $i)$ `id` – stores the name of the attribute being declared. Any kind of text can be used to define it, but it is always better to use the following notation: $X.a$, where X is the name of a symbol in $T \cup N$ and a is the name of an attribute in $A(X)$. As it is an identifier, it must be different from all other identifiers on the specification; $ii)$ `type` – stores the data type of the current attribute value and $iii)$ `class` – defines the class of the attribute. It must be one of: InhAttribute, SyntAttribute and IntrinsicValueAttribute.



```
<attributesDecl>
    <declaration>
        <attribute id="students.sum" type="int" class="SyntAttribute" />
    </declaration>
</attributesDecl>
```

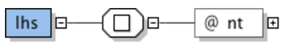
Fig. 10. $\mathcal{X}$AGra Schema – Element **Attribute Declarations**: definition and example

9.3. Element *semanticProds*

The element `semanticProds` represents the structure to define productions and associated semantic rules in $\mathcal{X}$`AGra` specifications. This structure is composed of a sequence of zero or more elements `semanticProd`. Each `semanticProd` has one single attribute, `name`, used to store the mandatory name of the production, as an identifier.

Element `semanticProd` has three direct descendants: `lhs`, `rhs`, `computation`, whose structure is explained in the next paragraphs and that are depicted in Figures 11, 12 and 13.

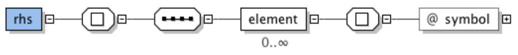
Element `lhs` (Figure 11) is used to refer to the nonterminal symbol on the `LHS` of the production. This element has a single attribute, `nt`, to refer to an existent `nonterminal`.



```
<lhs nt="students" />
```

Fig. 11. $\mathcal{X}$`AGra` Schema – Element **Semantic Productions**: `LHS` definition and example

Element `rhs` (Figure 12), stores the nonterminals on the `RHS` of a production. It is composed of a sequence of zero or more elements `element`. For this purpose, each `element`, has a single attribute, `symbol`, which is mandatory and represents a reference to a terminal or nonterminal symbol, already instantiated in the initial `symbols` structure.



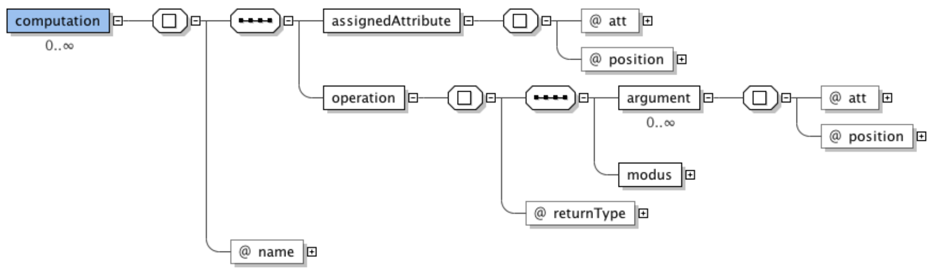
```
<rhs>
    <element symbol="student" />
    <element symbol="students" />
</rhs>
```

Fig. 12. $\mathcal{X}$`AGra` Schema – Element **Semantic Productions**: `RHS` definition and example

Element `computation` (Figure 13) is the last child of the element `semanticProds`. It represents an hard concept of `AG`s: the semantic rules.

This element has one attribute, `name`, used to give a name to the computation being declared. This attribute, despite being mandatory, is not a unique identifier: different computations can have equal names.

The structure of `computation` represents a pure abstraction of what is a semantic rule in an `AG` definition: the attribute to which a value is assigned, and



```
<computation name="getTheSum">
    <assignedAttribute att="students.sum" position="0" />
    <operation returnType="int">
        <argument att="student.age" position="1" />
        <argument att="students.sum" position="2" />
        <modus> $1 + $2 </modus>
    </operation>
</computation>
```

Fig. 13. $\mathcal{X}$AGra Schema – Element **Semantic Productions**: `Computation` definition and example

the operation that computes this value. Thus, the element `computation` has two children: the elements `assignedAttribute` and `operation`.

Element `assignedAttribute` is composed of two mandatory attributes: `att`, which is used to refer to an attribute; and `position`, which is a number that identifies the position of the symbol associated to the attribute in the list of elements of the production. That is, if the attribute is connected to the `LHS`, then the value for `position` must be 0. If the associated symbol belongs to the `RHS`, then its value should correspond to the position that the symbol occupies in the `RHS` sequence of symbols, starting with 1.

The element `operation` aggregates a sequence of zero or more elements `argument` and a single element `modus`. In addition to the elements, it has an attribute, `returnType`, used to store the data type of the value returned by the operation.

Elements `argument` are, in all aspects, equal to the `assignedAttribute` element. Each one has two attributes with the same name and the same semantic value underlaying, therefore they are used to refer to previous declared attributes. The difference is on the fact that this time, the attributes referenced are those used to compute the value in the operation.

The last element, `modus`[4], which is a simple text field to write the expression used to compute the value. Somehow, in this element's text, a reference to the argument attributes should be made. An example (and the convention established) is using $\$x$, where $x > 0$ is the position of the attribute in the sequence of arguments.

[4] *modus* is a latin expression for *way* (of computing something, in our case)

Aside the *importations* and the *functions* parts, the $\mathcal{X}$AGra's schema is now completely defined and explained, revealing the universality needed to store any AG for any AG-based compiler generator.

Next section briefly presents $\mathcal{X}$AGraAl, demonstrating one interesting applicability of the $\mathcal{X}$AGra dialect.

10. Conclusion

After many years working in specification and implementation of compilers supported by *Attribute Grammars*, it became clear that a modular and reusable approach to AG development is highly recommendable and necessary. On the other hand, the work on program comprehension tools emphasized the importance of software/data visualization. The combination of those two areas of R&D with a third one, the development of *Visual Languages*, gave rise to the proposal of creating a VL for AGs, since there are no other tools allowing it, according to our knowledge. The obligation to write text-based AG specifications imposed by several compiler generator tools and the habitual way of sketching AGs on paper in the form of a decorated tree, shortening the gap to the mental representation of an AG, reinforced the appropriateness of that proposal.

In this paper we introduced VisualLISA, a new *Domain Specific Visual Language*, which enables the specification of AGs in a visual manner and the translation of that visual AG into LISA or $\mathcal{X}$AGra (an XML notation to support generic AG specifications). $\mathcal{X}$AGra allows us to use this visual editor with other compiler-compiler tools.

We were mainly concerned with the design of the language, its formal and automatic implementation. In this phase of our project we neither focused on the usability of the language nor on its scalability. We focused on the specification, aiming at showing the formal work behind the visual outcome, and on the implementation of the underlying environment to specify AGs. At this point we highlighted the use of DEViL in order to create the desired environment, through a systematic approach of development. Also, an example was presented to show the steps to build an AG with VisualLISA.

In the future, it is our objective to perform at least, two experimental studies involving VisualLISA: one to assess the usability of the language regarding the visual vs textual approaches for developing AGs; and another one to test the scalability of the language and environment, regarding the hypothesis that it was created to cope with small AGs. We are also interested in assessing the comprehension of AGs; maybe VisualLISA would be very handy on this matter, working as AGs visualizer.

Concerning the applicability of $\mathcal{X}$AGra, we can translate it into the specific notation of any compiler generator tool. We call $\mathcal{X}$AGra loader to the program that performs this translation. As future work, the following translators are planed: $\mathcal{X}$AGra into LISA (a traditional LR parser generator); $\mathcal{X}$AGra into AntLR (an LL parser generator, based on an extended BNF grammar); $\mathcal{X}$AGra into `Eli` (an LR parser generator with special constructors). Finally, a translator from $\mathcal{X}$`AGra` to `Yacc` could be a challenging project.

Also, we developed a *Grammar Analyzer and Transformation* tool, $\mathcal{X}$`AGraAl`. that takes as input an `AG` written in $\mathcal{X}$`AGra`. Thus, the implementation of this tool shows the applicability of $\mathcal{X}$`AGra` as a universal and multi-purpose `AG` specification language.

References

1. Pereira, M.J.V., Mernik, M., da Cruz, D., Henriques, P.R.: VisualLISA: a visual interface for an attribute grammar based compiler-compiler (short paper). In: CoRTA08 — Compilers, Related Technologies and Applications, Bragança, Portugal. (July 2008)
2. Boshernitsan, M., Downes, M.: Visual programming languages: A survey. Technical report, University of California, Berkeley, California 94720 (December 2004)
3. Kastens, U., Schmidt, C.: VL-Eli: A generator for visual languages - system demonstration. Electr. Notes Theor. Comput. Sci. **65**(3) (2002)
4. Costagliola, G., Tortora, G., Orefice, S., De Lucia, A.: Automatic generation of visual programming environments. Computer **28**(3) (1995) 56–66
5. Mernik, M., Lenič, M., Avdičaušević, E., Žumer, V.: LISA: An interactive environment for programming language development. Compiler Construction (2002) 1–4
6. Mernik, M., Korbar, N., Žumer, V.: LISA: a tool for automatic language implementation. SIGPLAN Not. **30**(4) (1995) 71–79
7. Henriques, P.R., Pereira, M.J.V., Mernik, M., Lenič, M., Gray, J., Wu, H.: Automatic generation of language-based tools using the lisa system. Software, IEE Proceedings - **152**(2) (2005) 54–69
8. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. First edn. Pragmatic Programmers. Pragmatic Bookshelf (May 2007)
9. Solutions, U.: Ultragram - parser generator. http://www.ultragram.com/ (January 2010)
10. NorKen Technologies, I.: Programmar. http://www.programmar.com (January 2010)
11. Schmidt, C., Cramer, B., Kastens, U.: Usability evaluation of a system for implementation of visual languages. In: Symposium on Visual Languages and Human-Centric Computing, Coeur d'Alène, Idaho, USA, IEEE Computer Society Press (September 2007) 231–238
12. Ikezoe, Y., Sasaki, A., Ohshima, Y., Wakita, K., Sassa, M.: Systematic debugging of attribute grammars. In: AADEBUG. (2000)
13. da Cruz, D., Henriques, P.R.: Liss — the language and the compiler. In: Proceedings of the 1.st Conference on Compiler Related Technologies and Applications, CoRTA'07 — Universidade da Beira Interior, Portugal. (Jul 2007)
14. Golin, E.J.: A Method for the Specification and Parsing of Visual Languages. PhD thesis, Brown University, Department of Computer Science, Providence, RI, USA (May 1991)
15. Tolvanen, J.P., Rossi, M.: Metaedit+: defining and using domain-specific modeling languages and code generators. In: OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM (2003) 92–93
16. Costagliola, G., Deufemia, V., Polese, G.: A framework for modeling and implementing visual notations with applications to software engineering. ACM Trans. Softw. Eng. Methodol. **13**(4) (2004) 431–487

17. Ehrig, K., Ermel, C., Hänsgen, S., Taentzer, G.: Generation of visual editors as eclipse plug-ins. In: ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, New York, NY, USA, ACM Press (2005) 134–143
18. de Lara, J., Vangheluwe, H.: Atom3: A tool for multi-formalism and meta-modelling. In: FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering, London, UK, Springer-Verlag (2002) 174–188
19. Gray, R.W., Heuring, V.P., Levi, S.P., Sloane, A.M., Waite, W.M.: Eli: A complete, flexible compiler construction system. Communications of the ACM **35**(2) (February 1992) 121–131
20. Schmidt, C., Kastens, U., Cramer, B.: Using DEViL for implementation of domain-specific visual languages. In: Proceedings of the 1st Workshop on Domain-Specific Program Development, Nantes, France (July 2006)
21. Schmidt, C.: Generierung von Struktureditoren für anspruchsvolle visuelle Sprachen. Dissertation, Universität Paderborn (January 2006)
22. Oliveira, N., Pereira, M.J.V., da Cruz, D., Henriques, P.R.: VisualLISA. Technical report, Universidade do Minho (February 2009) `www.di.uminho.pt/~gepl/VisualLISA/documentation.php`.
23. GlassFish: Java architecture for XML binding. `https://jaxb.dev.java.net/` (June 2009)
24. GlassFish: Java API for XML processing. `https://jaxp.dev.java.net/` (June 2009)
25. da Cruz, D., Oliveira, N., Henriques, P.R.: Graal - a grammar analyzer (September 2009) Available at: `http://inforum.org.pt/INForum2009/programa`.

A. $\mathcal{X}$**`AGraAl`: A grammar analyzer based on $\mathcal{X}$`AGra`**

In this section we give a brief introduction to $\mathcal{X}$`AGraAl`, a *Grammar Analyzer and Transformation tool* that computes dependencies among symbols, grammar metrics, and grammar slices for a given criterion; moreover, $\mathcal{X}$`AGraAl` can also derive, from the original, shorter grammars combining slices or removing unitary productions (similar to re-factoring a program). $\mathcal{X}$`AGraAl` takes as input an `AG` written in $\mathcal{X}$`AGra`.

$\mathcal{X}$`AGraAl` is a platform independent tool, developed using Java. Java Architecture for XML Binding (`JAXB`) [23] and Java API for XML Processing (`JAXP`) [24] were used to process the input.

While parsing a $\mathcal{X}$`AGra` grammar using JAXB, $\mathcal{X}$`AGraAl` builds the identifiers table (`IdTab`) where it collects all grammar symbols and attributes; each identifier is associated with all its characteristics extracted or inferred from the source document. The identifiers table — that can be pretty-printed in `HTML` — complemented by the dependence graph (`DG`) — also printable using `Dot` and `GraphViz` — constitute the core of the tool. Traversing those internal representation structures, it is possible to implement the other $\mathcal{X}$`AGraAl` functionalities:

- Metrics, to assess grammar quality;
- Slicing, to ease the analysis producing sub-grammars focussed in a specific symbol or attribute;

– Re-factoring, to optimize grammars generating smaller and more efficient versions.

Metrics are organized in three groups of assessment parameters:

– Size metrics, that measure the number of symbols, productions, and so on (grammar and parser sizes);
– Form metrics, that describe the recursion pattern and measure the dependencies between symbols (the grammar complexity);
– Lexicographic metrics, that qualify the clearness/readablity of grammar identifiers, based on a domain ontology.

Slicing operation builds partial grammar with the elements that derive in zero or more steps on the criterion (backward slicing), or that are reachable from the criterion (forward slicing). The criterion can be either a symbol or an attribute. Slices are usually presented as paths over the dependence graphs. Figures 14 (a) and (b) illustrate a forward and a backward slice w.r.t the symbol *age*.

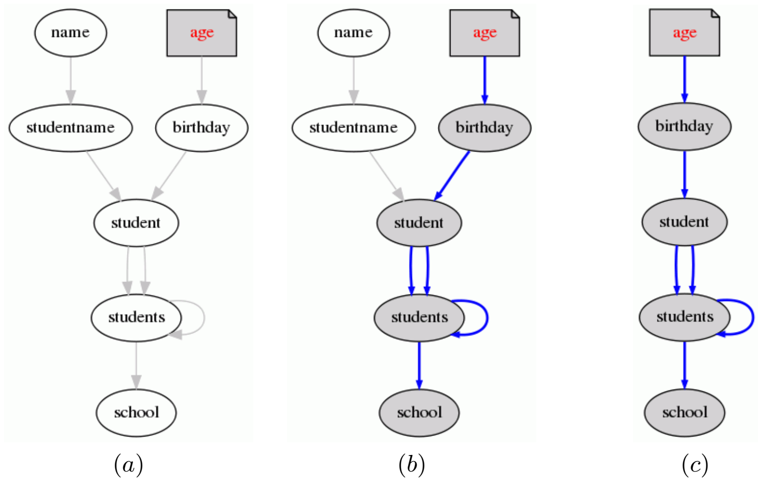


Fig. 14. Slices with respect to symbol *age*: (a) Forward slice; (b) Backward slice and (c) Combination of Forward and Backward slices

Re-factoring is a not so usual functionality that transforms the original grammar into a minimal one, removing all the *useless productions*. Another transformation also provided is the generation of a new grammar combining forward and backward slices with respect to the same symbol (see Figure 14 (c)).

Built in a similar way, `GraAL` [25] accepts as input a grammar written in `AntLR 3` and produces the same outputs. However, $\mathcal{X}$`AGraAl` beats `GraAL` in terms of generality as it consumes a grammar written in `XML`.

Nuno Oliveira*,* received, from University of Minho, a B.Sc. in Computer Science (2007) and a M.Sc. in Informatics (2009). He is a member of the Language Processing group at CCTC (Computer Science and Technology Center), University of Minho. He participated in several projects with focus on Visual Languages and Program Comprehension; VisualLISA and Alma2 the main outcome of his master thesis entitled ”Program Comprehension Tools for Domain-Specific Languages”, are the most relevant works. The latter came under ”Program Comprehension for Domain-Specific Languages”, a bilateral project between Portugal and Slovenia, funded by FCT. Currently, he is starting his PhD studies on Patterns for Architectures Coordination Analysis and Self-Adaptive Architectures, under MathIS, a research project also funded by FCT. Meanwhile he is an assistant-lecturer (practical classes) in a course on Imperative Programming, at University of Minho.

Maria João Varanda Pereira*,* received the M.Sc. and Ph.D. degrees in computer science from the University of Minho in 1996 and 2003 respectively. She is a member of the Language Processing group in the Computer Science and Technology Center , at the University of Minho. She is currently an adjunct professor at the Technology and Management School of the Polytechnic Institute of Bragança, on the Informatics and Communications Department and vice-president of the same school. She usually teaches courses under the broader area of programming: programming languages, algorithms and language processing. But also some courses about project management. As a researcher of gEPL, she is working with the development of compilers based on attribute grammars, automatic generation tools, visual languages and program understanding. She was also responsible for PCVIA project (Program Comprehension by Visual Inspection and Animation), a FCT funded national research project; She was involved in several bilateral cooperation projects with University of Maribor (Slovenia) since 2000. The last one was about the subject “Program Comprehension for Domain Specific Languages”.

Pedro Rangel Henriques*,* got a degree in “Electrotechnical/Electronics Engineering”, at FEUP (Porto University), and finished a Ph.D. thesis in “Formal Languages and Attribute Grammars” at University of Minho. In 1981 he joined the Computer Science Department of University of Minho, where he is a teacher/researcher. Since 1995 he is the coordinator of the “Language Processing group” at CCTC (Computer Science and Technologies Center). He teaches many different courses under the broader area of programming: Programming Languages and Paradigms; Compilers, Grammar Engineering and Software Analysis and Transformation; etc. Pedro Rangel Henriques has supervised Ph.D. (11), and M.Sc. (13) thesis, and more than 50 graduating trainingships/projects, in the areas of: language processing (textual and visual), and structured document processing; code analysis, program visulaization/animation and program comprehension; knowledge discovery from databases, datamining, and data-cleaning. He is co-author of the “XML & XSL: da teoria a prática” book, published by FCA in 2002; and has published 3 chapters in books, and 20 journal papers.

Daniela da Cruz*,* received a degree in “Mathematics and Computer Science”, at University of Minho, and now she is a Ph.D. student of “Computer Science” also at University of Minho, under the MAPi doctoral program. She joined the research and teaching team of “gEPL, the Language Processing group” in 2005. She is teaching assistant in different courses in the area of Compilers and Formal Development of Language Processors; and Programming Languages and Paradigms (Procedural, Logic, and OO). As a researcher of gEPL, Daniela is working with the development of compilers based on attribute grammars and automatic generation tools. She developed a completed compiler and a virtual machine for the LISS language (an imperative and powerful programming language conceived at UM). She was also involved in the PCVIA (Program Comprehension by Visual Inspection and Animation), a FCT funded national research project; in that context, Daniela worked in the implementation of “Alm”, a program visualizer and animator tool for program understanding. Now she is working in the intersection of formal verification (design by contract) and code analysis techniques, mainly slicing.

Bastian Cramer*,* received his degree in Computer Science from the University of Paderborn, Germany in 2005. Then he joined the research group ’Programming Languages and Compilers’ of Prof. Kastens at the same university. His research focus is the generation of software from specifications and especially the generation of environments for visual domain specific languages. He has several years of experience in language design in corporation with the automotive industry. Currently he is working on his PhD concerning simulation and animation of visual languages.

Received: November 15, 2009; Accepted: February 24, 2010.