Research on Visual Language Specification and Visual Language Editor Automatic Generator

LuYang ZhangLi DuanFang
Software Engineering Institute
Beijing University of Aeronautics and Astronautics
Xueyuan Road 37, Beijing 100083, P.R.China
luyang-sei@163.com zhangli@buaa.edu.cn

ABSTRACT
This paper focuses on Visual Language Specification (VLS) and Visual Language Editor Automatic Generator (VLEAG). We analyze the main problems in VLS and VLEAG implementation and summarize the core function modules of VLEAG. Then, we present the crucial requirements that VLS and VLEAG should achieve to.

KEY WORDS
Visual language, visual language specification, editor, editor automatic generator

1. INTRODUCTION

Nowadays, in computer application areas, visual language plays an important role in communication between human beings, also between human beings and computers [8]. Visual language is being applied in each domain. There are a lot of visual languages, i.e. UML, enterprise process model language, workflows diagram, software architecture description language, petri net, entity relationship diagram. In the near future, visual language modeling process may replace part of the coding process, i.e. MDA [3]. However, construction of visual language editor is a challenging task, which is usually time consuming and tedious. Researchers have developed some VLEAGs to facilitate the process, which pushes the application of visual language at a great extent.

The rest of the paper is organized as follows: Section 2 investigates the formalisms of visual language specification and presents the characteristics of visual language specification. Section 3 analyzes functional structure of VLEAG and summarizes the core function modules.

2. VISUAL LANGUAGE SPECIFICATION (VLS)

There is a three decades history of visual language specification, but until now there is not an approach accepted as a standard and common way. One of the most fundamental questions in visual language research is also how to specify a visual language. VLS should be effective to specify a large bound of visual languages and easy for syntax parsing. The Formalisms of visual language also make important influence on VLEAG structure.

2.1 Visual Language

A visual language is a set of diagrams that are a collection of visual symbols in a two or three dimension space. A visual language consists of three parts: primitives, syntax, and semantics. Primitives are correspondent to the visual symbols that have logic part (visual symbol means) and layout part (visual symbol appearance). Syntax defines rules set for primitives connecting and spatial layout. Visual languages distinguish between visual representations called concrete syntax and meaningful parts called abstract syntax. This is different from string text. Semantics describes a meaning to the sentence. Only the syntax is correct, the semantics is useful.

2.2 Approaches of VLS

Currently there are three main approaches to the specification of visual language: the grammatical approach, the logic approach, and the algebraic approach.

1. **The grammatical approach** is based on grammatical formalisms used in string languages specification. In contrast to string languages grammatical formalisms, it specifies the layout and spatial relationships between the primitives to be rewritten, i.e. Picture Description Language [12], Attributed Programmed Graph Grammars [2], Web grammars [9].
2. **The logic approach** uses forms of mathematical logic. Execution often starts with a query from the user, and determines if the query is a logical consequence of the program, i.e. Clause Set Grammars [13], Constraint Logic Programming [6].
3. **The algebraic approach** uses forms of algebraic specification. The main idea is to map the domain to be defined into typed abstract data structures and to define typed functions and predicates operating on these structures that specify the operations in application domain. Hierarchical typing is the key concept in algebraic specification, i.e. Algebraic specification [7], Graphical Assisted Reasoning [14].

Sometime, the three main approaches may work together. In grammatical approach, graph grammar is an extension of string grammar from one to two or three dimensions and used widely. Comparing to algebraic and logic approaches, grammatical approach is based on a firmer theory foundation. But the syntax parsing efficiency blocks the application of grammatical approach [10]. The algebraic and logic approach both use complex math symbols and expressions to define rules. Their formalisms are too obscure to understand.

Apart from approaches mentioned above，there is an approach using rules to specify visual language which was introduced by SEI of BUAA as Editor Customization Language [15] (ECL). ECL defines visual language editor in abstract graph and provides its visualizations. The abstract graph grammar describes the entity types contained in the graph and the connectivity of the entities. The objects in the graph are called entities and they are connected with connections. The abstract graph grammar is a grammar definition similar to BNF for textual languages. Visualization defines how entities in the abstract graph are presented and manipulated. ECL has an interface to semantics where customer can add semantic data and actions for the customized graph editor. ECL has simple formalism, great efficiency. But ECL only has static semantics and is not convenient for extension.

2.3 Conclusion of VLS

There are a large number of approaches mentioned above. How can we conclude which is better? Considering several aspects of visual language specification, we present the guidelines as follows:

1. **Efficiency.** VLS parsing algorithm efficiency. The parsing algorithm of most grammars takes exponential time. It does not suit for dealing with a graph with many visual symbols.
2. **Specifying Ability.** VLS should specify primitives, syntax and semantics of visual language. In contrast to text language, visual language should specify the layout constraints of primitives. VLS could specify context-sensitive visual language is better.
3. **Simplicity.** Easy for reading and writing. A visual environment is convenient for using.
4. **Extensibility.** A special VLS may be not suited for all visual language, so it should reserve an extension interface, which is useful for extending VLS.

Efficiency is contradictory to specifying ability. If VLS could specify all sides of visual language, the parsing algorithm may take great time. So we should balance VLS between efficiency and specifying ability.

4. VISUAL LANGUAGE EDITOR AUTOMATIC GENERATOR (VLEAG)

Visual language editor provides an environment to support writing, modifying, saving, and viewing visual language by the user, which is similar to text editor. There are two kinds of editing modes:

1. **Free-hand Editor.** User could write visual language freely, not restricted by the syntax. After writing, a

language-specific scanner and parser has to decide the membership. For a given graph and graph grammar, the parser tries to find a derivation sequence from the grammar's start graph to the given graph.

2. **Syntax-directed Editor.** When writing visual language, user has to abide by the syntax rules. Syntax-directed editor directs user to write visual language according to syntax, i.e. Rational Rose [5] EPMS [14].

Given a visual language specification, the VLEAG will automatically produce the visual language editor (cf. Figure1).

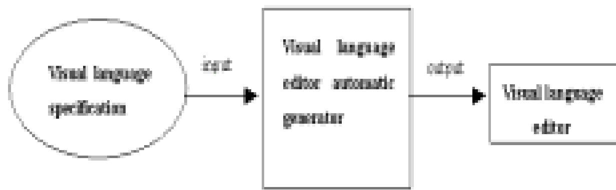

Figure 1. The Process of Producing Visual Language Editor

3.2 VLEAG

There are two main kinds of VLEAG: syntax-directed VLEAG, free-hand VLEAG.

3.2.1 Syntax-directed VLEAG

Roswitha and Bardohl introduced GENGED [1] basing on the algebraic graph grammars and graph grammar to define visual languages. The GENGED environment comprises of the GENGED-editor and the corresponding generated graphical editor (cf. figure 2).

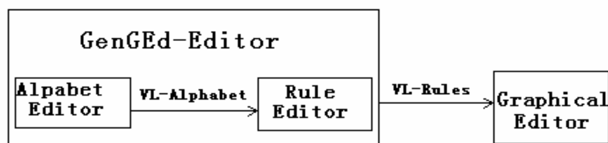

Figure 2. The GENGED Environment

The GENGED-editor allows the visual definition of a VL-alphabet and VL-rules. A syntax-directed graphical editor is generated from the definition, and supporting the editing of visual sentences. The GENGED-editor is divided into an alphabet editor and a rule editor allowing the definition of VL-rules. Alphabet editor consists of five parts:

1. **Graphic Object Editor** supports drawing and naming of graphical objects, which consist of primitive graphical objects as given by a rectangle, circle, arrow, etc.
2. **TypiEditor** defines the graphical objects representing a node or an edge, and gives a unique name to it. The name is used to as the graph type for the corresponding graphical symbol.
3. **ConEditor** defines the interrelations. The graphical symbols defined in TypiEditor can be selected and used for the interrelation definition, which is supported by this editor under intensively usage of graphical constraints.
4. **ConstrainHandling** offers a high-level constraint language.
5. **Graphical Constraint Solver** interprets and executes the low-level constraints.

Rule Editor consists of rule viewer, rule manager rule builder and AGG-system. It supports the visualization managing building and transformation of each rule.

SEI of BUAA introduced SGEG [17] (structured graphic editor generator) as a syntax-directed graphic editor generator. SGEG uses ECL (Editor Customization Language)[15] to define visual language.

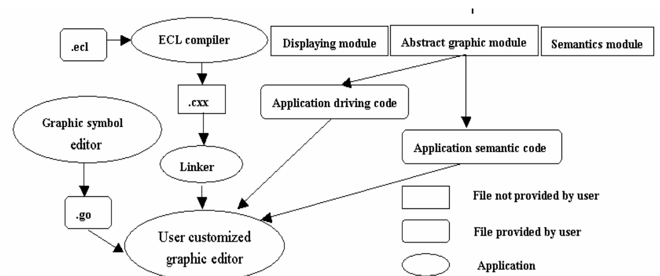

Figure 3. SGEG Command Package Process Model

On the structure viewpoint, SGEG consists of application supporting modules and supporting tools (cf. figure 3). Application supporting modules include abstract graphic module, semantics module and displaying module. Supporting tools include graphic symbol editor and ECL compiler. User makes an ECL file to define the visual language. ECL compiler generators a .cxx file from the

ECL file, the .cxx file is used in the graph editor application for checking rules. User uses graphic symbol editor to draw symbols and save them in a .go file. The graphic editor prototype can be tested and built immediately. The supporting modules link with semantics modules and the .cxx file to generate the target graphic editor.

Andy Schürr introduced PROGRES [11] basing on layered graph grammar, which can generate syntax-directed editor with the parsing algorithm. It integrates a set of language-specific tools supports editing, analyzing, and debugging of applications. The parsing algorithm reaches exponential time.

3.2.2 Free-hand VLEAG

Da-Qian Zhang and Kang Zhang introduced VisPro [16], basing on reserved graph grammar. The process of an editor construction consists of 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. The VisPro system consists of the framework and the specification tools. The specification tools consist of three parts (cf. Figure4):

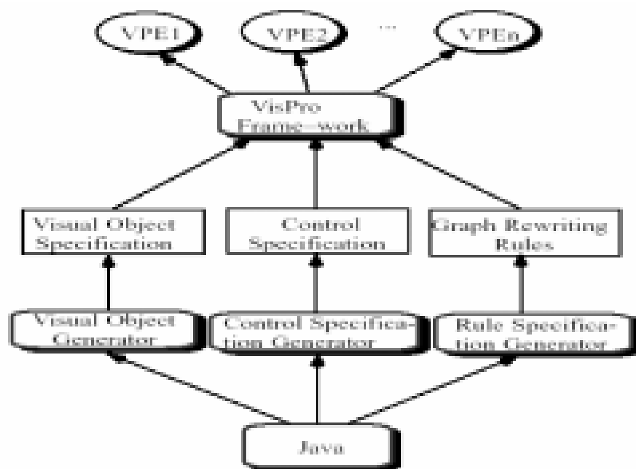

Figure 4. Constructing Visual Language Editor with VisPro

1. **Visual Object Generator** is a graph editor for constructing visual objects through direct manipulation.
2. **Control Specification Generator** is used to specify the control over visual objects constructed by the Visual Object Generator.
3. **Rule Specification Generator** specifies the domain knowledge in a graph rewriting rule formalism.

The parsing algorithm of VisPro has polynomial time complexity. The implementation of visual programming languages (VPLs) is automatically generated according to the specifications, which lacks a support for the layout. It uses java to specify the actions of its rules.

Eric J. Golin and Tom Magliery introduced SPARGEN [4], which is a compiler-compiler that automatically generates a visual language compiler from an Object-oriented picture layout grammars specification. Object-oriented picture layout grammar defines visual syntax that uses c++ to define graphical attributes and constraints. SPARGEN processes the grammar file and generates c++ code implementing the language specific elements of the compiler. This code is compiled and linked together with the supporting classes and SPAR library, which provides the base classes and the shell for the parsing algorithm. The result is a stand-alone visual language compiler, which reads an input picture (as a list of terminal symbols), then analyzes the picture to produce a parse tree and traverse the tree executing the actions associated with the productions. The parsing algorithm reaches the polynomial time for most visual languages.

3.3 Core Function Modules of VLEAG

Different VLAGs have varied structures and implementation methods, but there are some function modules similar in all VLAGs, which we call core function modules. Core function modules are the first class citizens and important characters of VLEAG.

Primitives Defining Module defines the logic and layout part of graphic symbols used in graph editor. User can draw or cite graphic symbols and give a unique name to each symbol representing its type.

VLS Defining Module uses a certain formalism to specify the visual language, and uses the names defined in primitives defining module to define the abstract and concrete syntax.

VLS Compiling or Interpreting Module compiles or interprets VLS to the file used in framework or other

modules.

VLS Parsing Module. Parsing a visual language takes two phases: syntax parsing and semantics parsing. Syntax parsing is to check whether the language is valid. Semantics parsing is to produce a result from a language. The result is meaningful only when the diagram is valid.

Syntax Checking Module checks whether the user operations are correct according to the syntax rules generated by VLS parsing module. It includes checking the connections and spatial layout.

Message Processing Module is in charge of processing messages. It translates outer operating messages into internal messages and sends the internal messages to the corresponding function.

Operation Servicing Module servers the basic functions of editor, i.e. creating, moving, copying, deleting and resizing a graphic symbol.

Displaying Module is in charge of drawing, arranging graph symbols and managing the user interface of editor. In the process of displaying graphic symbols, this module receives the internal messages and displays the result of the messages.

Saving Module saves information (position, size, state) of graphic symbol and data needed in VLEAG. This module provides a default storage device and an interface which users can assemble their saving modules to.

3.4 Conclusion of VLEAG

A better generator should meet the requirements provided by user and developers. We list the crucial requirements as follows:

1. **Reusability and Extensibility**. There are core function modules for every VLEAG as mentioned above. To reuse these modules, we should encapsulate the functions in components. Each component is loosely coupled with other one. At the same time, a VLS could not specify all visual languages, so extending VLS and application structure is necessary. VLEAG should isolate the influence produced by VLS extension to certain modules.
2. **Decoupling of Semantics**. Especially in modeling language, the semantics of different visual language varied much. Semantics depends on domain models. It is good to separate the semantics from other modules and provide an interface for user to define semantics.
3. **Decoupling of Application Logic and User Interface**. The application logic is a special part of visual language. This can avoid the influence bring by the interface changing.
4. **Simplicity**. Give a set of steps to produce a visual language editor and a uniform environment to use. Support user modifying the specification dynamically and not need to rebuild or link again.
5. **Independence of A Specific Storage Device** supports different kinds of storage device and the saving process is transparent.
6. **Distribution** allows users from different sites or computer to access or operate on the same project, i.e. distribution modeling.
7. **Platform Independence.** The application can be used on each platform.

References

[1] R. Bardohl. GenGEd: A generic graphical editor for visual languages based on algebraic graph grammars. In 1998 IEEE Symp. on Visual Languages, Halifax, Canada. Sept. 1998, pp. 48-55.

[2] H. Bunke. Attributed programmed graph grammars and their application to schematic diagram interpretation. IEEE Transactions on Pattern Analysis and Machine Intelligence, 4(6): 574-582,1982.

[3] D. Frankel, Model Driven Architecture: Applying MDA to Enterprise Computing, John Wiley&Sons, 2003

[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] IBM (2004). Rational Rose Official Web Site. Retrieved from: http://www-306.ibm.com/software/rational/

[6] J. Jaffar and J. L. Lassez. Constraint logic programming. Technical report, Monash University, June 1986.

[7] J. van Leeuwen, editor. Handbook of Theoretical Computer Science, volume 2, Elsevier, Amsterdam, New York, 1990.

[8] K. Marriott, B. Meyer. Visual Language Theory, Springer, New York, 1998, p1

[9] J.L. Pfaltz and A. Rosenfeld. Web grammars. In Proc. First Int. Joint Conference on Artificial Intelligence, pages 609-620,1969.

[10] 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.

[11] A. Schurr. PROGRES: A Visual Language and Environment for PROgramming with Graph REwrite Systems. Aachener Informatik Bericht 94-11, RWTH Aachen, Fachgruppe Informatik, 1994.

[12] A.C. Shaw. A formal picture description schema as a basis for picture processing systems. Information and Control, 14:9-52,1969.

[13] T. Tanaka. Definite clause set grammars: A formalism for problem solving. Journal of Logic Programming, 10:1-17,1991.

[14] D. Wang and J.R. Lee. Visual reasoning: its formal semantics and applications. Journal of Visual Languages and Computing, 4:327-345,1993

[15] 王雷、高仲仪，IPEE 结构化图形支持工具的研究与实现，软件学报，1997.8（增刊）

[16] D.-Q. Zhang, K. Zhang, VisPro: A visual language generation toolset. In Proceedings of the 1998 Symposium on Visual Languages, pages 195-202, Halifax, Canada, Sept. 1998.

[17] 张社英、刘又诚，过程模型建造环境 PMBE,软件学报，1997.8（增刊）