

Visual Language Semantics Specification in the VisPro System

Ke -Bing Zhang¹ Mehmet A. Orgun¹ Kang Zhang²

¹Department of Computing, ICS, Macquarie University, Sydney, NSW 2109, Australia
{kebing, mehmet}@ics.mq.edu.au

²Department of Computer Science, University of Texas at Dallas
Richardson, TX 75083-0688, USA
kzhang@utdallas.edu

Abstract

VisPro is a general-purpose visual language generation system, which can produce a wide range of diagrammatic visual programming languages (VPLs) based on Reserved Graph Grammar (RGG), a context sensitive graph grammar. This paper presents an approach to specify the semantic execution sequence of VPLs based on VisPro. In this approach, we use an ordering mechanism to facilitate the parsing formalism of Reserved Graph Grammar to determine semantic execution sequence of super-nodes in visual programs of the VPLs.

1 Introduction

The syntax of a visual programming language describes the rules of how to compose valid visual sentences of the language with its visual vocabularies. Its semantics describes the meaning or meaningfulness of visual sentences when they consist of the context of the language. However, visual programming languages are difficult to implement, because the two-dimensional feature makes them difficult to develop effective syntax and semantics specification formalisms (Zhang 1998).

Many works related to semantics of visual languages have been conducted, however, most of them are restricted to specific visual languages (Haarslev 1995). The other separation of concrete and abstract syntax proposed in (Rekers and Schürr 1995, Rekers and Schürr 1996, Erwig 1997). Concerning abstract visual syntax, some other authors also recommend the separation from concrete syntax (Andires, Engels and Rekers 1998, Min 2000).

Erwig (Erwig 1997) defined visual language semantics by using abstract syntax graphs that abstract from representation details of concrete diagrams. His approach does not restrict semantic definition to this representation, but uses denotational semantics to define diagram semantics based on abstract syntax. His approach, however, does not offer a method for translating a concrete diagram into its abstract syntax representation.

Minas (Minas 2000) uses hypergraphs and hypergraph grammars to describe concrete syntax and arbitrary data structures for semantic representations. Hypergraphs offer

a more natural representation of diagram components that have different “attachment areas” which link to other diagram components. Moreover there are restricted, yet powerful types of hypergraph grammars that allow for efficient parsing (Minas 1997, Bardohl, Minas, Schürr, and Taentzer 1999), which are not available for plain graph grammars. Arbitrary data structures for semantic representations, e.g., strings that are used in their system have the advantage that they can be customized for common compilers (Minas 2000).

The VisPro graph rewriting system is a tool for graph transformation based on the attribute graph grammar (Zhang and Zhang 1988). Its graph rewriting rules specify the high level graph transformation, and the low level equations associated with the rules specify the attribute computation.

Particularly, an attribute in the VisPro graph rewriting system is an object in the object-oriented formalism. The object-oriented form is more powerful than the original form (i.e, pure data structure) in that an attribute consists of both data and methods. In addition, an object can be active during parsing and can change its attributes. This is useful in interpreting a high level diagram. One application scenario may be like this: a diagram is an overall control system and its nodes are active during a control process. The control is governed by the current states of the nodes, which can be changed by the internal computations of the attributes of the nodes.

These features of VisPro system enable users to specify effective semantics of VPLs. However, the specification of the semantic execution sequence of VPLs by using a marking mechanism in the parsing formalism is not always applicable. We use an ordering mechanism to facilitate the parsing formalism to determine semantic execution sequence of visual programs of the VPLs in the VisPro system. The details of our approach are presented in the remainder of this paper.

This paper is organized as follows. The next section briefly introduces VisPro system and the reserved graph grammar that VisPro system is based on. In section 3, we point out the shortcoming of former semantics specification formalism of VisPro system. Section 4 addresses our idea of how to specify semantics execution sequence of VPLs in rewriting rules of reserved graph grammar. Section 5 demonstrates our approach by examples of VPL Flowchart. Finally, Section 6 concludes this paper.

2 Background

2.1 Reserved Graph Grammar

A graph grammar has a set of productions, also called graph rewriting rules. Each production has two graphs, which are called *left graph* and *right graph*. It can be applied to another graph (called *host graph*) in the form of an *L-application* or *R-application*. A *redex* is a subgraph in the host graph which is isomorphic to the right graph in an R-application or the left graph in an L-application.

The L-application defines the language of a grammar. A production's L-application to a host graph is to find a redex of the left graph of the production in the host graph and replace the redex with the right graph of the production. The language is defined by all possible graphs, which have only terminal labels and can be derived by using L-applications from a null graph (i.e. λ).

The R-application is used to parse a graph. If the graph is eventually a null after a series of R-applications, the graph is proven to belong to the language. An R-application is a reverse replacement of its corresponding L-application, i.e., from the right graph to the left graph.

Reserved Graph Grammar is a context-sensitive graph grammar, which is complete and explicit in describing the syntax of a wide range of diagrams using labeled graphs (Zhang and Zhang 1997). It is developed based on the *layered graph grammar* (Rekers and Schürr 1997) by using the layered formalism to let the parsing algorithm determine whether a graph is valid in finite steps. The parsing algorithm of RGG is only polynomial time complexity and more efficient than other context sensitive graph grammars if it satisfies a particular constraint (Zhang and Zhang 1998).

2.2 VisPro Graph Rewriting System

The semantics specification, such as translating a graph to another language or performing some actions on a graph, is necessary. In either case, some computations need to be performed on the source graph. This kind of a computation is syntax-directed. The best known tool for describing syntax-directed computation is an attribute grammar (Deransart, Jourdan and Lorho 1988).

Attribute grammars provide a means to associate data with the nodes of a tree structure (normally a syntax tree), and to specify relations between pieces of data on different nodes. In their pure form, attribute grammars do not care which kind of data are used, or which operations or functions can be applied to the data, although in practice these issues may be important.

VisPro is a general-purpose VPL generation system that can generate a wide range of diagrammatic VPLs based on the RGG formalism (Zhang and Zhang 1998). The graph rewriting system of VisPro is a tool for graph transformation based on the attribute graph grammar. It accepts a set of graph rewriting rules as its reasoning knowledge. The system can then parse a diagram and

produce some results by explaining the diagram with the rules.

A graph rewriting rule is a pair of graphs, like a production in the RGG. A node of a graph can be associated with semantic data called an attribute. An attribute is an object in object-oriented formalism. The semantics of a node is determined by its associated attribute. For example, a node representing a number should have an attribute about its value. The current value is the semantics of the node.

An attribute is a name for a piece of data which is associated with a node of a syntax tree. An attribute grammar should specify the relation between the attributes in such a way that it is possible to compute unique values satisfying the relations. A central principle of the attribute method is locality: for each production the relations are given for some attributes that are associated with the nodes corresponding to the production's application in the syntax tree. Hence one can concentrate on the relation of attributes for one production at a time.

2.3 Parsing in VisPro

Parsing visual program diagrams in VisPro is based on reserved graph grammars. The parsing process takes two phases: syntax parsing and semantic parsing.

Syntax parsing applies a series of R-applications to a host graph to check whether the program is valid. If an abstract diagram of a visual program is eventually transformed into a null graph (λ), it means that the visual program is valid. Syntax parsing is the process from a sub-graph in the host graph to whole host graph. The parse tree is produced in this phase if the host graph is valid.

Semantic parsing checks the source program for semantic errors and gathers type information for the subsequent code-generation phase by applying a series of L-applications according to the parse tree that is produced in syntax parsing. The result is meaningful only when the semantic parsing is successful.

2.4 Semantics Execution of VPLs

A visual programming language based on the VisPro system, its syntax and semantics specifications are usually specified in the same set of its graph rewriting rules. The interpreter of the VPL responds to execute the semantic transformations/actions in graph rewriting rules to the matching redexes during the syntax parsing process of the programs.

For the VPLs in VisPro, the semantic transformations are applied with the syntax parsing. When a redex is found in a host graph, the semantics to the redex is also applied according to the semantic codes in graph rewriting rules, and certain attributes of the associated objects (nodes and edges) in the physical diagram of the visual program are changed, such as the appearances of some nodes. If the syntax parsing of the program is completed, the semantic execution is completed as well.

The semantic execution sequences of visual programs in VisPro are specified by using a mark mechanism in the rewriting rules of the RGG according to the application of the graph rewriting system.

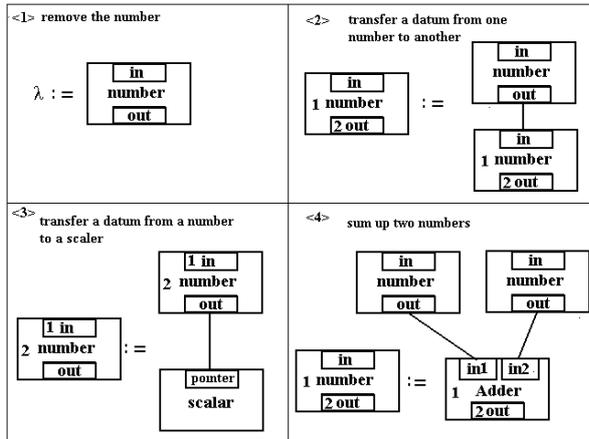


Figure 1. The reserved graph grammar for Adder

For example, Figure 1 shows the RGG of VPL Adder and Figure 2 gives a visual program of VPL Adder. According to the semantics of Adder, the top-right number node in graph (2) should get value from the top number node of graph (1). Hence, the execution sequence of the visual program diagram as shown in Figure 2 should be graph (1), graph (2), graph (3) and graph (4); otherwise a wrong result will appear. This semantics execution sequence is determined by the syntax of Adder. It is specified in rule <2> and <4> in Figure 1 that an unmarked visual object, such as nodes “out” and “in” in super-node “Number”, can be matched only when all of its edges are matched by the rule (Zhang and Zhang 1998).

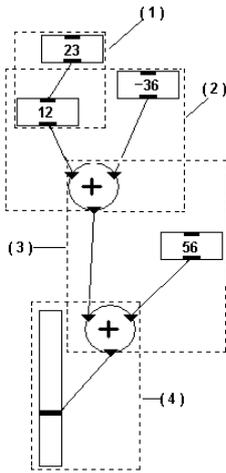


Figure 2. A visual program of Adder

3 Motivation

However, for more complicated reserved graph grammars, using the mark mechanism to specify the semantics execution sequence sometimes is not applicable.

For example, Figure 3 presents the RGG of VPL Flowchart, and Figure 4 gives a visual program diagram of VPL Flowchart. According to the semantics of Flowchart, the dataflow should be from top to bottom, this means the semantics execution sequence of the visual

program in Figure 4 should apply graph (1) first then apply graph (2). See Figure 3, rule <7> which matches graph (1) and graph (2) in Figure 4. We may find that either graph (1) or graph (2) are redexes because marked visual object, for example “stIN” with “1” and “stOUT” with “2” in “statement” in rule <7>, can be matched when it has any number of edges.

However, this mark mechanism in the graph rewriting rules of Flowchart cannot guarantee graph (1) first then graph (2) application sequence, because syntax-parsing sequence depends on the physical generation order of the visual object in the host graph for same the rewriting rule. Therefore, for some kinds of RGGs, for example Flowchart and Petri net, the mark mechanism would lead to incorrect results when the visual programs are executed.

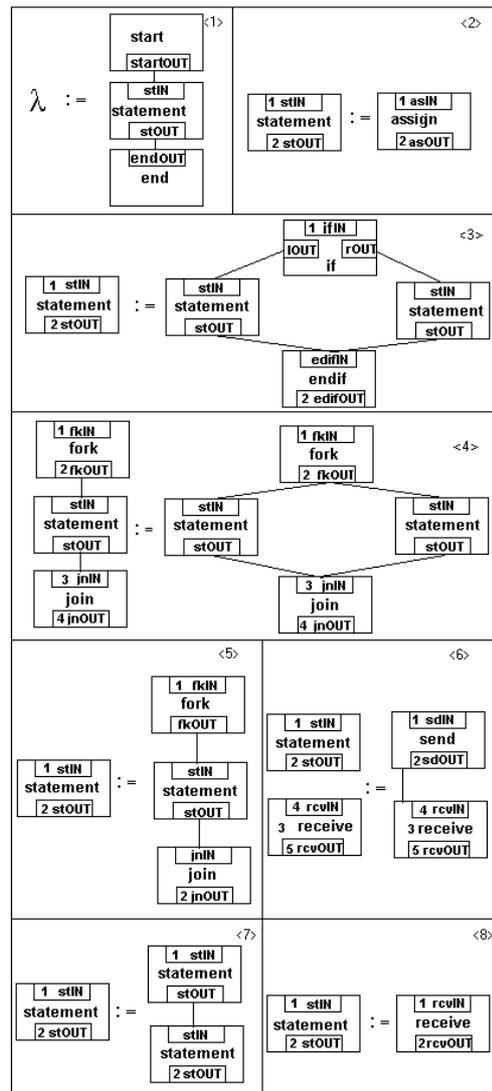


Figure 3. The reserved graph grammar of Flowchart

Furthermore, the compiler of a VPL in VisPro must check that the host graph follows both syntactic and semantic conventions of the VPL. Uniqueness check is a very important aspect of the semantic parsing. There are situations in which a visual object must be defined exactly once in a left graph and right graph of a production, for

example an identifier must be declared uniquely, labels in graph rewriting rules must be distinct.

For most reserved graph grammars, during the semantic parsing, a visual object in the redex can be identified by the name, type or edges of the visual object that matched in the right graph. But this is not always the case, for example, in the production <4> of VPL Flowchart as shown in Figure 3, there are two “statement”’s which cannot be identified by above the information because these information of the two “statement”’s are exactly same.

To overcome these shortcomings of the semantics execution of VisPro system, we adapted the parsing techniques of textual programming languages (Aho, Sethi and Ullman 1986) and an ordering mechanism to the parsing formalism of VisPro.

4 Semantic Execution Specification

4.1 Basic Idea

In the VisPro, semantic parsing process can be applied only when the syntax parsing is successful. For a valid visual program, after syntax parsing, the parse tree of the host graph is created. The parse tree can be used for tracing each parsing step in the subsequent semantic parsing phase. Then the compiler of the VPL can use the hierarchical structure of parse tree determined by the syntax parsing phase to identify the semantic execution sequence of redexes.

Because we neither limit the reserved graph grammar designers to generate the visual objects in certain order when they design graph grammars nor limit programmers to design visual programs in fixed sequence, for a visual program, it can have several possible parse trees. For example, there are four parse trees for the visual program given in Figure 4 based on different super-nodes (“a=3”, “b=a+4” and “c=a+b”) generation orders, as shown in Figure 5.

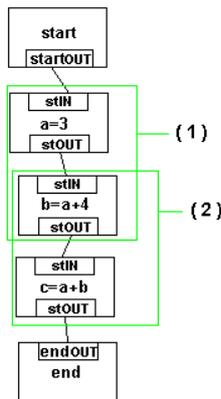


Figure 4. A visual program of Flowchart

If we can guarantee the super-nodes in a redex that mapping a right graph of RGG are matched in certain sequence, it will reduce possible parse trees that are generated in syntax parsing.

For example, if super-nodes in a redex of an L-application of rule <7> are matched in sequence of the top “statement” first, then the bottom “statement”, there will be only two parse trees for the program in Figure 4, parse tree I and parse tree II as shown in Figure 5.

In addition, analysing the parse tree I and parse II, we may find that whatever routes of the syntax parsing of the visual program, it does not affect the result of syntax parsing, as shown in the Figure 5 where st1’ and st’’ represent temporal redexes used in the syntax parsing. We may also find that according to the tree traversal rule (“from top down, from left right), although the routes of parse trees are different, the result of tree traversal are same. For this case, the tree traversal is “start” → ”a=3” → ”b=a+4” → ”c=a+b” → ”end”. This is just the dataflow of the visual program.

Therefore, it is natural to use a unique identifier to indicate the execution sequence for each visual object in right graphs graph rewriting rules.

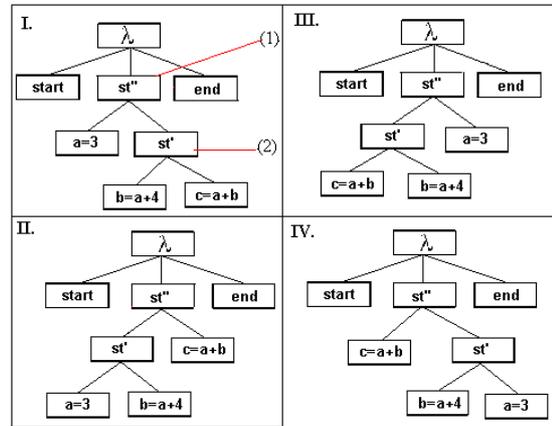


Figure 5. Parse trees of the visual program

We order the visual objects in right graphs according to their dataflow order. The ordered label can be also used as a unique identifier in the semantic parsing. Figure 6 illustrates three ordered productions <4>, <6> and <7> of RGG of VPL Flowchart, where [number] in super-nodes represent the semantic execution sequence of the super-nodes.

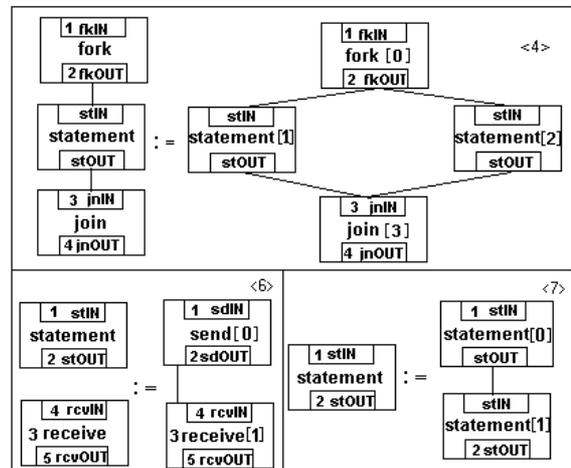


Figure 6. Production <4>, <6> and <7> of RGG Flowchart with ordering indication

We explain the semantic transformations of visual programs by ordering components in rewriting rules of RGGs as follows.

4.2 Semantics Transformation

In the case of linear textual languages, it is clear how to replace a non-terminal in a sentence by a corresponding sequence of (non-) terminals. But with visual languages, as there are two-dimensional relationships between language elements, a far more complicated mechanism is needed to establish relationships between the substitute of a redex and its adjacent elements.

The compilation techniques of textual languages can be borrowed to the visual languages based on context-free graph grammars, because a context-free grammar allows only a single non-terminal on the left-hand side of a production. However, context-free grammars cannot specify a large proportion of VPLs. Additional features of VPLs are required for context-free graph grammars to handle context-sensitivity (Zhang, Zhang and Deng 2001).

A context-sensitive graph grammar can allow left-hand and right-hand graphs of a production to have arbitrary number of nodes and edges. However, this feature of context-sensitive graph grammar makes the semantics specification of VPLs based on context-sensitive more complicated.

Reserved graph grammar is a context-sensitive grammar. It uses both an embedding rule and context elements to solve the embedding problem. With the context facility, this simple embedding rule can be powerful enough to deal with complicated situations (Zhang and Zhang 1997). Therefore, we use redex in semantic parsing as a parsing unit to solve the semantic transformations of context-sensitive cases, as shown in production <4> and <6> in Figure 3, like tokens in a textual language.

The parse tree produced by the syntax parsing is in a hierarchical structure, so a redex of L-application can have super node(s) and redex(s), as shown in Figure 5. With the context elements of RGG, a redex of R-application in a redex of L-application can be replaced by its corresponding right graph of rewriting rule according to the parse tree during the semantic parsing. The semantic execution of a redex in L-application is a process to execute the semantics actions and check the validity of semantics between the super nodes in the redex.

The semantic transformation of a host graph is a recursive process of applying L-applications to the last redex in the syntax parsing by using the hierarchical structure of a parse tree to trace each R-application in the syntax parsing, i.e., when a redex of an R-application in a host graph is found, the redex will be applied by its matched L-application and semantic transformations are executed according the semantic execution sequence in the right graph of the production. This process is repeated until no redex of R-application in the host graph.

For example, in parse tree I in Figure 5, the last redex of R-application is λ (i.e., null). Applying L-application of production <1> to λ , it is changed into “start” \rightarrow st’ \rightarrow “end”, as shown in Figure 7. In the semantic transformation process of this redex, the semantics are transformed from “start” to st’. However, the st’ is not a pure node, it is a redex of R-application. Then st’ is replaced by its corresponding right graph, i.e., “a=3” \rightarrow st’, the semantics are executed from “a=3” to st’. In this semantic transformation step, st’ is also a redex of R-application, this transformation is continued until no redex of R-application is found. The whole semantics transformation of parse I is shown in Figure 7. Figure 7 also illustrates the semantics transformation of parse II.

5 Examples

For a demonstration of our system, we designed a visual programming language, VPL Flowchart. The semantics of VPL Flowchart is quite simple; the function of the semantics is transforming the “message”, i.e., the attribute (name) of nodes, from one to another if they have an edge between them. The “message” can be shown on the “statement”, “receive” and “assign” nodes only.

A user designed visual program in VPL Flowchart is shown in diagram (1) of Figure 8. The diagram (2) in Figure 8 is the parsed visual program. We may notice that in diagram (2) the nodes of the visual program are laid out. A “smile face” icon, which indicates that syntax parsing of the program is successful, appears at the top-left of the diagram (2). Diagram (3) illustrates the executed visual program.

In diagram (2) of Figure 8, below the node “start”, there is a node “A-”, which is linked to “fork”. This “fork” is linked to two nodes “B-” and “N-”. After the program execution, the values of nodes “A-” and “fork” are transformed to the nodes “B-” and “N-”, their values are changed into “B-forkA-” and “N-forkA1” respectively, as shown in the diagram (3) of Figure 8.

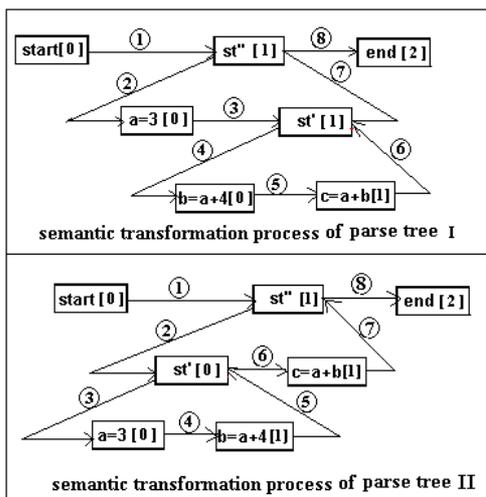


Figure 7. Semantic transformations

An L-application of RGG is the reverse process of its corresponding R-application, therefore, a valid host graph can be seen as a result of applying a series L-application of productions to an initial graph (null) with corresponding semantic transformation according to the parse tree.

We may find that the “statement” nodes “N-”, “S-” and “X-” are programmed in diagram (1) as the nodes “a=3”, “b=a+4” and “c=a+b” in the program as shown in Figure 4, see diagram (3), the messages of “N-”, “S-” and “X-” are transformed correctly.

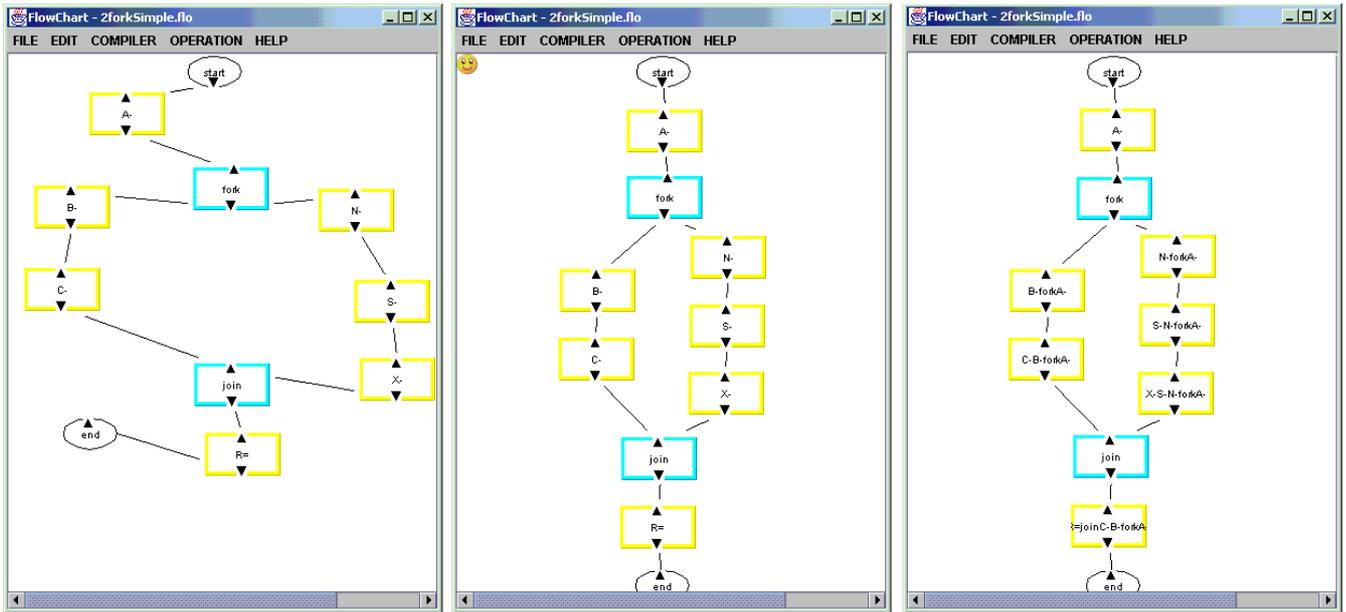
We present a more complicated program of VPL Flowchart in Figure 9. For simplification, only the original program, diagram (1) and the executed one, diagram (2) appear in Figure 9.

6 Conclusion

This paper has presented an approach to solving the problem of specifying sequence of semantic execution of VPLs in the VisPro system. In this approach, an ordering mechanism is introduced for indicating the sequence of semantic execution of super-nodes in visual programs in VPLs according to the dataflow of super-nodes in the right graphs of graph rewriting rules. This approach makes designing semantics for the VPLs in VisPro easily and effectively. It is intuitive, simple and suitable to specify semantics of VPLs based on VisPro system.

7 Reference

- ANDRIES, M., ENGELS, G. and REKERS, J. (1998): How to represent a visual specification. In *Visual Language Theory*. 245-260. MARRIOTT, M. and MEYER, B. (eds). Springer Verlag.
- AHO A.V., SETHI R. and ULLMAN J. D. (1986): *Compilers, Principles, Techniques and Tools*, Addison-wesley publishing company.
- BARDOHL R., MINAS M., SCHÜRR A., and TAENTZER G. (1999): Application of graph transformation to visual languages. In *Handbook of Graph Grammars and Computing by Graph Transformation*, volume II: Applications, Languages and Tools, EHRIG H., ENGELS G., KREOWSKI H.-J., and ROZENBERG G. (eds). 105-180. World Scientific.
- DERANSART P., JOURDAN M., LORHO B. (1988): Attribute Grammars: Definitions, Systems, and Bibliography. *Lecture Notes in Computer Science, Graph Grammars and Their Application to Computer Science* **323**. Springer Verlag, New York.
- ERWIG M. (1997): Semantics of visual languages. *Proc. VL'97-13th IEEE Symposium on Visual Languages*, Capri, Italy, 304-311, IEEE CS Press, Los Alamitos, USA.
- HAARSLEV V. (1995): Formal semantics of visual languages using spatial reasoning. *Proc. VL'95-11th Int. IEEE Symop. on Visual Languages*, Darmstadt, Germany, 156-163, IEEE CS Press, Los Alamitos, USA.
- MINAS M. (1997): Diagram editing with hypergraph parser support. VPLs. *Proc. VL'97-13th IEEE Symposium on Visual Languages*, Capri, Italy, 230-237, IEEE CS Press, Los Alamitos, USA.
- MINAS M. (2000): Creating semantic representations of diagrams. In M. Nagl and A. Schürr, editors, *Int. Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE'99), Selected Papers*, LNCS 1779, pp 209–224. Springer, Mar. 2000.
- REKERS, J., and SCHÜRR, A. (1995): A Graph Grammar Approach to Graphical Parsing, *Proc. VL'95-11th Int. IEEE Symop. on Visual Languages*, Darmstadt, Germany, 195-202, IEEE CS Press, Los Alamitos, USA.
- REKERS, J., and SCHÜRR, A. (1996): A Graph Based Framework for the Implementation of Visual Environments, *IEEE Symp. on Visual Languages*, 1996.
- REKERS, J., and SCHÜRR, A. (1997): Defining and Parsing Visual Languages with Layered Graph Grammars. *Journal of Visual Languages and Computing* **8**(1): 27-55.
- SCHÜRR, A., WINTER, A., and ZÜNDORF, A. (1995): Visual Programming with Graph Rewriting Systems. *Proc. VL'95-11th Int. IEEE Symop. on Visual Languages*, Darmstadt, Germany, 326-335, IEEE CS Press, Los Alamitos, USA.
- ZHANG D-Q. (1998): Generation of Visual Programming Languages, PhD thesis, Macquarie University, Australia.
- ZHANG D-Q. and ZHANG K. (1997): Reserved Graph Grammar: A Specification Tool for Diagrammatic VPLs. *Proc. VL'97-13th IEEE Symposium on Visual Languages*, Capri, Italy, 284-291, IEEE CS Press, Los Alamitos, USA.
- ZHANG D-Q. and ZHANG K. (1998): VisPro: A Visual Language Generation Toolset, *Proc. VL'98-1998 IEEE Symposium on Visual Languages*, Halifax, Canada, 195-202, IEEE CS Press, Los Alamitos, USA.
- ZHANG K., ZHANG D-Q. and DENG Y. (2001): A Visual Approach to XML Document Design and Transformation, *Proc. HCC'2001 - 2001 IEEE Symposium on Human-Centric Computing Languages and Environments*, Stresa, Italy, 312-319, IEEE CS Press., Los Alamitos, USA.

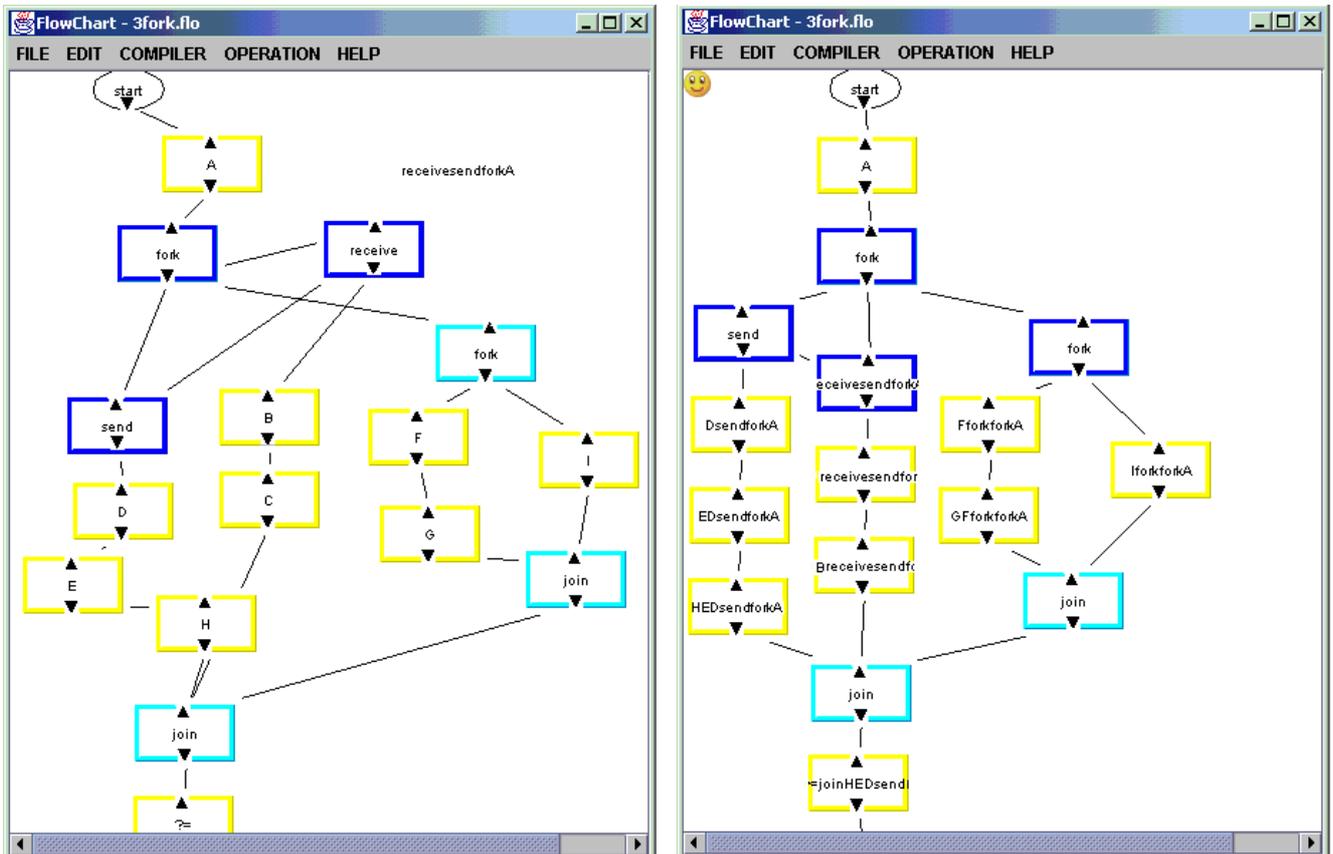


(1) Original program

(2) Parsed program

(3) Executed program

Figure 8. Demo I. A visual program of VPL Flowchart



(1) An original visual program in VPL Flowchart

(2) The executed visual program

Figure 9. Demo II. A more complicated visual program in VPL Flowchart