ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.19, No.8, August 2008, pp.1902−1919
DOI: 10.3724/SP.J.1001.2008.00842

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

可视化语言技术在软件开发中的应用*

孔 骏[1+], 赵春颖[2]

[1](Department of Computer Science, North Dakota State University, Fargo, ND 58105, USA)

[2](Department of Computer Science, University of Texas at Dallas, Richardson, TX 75080, USA)

Visual Language Techniques for Software Development

KONG Jun[1+], ZHAO Chun-Ying[2]

[1](Department of Computer Science, North Dakota State University, Fargo, ND 58105, USA)

[2](Department of Computer Science, University of Texas at Dallas, Richardson, TX 75080, USA)

+ Corresponding author: E-mail: jun.kong@ndsu.edu

Kun J, Zhao CY. Visual language techniques for software development. *Journal of Software*, 2008,19(8): 1902−1919. http://www.jos.org.cn/1000-9825/19/1902.htm

Abstract: Visual language techniques have exhibited more advantages in describing various software artifacts than one-dimensional textual languages during software development, ranging from the requirement analysis and design to testing and maintenance, as diagrammatic and graphical notations have been well applied in modeling system. In addition to an intuitive appearance, graph grammars provide a well-established foundation for defining visual languages with the power of precise modeling and verification on computers. This paper discusses the issues and techniques for a formal foundation of visual languages, reviews related practical graphical environments, presents a spatial graph grammar formalism, and applies the spatial graph grammar to defining behavioral semantics of UML diagrams and developing a style-driven framework for software architecture design.

Key words: visual language; graph grammar; UML semantics; software architecture

摘 要: 可视化语言技术比一维文本语言在描述软件组成方面具有优越性.由于图表和图形概念在系统建模中的广泛使用,可视化语言可以应用于需求分析、设计、测试和维护等软件开发的各个阶段.除了具有直观易见的特点之外,图文法在计算机上的精确建模和验证能力,为设计可视化语言提供了一个坚实的理论基础.讨论了可视化语言的形式理论基础,回顾了相关的可视化图形编程环境.特别提出了一种空间图文法,并且用该图文法定义了统一建模语言的行为语义.基于空间图文法,开发了一种基于模式驱动的框架,以帮助软件架构与设计.

关键词: 可视化语言;图文法;统一建模语言的语义;软件架构

中图法分类号: TP311 **文献标识码**: A

1 Introduction

Visual notations have been used in many disciplines and ranged from an architect's initial design to precise

* Received 2008-01-17; Accepted 2008-04-18

technical communication using rigorously defined notations[1]. Compared with texts, graphs are more intuitive to express structural and comparative information. Information that needs a lot of textual explanations may be intuitively and directly conveyed through a graph. There are many benefits of graphic notations. First, they can be used to model complex concepts and designs, such as object-oriented design. Notations as those in the *Unified Modeling Language* (UML) serve the useful purpose in communicating designs and requirements. Second, they can help people grasp large amount of information more quickly than text. Third, graphical notations cross language boundaries and can be used to communicate with people of different cultures.

With a support from computer hardware, the past two decades have seen a remarkable increase in the presence of graphs in computer systems. The graphical user interface makes computer accessible even to those people without much training in computer science. Users communicate with computers through exchanging digital messages with a graphical representation. The visual human-computer interaction proceeds by manipulating graphical notations: generation and recognition of graphs characterizing exchanged messages. Visual techniques also support the *visual software engineering*, which refers to the use of various visual means in addition to text in software development. The forms of the development include graphics, sound, color, gesture, and animation etc. Currently, graphs ranging from general-purpose notations to domain-specific ones have been widely used in software design. For almost every activity in the software process, a variety of graphical notations have been proposed.

Without a formal foundation, informal graphical notations, however, are not amendable for automatic analysis and verification on computers. Like the standard formal language theory supporting the definition of textual languages, a formalism supporting the specification of languages with a multi-dimensional representation is urgently needed. Such a formalism makes it feasible for designers to visually model software artifacts with a precise expression and for computers to correctly understand the information carried by a graph. This paper focuses on the issues and techniques for a formal foundation of visual languages (VLs)[2], which refer to those languages possessing a multi-dimensional representation, and applies visual techniques to software development. According to the type and extent of visual expressions, we can classify visual languages as icon-based languages, form-based languages and diagram-based languages. Although this paper focuses on diagram-based languages, which use nodes and edges as visual expressions, it will simply use the general term "*visual languages*" that is commonly used in the academic community.

The rest of this paper is organized as follows: Section 2 introduces the visual language specifications and compares different graph grammar formalisms. Section 3 discusses several well-known general purpose graphical environments and tools supporting graph transformation. Section 4 illustrates the needs for a new visual language specification formalism and briefly presents the new formalism, i.e. the *Spatial Graph Grammar*. Section 5 specifies the behavior semantic for UML diagrams using visual language techniques. Section 6 illustrates the application of graph grammar formalism to software development with a focus on software architecture and compares different approaches. Section 7 summarizes this paper.

2 Visual Languages and Graph Grammars

Due to its intuitive feature, graphical notations benefit the distribution and communication of software artifacts among designers, and thus have been extensively used in almost every stage of software design and development. For example, the UML[3] has been widely accepted as a standard modeling language. Featured by multidimensionality, visual languages, which take advantage of icons, spatial relations, or even temporal relations etc as language elements, facilitate the design and development of sophisticated systems by specifying computations

in a visual fashion. The multi-dimensional nature of visual languages offers an efficient way to interact with computers using languages with a graphical and intuitive representation, which can make computation more accessible to users without much training in programming and improve the productivity of software design and development.

Applying graphical notations to conceptualize and model systems, visual languages frequently use nodes to represent objects and edges to represent pre-determined relationships between objects. Similar to standard textural languages, a visual language needs a formalism to precisely specify its abstract syntax, which is indispensable for automatic analysis and verification. Being a natural offspring of a string-based formal language theory, graph grammars provide a well-established foundation[4] for defining visual languages with a precise expression. A graph grammar consists of a set of graph transformation rules (i.e. *productions*), which dictate the way of constructing a complete graph by a variety of nodes. Since all possible inter-connections among the nodes have been stated in the grammar, any connection between a pair of objects in a graph with a valid meaning can be eventually derived from a sequence of production applications, which incrementally rewrite one graph to another. Conversely, an unexpected connection indicates a violation on structural requirements.

The terms "graph transformation system" and "graph grammar" are often used as synonyms to each other. Precisely speaking, a graph transformation system[5] includes a set of productions, on which an application priority may be imposed. On the other hand, a graph grammar is constructed on a graph transformation system supplemented with 1) an initial graph and 2) a distinction of terminal and non-terminal nodes. Given a graph grammar, beginning from the initial graph, a generating process, i.e. a graph grammar is applied in a forward direction[6], can generate all well-formed graphs defined by the graph grammar; a parsing process, i.e. a graph grammar is applied in a reverse direction[6], can recognize the membership of a graph. In summary, a collection of graph transformation rules can be organized in a variety of ways[6]:

- An *unordered rewriting system* includes a collection of productions, which are applied in any order continuously until no appropriate rule can be applied.
- A *graph grammar* is made up of a collection of productions, in which labels are distinguished as "terminal" and "non-terminal". An initial graph is the start point of a generating process and the end point of a parsing process. In the generating process, the system starts from an initial graph to generate a specific graph without non-terminal labels. In the parsing process, the system starts from an input graph (called a *host graph*) and applies productions to induce it into the initial graph.
- An *ordered graph rewriting system* imposes a specific application order on productions. An ordered rewriting system is also referred to as a "programmed graph grammar", since the ordering of productions is reminiscent of the ordering of statements in an imperative programming language.
- An *event-driven graph rewriting system* gives an ordering of productions based on an external sequence of events.

Graph grammars provide an expressive mechanism to specify computations on graphs in a visual fashion, i.e. a computation is performed through a production application. Specially, each production consists of two sub-graphs, called the *left graph* and the *right graph*. A production application, i.e. a graph transformation, applies a production to a host graph and rewrites the host graph into another. The process of a graph transformation is modeled as recognize-select-execute[7]. In other words, a graph transformation is to find a *redex*, which denotes a sub-graph in the host graph isomorphic to a left/right graph upon the application direction, and to replace it with a copy of the right/left graph of the production. A production application denotes a local computation and the sequence of production applications can assemble local computations to realize a complex computation.

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 in the case of visual languages with many possible relationships among graphical objects, we need a more complicated mechanism to (re-)establish relationships between the surrounding of a replaced non-terminal and its replacing (non-)terminals, i.e. the embedding issue. Reker and Schürr[8] classified the embedding solutions into three categories:

- Context element: As applied in the *Layered Graph Grammar*[8], this solution identifies a common set of nodes and edges (i.e. *context elements*) in the left and right graphs of a production. Context elements are preserved in a graph transformation and newly created nodes are embedded into the host graph by establishing relations with context elements.
- Implicit embedding: Unlike graph grammars using edges to define relations between objects, some visual language specification formalisms (e.g. the *Picture Layout Grammar*[9] and the *Constraint Multiset Grammar* (CMG)[10]) use constraints over attribute values to implicitly specify all relations between objects. The attribute value assignment within a production establishes relations between newly created objects and their surrounding objects.
- Embedding rules: This solution introduces separate embedding rules to address the embedding issue. The embedding rules allow for the redirection of arbitrary sets of relations in a graph transformation. The *Reserved Graph Grammar* (RGG)[11] and the *Spatial Graph Grammar* (SGG)[12] encode embedding rules into productions through a *marking* technique.

Lakin[13] presented the concept of spatial parsing and described a type of grammar for specifying visual programming languages. In particular, the proposed grammatical approach allows the specification of a spatial arrangement among symbols on the right hand side. However, the annotated grammars are not formalized.

Using *multisets* (i.e. unordered collections) as the underlying data model, the formalism of *Picture Layout Grammars* (PLGs)[14] has been proposed to define pictures, which are viewed as unordered collections of visual symbols with attributes containing positional information. Improving the PLG, CMG[10] is a high level framework for the definition of visual programming languages. A production in a Constraint Multiset Grammar is of the form:

$$P ::= P_1,\ldots, P_n, \text{ where exist } P'_1, \ldots, P'_m \text{ where } C.$$

This production indicates that the non-terminal symbol P can be rewritten by the multiset of symbols $P_1,\ldots,P_n$ whenever there exist $P'_1,\ldots,P'_m$ such that the attributes of all symbols satisfy the constraint C. As a high-level grammatical specification language, the CMG supports the generation of diagram editors in the Penguins system[1]. Taking the specification of a diagram language as the input, the system automatically generates an incremental diagram parser, which interprets diagrams according to the relationships between objects captured by geometrical constraints. In particular, such a diagram editor combines the best features of the freehand and syntax-directed modes.

The *Relational Grammar* extends traditional one-dimensional string languages to higher dimensions through user-supplied domain relations[15]. In other words, sentential forms are specified as multisets of symbols with a set of relations in the symbol set[16]. Both the CMG and the Relational Grammar are characterized by generating or parsing languages according to relational constraints among objects in a textual form.

The above three formalisms fall in the category of multiset rewriting. Unlike graph grammars enforcing a strict distinction between objects (nodes) and relations (edges)[7], they manipulate all needed spatial or abstract relationships implicitly through constraints over attribute values of objects. Furthermore, abstract syntax graphs do not exist as a distinct notion within their syntax definitions.

Many proposed graph grammars, such as the NLG graph grammar[17], fall in the category of node-replacement

graph grammars, where a node of a given graph is replaced by a new sub-graph connecting to the remainder of the graph by new edges. Brandenburg investigated the complexity of node rewriting graph grammars, and reviewed the features of polynomial time graph grammars[18].

As a context-sensitive graph grammar, the *Layered Graph Grammar* (LGG) is equipped with an exponential parsing algorithm. Instead of exhaustive search, the parser uses a *breath-first search algorithm* such that possible sub-derivations are constructed and extended in parallel instead of re-computing them multiple times[8]. Filters are used to discard useless sub-derivations as soon as possible. Extending the LGG, Bottoni *et al*. proposed the *Contextual Layered Graph Grammar* (CLGG)[19], which provides new constructs, such as negative application conditions and complex predicates. The parsing algorithm of the CLGG is improved over that of the LGG through the application of critical pair analysis[20]. More specifically, non-conflicting rules are first applied to reduce the graph as much as possible. Afterwards, rule application conflicts are handled by creating decision points for the backtracking[19].

A *Triple Graph Grammar*[21], i.e. the coupled graph grammar, is used to build up interdependencies between spatial relations graphs (SRGs) and abstract syntax graphs (ASRs). The coupled graph grammar is used to define a diagram editor[22]. The editor updates the layout of a diagram on the basis of changes in the SRG. Since multiple layouts may exist to satisfy spatial relationships in an SRG, Rekers and Schürr proposed the *layout editing*, which allows users to change the layout while keeping the updated layout still satisfying all spatial constraints. In the coupled graph grammar, spatial relationships are implicitly defined as edges between nodes.

Bottoni, *et al*. introduced the Visual Conditional Attributed Rewriting system to specify visual interactive systems based on direct manipulation[23]. In an interactive session, images denoting exchanged messages are materialized and interpreted by the human and computer. The human interprets an image by recognizing characteristic structures. On the other hand, the computer uses a set of attributed symbols to capture the meaning of an image. Necessary spatial information, such as the physical position of an object, is identified through corresponding symbols, but does not participate in spatial transformation.

Brandenburg[24] presented a Layout Graph Grammar consisting of an underlying context-free graph grammar and layout specifications. Spatial relationships are derived according to the drawing of productions. A desirable layout is achieved by satisfying those constraints. One drawback of the approach is that grids and planar graphs cannot be captured by context-free graph grammars[24]. Another grammatical approach for graph drawing[25] embedded layout rules into the productions of a context-sensitive graph grammar formalism. The spatial relationship is represented by labels. This approach is suitable for graph drawing of visual programs. The above two formalisms only focus on the graph layout.

Of the three different approaches to visual language specification[26], the logical approach uses mathematical logic to axiomatize the possible spatial relationships between objects. Meyer[27] proposed a visual logic formalism, i.e. the *Picture Clause Grammar* (PCG), which integrates logic programming with graphical expressions. The PCG introduces visual terms into logic programming, i.e. *picture term* partially describing the contents of a picture, and is distinguished by the combination of spatial reasoning with the expressiveness of graphical communications. Another prominent logic approach is based on the *spatial logic*[28], which is made up of three components, i.e. geometric objects, spatial relations and description logic[29]. Capable of describing syntax and semantics of a visual language in a uniform framework, the spatial logic serves as the theory underlying an object oriented editor, i.e. GenEd[30], which can define visual languages such as Petri nets. Falling into the category of grammatical approaches, graph grammar has a different theoretical ground from the logic approaches mentioned above.

There are also some language-based techniques addressing the issues of image retrieval using symbolic

descriptions. Due to the large amount of information associated with images, efficient retrieval needs suitable representation of spatial information to index image contents. Chang, *et al*.[31] proposed 2-D symbolic strings to encode spatial relationships between objects as projected along the two coordinate axes. Del Bimbo, *et al*.[32] presented a logic-based language for describing the contents of stored images and specifying queries. This language specializes in the syntax and the semantics of *temporal logic* to deal with spatial ordering between the projections of objects in a scene. Soffer and Samet[33] presented a pictorial query specification technique for image databases and addressed the issues of matching, contextual and spatial ambiguity inherent in pictorial queries. Each query is composed of one or more query images by selecting required objects and positioning them according to the desired spatial configuration. These image retrieval techniques have inspired us to develop a precise semantics for spatial relationships within a grammatical framework.

In summary, different formalisms have been proposed to define VLs. However, those formalisms convert concrete spatial relations to abstract edges/attributes to specify spatial relations between nodes. We will introduce new graph grammar formalism - the Spatial Graph Grammar, which directly supports spatial specifications in the abstract syntax, in Section 4.

3 Graphical Environments for Visual Languages

The practical use of graph grammar and graph transformation is highly determined by the availability of development and debugging tools supporting the specification of visual languages. Currently, some general-purpose tools supporting the specification of visual languages have been proposed, such as the *VisPro*[34] and the *DiaGen*[35]. Figure 1 illustrates the concept of a graphic editor generator used to automatically generate a visual language supporting environment. Based on a set of productions, a graphical environment supporting the construction of visual languages can be automatically generated. The use of graphic editor generator separates the responsibility of language design from that of language usage, which complies with a 2-level meta-tool concept. A language user directly interacts with the generated graphical environment to manipulate visual languages. A graph transformation engine underlying the environment can interpret user-designed graphs and then take corresponding reactions based on a graph grammar designed by a language designer. More disciplined software development is encouraged by the 2-level process, i.e. the meta-tool used by language designers and the visual language environment by language users.

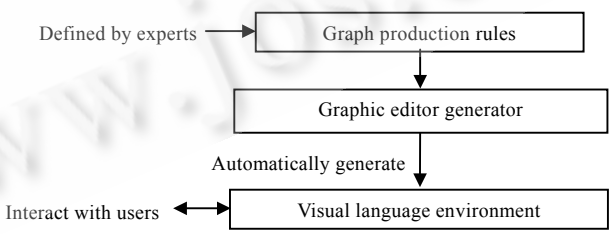

Fig.1 Support for visual language generation

In addition to the tools supporting the automatic generation of a visual language supporting environment, some general purpose environments for modeling graph-centric problems[36−38] based on graph transformation have been developed. Those environments in general provide a set of tools to assist the task of visual computing, such as tools for creating, analyzing, compiling and debugging graph transformation rules as well as rapid prototyping activities.

The *PROgrammed Graph Rewriting System* (PROGRES)[37,39,40] offers a hybrid visual language and a set of

tools for a systematic design and implementation of software artifacts. The PROGRES environment uses directed attributed graphs as data models and applies graph schema to specify static properties of a class of directed attributed graphs. Operations on those well-formed graphs are defined by high-level notations of graph transformation rules through the PROGRES language. Particularly, the PROGRES environment supports static check to ensure consistency-preserving operations.

The PROGRES, which combines rule-based, object-oriented, deductive and active database system as well as imperative programming concepts[37], tightly integrates graphical elements with textual elements: A graph transformation rule including visual notations and textual description for additional specifications. The backtracking, which is used to escape out of dead-ends of graph transformation processes, is supported by the implementation of the PROGRES and thus makes the PROGRES more powerful.

Pragmatically, the PROGRES environment includes a set of highly integrated tools for creating, analyzing, compiling and debugging graph transformation specifications, such as a mixed textual/graphical syntax-directed editor for editing graph schemata and productions etc. In general, the development of a software system using the PROGRESS language and its accompanied tools follows the design manner in the *Graph Grammar Engineering*[37,40−42], which offers a methodology for modeling data structures as graphs and performing operations through graph transformations.

The *Attributed Graph Grammar* (AGG) system[36] provides a visual programming environment based on a hybrid programming language (i.e. the AGG language), which integrates graph transformation rules with Java. In other words, AGG programs consist of a set of productions attributed by Java expressions and the standard Java library can be exploited to compute objects' attributes. The AGG language implements the *single pushout* of an algebraic approach[43] to support graph transformation, which is different from the algorithmic approach applied in the PROGRES language. The AGG language combines attributed graph transformation with negative application conditions[44,36]. A negative application condition allows designers to precisely specify a sub-graph, which must not be presented in order to perform a graph transformation. The tools in the AGG environment include an editor, which is used for manipulating directed attributed graphs, designing graph transformation rules and editing Java expressions, as well as supports for visual interpretation and debugging.

The *Fujaba* environment[38] provides a UML-like CASE tool for the round trip engineering. In other words, the Fujaba supports code generators for class diagrams, activity diagrams, state diagrams and collaboration diagrams as well as the reverse engineering functionality from Java code to UML diagrams[45]. Graph transformation in the Fujaba is used to specify behavioral aspects of models through a language called *story diagrams*[46]. The story diagram is a graph rewriting language using UML diagrams as presentation notations. More specifically, it combines class diagrams, collaboration diagrams and activity diagrams: the class diagram is used to specify graph schema, the collaboration diagram defines graph transformation rules and the activity diagram illustrates controls over graph transformations. Furthermore, the story diagram can be translated to a Java class, which provides an integration of the object-oriented language and the graph rewriting system.

The *VisPro* framework is a generic visual language generation environment with a hierarchical specification structure and multiple programming paradigms[34]. In order to allow independent development and integration, the VisPro decouples different functional modules interconnected through a protocol. A set of tools are provided in the VisPro. Based on those tools, the VisPro can automatically generate a graphical environment supporting a domain-specific visual language by (1) specifying visual objects with a desired appearance used in the target visual language and (2) defining a graph grammar for the syntax of the target visual language. The underlying graphical formalism used in the VisPro is the RGG[11]. As a context-sensitive formalism, the RGG is expressive in specifying

various types of graphs. Furthermore, it is distinguished from other graph grammar formalisms by organizing nodes in a two-level hierarchy and developing a marking technique to address the embedding issue. The RGG is equipped with an efficient parsing algorithm with polynomial time when working on confluent graph grammars.

DIAGEN, which uses hypergraphs to model various types of diagrams[47, 48], is a prototyping tool for creating diagram editors. In the hypergraph model, several distinct *attachment areas* of a diagram component connect to other diagram components and establish spatial relationships. In particular, edges in the model are distinguished as *component edges* and *relation edges*. The former represents diagram components and the latter indicates relationships between attachment areas. Through a hyper-edge graph grammar, the DIAGEN can recognize syntactic correctness of graphs during the editing process. The DIAGEN supports both the freehand and syntax-directed editing modes, which provide the flexibility for users and allow for efficient parsing of user specifications.

4 The SGG-Based Visual Language Approach

Designers of complex systems typically use graphical methods as conceptual devices to organize their design spaces. Contrast to traditional textual languages encoding multi-dimensional structures into one-dimensional strings, visual languages provide a direct representation of structures and concepts in a visual fashion[49]. In a typical visual language, nodes represent objects and edges denote relations between nodes. A graph glues nodes through edges into a complete system structure.

4.1 Need for spatial specification

The standard formal language theory can only define one-dimensional textual languages, while the multi-dimensional nature of visual languages needs a more powerful formalism to support multi-dimensional representations. Picture description formalisms, such as the *CMG*[10], are expressive in specifying languages of multi-dimensional representation by defining spatial relationships among visual objects. In those grammars, all needed relations are implicitly represented as constraints over attribute values of objects, which may cause side effects that create new relationships to unknown context elements. In general, those grammars use textual notations instead of visual ones to define the syntax of visual languages.

Unlike picture description formalisms, graph grammars make a strict distinction between objects (nodes) and relations (edges), and provide a more visual mechanism to define visual languages by taking advantage of visual notations. In addition to the *left* and *right* relationships found in a text, graphical objects in a visual language can hold various spatial relationships, such as *above* and *contain* etc. As noted by Reker and Schürr[22], the physical layout and the meaning of a diagram are two important aspects of a *visual sentence* (i.e. a graph with only terminal labels). Correspondingly, an SRG explores spatial relationships between graphical objects while an ASG provides structural information in a succinct form. The SRG is geared toward visualization, and the ASG toward interpretation[22]. The distinction between SRGs and ASGs offers different representations of the same concept simultaneously[7].

Some people also introduced spatial information to the abstract syntax. Andy Schürr proposed a Triple Graph Grammar to specify interdependencies between graph-like structures at a high level[21]. With one graph grammar specifying the SRGs and the other defining the ASGs, the triple graph grammar maintains a loose correspondence between the abstract and spatial aspects of VLs by introducing additional edges connecting SRG objects to the corresponding ASG objects[22]. Aiming at syntax-directed layouts, Brandenburg proposed the *Layout Graph Grammar*[24], which directly draws productions on a plane and generates a desirable layout according to the spatial relationships defined in productions. However, those graph grammars only explore the spatial relations from the layout aspect without a direct contribution to the interpretation of a graph.

Due to the visual nature of VLs, spatial information should not only contribute to the representation, but also explicitly and intuitively convey structural and semantic information over involved objects. A visual sentence can be viewed as a collection of graphical objects with a spatial arrangement. In an ASG, abstract relations are abstracted from significant spatial relations of a concrete representation of a visual language and modeled as edges connecting related objects. A visual sentence can produce a complex ASG with a large number of edges to model sophisticated relations between objects. The large number of edges in an ASG not only makes the abstract model hard to understand but also challenges the effort of designing a graph grammar validating the ASG. In order to generate a more concise ASG, some spatial relations instead of modeled as edges can be preserved to explicitly specify relationships between objects in the ASG when abstracting an ASG from an SRG. In other words, an ASG can use spatial relations together with edges to model abstract relations between objects. For example, rather than using directed edges or attributes to model an order over a collection of objects in an ASG, the order can be visually represented through spatial relationships between related objects (e.g. the left object has a smaller index than the right one). Figure 2 shows a graphical representation of a sequential specification in the XML schema. In a traditional abstract syntax graph, directed edges or attributes are needed to model the sequential order on the elements *FirstName* and *LastName*. By introducing spatial information to the abstract syntax, the sequence of the elements can be directly specified through their spatial configuration without using an extra edge or attribute, e.g. *FirstName* should occur before *LastName* by a left-right order. Especially in the case of a complex visual sentence, spatial information, when efficiently used, can significantly reduce the number of edges and produce a more concise and readable abstract model.

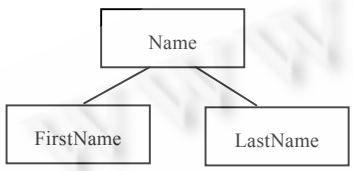

Fig.2 A sequence in the XML schema

4.2 Spatial graph grammars

All current graph grammars model spatial information either by edges or implicitly in terms of attribute values without a direct support for spatial information in the abstract syntax. In order to provide a formal foundation for interpreting abstract syntax graphs and integrating spatial information with edges to model abstract relations, a new graph grammar formalism is urgently needed. Recently, the *Spatial Graph Grammar*(*SGG*)[12] was proposed to introduce spatial notions into the abstract syntax. In the SGG, nodes and edges together with spatial relations construct the precondition of a production application. The direct representation of spatial information in the abstract syntax reduces the gap between the specifying language and the specified language since productions are often specified in a similar way to the concrete representation of specified graphs. With the capability of spatial specifications in the abstract syntax, the SGG is especially suitable for modeling complex systems by demonstrating abstract relations between objects through both layout and edge, which can make the specification more concise and understandable. Furthermore, the spatial specification can reduce the search space and derive an efficient parsing algorithm.

Figure 3 illustrates the concept of visual interaction with software artifacts based on the SGG. Upon a graph grammar in the form of the SGG, a meta-tool like *VisPro*[34] can automatically generate a graphical environment. In the generated environment, users can directly communicate with computers by drawing graphs. All communicated messages with a graphical representation will be validated and interpreted by the SGG parser. Therefore, based on the SGG, a developer uses a meta-tool to generate a graphical environment, which can be used by end users.

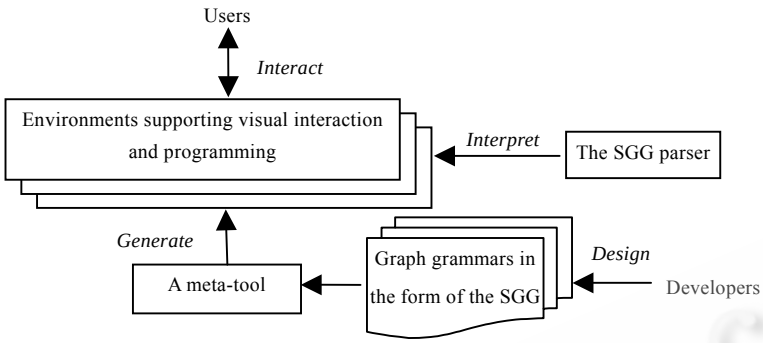

Fig.3 Visual interactions based on the Spatial Graph Grammar

In general, the description of a scene of objects in space involves spatial aspects that have an expression both in terms of inherent characteristics of each object and in the context of other objects[50]. The size and shape of an object illustrate its own properties while spatial relations express configurations among distinct spatial objects. Spatial relations, which the SGG takes as language constructs, can be classified into several types[51] including *direction* relations that describe an order in space, *topological* relations that express neighborhood and incidence, and *distance* relations such as near and far. Furthermore, *alignment* relations are introduced to illustrate projections of two objects on the x/y-axis. Developers may design a set of graphical notations to represent spatial relationships, and every SGG rule can have an equivalent graphical representation when replacing textual definitions with corresponding graphical notations.

In the SGG, a graph transformation may create new objects or adjust current spatial configuration. The SGG applies syntax-directed computation, called *action codes*, to derive new spatial properties from current spatial attributes. An *action code* is associated with a production, and is triggered when the production is applied. Writing an action code is like writing a standard event handler in Java. Therefore, in addition to a left graph and a right graph, each SGG production also includes a spatial specification and action codes.

The parser of the SGG takes advantage of spatial information defined in productions to reduce the search space, and performs in polynomial time with an improved parsing complexity over its non-spatial predecessor (i.e. the Reserved Graph Grammar[11]) when working on confluent graph grammars. More specifically, a *pattern sequence*, which is made up of objects in the right graph of a production, is generated based on spatial specifications between objects. Objects in a host graph are sequenced to make an ordered list, called a *host sequence*, based on the same criteria for generating the pattern sequence. Efficient string-matching techniques are used to search for the pattern sequence in the host sequence, and then a match of the pattern sequence is extended to construct a redex by searching for the remaining objects, which are in the right graph of the production but not present in the pattern sequence.

To the authors' knowledge, research on combining both spatial and abstract specifications in graph grammars has been scarce. The Spatial Graph Grammar (SGG) differs from other graph grammars by introducing spatial information to the abstract syntax. Layout does not have meaning in other graph grammars, which is changed in our approach by an explicit representation for layout. The direct representation of spatial information in productions can make the productions easier to understand since grammar designers often design rules in a similar way to their specified diagrams. Furthermore, the SGG is equipped with a parser, which is more efficient than its predecessor, i.e. the Reserved Graph Grammar[11], by using spatial information.

5 Defining Behavioral Semantics of UML Diagrams

Capable of modeling computing in a visual fashion, graph grammars have found applications in software architectures[52,53], pattern recognition[54], and many other areas[55−60]. In those applications, data are abstracted and modeled as graphs. Working on graphic data models, graph transformation and graph grammar provide a visual means to state how those models are dynamically evolved. The UML[3] provides a foundation for the Model Driven Architecture. The intuitive appearance of UML notations benefits the distribution and communication of software designs among different designers. However, UML diagrams lack a precise semantics, which restricts their applicability for automatic verification and analysis. The Spatial Graph Grammar provides a visual means to specify behavioral semantics and dynamic reconfigurations on UML models.

In general, the run-time system state is determined not only by the structure but also by the active state of each object. We model the system state through an integrated model combining the object and state diagrams, and give an integrated behavioral semantics, which specifies the dynamic reconfiguration in an object diagram taken together with state transitions of objects, to simulate the execution of the integrated model. The process of dynamic reconfiguration in object diagrams is realized through a sequence of graph transformations, on which an application order is imposed according to the occurrence of events. The state hierarchy in a statechart diagram is automatically formalized as a graph grammar, which can ease the effort in specifying the behavioral semantics. In order to interpret state transitions of simple and composite states, the automatically produced graph grammar is used to validate, recognize and generate active state configurations. The automation mechanism for generating graph grammars and their validation and recognition capability would populate the automated design of model-driven architectures. Therefore, an integrated behavioral semantics of object and state diagrams is obtained by an event-driven graph transformation and grammar system.

In their seminal work, Harel and Namaad[61] described the semantics of the language of the statecharts as implemented in the STATEMATE system, which is a commercial tool designed for the specification and design of real-life complex systems. Maggiolo-Schettini and Peron[62,63] interpreted statecharts through graph rewriting systems, and the evolution of a statechart is described by means of derivation sequences.

Kuske, *et al*.[64] used graph transformation rules to describe operations from class specifications and transitions from state transitions respectively. An integrated semantics is then given based on a combination of graph transformation rules associated with operations and those associated with state transitions. Kuske's approach is limited to simple states while the SGG based approach is applicable to composite states, which complicate the semantic specification since the active state could be represented as a tree instead of a single node. Furthermore, to handle complex models (such as an integrated diagram of object and state diagrams), the SGG can use spatial relations to denote structural relations between objects, and thus offer a concise representation. Such a spatial modeling can be directly specified in the SGG through the spatial specification in the abstract syntax.

Kuske[65] also explored the behavioral semantics of state diagrams through structured graph transformation. State hierarchies over a fixed state set S are defined by using a transformation unit, which includes a set of productions, initial and terminal graph class expressions, a control condition imposed on the production applications, and other transformation units. An active state is considered a sub-graph inside a state hierarchy. A state transition is performed as the application of a production, where the left graph indicates the current active state and the right graph denotes the new state configuration. Kuske's work does not discuss dynamic reconfigurations in object diagrams.

Graph transformation has been recognized as a powerful tool to formalize the semantics of languages with dynamic features[63]. Engels, *et al*.[66] used graph transformation rules to specify a goal-oriented interpreter for state

machines. This approach extends the static meta-model with a specification describing a system's behavioral semantics. The semantics is given in the form of collaboration diagrams, which can make the specification compatible with the UML standard. Engels's approach does not consider composite states. Instead of specifying an abstract interpreter for all state machines, the SGG based approach automatically abstracts a state machine as a graph grammar, and a state transition is performed as a combination of the parsing and generating processes.

Engels, *et al*.[67] later interpreted state diagrams by means of a mapping onto a semantic domain of the *Communicating Sequential Processes*[68]. The mapping is defined by a hybrid rule-based language, which combines textual grammar rules (specifying CSP expressions) with graphical patterns (specifying the abstract syntax of state diagrams). Baresi, *et al*.[69] provided a general method to interpret diagram notations in terms of denotational semantics by mapping the abstract syntax to some semantic domain through two sets of productions. One set defines the abstract syntax models and the other specifies the semantic models. The application of a production in the abstract syntax grammar can trigger the application of the corresponding production in the semantic grammar.

Fujaba[38] provides a round-trip engineering support for UML and Java. Especially, it tightly integrates the UML class and UML behavior diagrams with a visual programming language. Geiger and Zündorf[70] used the Fujaba to exemplify the specification of the behavioral semantics of state diagrams. In order to process composite states, this approach needs to perform a model transformation to obtain a plain state machine by removing or-state from an original state machine while the SGG based approach can directly work on state machines including composite states.

The intended meaning of the arrows touching the boundary of a composite state is asymmetric: the arrow pointing to the boundary denotes an initial pseudo-state while the arrow pointing away from the boundary denotes the internal structure of the state[71]. Gogolla and Presicce[71] flattened a state diagram by means of graph transformation to make the intended semantics explicit. This approach, however, does not discuss the execution semantics of state transitions.

Lilius and Paltor[72] translated a state machine into a term rewriting system. Kwon used conditional term rewriting systems to translate state diagrams into the input language of the model checker SMV[73] for a model checking[74]. Latella, *et al*.[75] first mapped state diagrams to extended hierarchical automata and then an operational semantics is defined for these automata. Varró[76] gave an operational semantics for UML statecharts based on meta-modeling techniques (using Extended Hierarchical Automaton as the underlying static structure) and model transition systems (for defining operational semantics). Labeled transition system is used as the underlying model to precisely specify UML active classes with an associated statechart[77]. Von der Beeck[78] developed a precise structured operational semantics (SOS) for UML-statecharts. *Abstract state machine* is used by Börger, *et al*.[79] to define the dynamic semantics of UML state machines. None of those approaches uses the graph transformation.

6 Style-Driven Architecture Design

In addition to behavioral semantics, the SGG also provides an expressive mechanism to specify the structure of models. In general, class diagrams provide a declarative approach to defining the static structure of models. On the other hand, a grammatical approach specifies structures in a constructive and incremental fashion. Though the declarative approach is easier to understand, the constructive and incremental method is more suitable for analysis and evolution. We propose a style-driven framework for software architecture specification based on the Spatial Graph Grammar. In this framework, the designer uses graphical notations to define architectural styles. Given the graphical specifications, a visual language generator, i.e. a meta-tool, can automatically generate a specific design environment, which is suitable for the users without any knowledge of formal methods to directly manipulate

software architectures by drawing graphs. The graph transformation engine underlying the design environment can automatically validate structural integrity and reveal the hierarchical structure of a user-defined software architecture.

Many architecture description language (ADLs), such as Wright[80] and Rapide[81], have been proposed to model and analyze software architectures. Based on formal models, those languages allow users to define software architectures without ambiguity, and thus are suitable for automatic reasoning on architectures. There is, however, a mismatch between the abstraction level at which users usually model the software architectures and the abstraction level at which users should work with these languages[82]. In order to model software architectures using those languages in their current forms, users need expertise with a solid technique background. With the meta-tool capability, the graph grammar approach can overcome the problem by automatically generating a style-specific design environment. Then, users without any formal method knowledge can directly specify and manipulate software architectures in terms of box-and-line drawings, which are verified by a graph transformation engine.

The UML[3] provides a family of design notations to model various aspects of systems. As a general-purpose modeling language, the UML has also been applied to defining software architectures. Medvidovic, *et al*. systematically presented two strategies to model software architectures in the UML[83]. Focusing on the usability of concepts, Garlan, *et al*. proposed several alternatives to map concepts from ADLs to the UML[84]. Both works exhaustively discuss the strength and weakness of each method. Emphasizing on the analysis and validation of designed models, Baresi *et al*. used the UML notations to specify the static aspect of structural styles paired with graph transformation for modeling dynamic reconfiguration[82]. Selonen and Xu applied UML profiles to software architecture design process and software architecture description[85]. Such profiles, called *architectural profiles*, define the structural and behavioral constraints and rules of the architecture under design, and are used to drive, check and automate the software architecture design process and the creation of all architectural views.

Some researchers apply the typed graph approach to defining architectural styles. For example, Wermelinger and Fiadeiro[86] used typed graphs to specify all possible connections between components. Briefly, a typed graph $\langle G,t\rangle$ is a graph G equipped with a morphism $t: G\rightarrow TG$, where TG is a fixed graph, i.e. the type graph[87]. The typed graph approach also leads to a declarative fashion as the UML. We argue that graph grammars are more expressive in specifying architectural styles than typed graphs by associating attributes to nodes.

Formalizing graphs through set theories, Dean and Cordy[52] applied the diagrammatic syntax to express software architectures, and used it for pattern matching. Their work focuses on exploiting the composition of software architectures. Taentzer, *et al*.[88] used the distributed graph transformation to specify dynamic changes in distributed systems. The changes are organized in a two-level hierarchy. One is related to the change in a local node and the other is associated with the structure of the distributed system itself. This work emphasizes modeling dynamic changes of distributed systems rather than specifications of structural composition.

Métayer[53] presented a formalism for the definition of software architectures in terms of graphs. Dynamic evolution is defined independently by a coordinator. Métayer's approach only allows a single node to be replaced with a sub-graph, and thus is limited to those graphs, which can be specified by context-free graph grammars. On the other hand, our approach is more expressive in specifying software architectures by allowing arbitrary graphs in both left and right graphs of a production. Furthermore, our methodology is supported by a set of tools. Without spatial specifications, the Spatial Graph Grammar is the same as the Reserved Graph Grammar and thus can directly use the VisPro[34], which is based on the Reserved Graph Grammar, to automatically generate an application-specific design environment. The environment is able to verify the structural properties of software architectures.

Based on the theoretic foundation of term rewriting systems, Inverardi and Wolf[89] applied the *Chemical*

Abstract Machine (CHAM)[90] model to the architectural description and analysis. Briefly speaking, software systems are viewed as chemicals whose reactions are controlled by explicitly stated rules. Wermelinger[91] further proposed two CHAMs, i.e. the creation CHAM and the evolution CHAM, to define the architectural style and the reconfiguration respectively.

Karsai, *et al*.[92,93] proposed the Model-Integrated Computing (MIC) to address essential needs of embedded software development. The MIC uses domain-specific modeling languages, which are tailored to the needs of a particular domain, to represent static and dynamic properties of a system. Similar to the meta-tool capability of our approach, the MIC supports to transform a meta-programmable generic modeling environment into a domain-specific environment, which only allows the creation of models complying with the abstract syntax of a modeling language. Instead of using the UML class diagram as the meta-model to define the abstract syntax of a domain-specific modeling language, we apply the Spatial Graph Grammar to define a language. Supported by a formal basis of graph grammars, the rule-based specification is more suitable for reasoning and verification.

In an effort to adapt the principles and technology of generic software development environment to provide style-specific architectural support, the Aesop system[94] defines a style-specific vocabulary of design elements by specifying subtypes of the seven basic architectural classes. Then, designers have to over-load the methods of these subtypes to support stylistic constraints. Taking the advantage of conciseness and intuitiveness of graph transformation, our approach supports a high level specification of architectural styles through graph grammars.

7 Summary

This paper has reviewed issues related to the visual languages technique and compares various graph grammars. The Spatial Graph Grammar formalism developed from a context-sensitive graph grammar formalism, the Reserved Graph Grammar[11], is highlighted. Though the RGG is powerful in specifying structural relationships among objects, it has no support for describing what a graph looks like. The SGG enhances the RGG with a spatial extension. In general, the new formalism satisfies the following criteria:

- **Expressive**: Potential spatial relationships can be clearly specified without sacrificing the expressiveness of structural specifications.
- **Flexible**: Though tightly coupled with spatial configurations, the structural part of the grammar can be independently used, just as the original RGG, without relying on spatial specifications.

The SGG takes spatial notions as language constructs, which distinguishes it from other graph grammar formalisms. Existing graph grammars model structures with nodes and edges while the SGG further introduces spatial information to the abstract syntax. The spatial information benefits the SGG on the following aspects:

- The SGG extends the expressive power of the RGG by introducing a spatial specification mechanism. It provides a high-level language for explicitly expressing both abstract and spatial relationships within a single grammatical definition.
- The direct representation of spatial information in the abstract syntax can make productions easy to understand since it reduces the gap between the concrete representation of languages and the specification of productions.
- With the help of spatial specifications, the parser of the SGG can reduce the search space, and thus achieve a better performance than that of the RGG.

Featured with spatial specifications in the abstract syntax, the SGG is especially suitable for complex systems since it can model structural relations between objects directly through layout instead of edges and attributes, which makes the specification more concise and intuitive. To exemplify the application of the SGG in software

engineering, we use the SGG to specify behavioral semantics of UML diagrams and propose a style-driven framework for software architecture design:

- The expressiveness and reasoning power of an event-driven graph transformation and grammar system support specifying an integrated behavioral semantics of UML object and state diagrams with composite states.
- The state hierarchy of a state machine is automatically compiled as a graph grammar, which greatly eases the efforts in interpreting state transitions. The recognition and generation of state configurations are performed as a parsing process and a generating process, respectively.
- Spatial information is used to model relations in the abstract syntax.
- The meta-tool capability automatically generates a graphical authoring environment for users without knowledge of formal methods. The generated environment allows users to represent design policies and requirements through graphical notations. The graph transformation engine underlying the environment validates users' designs.

References:

[1] Chok SS, Marriott K. Automatic generation of intelligent diagram editors. ACM Trans. on Computer-Human Interaction, 2003, 10(3):244−276.

[2] Burnett MM. Visual language research bibliography. 2008. http://www.cs.orst.edu/~burnett/vpl.html

[3] Rumbaugh J, Jacobson I, Booch G. The Unified Modeling Language Reference Manual. 2nd ed., Addison-Wesley, 2005.

[4] Rozenberg G. Handbook on graph grammars and computing by graph transformation: Foundations. World Scientific, 1997.

[5] Andries M, Engels G, Habel A, Hoffmann B, Kreowski HJ, Kuske S, Plump D, Schürr A, Taentzer G. Graph transformation for specification and programming. Science of Computer Programming, 1999,34:1−54.

[6] Blostein D, Fahmy H, Grbavec A. Issues in the practical use of graph rewriting. In: Proc. of the 5th Int'l Workshop on Graph Grammars and Their Application to Computer Science. LNCS 1073, 2005. 38−55.

[7] Bardohl R, Taentzer G, Minas M, Schürr A. Application of graph transformation to visual languages. In: Ehrig H, Engels G, Kreowski HJ, Rozenberg G, eds. Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools, Vol.2. 1999. 105−180.

[8] Rekers J, Schürr A. Defining and parsing visual languages with layered graph grammars. Journal of Visual Languages and Computing, 1997,8(1):27−55.

[9] Golin EJ. Parsing visual languages with picture layout grammars. Journal of Visual Languages and Computing, 1991,4(2): 371−394.

[10] Marriott K. Constraint multiset grammars. In: Proc. of the 1994 IEEE Symp. on Visual Languages. 1994. 118−125.

[11] Zhang DQ, Zhang K, Cao J. A context-sensitive graph grammar formalism for the specification of visual languages. The Computer Journal, 2001,44(3):187−200.

[12] Kong J, Zhang K, Zeng XQ. Spatial graph grammar for graphic user interfaces. ACM Trans. on Human-Computer Interaction, 2006,13(2):268−307.

[13] Lakin F. Visual grammars for visual languages. In: Proc. of the American Association for Artificial Intelligence. 1987. 683−688.

[14] Golin EJ, Reiss SP. The specification of visual language syntax. In: Proc. of the IEEE Workshop on Visual Languages. 1989. 105−110.

[15] Weitzman L, Wittenburg K. Relational grammars for interactive design. In: Proc. of the IEEE Symp. on Visual Languages. 1993. 4−11.

[16] Wittenburg K. Earley-Style parsing for relational grammars. In: Proc. of the IEEE Workshop on Visual Languages. 1992. 192−199.

[17] Engelfriet J, Rozenberg G. Node replacement graph grammars. In: Rozenberg G, ed. Handbook on Graph Grammars and Computing by Graph Transformation: Foundations. 1997. 1−94.

[18] Brandenburg FJ. On polynomial time graph grammars. In: Proc. of the 5th Conf. on Theoretical Aspects of Computer Science. 1988. 227−236.

[19] Bottoni P, Taentzer G, Schürr A. Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation. In: Proc. of the 2000 IEEE Int'l Symp. on Visual Languages. 2000. 59−60.

[20] Plump D. Hypergraph rewriting: Critical pairs and undecidability of confluence. In: Sleep MR, Plasmeijer MJ, van Eekelen M, eds. Term Graph Rewriting. 1993. 201−214.

[21] Schürr A. Specification of graph translators with triple graph grammars. In: Proc. of the 20th Inte'l Workshop on Graph-Theoretic Concepts in Computer Science. 1994. 151−163.

[22] Rekers J, Schürr A. A graph based framework for the implementation of visual environments. In: Proc. of the IEEE Symp. on Visual Languages. 1996. 148−155.

[23] Bottoni P, Costabile MF, Mussio P. Specification and dialogue control of visual interaction through visual rewriting systems. ACM Trans. on Programming Languages and Systems, 1999,21(6):1077−1136.

[24] Brandenburg FJ. Designing graph drawings by layout graph grammars. In: Proc. of the Int'l Workshop on Graph Drawing. LNCS 894, 1995. 416−428.

[25] Zhang KB, Zhang K. Grammar-based approach for a visual programming language generation system. In: Proc. of the 2nd Int'l Conf. on Theory and Application of Diagrams. LNAI 2317, 2002. 106−108.

[26] Marriott K, Meyer B, Wittenburg KB. A survey of visual languages specification and recognition. In: Marriott K, Meyer B, eds. Visual Language Theory. Springer-Verlag, 1998. 5−85.

[27] Meyer B. Pictures depicting pictures on the specification of visual languages by visual grammars. In: Proc. of the IEEE Workshop on Visual Languages. 1992. 41−47.

[28] Haarslev V. A fully formalized theory for describing visual notations. In: Marriott K, Meyer B, eds. Visual Language Theory. Springer-Verlag, 1998. 261−292.

[29] Brachman RJ, Schmolze JG. An overview of the KL-ONE knowledge representation system. Cognitive Science, 1985. 171−216.

[30] Haarslev V, Wessel M. GenEd−An editor with generic semantics for formal reasoning about visual notations. In: Proc. of the 1996 IEEE Symp. on Visual Languages. 1996. 204−211.

[31] Chang SK, Shi QY, Yan CW. Iconic indexing by 2-D string. IEEE Trans. on Pattern Analysis and Machine Intelligence, 1987(9): 413−427.

[32] Del Bimbo A, Vicario E, Zingoni D. A spatial logic for symbolic description of image contents. Journal of Visual Languages and Computing, 1994,5(3):267−286.

[33] Soffer A, Samet H. Pictorial query specification for browsing through Spatially-referenced image databases. Journal of Visual Languages and Computing, 1998,9(6):567−596.

[34] Zhang K, Zhang DQ, Cao J. Design, construction, and application of a generic visual language generation environment. IEEE Trans. on Software Engineering, 2001,27(4):289−307.

[35] Minas M, Viehstaedt G. DiaGen: A generator for diagram editors providing direct manipulation and execution of diagrams. In: Proc. of the 11th IEEE Symp. on Visual Languages. 1995. 203−210.

[36] Ermel C, Rudolf M, Taentzer G. The AGG approach: Language and tool environment. In Ehrig H, Engels G, Kreowski HJ, Rozenberg G, eds. Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools, Vol.2. 1999. 551−601.

[37] Schürr A, Winter AJ, Zündorf A. The progres approach: Language and environment. In: Ehrig H, Engels G, Kreowski HJ, Rozenberg G, eds. Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools, Vol.2. 1999. 487−550.

[38] Fujaba. 2005. http://www.fujaba.de

[39] Schürr A, Winter AJ, Zundorf A. Visual programming with graph rewriting systems. In: Proc. of the IEEE Int'l Symp. on Visual Languages. 1995. 326 −333.

[40] Schürr A, Winter AJ, Zündorf A. Graph grammar engineering with PROGRES. In: Proc. of the 5th European Software Engineering Conf. LNCS 989, 1995. 219−234.

[41] Engels G, Lewerentz C, Schäfer W. Graph grammar engineering: A software specification method. In: Ehrig H, Nagl M, Rozenberg G, Rosenfeld A, eds. Graph Grammars and Their Application to Computer Science. LNCS 291, 1987. 186−201.

[42] Engels G, Lewerentz C, Nagl M, Schäfer W, Schürr A. Building integrated software development environments−Part 1: Tool specification. ACM Trans. on Software Engineering and Methodology, 1992,1(2):135−167.

[43] Ehrig H, Heckel R, Korff M, Löwe M, Ribeiro L, Wagner A, Corradini A. Algebraic approach to graph transformation II: Single Pushout approach and comparison with double Pushout approach. In: Rozenberg G, ed. Handbook of Graph Grammar and Computing by Graph Transformation: Foundations. 1997. 247−312.

[44] Habel A, Heckel R, Taentzer G. Graph grammars with negative application conditions. Fundamenta Informaticae, 1996,26(3,4):

287−313.

[45] Maier T, A Zündorf. The fujaba statechart synthesis approach. In: Proc. of the 2nd Int'l Workshop on Scenarios and State Machines: Models, Algorithms, and Tools. 2003.

[46] Fischer T, Niere J, Torunski L, Zündorf A. Story diagrams: A new graph rewrite language based on the unified modeling language and Java. In: Proc. of the Theory and Application to Graph Transformations. LNCS 1764, 1998. 296−309.

[47] Minas M. Diagram editing with hypergraph parser support. In: Proc. of the 13th IEEE Symp. on Visual Languages. 1997. 226−233.

[48] Minas M. Concepts and realization of a diagram editor generator based on hypergraph transformation. Science of Computer Programming, 2002,40:157−180.

[49] Cox PT, Smedley T. Building environments for visual programming of robots by demonstration. Journal of Visual Languages & Computing, 2000,11(5):549−571.

[50] Clementini E, Felice PD, Hernandez D. Qualitative representation of positional information. Artificial Intelligence, 1997,95: 317−356.

[51] Pullar D, Egenhofer M. Toward formal definitions of topological relationships. In: Proc. of the 3rd Int'l Symp. on Spatial Data Handling. 1988. 225−241.

[52] Dean TR, Cordy JR. A syntactic theory of software architecture. IEEE Trans. on Software Engineering, 1995,21(4):302−313.

[53] Métayer DL. Describing software architecture styles using graph grammars. IEEE Trans. on Software Engineering, 1998,24(7): 521−533.

[54] Blosetin D, Schürr A. Computing with graphs and graph transformation. Software-Practice and Experience, 1999,29(3):197−217.

[55] Ehrig H, Engels G, Kreowski HJ, Rozenberg G. Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools. World Scientific, 1999.

[56] Ehrig H, Kreowski HJ, Montanari U, Grzegorz Rozenberg. Handbook of Graph Grammars and Computing by Ggraph Transformation: Concurrency, Parallelism, and Distribution. World Scientific, 1999.

[57] Mens T, Schürr A, Taentzer G. Proc. GraBaTs 2002-Graph-Based Tools, Electronic Notes in Theoretical Computer Science. 2002,72(2).

[58] Bottoni P, Minas M. Proc. GT-VMT'2002-Graph Transformation and Visual Modeling Techniques, Electronic Notes in Theoretical Computer Science. 2003,72(3).

[59] Heckel R, Mens T, Wermelinger M. Proc. Workshop on Software Evolution Through Transformations Toward Uniform Support Throughout the Software Lift-Cycle. Electronic Notes in Theoretical Computer Science, 2003,72(4).

[60] Kreowski HJ, Knirsch P. Proc. AGT2002-Applied Graph Transformation. Grenoble, 2002.

[61] Harel D, Namaad A. The STATEMATE semantics of StateCharts. ACM Trans. on Software Engineering and Methodology, 1996, 5(4):293−333.

[62] Maggiolo-Schettini A, Peron A. Semantics of full statecharts based on graph rewriting. In: Proc. of the Graph Transformation in Computer Science. LNCS 776, 1994. 265−279.

[63] Maggiolo-Schettini A, Peron A. A graph rewriting framework for statecharts semantics. In: Proc. of the 5th Int'l Workshop on Graph Grammars and Their Application to Computer Science. LNCS 1073, 1996. 107−121.

[64] Kuske S, Gogolla M, Kollmann R, Kreowski HJ. An integrated semantics for UML Class, Object and State Diagrams Based on Graph Transformation. In: Proc. of the 3rd Int'l Conf. on Integrated Formal Methods. LNCS 2335, 2002. 11−28.

[65] Kuske S. A formal semantics of UML state machines based on structured graph transformation. In: Proc. of the UML. 2001. 241−256.

[66] Engels G, Hausmann JH, Heckel R, St. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In: Proc. Of the UML 2000. LNCS 1939, 2000. 323−337.

[67] Engels G, Heckel R, Küster JM. Rule-Based specification of behavioral consistency based on the UML meta model. In: Proc. of the UML. 2001. 272−286.

[68] Hoare CAR. Communicating Sequential Processes. Prentice Hall Int'l Series in Computer Science, Prentice-Hall International, 1985.

[69] Baresi L, Milano PD, Pezzè M. Formal interpreters for diagram notations. ACM Trans. on Software Engineering and Methodology, 2005,14(1):42−84.

[70] Geiger L, Zündorf A. Statechart modeling with fujaba. In: Proc. of the Int'l Workshop on Graph-Based Tools. 2004.

[71] Gogolla M, Presicce FP. State diagrams in UML: A formal semantics using graph transformations. In: Proc. of the ICSE'98 Workshop Precise Semantics of Modeling Techniques. Technical Report, TUM-I9803, 1998. 55−72.

[72] Lilius J, Paltor IP. Formalizing UML state machines for model checking. In: Proc. of the UML'99. LNCS 1723, 1999. 430−444.
[73] McMillan KL. Symbolic model checking: An approach to the state explosion problem [Ph.D. Thesis]. Department of Computer Science, Carnegie Mellon University, 1992.
[74] Kwon G. Rewrite rules and operational semantics for model checking UML statecharts. In: Proc. of the UML. 2000. 528−540.
[75] Latella D, Majzik I, Massink M. Towards a formal operational semantics of UML statechart diagrams. In: Proc. of the IFIP TC6/WG6.1 3rd Int'l Conf. on Formal Methods for Open Object-Oriented Distributed Systems. 1999. 331−347.
[76] Varró D. A formal semantics of UML statecharts by model transition systems. In: Proc. of the ICGT 2002. LNCS 2505, 2002. 378−392.
[77] Reggio G, Astesiano E, Choppy C, Hussmann H. Analyzing UML active classes and associated state machines-A lightweight formal approach. In: Proc. of the FASE. LNCS 1783, 2000. 127−146.
[78] von der Beeck M. A structured operational semantics for UML-StateCharts. Software and Systems Modeling, 2002,1(2):130−141.
[79] Börger E, Cavarra A, Riccobene E. Modeling the dynamics of UML state machines. In: Proc. of the ASM. LNCS 1912, 2000. 223−241.
[80] Allen R, Garlan D. A formal basis for architectural connection. ACM Trans. on Software Engineering and Methodology, 1997,6(3): 213−249.
[81] Luckham D, Kenney J, Augustin L, Vena J, Bryan D, Mann W. Specification and analysis of system architecture using rapide. IEEE Trans. on Software Engineering, 1995,21(4):336−355.
[82] Baresi L, Heckel R, Thöne S, VarróD. Modeling and validation of service-oriented architectures: Application vs. style. In: Proc. of the ESEC/FSE 2003. 2003. 68−77.
[83] Medvidovic N, Rosenblum DS, Redmiles DF, Robbins JE. Modeling software architectures in the unified modeling language. ACM Trans. on Software Engineering and Methodology, 2002,11(1):2−57.
[84] Garlan D, Cheng SW, Kompanek AJ. Reconciling the needs of architectural description with object-modeling notations. Science of Computer Programming, 2002,44:23−49.
[85] Selonen P, Xu J. Validating UML models against architectural profiles. In: Proc. of the ESEC/FSE. 2003. 58−67.
[86] Wermelinger M, Fiadeiro JL. A graph transformation approach to software architecture reconfiguration. Science of Computer Programming, 2002,44(2):133−155.
[87] Corradini A, Montanari U, Rossi F. Graph Processes. Fundamenta Informaticae, 1996,26(3−4):241−265.
[88] Taentzer G, Geodicke M, Meyer T. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In: Proc. of the 6th Int'l Workshop Theory and Application of Graph Transformations. LNCS 1764, 1998. 179−193.
[89] Inverardi P, Wolf AL. Formal specification and analysis of software architectures using the chemical abstract machine model. IEEE Trans. on Software Engineering, 1995,21(4):373−386.
[90] Berry G, Boudol G. The chemical abstract machine. Theoretical Computer Science, 1992,96:217−248.
[91] Wermelinger M. Towards a chemical model for software architecture reconfiguration. In: Proc. of the 4th Int'l Conf. on Configurable Distributed Systems. 1998. 111−118.
[92] Karsai G, Sztipanovits J, Ledeczi A, Bapty T. Model-Integrated development of embedded software. Proc. of the IEEE, 2003,91(1): 145−164.
[93] Sztipanovits J, Karsai G. Generative programming for embedded systems. In: Proc. of the Generative Programming and Component Engineering. LNCS 2487, 2002. 32−49.
[94] Garlan D, Allen R, Ockerbloom J. Exploiting styles in architectural design environments. In: Proc. of the 2nd ACM SIGSOFT Symp. on Foundations of Software Engineering. 1994. 175−188.

KONG Jun was born in 1976. He is an assistant professor at North Dakota State University, USA. His research areas are visual languages, human computer interaction and model driven architecture.

ZHAO Chun-Ying was born in 1979. She is a Ph.D. candidate at the University of Texas at Dallas, USA. Her current research areas are software engineering and visual languages.