VisPro: A Visual Language Generation Toolset

Da–Qian Zhang Kang Zhang

Department of Computing, Macquarie University, NSW 2109, Australia
{daqian, kang}@mpce.mq.edu.au

Abstract

The implementation of visual programming languages (VPLs) and their supporting environments is time–consuming and tedious. To ease the implementation, researchers have developed some high level tools, which can greatly reduce the effort of developing VPLs. None of them, however, can be easily used to create a complete visual language in a seamless way like the lex/yacc tools for textual language constructions.

This paper presents VisPro, a toolset for developing diagrammatic VPLs in a way similar to the lex/yacc. VisPro consists of a set of visual programming tools. It divides the process of a VPL construction into two steps: lexicon definition and grammar specification. The lexicon definition defines visual objects and a visual editor, and the language grammar is specified with graph rewriting rules (associated with actions written in Java). The compiler for the VPL is automatically created according to the grammar specification. A target VPL is a visual programming environment which contains the compiler and the visual editor.

1. Introduction

The implementation of visual programming languages (VPLs) and their supporting environments is usually time–consuming and tedious. The task is much harder than implementing textual languages [1]. Researchers have developed some high level tools to ease the implementation of visual languages. For example, Escalante supports the construction of applications for visual languages that are based on object–relationship abstractions [2]. It provides mechanisms for iterative design, rapid prototyping and generation of visual language applications within an integrated environment. DiaGen [3] is a tool for producing diagram editors, which can be used to construct visual programs. These tools can greatly reduce the effort of developing visual languages, although they focus mainly on the construction of user interface aspects of VPLs.

SPARGEN [4] is a visual language compiler generator. Its generated parser supports additional action routines written in C++. This allows it to specify complicated actions in the form of rules. It, however, does not support the generation of a visual programming environment.

PROGRES [5,6,7] can generate both programming environments and parsing algorithms. But it cannot use existing programming languages (e.g. C, C++, Java) to specify the actions of its rules directly. Instead, it uses a simple textual language which is a part of the system. This limits the functionality of target VPLs. Moreover, it has limited visual object construction ability and can only produce syntax–directed editors, which are often difficult to use. PROGRES uses layered graph grammar to specify VPLs. This results in a very complicated parsing algorithm, and its performance reaches exponential time.

Tools such as SIL–ICON [8] and VLCC [9] use parameterisable frameworks to support VPL generation. They are easy to use since generating target VPLs is simply done by customising the predefined frameworks through domain specifications. The SIL–ICON [8] compiler is based on the generalised icon theory and limited to iconic VPLs. Using positional grammars as the underlying theory and pure images as single–level visual objects, VLCC [9] has limitations in its visual representations.

A lex/yacc tool divides the process of a language creation into two steps: lexicon definition and grammar specification. The created lexicon and grammar analyzers are combined together to serve as a language parser. In particular, its rules (i.e. grammar) can have actions written in C, which allows lex/yacc to specify a wide range of textual languages.

This paper presents a toolset called VisPro which provides the similar mechanism as lex/yacc in the process of constructing VPLs. It is very easy to use, since the tools in VisPro are meta visual languages. To formally represent VPLs, VisPro uses reserved graph grammars [10] to express a wide range of diagrammatic VPLs. A graph grammar in VisPro is a set of graph rewriting rules associated with actions written in Java. The target language compilers for a large class of diagrams can be automatically generated in polynomial time by VisPro according to the grammar specifications. Moreover, a set of language components (i.e. visual objects) can be created through direct manipulation and a visual editor can be produced according to control specifications. A visual programming environment integrating the visual editor and the compiler is then created.

VisPro provides a high level support for VL developers to rapidly construct a wide variety of domain–oriented diagrammatic VPLs. It can easily create both the user interface and the underlying language of any specific visual language.

2. The VisPro Toolset

With VisPro, a complete VPL is specified by a lexicon definition and a grammar specification. A lexicon definition describes the VPL's visual objects and the editor with which the visual objects can be used to create a program. A grammar specification (syntax and semantics) defines whether the program is valid and what it means. A visual programming environment is then created automatically based on the definition and the specification.

2.1. The Tools

A VPL's lexicon means the classes of visual objects (VOs). Each specific VPL has its own lexicon. In VisPro, the lexicon definition is independent of the design of a VPL (syntax and semantics), so that a general lexicon library can be predefined. This can avoid repetitive work when a number of VPLs need to be designed and they use some common VOs.

Each tool in a VisPro framework can be customized to a domain–oriented functional module when domain specifications are provided. The VisPro framework and the specification tools, together called the VisPro system, provide a complete and effective support for VPL generation.

The specification tools are developed based on the VisPro framework architecture. They are Visual Object Generator (VO Generator), Control Specification Generator (CS Generator), and Rule Specification Generator (RS Generator), which are used to generate view specifications, control specifications and parsing specifications respectively, as shown in Figure 1.

A *VO generator* is a graph editor for constructing visual objects through direct manipulation. A constructed visual object is not just an image. It is manipulatable and may be a composite graph, whose components can be manipulated independently. A *CS generator* is used to specify the *control* over visual objects constructed by the VO generator. An *RS generator* specifies the domain knowledge in a graph rewriting rule formalism. Consequently, the domain specifications can be produced automatically by the specification tools in a format that the framework can recognize.

In VisPro, the object–oriented language Java serves as a low level specification tool to specify details which are not desirable to be specified in these visual specification tools. This arrangement allows us to precisely construct simple but effective visual tools while maintaining completeness.

Figure 1 illustrates the use of the VisPro system for the VPE

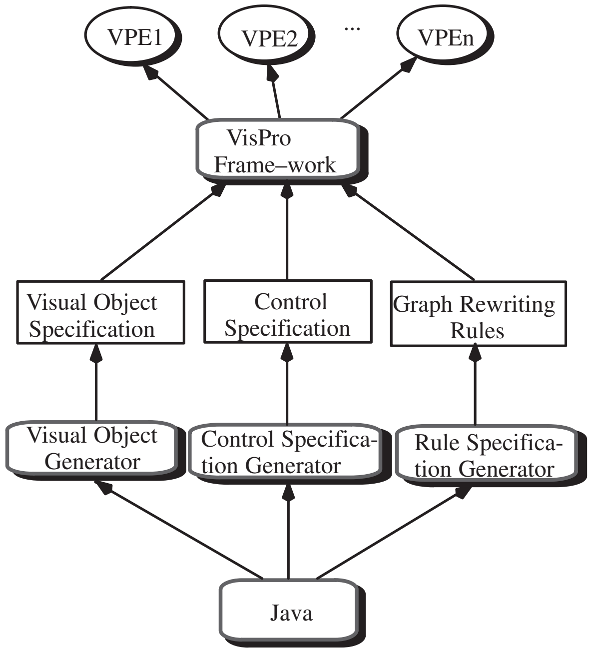


Figure 1: Constructing VPEs with VisPro

generation. First, the user creates desired visual objects using the VO generator. He/she then uses the CS generator to choose the pre–defined or construct more sophisticated editing commands for each created visual object, and use the RS generator to specify the grammar rules and actions. During the construction process, the low level details such as actions may be specified in Java in a provided text editor. Once all the necessary specifications are provided, they are fed into the VisPro framework which then generates a customized target VPE.

2.2. A Case Study

In the following sections, we will explain the functions and the use of each specification tool by demonstrating the construction of a simple visual programming environment called *summation*. More sophisticated VPEs can be similarly built through the same process but with more interactions. Figure 2 is a snapshot during the use of the generated *summation*. In *summation*, one can sum up integers and visualize results. It has three visual classes: *numbers, pluses,* and *scalers*. A *number* stores an integer which can be entered through keyboard. A *plus* receives integers from two *numbers* and produces their summation also as an integer, which can be stored as a *number*, sent to another *plus*,

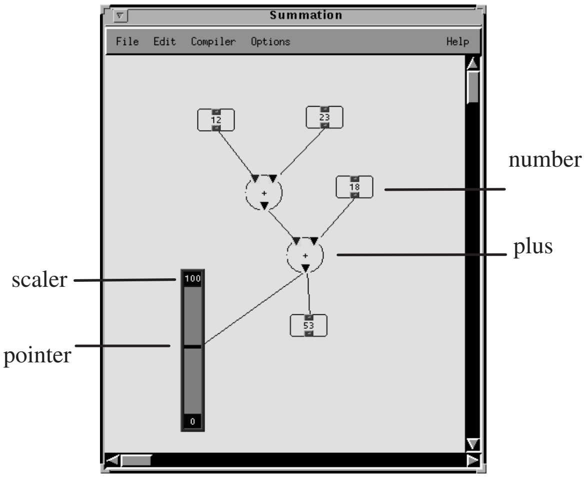


Figure 2: A snapshot of the *summation* VPE

or sent to a *scaler* for visualization. A *scaler* visualizes an integer in a vertical bar, where a pointer (the little horizontal mark) indicates the integer value through its height on the vertical bar. In Figure 2, the maximal and minimal values of the *scaler* are 0, and 100 respectively, which can be changed by entering new values through keyboard. The pointer of the *scaler* is in a position representing 53 , which is the execution result of the simple visual program shown in the figure. During the program execution, the displays of the *numbers* and *scalers* are changed according to the values sent to them. The following sections introduce the specification tools and explain how *summation* can be created using these tools.

3. Visual Object Generator

In the VisPro framework, a VO generator generates a set of visual classes to suit any special–purpose visual languages by editing the predefined visual objects called *VO prototypes*. We call such a process *customisation*.

A VO prototype is customisable. Figure 3(a) shows two visual classes which have been customised, and a VO prototype which is a black box in its original form. One can edit the VO prototype by triggering the editing commands attached to the box, i.e. by clicking the right mouse button when the mouse is in the box. Figure 3(a) shows the menu commands of the box. For example, by selecting the command *create[vertex]* in the menu items, a vertex can be created in the box. A subgraph or a vertex has its own commands which can be popped up in a menu and used for editing the subgraph to obtain a desired shape and color.

To construct a *scaler*, for example, a command called *selectShape* can be triggered from the pop–up menu. This command opens a dialog box which contains a set of graph patterns. If a *scaler* is selected, the black box will be reshaped to a *scaler* as shown in Figure 3(b). The dimension and color of the *scaler* can be edited and also labelled if necessary.

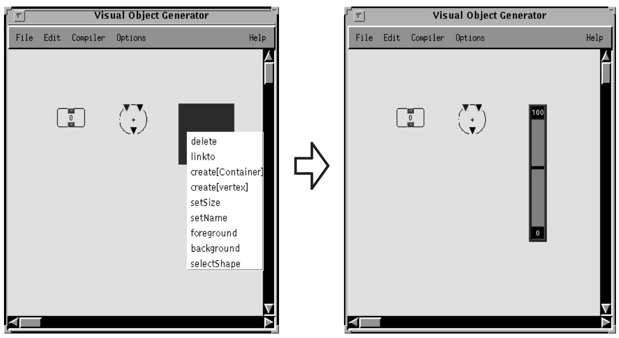
Figure 3(b) shows three nodes that have been created: *number, plus,* and *scaler*. One may notice that the *scaler* has a fixed pointer. According to the semantics, the pointer should be created dynamically using the mouse during program editing, and a *scaler* can have more than one pointer at a time. This is done in the VO generator by specifying its construction style as “dynamic” (by selecting the menu item named *construction* in the command menu popped up from a vertex).

We use attributes to parameterise all the three node classes of *summation*. The domain attribute for the *scaler* is *(pointer, integer)*, for the *number* is *(“value”, integer)*, and for the *plus* is *(“in1”, integer)*, *(“in2”, integer)* and *(“sum”, integer)*. For example, when we need to set a value 3 to a *number* object called *num*, we simply write: *num.put(“value”, 3)*, where *put* is a method of the *number* class.

For a *scaler* class, a method of its attributes called *put_do* can be rewritten so that when the value of a *number* connected to the pointer of the *scaler* changes, the position of the pointer will be adjusted accordingly. This modification is done in Java. Other modules do not need to know this modification when using the *scaler*, as the *put* method will automatically call the *put_do* method. Therefore, a method call like *scaler.put(pointer, 3)* will put an integer 3 into the attributes associated with the *scaler* and adjust the position of the pointer geometrically.

An edge class can also be created in a VO generator. The edge named *flow–to* used in *summation* is defined as shown in Figure 4, where two little filled rectangles are supposed to be replaced by two vertices in an application when the edge is used. The VO prototype of an edge can be customized by changing its shape, color and label through the menu items *selectShape*, *selectColor* and *setLabel* respectively.

A diagram workbench prototype can be customized to a workbench for a specific VPL with a set of node classes and



(a) (b)

Figure 3: Snapshots of visual object construction

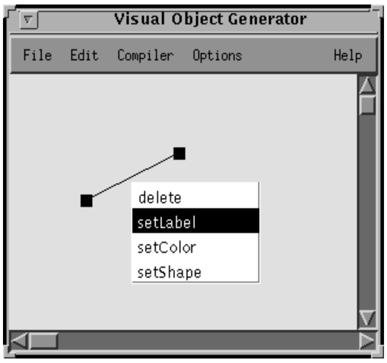


Figure 4: Generating an edge class with the VO generator

edge classes. This is specified in the control specification. There is a VO prototype which can be edited to a graph representing the icon of a workbench. A workbench can be accessed, e.g. opened, through its icon. Figure 5 shows an icon for the *summation* workbench. A *summation* workbench can be created in another diagram workbench as a special icon.

4. Control Specification Generator

4.1. Object–Oriented Editing Commands

The process of editing a diagram can be seen as a number of steps, each of which is an execution of a command on a set of graphs. Usually in a visual editor, commands and visual objects are independent of each other. Execution of a command is a selection of both the command and its target graph. In our object–oriented formalism, a graph is an object which encapsulates a set of related commands. The problem of how a command knows its targets arises when a command has more than one target. To solve this problem, we designed a set of operations, each of which is a series of interactions between a user and the editor. An operation can provide certain information to a command, so that a command can be executed correctly if it has been combined with a proper operation.

The CS generator is used to visually generate a control specification which can be understood by the object–oriented controller. The controller allows its basic commands to be triggered from its canvas and user–defined commands to be triggered from the created visual objects.

The CS generator assigns a set of editing commands and relationships to each visual class. The visual objects instantiated from a visual class can then trigger the assigned commands. The basic editing commands include *cut*, *copy*, *paste*, *create*, *link*, *open*, *property* and *properties*, which are pre–defined in the VisPro framework. If a user wishes to define additional commands, he/she can specify them in Java.

4.2. Command Specification

The visual objects handled in the CS generator include *edge objects*, *node objects* and *command objects*. An edge object, e.g. an ellipse in Figure 6, is an instance of the edge class defined in the target VPL. Its value is the label type of an edge class, i.e. *flow–to*, which can be entered or modified through keyboard. A node object, drawn as an unfilled rectangle, is an instance of the node class. It can be edited to

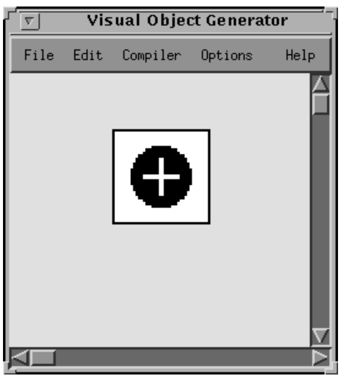


Figure 5: An icon for *summation*

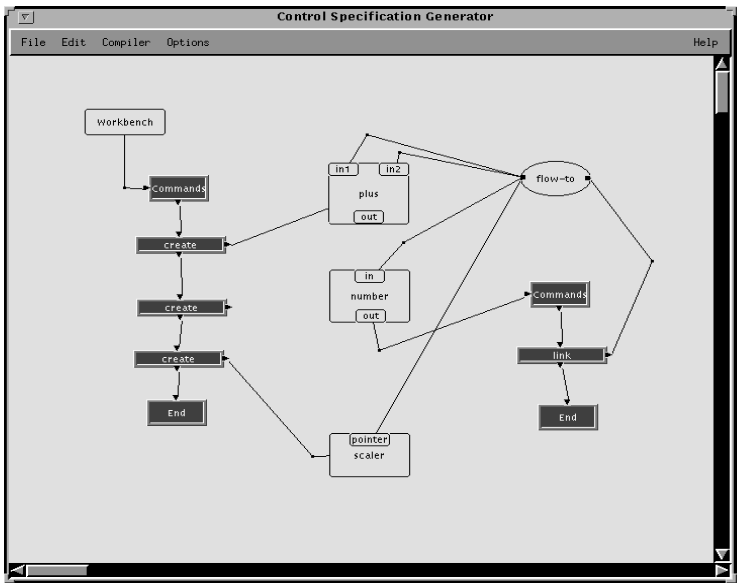


Figure 6: Examples of the command specification

form a super vertex embedded with some vertices. For example, the node labelled "plus" represents the *plus* class of *summation*. A command object, i.e. an gray box in Figure 6, represents an editing command, and its value (i.e., delete, link, etc.) can also be entered through keyboard.

Figure 6 shows a visual sentence which specifies a part of *summation* control. The node object labelled "Workbench" represents the visual editor of *summation*. The "Workbench" label is a reserved word in the CS generator. The *Workbench* node object links to a command list which includes three *create* commands. A *create* specification can be generated by interpreting the link between a command object and a node. A *create* command node linking a *plus* node, for example, will be interpreted to produce a command specification *create create[plus] plus*, where *create[plus]* is the name of the command menu item.

The *number* node object has a super vertex labelled "number", and two vertices labelled "in" and "out". The *out* vertex (i.e., the vertex labelled "out") has one command *link*, which links to an edge object labelled "flow–to". The *flow–to* edge object links to the vertices labelled "in1", "in2", "in", and "pointer". This indicates that an *out* can link to *in1*, *in2*, *in*, or *pointer* in *summation*.

The control specification diagram in Figure 6 will produce a list of control specifications as follows:

```
Workbench
3
create create[plus] plus
create create[number] number
create create[scaler] scaler
```

It indicates that three *create* commands can be triggered from the visual editor canvas. The specification

```
number.out
1
link connect flow–to
```

indicates that *number.out* has a *link* command named *connect* and can be linked to other vertices through the edge object instantiated from the *flow–to* edge class.

The specifications

```
VE number.out flow–to plus.in1
VE number.out flow–to plus.in2
VE number.out flow–to number.in
VE number.out flow–to scaler.point.
```

describe that the *number.out* may be linked to visual objects instantiated from *plus.in1*, *plus.in2*, *number.in*, and *scaler.point* visual classes, where *number.out* means a vertex labeled "out" in a super vertex labeled "number". The *number.out* cannot link to other vertex classes which are not indicated in the specifications.

The control specification diagram in Figure 6 can be extended to specify the complete behaviour of *summation*. Thus, the CS generator provides an intuitive and easy way to produce the control specification for a visual editor.

5. Rule Specification Generator

5.1. Graph Grammar

The grammar of a VPL is a collection of graph rewriting rules. A visual sentence can be interpreted by the rules through graph transformations. There, however, has been no graph grammar that can handle a wide range of visual languages both effectively and efficiently. For example, if a grammar is expressive enough to represent various types of VPLs, its parsing algorithm usually reaches exponential time [5]. We have proposed a new type of graph grammar known as *reserved graph grammar* [10], which can effectively represent most existing types of diagrams with a parsing algorithm of polynomial time complexity.

Figure 7 shows a reserved graph grammar for *summation*. The grammar completely describes the syntax of a valid *summation* diagram. For example,

- *plus.in1* (or *plus.in2*, or *number.in*) can connect to *plus.out* or *number.out* and but cannot connect to more than one destination.
- *plus.out* (or *number.out*) can link to one or more destinations, which include *number.in, plus.in1, plus.in2* and *scaler.pointer*.

The graph grammar also specifies the semantic aspects. Figure 8 shows a valid *summation* diagram. According to the semantics, graph (1) must be applied first with the grammar rule <3>. Graph (2) should be applied next, followed by graph (3). Otherwise, a wrong result will appear. For example, if graph (2) is applied first, as its numbers do not have the correct values (from graph (1)), the result of graph (2) will be incorrect. Such an order of applications is not allowed according to the graph rewriting system which dictates that a rule can be applied to an unmarked visual object only if all of the object's edges are matched by the rule.

5.2. Parsing

Parsing a diagram takes two phases: syntax parsing and semantics parsing. Syntax parsing is to check whether the diagram is valid. If a diagram is eventually transformed into an initial graph (i.e. λ) by the graph rewriting rules, it is valid. Semantics parsing is to produce a result from a diagram. The result is meaningful only when the diagram is valid. In a translation process, say from a diagram to a textual specification, the syntax and semantics can usually be specified in the same set of graph rewriting rules. In this case, the graph transformation process checks the syntax and translates a graph into a textual specification at the same time. For an executable diagram, this is not always the case. The syntax and semantics specifications of a Petri net visual language, for example, should be specified separately. This is because that a Petri net can be executed repeatedly, while the syntax checking must be done in finite steps. For *summation*, work the syntax and semantics can be speci-

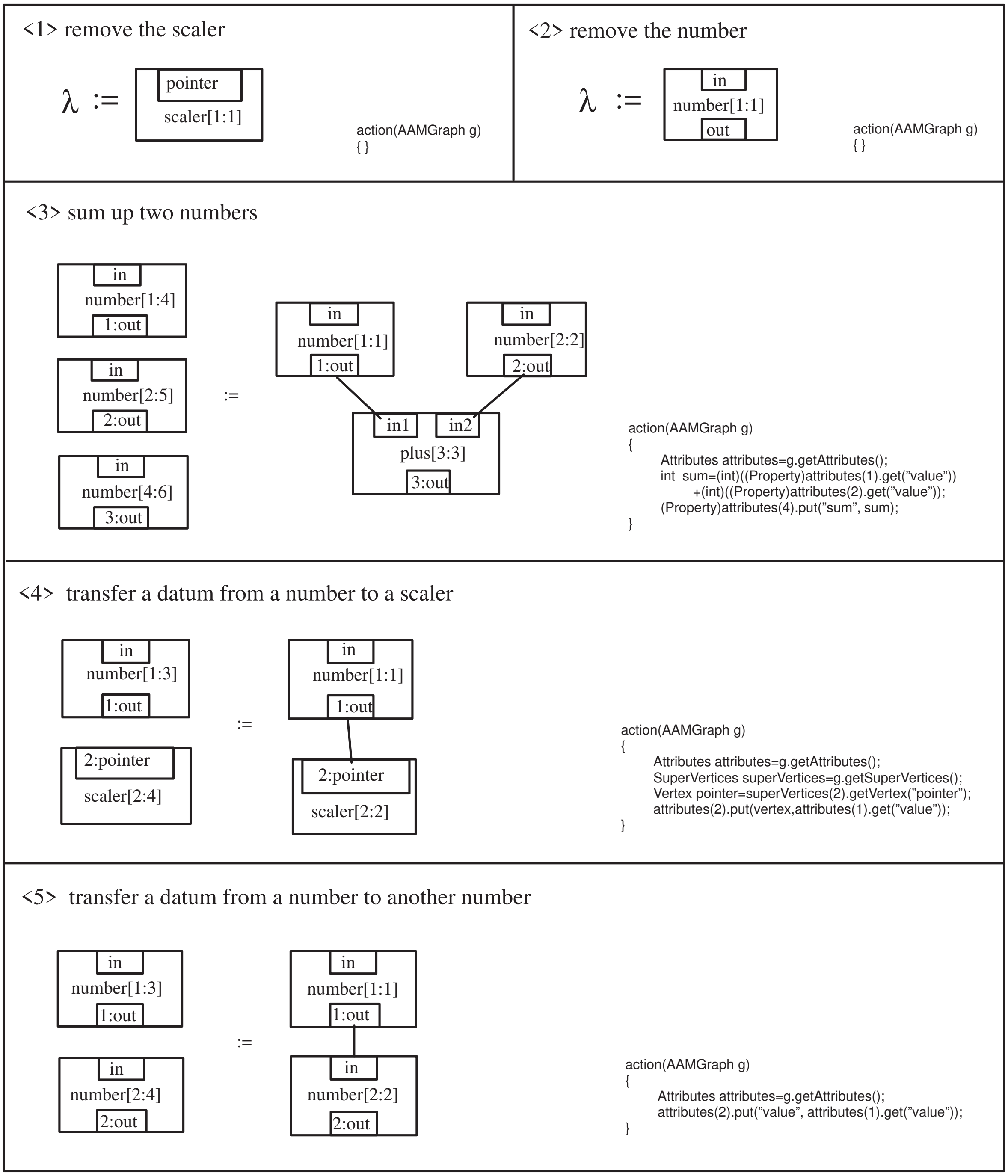


Figure 7: A reserved graph grammar for *summation*

fied with the same set of graph rewriting rules, as shown in Figure 7.

To create a parser for a VPL, one must write a piece of code (i.e. action) for each graph rewriting rule. An action performs computation over the attributes of a redex (a subgraph of the program) of the productions when the redex is applied. The actions of the graph rewriting rules of *summation* are listed in Figure 7. With the actions, the desired results can be produced after the graph transformation. For example, the action of Production <3> is as follows.

```
action (AAMGraph g)
{
  Attributes attributes=g.getAttributes();
  int sum=(int)((Property)attributes(1).get("value"))
```

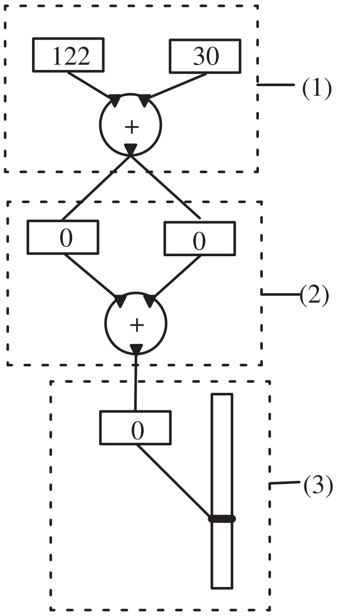


Figure 8: Application of the grammar

```
            +(int)((Property)attributes(2).get("value"));
    (Property)attributes(3).put("sum", sum)
}
```

The action takes a graph *g* as its input. This graph has a matching redex isomorphic to the right graph of Production <3>. To facilitate the access of attributes in the redex, an array referring to required attributes is first produced through the method *g.getAttributes()*. The array member *attributes(2)*, for example, refers to the attribute of the super vertex which has a *A_ref=2, i.e., number[2:2]* in the figure. The sum is calculated by summing up values of two matched numbers. It is then stored in *attributes(3)* as a result.

5.3. Rule Specification

The RS generator facilitates the rule specification. Figure 9 is a snapshot when using the RS generator, where two kinds of nodes (*left graph node* and *right graph node*) are used to represent the left graph and the right graph of a production. For example, the node labeled L<3> is the left graph node for Production <3>. The node labeled "Duplex" is the head of the rule list. It indicates that the rules are applied in a *duplex* mode. A production is created by linking a left graph node and a right graph node. Each graph node has a subeditor for defining a graph. In addition, a textual editor workbench can be triggered from the right graph for writing the action code. Figure 9 shows a snapshot of creating a production (rule <3> of the *summation* graph grammar), where two windows are opened for editing the left graph and the right graph of Production <3> respectively. Also there is a textual window for editing the action. Thus, the RS generator provides visual editors for specifying the graph rewriting rules, and a textual editor for specifying actions. The RS generator can produce graph rewriting rules automatically by interpreting the connected editors.

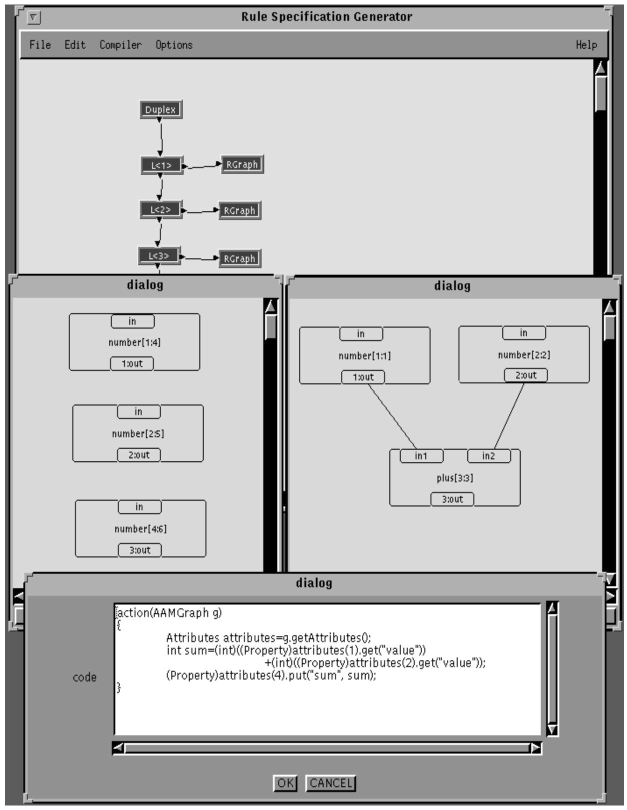


Figure 9: A snapshot of the rule editor

6. Implementation

VisPro is implemented in Java, an object–oriented language. One advantage of using Java is that it is platform independent, so that the system can be ported to different platforms. Another advantage is that Java is developed for network programming. This characteristic can support the construction of VPLs which allow visual programming for the Internet and distributed applications.

The implementation of VisPro includes seven functional modules, as shown in Figure 10:

- The configuration interpreter receives the configuration specification, and transfers the lexicon definition of the specification to the user interface and the grammar specification to the parsing module.
- The user interface controls the interaction between users and the VisPro tools.
- The underlying structure manages the diagrams which are being edited.

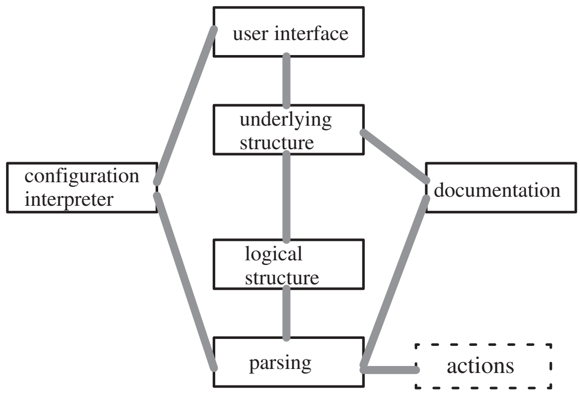


Figure 10. Modules of PPE

- During parsing, the logical structure module creates and manages the logical graph converted from the underlying structure of a diagram.
- The parsing module is designed to parse the logical structure of a diagram using the reserved graph grammar formalism.
- The documentation module automatically records edited diagrams and parsing results.
- The actions module collects actions for each VPL from grammar specifications. The collected actions are represented as a Java program and dynamically linked to the parsing module during execution.

7. Conclusion and Future Work

The paper has described VisPro, which can be used to generate diagrammatic VPLs. VisPro consists of three tools, each of which facilitates one aspect of the construction of VPLs. The VPL construction process with VisPro is similar to the textual language construction process using lex/yacc.

The VisPro framework can be customized to any target visual language with a set of domain specifications. To further ease the visual language generation, tools for domain specifications are developed. These specification tools are visual languages themselves so that the graph properties of the domain specifications can be easily described via direct manipulation.

The VisPro framework and the specification tools together provide a complete support for the VPL generation. They can be used to generate a wide range of visual programming environments easily.

Though the current implementation of VisPro is suitable for constructing many kinds of VPLs, it lacks a flexible support for the layout of a target visual editor. With the framework of VisPro, it is not difficult to add this mechanism. There are two kinds of layout for a diagram: local layout and global layout. A local layout is only related to the operated objects. For example, when a relation is created between two objects, their geometrical relationships may need to be changed. Sometimes a geometrical change of one object may result in changes of other objects, which is called global layout. In VisPro, the grammar system can be extended to solve this problem. As a local layout is concerned with a single command, a graph rule can be attached to that command to perform layout. For a global layout, a layout grammar, which is similar to the language grammar, can be specified and a layout parser can be produced. This approach is general enough to solve complicated layout problem.

References

[1] B.A. Myers, Taxonomies of Visual Programming and Program Visualisation, *Journal of Visual Language and Computing*, 1990, 1 , pp.97–123.

[2] J.D. McWhirter, and G.J. Nutt, Escalante: An Environment for the Rapid Construction of Visual Language Applications, *Proceedings of 10th IEEE Symposium on Visual Languages*, Oct. 4–7, 1994, pp.15–22.

[3] M. Minas and G. Viehstaedt, DiaGen: A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams, *Proceedings of 11th IEEE Symposium on Visual Languages*, Darmstadt, Germany, Sep. 5–9, 1995, pp.203–210.

[4] E.J. Golin and T. Magliery, A compiler generator for visual languages, *Proceedings of 9th IEEE Symposium on Visual Languages*, Bergen, Norway, August 1993, pp.314–323.

[5] J. Rekers, and A. Schürr, A Graph Based Framework for the implementation of Visual Environments, *Proceedings of 12th IEEE Symposium on Visual Languages*, Boulder, Colorado, USA, Sep. 3–6, 1996.

[6] A. Schürr, A. Zundorf, and A. Winter, Visual programming with graph rewriting systems, *Proceedings of 11th IEEE Symposium on Visual Languages*, Darmstadt, Germany, Sep. 5–9, 1995, pp.326–333.

[7] J.Rekers, and A. Schürr, Defining and Parsing Visual Languages with Layered Graph grammar, *Journal of Visual Languages and Computing*, 8(1), Feb. 1997.

[8] S.K. Chang, M.J. Tauber, B. Yu, and J.S. Yu, A Visual Language Compiler, *IEEE Transactions on Software Engineering*, 15(5), May 1989, pp. 506–525.

[9] G. Costagliola, G. Tortora, S. Orefice, and A.D. Lucia, Automatic Genetaion of Visual Programming Environments, *IEEE Computer*, 28(3), March 1995, pp.56–66.

[10] D–Q. Zhang, and K. Zhang, Reserved Graph Grammar: A Specification Tool for Diagrammatic VPLs, *Proceedings of 13th IEEE Symposium on Visual Languages*, Capri, Italy, Sep. 23–26, 1997, IEEE CS Press, 284–291.