View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Biblioteca Digital do IPB

VisualLISA: Visual Programming Environment for Attribute Grammars Specification

Nuno Oliveira*, Pedro Rangel Henriques*, Daniela da Cruz*, Maria João Varanda Pereira†,

* University of Minho - Department of Computer Science,
Campus de Gualtar, 4715-057, Braga, Portugal
Email: {nunooliveira, prh, danieladacruz}@di.uminho.pt
† Polytechnic Institute of Bragança
Campus de Sta. Apolónia, Apartado 134 - 5301-857, Bragança, Portugal
Email: mjoao@ipb.pt

***Abstract*—The benefits of using visual languages and graphical editors are well known. In some specific domain it is really crucial to program with graphical representations, icons, geometric objects, colors and so on. Nowadays it is possible to easily implement a visual language, constructing, automatically, visual editors for it.**

In this paper we want to emphasize how it is possible to easily specify a huge amount of complex information, associated with an attribute grammar, using graphical objects and a very intuitive modular approach. For that purpose we present a new visual language to specify attribute grammars (called VisualLISA) and we present also a modular approach that uses VisualLISA in an integrated editor to draw attribute grammars.

I. Introduction

Attribute Grammars (AG), introduced by Knuth [1], are Context-Free Grammars (CFG) where productions are augmented by semantic rules, so that the terminal and non-terminal symbols can have attributes associated which have different values depending on the context they appear in.

In this paper we will present a new Visual Language (VL) and a Visual Programming Environment (VPE) to visually draw Attribute Grammars — VisualLISA. We constructed such an environment following a systematic approach based on the compiler construction theory [2] and resorting to DEViL[1], in order to generate the environment with little effort. Although we present some steps on the development of VisualLISA, our focus is on the user interaction with this environment.

VisualLISA's main purpose is to be used as a graphical front-end for LISA [3]. The environment is generated from the specification of a visual language, and ensures the possibility of drawing, syntactically and semantically correct, attribute grammars, in an integrated editor. The visual specification of the attribute grammar is production-oriented and incremental.

[1]http://devil.cs.upb.de

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 into LISA notation or, alternatively, into a universal XML representation designed to support AG specifications.

The remainder of this paper is organized as follows. In Section II, visual languages theme is addressed. VisualLISA, is presented in Section III; although have been formalized by an AG, in this section we just define its syntax and semantics. In Section IV, the user interaction is illustrated step-by-step through an example. In Section V, the usability of the language is discussed based on the Cognitive Dimensions Framework. We close the paper, in Section VI, with conclusions about the usage of visual languages and in specific about our visual environment and associated language.

II. Visual Languages

There is not a consensual definition for visual languages. The intuition says that almost everything that uses composition of figures, instead of words, in order to transmit a message, can be considered a VL. In this sense, there are many types of VL. Examples cover a large range from the daily used musical scores or traffic signals, and those more specific like modeling languages for definition of Entity-Relation Diagrams (ERD), Class Diagrams, and so forth.

Modeling Languages fall within a restrict area of VLs: the Visual Programming Languages (VPL). A VPL aims at offering possibilities of solving problems by describing their properties or their behavior using graphical/iconic definitions [4]. Icons are used to be composed in a space with two or more dimensions, defining sentences that are formally accepted by parsers. The shape, color and relative position of these icons are relevant issues.

VPLs are used in many areas of computing. In databases, the main usage of visual languages is to help on drawing the tables and relations between them, rather then using SQL notation. In software development, they are mostly used to draw the system's structure and its behavior with modeling languages as referred before. In interface design for stand-alone or web-based applications, visual programming languages that allow the drag, drop and composition of interface elements like buttons, textboxes, windows, and so on.

Roughly speaking, VPLs can be classified [5] concerning the language paradigm (functional, imperative, rule-based, etc), and visual representation in use (diagrammatic, iconic, pictorial sequences or sound-based). The VL/HCC Symposium bibliography—reachable at the URL http://web.engr.oregonstate.edu/~burnett/vpl.html—has become a widely-used resource for people seeking information on visual languages, visual programming, visual software engineering, human use of programming languages and tools, and so on. There, papers on Software Visualization are classified according to Margaret Burnett's criteria based on the following parameters: Technique (interaction with the user, output generation); Applications; Performance; and Visualization Domains (kind of drawings, kind of visualizations and kind of source language).

Textual languages are the common artifacts used to develop programs and complex systems. The history says that programming languages have passed through many conceptual levels, from machine level to higher levels, but all of them with a common characteristic: specifications are sequential (from left to right) compositions of textual characters. This implies the knowledge of different syntaxes and the necessity of being aware of little details like semicolons at the end of instructions, matching of braces, among others. Also, it is needed to know reserved words or constructors that are always different from language to language. For example, the SQL notation to create tables and the relations between them may be very complex, because it is needed to know how a table is created, how the several fields are declared and how the primary or foreign keys are defined. In the other way around, visual programming frees the developers from these small details. The creation of a table in a database is as simple as dragging an icon into a drawing area. To create a relation between tables is as simple as connecting two tables with an arrow, a line, and so forth.

However, the specification of a VL is costly. While a traditional language can be defined only by an attribute grammar, a visual language must be specified resorting to an attribute grammar (or other formalism) that builds the syntax, images that define the language icons, visual patterns to compound the layout and interaction, among other things related to the construction of the programming environment associated. The processing of visual specifications also takes more time than processing textual specifications.

III. VisualLISA

In this section we present a brief overview about VisualLISA, adressing its architecture, the formal specification of various aspects concerning the implementation and the implementation resorting to a systematic approach. For more details about the implementation, see [6].

A. The Concept

VisualLISA is a visual programming environment for specification of AGs. It has not a complex architecture because its purpose is to be a graphical front-end for LISA and other compiler-generators. As shown in Figure 1, an editor, where the attribute grammar is specified, mechanisms used to validate the grammar drawn, and a processor to translate the iconic sentences into LISA notation or XML, compose the programming environment. The generated LISA specifications can be passed straightforward to LISA system in order to create the compiler for the language defined in VisualLISA. The development of an XML dialect to support the abstract structure of AGs gives the system more versatility, because it allows a functional separation between VisualLISA and the compiler generator tools. Moreover, this XML notation opens doors for new ways of using VisualLISA, rather then just as an AG specification tool.

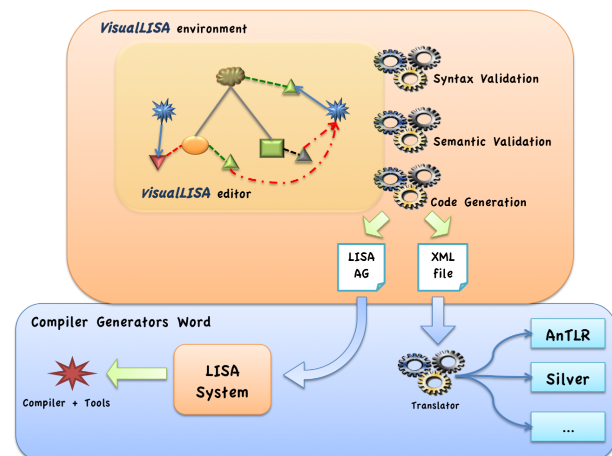

Fig. 1. Architecture of VisualLISA

B. Specification

The specification of the VisualLISA's environment is directly related with the formal specification of the VL. That specification lies on three main definitions:

the syntax, the semantics, and the translation. As the description of the development process of this tool is not in the scope of this article, we are not going to focus in detailing this formal specification. However, we present a brief overview of what was done.

We used the Picture Layout Grammars (PLG) [7] formalism to formalize the syntax of VisualLISA, which can be described by the following summary: *The terminal and non-terminal symbols of the Right-Hand Side (RHS) of a production should be connected to the Left-Hand Side (LHS) symbol. The production should be decorated with attributes, so, connections between the terminal/non-terminal symbols and the attributes are mandatory to understand to which symbol the attribute belongs. At the end, the attributes should be associated to semantic rules defining their values. These rules should be defined reusing the layout of a production, but in a separated view*.

Besides of being used to define this visual aspect of the language, PLG was also used to define hard syntactic constraints concerning the connections between the symbols. The semantic constraints (or contextual conditions) of VisualLISA are directly related with the attributes of the language and their values in each context. Inside these contexts, the attribute values must converge to hold a condition. The most important constraints that must hold in VisualLISA are concerned with the correct specification of an AG. These constraints can be separated into two major groups: one concerns with the syntactic rules and another with the respective semantic rules. The complete set of constraints, with their formal specification, can be found in [8]. The following sentences present two examples of these constraints.

1) Each production should have one and only one LHS symbol;
2) The data-type of an assigned attribute must match with the data-type of the operation's output.

Once the drawing of an AG is complete and semantically correct, it can be translated into textual notation. A translation ($\mathcal{L}_s \rightarrow \tau \rightarrow \mathcal{L}_t$) is the transformation of a source Language (Ls) into a target language (Lt). τ is the mapping between the productions of the $\mathcal{L}_s$ (in our case VisualLISA) and the fragments of $\mathcal{L}_t$. LISA and XML are the target languages of VisualLISA's translation process.

LISA is a compiler generator tool based on attribute grammars. It generates a compiler from the specification of an AG, and also other tools, as can be seen in [9]. Based on the CFG of the LISA language we were able to find sections that divided the language into fragments.

Based on the knowledge about attribute grammars, and in the study made to conceive the structure of LISA, the definition of an XML notation to support attribute grammars in an abstract way was a straightforward task. We defined such dialect, XAGra — XML dialect for Attribute Grammars — resorting to a schema. The complete structure of XAGra can be separated into five elements: $i)$ *symbols* - where terminal, non-terminal and the start symbol are declared; $ii)$ *attributesDecl* - where the attributes are associated to the symbols; $iii)$ *semanticProds* - where the productions and the semantic rules are defined; $iv)$ *importations* - used to store the modules or packages necessary to perform computations and $v)$ *functions* - where the users should declare their auxiliary functions.

C. Implementation

In order to achieve a systematic and effortless implementation of VisualLISA, we submit VPE tools to some experiments. From these experiments we chose DAViL because it gave us more comfort about the features, usage and final output. With DEViL (but not exclusively because of it) the development of the VL and associated VPE can be systematized in four main steps: i) Abstract Syntax Specification; ii) Interaction and Layout Definition; iii) Semantics Implementation and iv) Code Generation.

To define the abstract syntax we translated the PLG formal definition of VisualLISA into the modular (object-oriented AG) notation of DEViL. In order to make possible the specification of separated computation rules, it was used a DEViL specific feature: the coupling of structures [10]. This feature copies a part of the main structure and maintains synchronization between the original structure and the copied one. In VisualLISA it means that the layout of production is replicated for every semantic rule and both are always synchronized. The specification of the layout and the interaction consists in the definition of the buttons of the dock and the creation of figures to define the icons of the language. Figure 2 shows a button (rectangular shape) and an icon of the language (cloud shape), which was used to identify the LHS symbols of VisualLISA's productions.

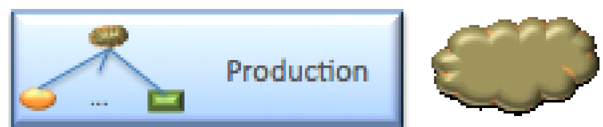

Fig. 2. Example of a Button (rectangular-shaped image) and an Icon (cloud-shaped image) of VisualLISA Interface.

In DEViL, besides these steps we also specified views of the language, in order to ease the comprehension and the modularity of it. All the layout specifications are based in the inheritance of already made interfaces called Visual Patterns [11]. From the abstract structure, DEViL creates an Abstract Syntax Tree (AST). This

allows the definition of a tree-walker function in order to traverse the tree and execute some actions in given contexts. With this approach we defined the module of semantic verification based on the constraints specified before. However, for implementing the code generation (or translation) module, it is used the traditional AG approach. We defined the semantic rules to translate the iconic sentences into text, associating some necessary attributes to the grammar symbols.

As seen in the specification, there could be found some static fragments in the target code notations. Regarding this, we used template files to structure out the output. This eases the translation process and future maintenance of this module. Besides the templates, we used auxiliary functions so that we could solve problems like ordering the RHS of a production based on their position along the X-axe. With the specification of these four modules, we were able, using DEViL, to generate VisualLISA as a stand-alone visual programming environment.

IV. User Interaction Through an Example

In this section, the interaction with VisualLISA editor will be shown using an example. Although bigger grammars have been tested, we use this small example for an easier explanation of the interaction. In this example, the source text defines a set of students and for each student a name and an age are specified. The objective of the example is to visually define the AG (henceforth called Students Grammar) where the sum of the ages will be the generated output. For Instance, taking a concrete source text like the one shown in Listing 1, the output will be 37.

Listing 1. Source Text Example for the Students Grammar

```
1
2 Peter Gabriel        12
3 John Lennon          13
4 Maria Callas         12
```

Listing 2 shows the context-free grammar in BNF notation for the Students Grammar, and also presents the semantic rules associated to each production in order to calculate the sum of the ages. The terminal symbols must be lexically defined as [a-zA-Z]+ and [0-9]+ for *name* and *age* respectively.

Listing 2. Formal Specification of the Students Grammar

```
1
2 P1: Students: Student Students
3            {Student.sum = Students.sum + Student.sum}
4 P2:          | Student
5            {Students.sum = Student.sum}
6 P3: Student : name age
7            {Student.sum = name.value}
```

A. Developing with VisualLISA

Before starting the specification, we present VisualLISA's work-area interface in Figure 3.

In this image are represented the four windows used for an AG specification in VisualLISA. The first window (rootView) is the one presented when a new specification is started. Here the user can declare productions and access the global definitions of the AG being specified. The latter (defsView) corresponds to the second window in Figure 3 and it is where the user defines the global lexemes, functions, new data-types and modules to import. The third window (prodView) is used to model the productions declared in the first one. Its drawing area is separated into two parts: the largest is to draw the production layout and the thinnest is used to associate computation rules to the production. The computation rules are modeled in the fourth and last window (ruleView) reusing the production layout.

All these windows have a dock with buttons. These buttons are the main way to model the AG, by dragging them into the drawing area.

Regarding the formal definition of the Students Grammar, its specification in VisualLISA will be divided into three productions and one computation rule associated to each production. This specification is depicted through Figure 4 to Figure 6.

A production is always defined by a LHS and a RHS. In VisualLISA, the cloud-shaped icon identifies the LHS symbol. The oval and the rectangular icons identify the non-terminal and the terminal symbols, respectively.

To build a production, the user start by specifying the RHS because the LHS is automatically drawn when a terminal or non-terminal is pushed into the drawing area for the first time, besides it, those two symbols are automatically connected to create the production tree. This connection is always created whenever another RHS symbol is dragged into the drawing area.

The same does not happen with the attributes. To associate an attribute to a symbol, the user should drag the attribute icon and the respective connector line into the drawing area, and then attach the connector to the attribute and the symbol. There are three types of attributes (inherited (red inverted triangle), synthesized (green triangle) and intrinsic (grey triangle)). In order to avoid bad constructions, the edition is directed by the syntax of VisualLISA. For example, the intrinsic attribute can only be associated to a terminal symbol; the editor does not allow any other connection of this attribute.

When double clicking in an icon, a simple form with the symbol's properties appears. There, is possible to write the name, or choose a data-type for that symbol, when applied.

After a production modeling, is possible to associate computation rules, by dragging the respective icon into the drawing area. This declares a new instance of computation rule that can be opened in the ruleView. In this new window, the layout of the production is presented

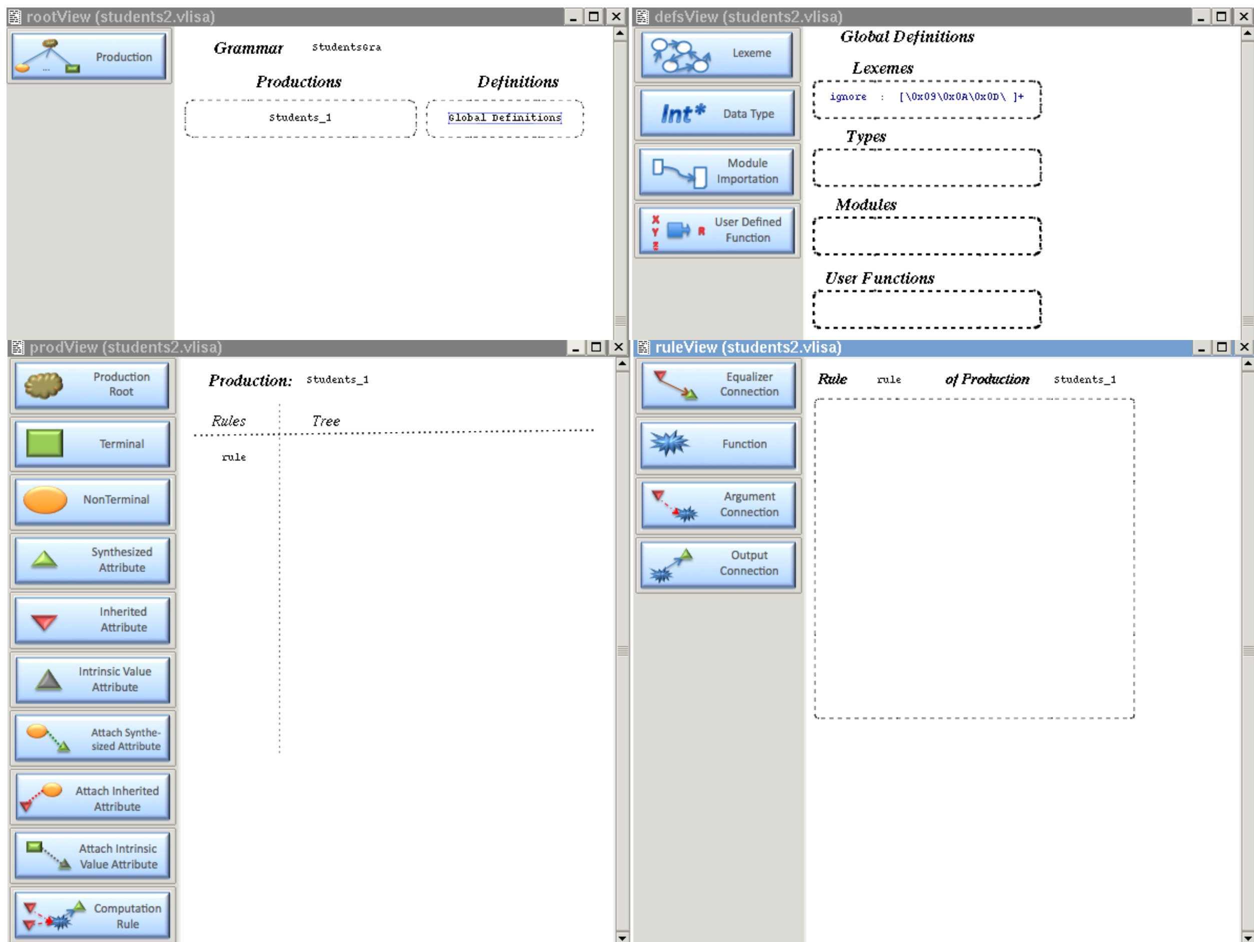

Fig. 3. Main Windows for Grammar Specification

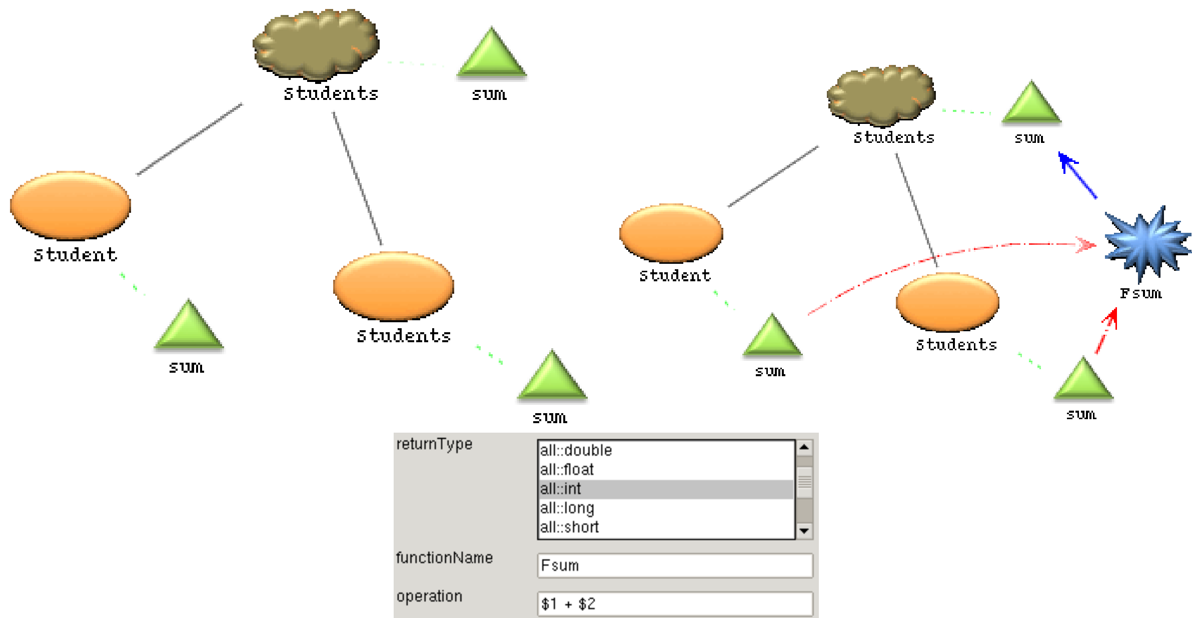

Fig. 4. Production P1 (left), associated Computation Rule (right) and Definition of the Operation (center-below).

by default. The user only has to concern about the construction of the semantic rule.

A semantic rule is modeled using four different icons. In Figure 4 we show three of them: $i)$ function (star-shaped icon) - where the mathematical operation to compute a value is specified. Figure 4 at the right, shows an example of such specification; $ii)$ argument connector (red dashed-arrow) - is always attached to an attribute and to a function to specify that the value of the attached attribute is used as argument in the function's operation; $iii)$ output connector (blue full-arrow) - is used to attach a function to an attribute. It means the assignment of the function's output to an attribute. In Figure 5 we present the fourth: $iv)$ identity function (brown full-arrow) - it connects two different attributes, and means that the target attribute is being assigned with the value of the source one.

In Figure 6 is depicted the production defining the nonterminal Student. There can be seen the terminal symbols and their association with an intrinsic attribute. Notice the name given to that attribute; in fact it is a function used in LISA to access the value of the attribute. We used this name because we will generate code for LISA. The regular expression for a terminal is defined locally on that terminal, as can be seen in Figure 6.c).

The Students Grammar was completely defined in VisualLISA. But there is not any instruction to print the final output. To achieve this, we have to define a new production (with a new start symbol). This happens because we must define a function to print the result, and a function must always have an output value associated to an attribute.

The new production is depicted in Figure 7. Creating a new production is not the only way to achieve the requirement proposed. With this approach we can address an important issue: the order of the production specification. In the list of productions, the user must set the production with the start symbol at the top of the list. This is the way to define the start symbol of the grammar. The reminder productions have not a predefined order.

After specifying the AG, the user can generate code. Before generating code, VisualLISA automatically performs semantics verification, in order to warn the user of possible errors. VisualLISA generates LISA specification that is the input for LISA compiler generator. It is also possible to generate a XAGra specification, with the same information, that can be adapted to be used as an input of other compiler generator. In Figure 8 we show the generated code for both notations. As the resultant files are big, we only show a small part that depicts the translation respective to the production and computation rule associated in Figure 4.

Using LISA our example will be compiled, the output will be evaluated and a set of visualizations generated.

V. USABILITY ANALYSIS

VisualLISA is a domain-specific visual language created to make easier the specification of attribute grammars. There are cognitive dimensions that can be used to test the usability of this kind of programming language [12]. Some of them are: closeness of mapping, role-expressiveness, consistency, viscosity, visibility and error-proneness. The following analysis is based on our beliefs and on the feedback received from several experiments that have been done (involving the members of the development team, as well as, some students).

The language was crafted based on the user mental representation of attribute grammars: a decorated tree. In this case, the gap between the problem domain (what we want to solve) and the program domain (how to solve) is smaller. It is easier for the user to specify the attribute grammar using a graphical representation of that decorated tree (closeness of mapping).

On the other hand, a visual programming language must provide facilities for coloring, commenting, grouping, modularizing and so on. So, different colored graphical icons were chosen for VisualLISA and an intuitive composition process was used to create the decorated tree. This kind of features improves the role-expressiveness of the language.

VisualLISA is a consistent language because it is easy for the user to infer how to add a new symbol or how to specify an attribute evaluation. Specifications are created in a very systematic way (consistency).

VisualLISA uses a modular specification approach, which turns less hard to perform changes (viscosity). The modular approach can also solve problems related with scalability. Since every production has a new specification window we never get huge and confuse graphical representations. The same happens with semantic rules. In order to specify one rule we just have to choose the production and decorate it with attributes, relations and functions and to do that a new window is used.

Beside this, it is possible to get a global view of the grammar (list of productions and rules) avoiding loosing the connection between each production and the whole grammar (visibility).

In visual programming languages there are fewer syntactic details to take into account: situations like unpaired delimiters, discontinuous constructors, missing separators, missing variable initialization and so on can not happen in this kind of language. Instead of that, the programming style is based on drag and drop operations and it is possible to restrict that actions in order to follow the correct syntax of the language. There are also semantic constraints to attain but the user can be guided in order to avoid both syntactic and semantic errors (low or inexistent error-proneness).

Fig. 5. Production P2 (a) and associated Computation Rule (b).

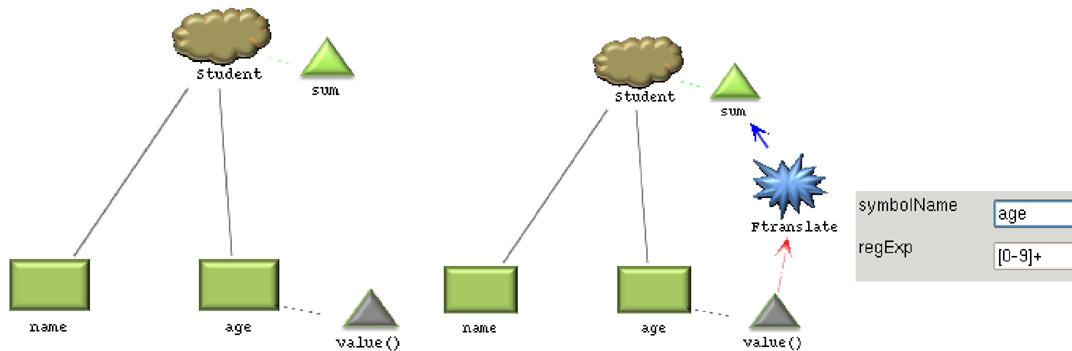

Fig. 6. Production P3 (a), associated Computation Rule (b) and Terminal symbol Properties (c)

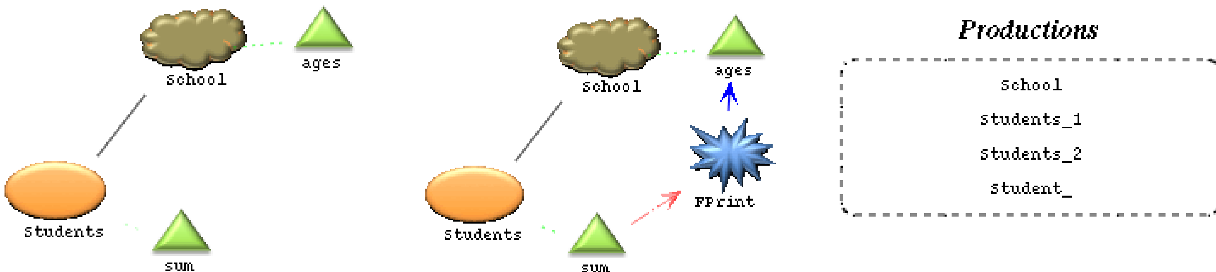

Fig. 7. Production P0 (left), associated Computation Rule (center) and the List of Productions of the Grammar (right).

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

(a)

```
<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>
```

(b)

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

This study is just our point of view about the language and the programming environment we developed; we are, obviously, suspects! A concrete usability test must be done in order to confirm our beliefs. With that purpose we will measure the user interaction according to the cognitive dimensions above. In the future, we expect to pick a group of students with similar experience on attribute grammars, and propose a set of questionnaires with problems to be developed using VisualLISA. So that we can gather information about these results to compare against our beliefs.

VI. Conclusion

Attribute Grammars (AG) specification is not as easy as people would desire. The difficulties of choosing the appropriate attributes and conceiving the respective evaluation rules are significant, as well as the effort to write the complete specification. Normally is easier to sketch it on paper. This strategy allows the developers to create a syntax-independent abstract mental model. However, after being sketched, the productions and the semantic dependencies between attributes are not more than scribbling on paper. The person who drew it must go through the translation of the pencil strokes into the concrete syntax of the compiler generator.

In this paper we presented a new visual language to create AGs, where the improvement of the user interaction was the basis for choosing the icon shapes, colors and general schemas for the specification. We also showed, briefly, a systematic approach to develop its underlying programming environment (VisualLISA) taking advantage from the usage of DEViL. Resorting to a running example, we described, how the user should interact with VisualLISA to create a new AG. This method of specifying AGs is closer to the mental model adopted by the users to sketch the computation of the attributes in a semantic-directed translation. Therefore, the gap between the mental model and that methodology is small. We made a small and self-critic usability analysis based on the cognitive dimensions framework, to enhance the last sentence. However, a concrete usability test with target users is lacking and was left for further work.

VisualLISA, as a graphical front-end for LISA, can be used to generate compilers and other language-based tools. Moreover, it can be easily adapted to work with other compiler generator tools since it produces an XML dialect as output. We are aware of the non-scalability of VLs. For long AGs, maybe the textual approach is better. Our approach is chiefly appropriate for beginners and small specifications.

From long time ago, visual programming languages were created for databases management, image-processing, user-interface (GUI) generation, and so on. Several grammar formalisms and compiler generation techniques were created in order to implement VLs. An effort was made in order to systematize the generation of visual languages and some tools, like VLDesk, DEViL, and so on, appeared. When creating our VL we did not intend to define a new visual interaction paradigm, instead we desired to take profit of visual interactions to implement a new framework for describing AGs, because, reviewing the literature, we did not find a similar work.

We hope we have emphasized how it is possible to easily specify a huge amount of complex information, as it is the case of an AG, using graphical objects and a very intuitive modular approach.

Acknowledgemnts

We would like to thank *Bastian Cramer*, from University of Paderborn, for his constant support during the development of VisualLISA, and his continuous interest on collaborating with us on other projects.

References

[1] D. E. Knuth, "Semantics of context-free languages," *Theory of Computing Systems*, vol. 2, no. 2, pp. 127–145, June 1968.

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

[3] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer, "LISA: An interactive environment for programming language development," *Compiler Construction*, pp. 1–4, 2002.

[4] M. Boshernitsan and M. Downes, "Visual programming languages: A survey," University of California, Berkeley, California 94720, Tech. Rep., December 2004.

[5] M. Burnett and M. Baker, "A classification system for visual programming languages," *Journal of Visual Languages and Computing*, vol. 5, pp. 287–300, 1994.

[6] N. Oliveira, M. J. V. Pereira, P. R. Henriques, D. da Cruz, and B. Cramer, "Visuallisa: A domain specific visual language for attribute grammars," in *Proceedings of the 3rd Compilers, Programming Languages, Related Technologies and Applications (CoRTA'2009)*. Lisbon, Portugal: Faculdade de Ciências da Universidade de Lisboa, September 2009, (To Appear).

[7] E. J. Golin, "A method for the specification and parsing of visual languages," Ph.D. dissertation, Brown University, Department of Computer Science, Providence, RI, USA, May 1991.

[8] N. Oliveira, M. J. V. Pereira, D. da Cruz, and P. R. Henriques, "VisualLISA," Universidade do Minho, Tech. Rep., February 2009, www.di.uminho.pt/~gepl/VisualLISA/documentation.php.

[9] P. R. Henriques, M. J. V. Pereira, M. Mernik, M. Lenič, J. Gray, and H. Wu, "Automatic generation of language-based tools using the lisa system," *Software, IEE Proceedings* -, vol. 152, no. 2, pp. 54–69, 2005.

[10] U. Kastens and C. Schmidt, "Vl-eli: A generator for visual languages - system demonstration," *Electr. Notes Theor. Comput. Sci.*, vol. 65, no. 3, 2002.

[11] C. Schmidt, U. Kastens, and B. Cramer, "Using DEViL for implementation of domain-specific visual languages," in *Proceedings of the 1st Workshop on Domain-Specific Program Development*, Nantes, France, Jul. 2006.

[12] T. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131–174, June 1996.