Visual Languages and Applications

Kang Zhang

Visual Languages and Applications

Visual Languages and Applications

Kang Zhang
The University of Texas at Dallas
USA

Springer

Kang Zhang
University of Texas at Dallas
Dept. Computer Science
Richardson TX 75083-0688
kzhang@utdallas.edu

Visual Languages and Applications by
Kang Zhang

Library of Congress Control Number: 2006939508

ISBN-10: 0-387-29813-4 e-ISBN-10: 0-387-68257-0
ISBN-13: 978-0-387-29813-9 e-ISBN-13: 978-0-387-68257-0

Printed on acid-free paper.

Printed in the United States of America.

9 8 7 6 5 4 3 2 1

springer.com

Contents

Preface

Visual languages have long been a pursuit of effective communication between human and machine. Today, they are successfully employed for end-user programming, modeling, rapid prototyping, and design activities by people of many disciplines including architects, artists, children, engineers, and scientists. Furthermore, with rapid advances of the Internet and Web technology, human-human communication through the Web or electronic mobile devices is becoming more and more prevalent.

This manuscript provides a comprehensive introduction to diagrammatical visual programming languages and the technology of automatic generation of such languages. It covers a broad range of contents from the underlying theory of graph grammars to the applications in various domains. The contents were extracted from the papers that my Ph.D. students and I have published in the last 10 years, and are updated and organized in a coherent fashion. The manuscript gives an in-depth treatment of all the topic areas. Pointers to related work and further readings are also facilitated at the end of every chapter except Chapter 9.

Rather than describing how to program visually, the manuscript discusses what are visual programming languages, and how such languages and their underlying foundations can be usefully applied to other fields in computer science that need graphs as the primary means of representation.

Assuming the basic knowledge of computer programming and compiler construction, the manuscript can be used as a textbook for senior or graduate computer science classes on visual languages, or a reference book for programming language classes, practitioners, and researchers in the related field.

The manuscript cannot be completed without the helps of many people. First of all, I am very grateful to Shi-Kuo Chang, a pioneer of visual languages and one of the greatest computer scientists and Chinese novelists, for writing a foreword for this manuscript. I would like thank my past and present Ph.D. students who have contributed to its rich contents, particularly Jun Kong, Guanglei Song, Da-Qian Zhang, and Chunying Zhao. My thanks also go to Maolin Huang (University of Technology, Sydney,

Australia) for allowing me to apply his work to Web visualization and browsing; and to Xiaoqin Zeng (Hohai University, China) for his contribution to the generalization of Reserved Graph Grammars. Publishers of the original papers including IEEE, Oxford Press, and Springer are acknowledged for their permission to reuse the contents previously published in their respective journals and conferences. Finally, I would like to thank Susan Lagerstrom-Fife and Sharon Palleschi at Springer USA for their assistance in publishing this manuscript in a timely fashion.

Kang Zhang

Department of Computer Science
The University of Texas at Dallas
Richardson, Texas, U.S.A.

Foreword

Visual computing is computing on visual objects. Some visual objects such as images are **inherently visual** in the sense that their primary representation is the visual representation. Some visual objects such as data structures are **derivatively visual** in the sense that their primary representation is not the visual representation, but can be transformed into a visual representation. Images and data structures are the two extremes. Other visual objects such as maps may fall somewhere in between the two. Visual computing often involves the transformation from one type of visual objects into another type of visual objects, or into the same type of visual objects, to accomplish certain objectives such as information reduction, object recognition and so on.

In visual computing it is important to ask the following question: who performs the visual computing? The answer to this question determines the approach to visual computing. For instance it is possible that primarily the computer performs the visual computing and the human merely observes the results. It is also possible that primarily the human performs the visual computing and the computer plays a supporting role. Often the human and the computer are both involved as equal partners in visual computing and there are visual interactions. Formal or informal **visual languages** are usually needed to facilitate such visual interactions. With the advances in biocomputing it is conceivable that visual computing may involve animals, robots, cyborgs and other hybrid life forms so that visual languages can be either natural or artificial. It is clear that visual languages are both vehicles for communication and also tools for programming.

A visual language is a pictorial representation of conceptual entities and operations and is essentially a tool through which users compose visual sentences. Compilers for visual languages must interpret visual sentences and translate them into a form that leads to the execution of the intended task. This process is not straightforward. The compiler cannot determine the meaning of the visual sentence simply by looking at the visual objects. It must also consider the context of the sentence, how the objects relate to one another. Keeping the user intent and the machine's interpretation the same is one of the most important tasks of a visual language.

Diagrammatical visual programming languages are important because they are based upon relational graphs and capable of specifying complex relationships precisely. Kang's book provides a comprehensive introduction to diagrammatical visual programming languages and the technology of automatic generation of such languages. It covers a broad range of contents from the underlying theory of graph grammars to the applications in various domains. As Kang points out himself, this book is not about how to program visually. Rather, it is about what are visual programming languages, and how such languages and their underlying foundations can be usefully applied to other fields in computer science that need graphs as the primary means of representation. The book gives a comprehensive treatment of graph grammars and their various applications. The Reserved Graph Grammar (RGG) formalism is extended to Spatial Graph Grammar (SGG), while both being relational grammars, by integrating the spatial and structural specification mechanisms in a single framework. What is unique about this book is the extensive discussion of several important application areas of visual languages and graph grammars, including multimedia authoring and presentation, data interoperation, software engineering and web design. Thus the book can be used in a graduate course on visual programming languages and applications.

I have known Kang for many years, but I only found out he is also an artist when he asked me to write the foreword for his book and disclosed to me that the cover design is from one of his paintings. Small wonder Kang is so persistent in his research on visual languages! His book is an important contribution to the growing collection of textbooks and monographs on visual languages.

Shi-Kuo Chang

Department of Computer Science
The University of Pittsburgh
Pittsburgh, USA

Chapter 1 Introduction

1.1 Visual Languages and Programming

Visual communications have existed as long as the history of mankind. People communicated through symbols and drawings long before spoken languages have been developed. Carrying specific meanings, or semantics, those symbols and drawings may be considered *visual languages*[1], that serve effective communication purposes. In a broader sense, visual languages refer to any kinds non-textual but visible human communication medias, including art, images, sign languages, maps, and charts, to name a few.

Since the invention of digital computers, researchers have been seeking intuitive and effective communication means between human and computers (Sutherland 1963). The most important communication is for human to instruct computers what to do and how to do to complete intended tasks in the form of "programs".

"Programs describe the computational steps carried out by computer devices, both physical and conceptual, and are the basis for automatic control of a wide variety of machines" (Graham 1987). Programming refers to the activities of constructing programs. Programming languages are the means by which different types of computations are expressed in the programs.

Visual programs are a type of programs that describe computational steps in two or more dimensional fashion. This is in contrast to the conventional programs that are expressed textually and considered one dimensional. By the above definition, a visual program could be a diagram or any kind of meaningful, possibly high dimensional, structures.

Visual programming refers to a process in which the user specifies programs in a two or more dimensional fashion (Burnett 1999). Visual

[1] *Languages* are symbol systems, such as the languages of art (Goodman 1968).

programming aims at effectively improving the programming productivity by applying visual technologies to support program construction.

Visual programming languages (VPLs) are the languages that support visual programming, or the visual languages that support programming. This implies that VPLs are a special class of visual languages used for computer programming. However, the remaining chapters of this book will simply refer to visual languages, rather than explicitly VPLs, if the programming context is clear. In many of the application contexts, the term "visual language" is more appropriate than the term "visual programming language".

An important class of visual programming languages is the diagrammatic one, which is based on object-relationship abstractions (e.g. using nodes and edges). Frequently used diagrammatic visual languages include Entity-Relationship database design languages, data-flow programming languages (e.g. Petri nets), control flow programming languages, state transition specifications, and so on. Other classes of VPLs include form-based languages, notably spreadsheet style of languages (Burnett and Gottfried 1998), and iconic languages based on iconic theory (Chang et al. 1987). This book covers a specific family of diagrammatic visual languages and their applications.

Rapid advances of the display and interaction technologies have made visual programming an effective and attractive communication means. VPLs have been successfully used in several application areas: teaching children and adults such as KidSim and later Cocoa (Smith et al. 1994), programming assistance such as the dataflow programming language Prograph (Cox and Pietrzykowski 1985), development of user-interfaces such as WIPPOG (Bottoni and Levialdi 2005), sketch recognition (Costagliola et al. 2006; Plimmer et al. 2006), etc. VPLs have also been widely used in the design and analysis of software systems. Well-known examples of software modeling and specification languages include UML - the Unified Modeling Languages (Booch et al. 1999), automata, Petri nets, etc. In fact, visual modeling is becoming an increasingly important area of research in visual languages. Since the late 1990s, there has been a dramatic increase in the literature in visual modeling languages. For pointers to a comprehensive literature coverage of the applications of visual languages, the reader may refer to Section 1.4.

1.2 Visual Programming vs. Program Visualization

To explain the visual programming process supported by a VPL, we use a high level conceptual model to illustrate the roles of the user, user interface, and visual program. This model also clearly defines the differences between visual programming and program visualization, which play complementary roles in software development (see Preface in Zhang 2003).

We adapt the model of van Wijk (2006) that was used to identify the major ingredients, costs and gains in the field of visualization. The adapted model, illustrated in Fig. 1.1, considers the major ingredients of the *user*, *user-interface*, and *program*, rather than those of the *user*, *visualization*, and *data* in the context of visualization.

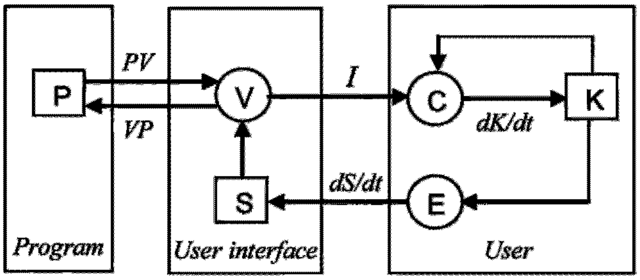


Fig. 1.1. A conceptual model of visual programming and program visualization

The boxes in Fig. 1.1 denote containers, and circles denote processes that transform inputs into outputs. The user is modeled with his/her knowledge *K* about the program to be constructed, or to be understood and analyzed. The knowledge *K* is obtained through the user's cognitive capability *C*, particularly the perceptual ability in the context of visual programming and program visualization. The knowledge also enhances the cognitive capability and plays the key role in driving the interactive exploration *E* through the user interface.

Through the user interface, the user provides specifications *S* for the program to be developed or the algorithms and their parameters to be applied. Upon the specification supplied, a visual program *V* is displayed and edited as an image *I*. In the context of program visualization, *V* represents the visualization of a program's properties, such as status, structure, interaction among its components, or output results. Its image *I* is perceived by the user, with an increase in knowledge *K* as a result.

Program *P* is what the user is interested. It is to be developed in the case of visual programming, or comprehended and analyzed in the case of program visualization. The program in this conceptual model has a broader sense than the traditionally understood program as defined by Graham (1987). *P* can be any of the following

- A code sequence conforming to a traditional programming language such as Java,
- A code sequence conforming to a mark-up language such as XML which may not necessarily carry any computation, or
- A binary code or data structure generated (e.g. by a parser) from a high-level specification.

As defined in Section 1.1, visual programming (VP) refers to a process in which the user specifies programs in a two or more dimensional fashion, i.e. in the direction of *V* to *P*. Program visualization (PV), on the other hand, refers to a process in which certain properties of a program are displayed in a two or more dimensional fashion according to the user's selection of parameters and/or algorithms. The process of PV is clearly in the direction of *P* to *V*. Usually, a VPL or a VP system aims at easing the process of program specification *S* through graphical interaction and direct manipulation with a minimal requirement of the programming knowledge. The easiness of the specification *S* for a given program *P* is measured by the amount of time *T* required, represented by *dS/dt*. While a PV system aims at maximizing the user's gain in his/her knowledge *K* about the program *P* under analysis. The measurement of PV's effectiveness is made when the user takes time *T* to gain additional knowledge *K(T)-K(0)* about the program *P*, represented by *dK/dt*.

An ideal visual software engineering system should support round-trip visual engineering by incorporating both visual programming and program visualization. Consistent graphical formalisms in both VP and PV are desirable in order to maintain the user's mental map (Misue et al. 1995) throughout the life-cycle of the development. Program visualization, however, is a topic beyond the scope of this book, and is usually within the scope of software visualization (Eades and Zhang 1996; Stasko et al. 1998; Zhang 2003).

1.3 Organization of the Book

This book is not about how to program visually. Rather, it is about what are visual programming languages, and how such languages and their underlying foundations can be usefully applied to other fields in computer science that need graphs as the primary means of representation. The remaining of the book is organized as the following.

Graph grammars may be used as a natural and powerful syntax-definition formalism for visual programming languages. Chapter 2 (extended and updated on Zhang et al. 2001a) presents a context-sensitive graph grammar formalism called *reserved graph grammar* (RGG), which can explicitly and completely describe the syntax of a wide range of diagrams using labeled graphs. The parsing algorithm of a reserved graph grammar uses a marking mechanism to avoid ambiguity during parsing and has polynomial time complexity in most cases.

Chapter 3 (Kong and Zhang 2004a; Kong and Zhang 2004b) presents an extension of the RGG formalism, called *Spatial Graph Grammar* (SGG), by integrating both the spatial and structural specification mechanisms in a single framework. In addition to nodes and edges, this formalism treats spatial constraints as a type of language constructs in the abstract syntax. With the extended expressive power, semantic and structural requirements can be intuitively specified through spatial notations.

The next four chapters present some typical applications of visual languages and graph grammars.

The first application is multimedia authoring and presentation. On-line multimedia presentations, such as news, need to be constantly updated. There are increasing demands for accessing on-line multimedia documents from mobile devices such as PDAs. A sound but practical formalism is needed to support automatic adaptation to the change of media contents, display environments, and the user's intention. Chapter 4 (Zhang et al. 2005a) presents a visual language approach to the layout adaptation of multimedia objects, based on the RGG. The chapter focuses on the issues and techniques for size adaptation and style adaptation in response to the change of device requirements and user's interactions.

The next application is data interoperation. As an increasing amount of scientific and societal data is accessible as in digital forms and possibly represented in various styles of XML-based languages, there is a need for seamless and user-friendly tools that could reuse and integrate the heterogeneous digital artifacts. Aiming at providing user-friendly means for exchange of digital artifacts, Chapter 5 (Zhang et al. 2001c; Song et al. 2004a; Song et al. 2004b; Zhang et al. 2005b) presents a language generation mechanism that allows graphical data-encoding languages and schemas to be specified and automatically generated. The generated language environments can automatically verify the syntactical structure of any constructed digital artifacts and, when translation specifications are provided, automatically translate a source artifact expressed in one encoding language or schema to its equivalent in another language or schema.

The third application is software engineering. Software architecture and design are usually modeled and represented by informal diagrams, such as architecture diagrams and UML diagrams. While these graphical notations are easy to understand and convenient to use, they are not amendable to automated verification and transformation. Chapter 6 (Kong et al. 2003; Kong et al. 2005) applies graph grammars to the specification of software architectures through UML class diagrams and design patterns. These grammars enable a high level of abstraction for the general organization of a class of software architectures, and form a basis for various analysis and transformations. In this approach, software verification is performed through a syntax analyzer. Architecture transformation is achieved by applying predefined transformation rules.

The last application is Web design. The design of Web sites has been largely ad hoc, with little concern about the effectiveness of navigation and maintenance. Chapter 7 (Zhang et al. 2002) presents a general framework with a human-Web interface that supports Web design through visual programming and reverse Web engineering through visualization. The chapter describes the framework in the context of a Web tool, known as HWIT, which has been developed for a pilot study.

Chapter 8 (Zhang et al. 2001b) presents the design, construction, and application of a generic visual language generation environment, called VisPro. The VisPro design model improves the conventional Model-View-Controller framework in that its functional modules are decoupled to allow independent development and integration. The VisPro environment consists of a set of visual programming tools. Using VisPro, the process of VPL construction can be divided into two steps: lexicon definition and grammar specification. The former step defines visual objects and a visual editor, and the latter step provides language grammars with graph rewriting rules. The compiler for the VPL is automatically created according to the grammar specification. A target VPL is generated as a programming environment which contains the compiler and the visual editor. The chapter demonstrates how to use VisPro by building a simple visual language and a more complex visual modeling language for distributed programming.

Finally, Chapter 9 summarizes the challenges faced by visual language researchers and provides a future perspective in addressing the challenging issues and in other application potentials.

1.4 General Readings on Visual Languages

Representative pioneering work in visual programming and visual languages includes Chang (1971), Smith (1975), and Pong and Ng (1983). The milestone work of Sutherland (1963) sets a lasting foundation for graphical interactions.

There have been several surveys and reviews, mostly in early days of visual language research, including Shu's dimensional perspective (1986), Myers' taxonomy on visual programming, programming by examples, and program visualization (1990), and Chang's tutorial and survey (1987). More recently, Marriott and Meyer (1997) proposed a classification of visual languages based on grammar hierarchies, and Bottoni and Grau (2004) presented a family of meta-models, expressed as UML diagrams, for classifying visual languages.

Readers may find collections of papers in visual language research in early years in Chang at al. (1986), representative papers in visual programming environments in Glinert (1990a; 1990b), object-oriented visual programming in Burnett et al. (1995), and visual language theory in Marriott and Meyer (1998). Burnett (2006) maintains a Web page (http://web.engr.oregonstate.edu/~burnett/vpl.html), which is perhaps the most comprehensive and updated collection of references.

There had been an annual conference, the *IEEE Symposium on Visual Languages* (VL), that was started as a workshop in 1985 in Hiroshima University, Japan, and then became the symposium until 2002. Due to the increasing role of human aspects and cognitive science, the conference title was changed to the *IEEE Symposia on Human-Centric Computing – Languages and Environments* in 2003, organized as three separate symposia. The title has now been changed to the *IEEE Symposium on Visual Languages and Human-Centric Computing* (VL/HCC) since 2004, having realized the loss of the *visual language* identity. The most recent symposia in this series can be found at the following URLs:

- VL/HCC'07, Coeur d'Alene, USA: http://vlhcc07.eecs.wsu.edu/
- VL/HCC'06, Brighton, UK: http://www.cmis.brighton.ac.uk/vlhcc/
- VL/HCC'05, Dallas, USA: http://viscomp.utdallas.edu/vlhcc05/
- VL/HCC'04, Rome, Italy: http://vlhcc04.dsi.uniroma1.it/index.php
- HCC'03, Auckland, New Zealand: http://projects.cs.dal.ca/HCC03/

A less formal and smaller annual forum is the *Visual Languages and Computing Workshop* (VLC), started in 2002. VLC has always been a satellite workshop at the *International Conference on Distributed Multimedia.*

The *Journal of Visual Languages and Computing*, Elsevier (previously published by Academic Press), is the premier archived journal dedicated to the publication of research in visual languages and related topics in visual computing.

Chapter 2 The Foundation - Graph Grammars

2.1 Introduction

In the implementation of textual languages, formal grammars are commonly used to facilitate the language understanding and the parser creation. When implementing a diagrammatic visual programming language (in the rest of the chapter, diagrammatic visual programming languages will simply be referred to as visual languages), this is not usually the case. A visual language requires a formal syntactic definition, which is indispensable for automatic analysis, transformation, and non-ambiguous expression. Graph grammars with their well-established theoretical background may be used as a natural and powerful syntax-definition formalism (Rozenberg 1997) and the parsing algorithm based on a graph grammar may be used to check the syntactical correctness and to interpret the language semantics.

One obstacle for the application of graph grammars is that even for the most restricted classes of graph grammars the membership problem is NP-hard (Rozenberg and Welzl 1986). Consequently, most of the existing graph grammar parsing algorithms are either unable to recognize interesting languages of graphs or tend to be inefficient when applied to graphs with a large number of nodes and edges.

Another problem is that nearly all known graph grammar parsing algorithms (Rozenberg and Welzl 1986; Bunke and Haller 1989; Golin 1991; Kaul 1982; Wills 1992; Wittenburg 1992) deal only with context-free productions. A context-free grammar requires that only a single non-terminal is allowed on the left-hand side of a production (Wittenburg and Weitzman 1996). A context-sensitive graph grammar, on the other hand, allows left-hand and right-hand graphs of a production to have arbitrary number of nodes and edges. Most existing graph grammar formalisms for visual languages are context-free. Yet not many visual languages can be specified by purely context-free productions. Additional features are required for context-free graph grammars to handle context-sensitivity. It is therefore difficult for context-free grammars to specify many types of visual languages.

Rekers and Schürr (1997) proposed *layered graph grammars* (LGGs) for specifying visual languages. LGGs differ from most other grammars in two aspects: context-sensitivity and graph formalism. Being context-sensitive makes the graph grammars expressive. The graph formalism in LGGs is intuitive and thus easier to understand and to use than textual formalisms for specifying visual languages. However, although being expressive, the layered graph grammar is inefficient in its implementation. Its parsing algorithm is complicated and the parsing complexity generally reaches exponential time.

This chapter presents a context-sensitive graph grammar called *reserved graph grammar* (RGG) (Zhang 1997; Zhang and Zhang 1997), which was motivated by the development of a general-purpose visual language generator (see Chapter 8). Because the targets of the generator are visual languages, their grammars are better specified using a graph formalism. As a part of the generator, a visual editor should be used to create visual programs based on the grammar specifications and parsing algorithms should be automatically created according to the grammar.

The RGG is developed based on the layered graph grammar by using the layered formalism to allow the parsing algorithm to determine in finite steps whether a graph is valid. It uses labeled graphs to support the linking of newly created graphs into a parsed graph (traditionally called embedding). The node structure enhanced with additional visual notations in the RGG simplifies the transformation specification and also increases the expressiveness.

An RGG is complete and explicit in describing the syntax of a wide range of diagrams. Compared to the LGG where the context-graph (Rekers and Schürr 1997) must explicitly appear in the production, the embedding mechanism in the RGG allows the grammar representation to avoid most of the context-specifications while being more expressive. This greatly reduces the expression complexity, and in turn increases the efficiency of the parsing algorithm.

A general RGG parsing algorithm, however, has the exponential time complexity. This is solved by introducing a constraint into the RGG. It is not yet clear how this constraint limits the application scope, but we find that even the grammar of a complicated control flow diagram satisfies the constraint. With this constraint, a parsing algorithm of polynomial time complexity can be developed. An algorithm for checking whether an RGG satisfies the constraint is also developed.

The RGG formalism has been used in the implementation of a toolset called VisPro, which facilitates the generation of visual languages using the lex/yacc approach (Chapter 8; Zhang 1997; Zhang and Zhang 1998b; Zhang et al. 2001a).

The rest of the chapter is organized as follows: Section 2.2 describes a case study that demonstrates the basic idea of the RGG. Section 2.3 provides a formal definition of the RGG formalism. Section 2.4 defines a selection-free condition which allows an RGG to be parsed in polynomial time. Section 2.5 compares the RGG formalism with its predecessor, the LGG, followed by the chapter summary in Section 2.6.

2.2 A Case Study

2.2.1 Process Flow Diagrams

We use a process flow diagram (PFD) as an example to illustrate how an RGG works. A process flow diagram has two types of constructs: structured and non-structured. For example, a *fork-join* construct provides a structure in a diagram, while a *send-receive* construct does not affect the structure of a diagram. Many diagrams used in computer science have such a mixture of constructs, which are difficult to specify using existing graph grammars except the layered graph grammar (Rekers and Schürr 1997).

In the PFD shown in Fig. 2.1, the *fork* statement splits one thread into multiple threads (three in the example). There are two *send* statements that send different messages to the same *receive* statement. Assuming syntactically, a *receive* statement can receive information from any number of send statements, while a *send* statement can send to only one *receive*. A *fork* statement can split one thread into any number of threads.

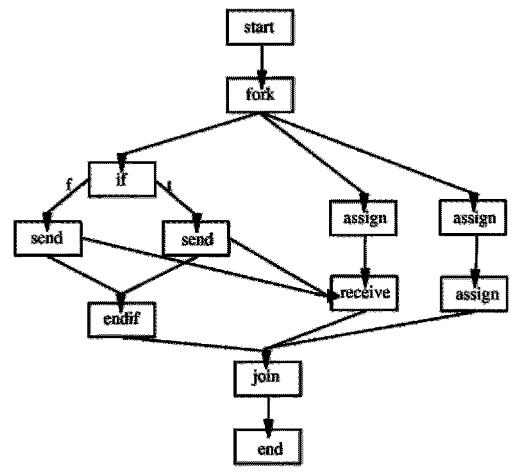
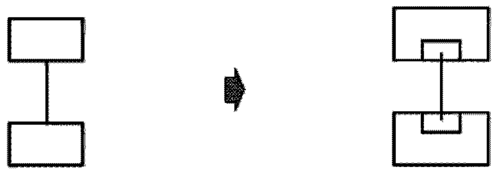


Fig. 2.1. A process flow diagram

We first translate the diagram in Fig. 2.1 into a graphical form whose syntax is suitable for the RGG interpretation. We will call such a graphical form a *node-edge diagram*. The translation is very straightforward as shown in Fig. 2.2, ignoring all the arrows since the direction is unimportant in our graph grammar representation. A node in the node-edge representation is a two-level structure. Fig. 2.3 depicts an example node called *join*. The first level is the large surrounding rectangle, which is called a *super vertex*. The small rectangles embedded in a super vertex are the second level called *vertices*. A vertex or super vertex can be connected to one or more edges. An edge is uniquely determined by two vertices in the involved nodes. RGG does not impose semantic difference between connecting to a vertex and connecting to a super vertex. The translated node-edge representation of the process flow diagram is shown in Fig. 2.4.



(a) normal representation (b) node–edge representation
(a) usual representation (b) node-edge representation

Fig. 2.2. From a diagram to a node-edge representation

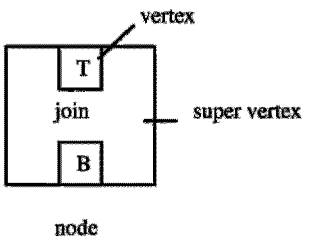


Fig. 2.3. Node structure

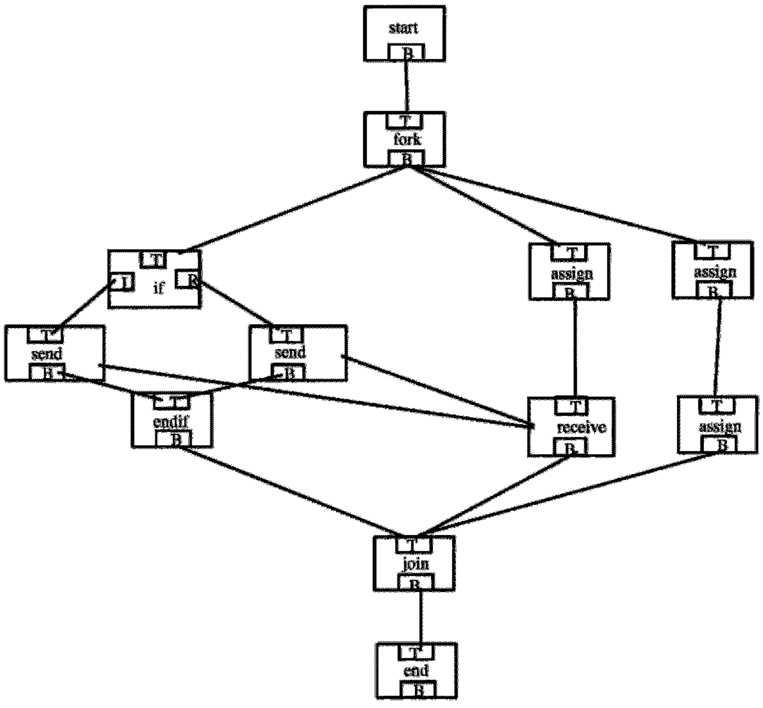


Fig. 2.4. The node-edge form of the process flow diagram

In a node-edge diagram, all vertices should be labeled. For simplicity, we use T (top), B (bottom), L (left), R (right) to label the vertices according to their positions in a node. Vertex labels uniquely identify the vertices in each node.

2.2.2 Graph Rewriting Rules

A graph rewriting rule, also called a *production*, has two graphs which are called *left graph* and *right graph*. It can be applied to an application graph (called *host graph*) in the form of an *L-application* or *R-application*. A production's L-application to a host graph is to find in the host graph a *redex* of the left graph of the production and replace the redex with the right graph of the production. An R-application is a reverse replacement (i.e. from the right graph to the left graph). A *redex* is a sub-graph in the host graph which is isomorphic to the right graph in an R-application or to the left graph in an L-application.

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. However, with a visual language that has two-dimensional relationships among the language elements, a far more complicated mechanism is needed to establish relationships between the substitute of a redex and its adjacent elements.

There are three approaches to embedding a graph into a host graph (Rekers and Schürr 1997):

- *Implicit embedding*: formalisms such as picture layout grammars (Golin 1991) and constraint multiset grammars (Chok and Marriott 1995) do not distinguish between vertices and edges. Relationships are implicitly defined as constraints over their attribute values. Attribute assignments within productions have the implicit side effect that creates new relationships to unknown context elements. Users are, therefore, not always aware of the consequences of attribute assignments, and parsers require considerable time to extract, from attributes and constraints, implicitly defined knowledge about the relationships.

- *Embedding rules*: some graph grammars such as the NLC graph grammar (Rozenberg and Welzl 1986) and the DNECL graph grammar (Brandenburg 1988) have separate embedding rules which allow the redirection of arbitrary sets of relationships from a redex to its substitute. This approach is easy to implement. However, the embedding rules are often difficult to understand and all known parsing algorithms for productions with embedding rules are either inefficient or imposing very strict restrictions on the left- and right-hand sides of the productions. Furthermore, embedding rules are only able to redirect or re-label existing relationships. They cannot be used to define such productions as the one in Fig. 2.5, which establishes new relations between previously unconnected vertices.

- *Context elements*: context elements can be used to establish the relationships between a newly created graph and the host graph. This approach is the easiest to understand, but an unrestricted use of context elements may complicate the graph rewriting rules. Furthermore, it is difficult to rewrite elements which may participate in a statically unknown number of relationships.

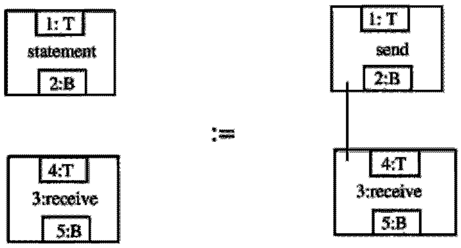


Fig. 2.5. A graph rewriting rule

The reserved graph grammar combines the approaches of the embedding rule and the context elements to solve the embedding problem. By introducing context information, simple embedding rules can be sufficiently expressive to handle complicated programs. Moreover, the wildcards formalism used in the LGG is not needed in the RGG. The following paragraphs explain our new embedding approach by showing its application in the graph transformation process. In order to identify any graph elements which should be reserved during the transformation process, we mark each isomorphic vertex in a production graph by prefixing its label with a unique integer. The purpose of marking a vertex is to preserve the context.

We impose an embedding rule which states that if a vertex in the right graph of the production is unmarked and has an isomorphic vertex *v* in the redex of the host graph, then all edges connected to *v* should be completely inside the redex. With the above embedding rule which is usually called the *dangling condition* (Rozenberg 1997), each application of a production can ensure that a graph can be embedded in a host graph without creating dangling edges. The examples in Fig. 2.6 illustrate the R-application process, where some host graphs have isomorphic graphs (enclosed in dashed boxes) of the right graph of the production in Fig. 2.5. In Fig. 2.6(a)(1), the isomorphic graph is a redex. The vertices corresponding to the isomorphic vertices marked in the right graph of the production are painted gray. The transformation deletes the redex while keeping the gray vertices, as shown in Fig. 2.6(a)(2). Then the left graph of the production is embedded into the host graph, as shown in Fig. 2.6(a)(3), while treating a marked vertex in the left graph the same as a gray vertex that has the same mark. We can see that the marking mechanism allows some edges of a vertex to be reserved after transformation. For example, in Fig. 2.6(a), two edges from B to T are reserved after transformation. Note that Fig. 2.6(a)(2) serves only as an illustration of "reserving", and is not the result of a transformation.

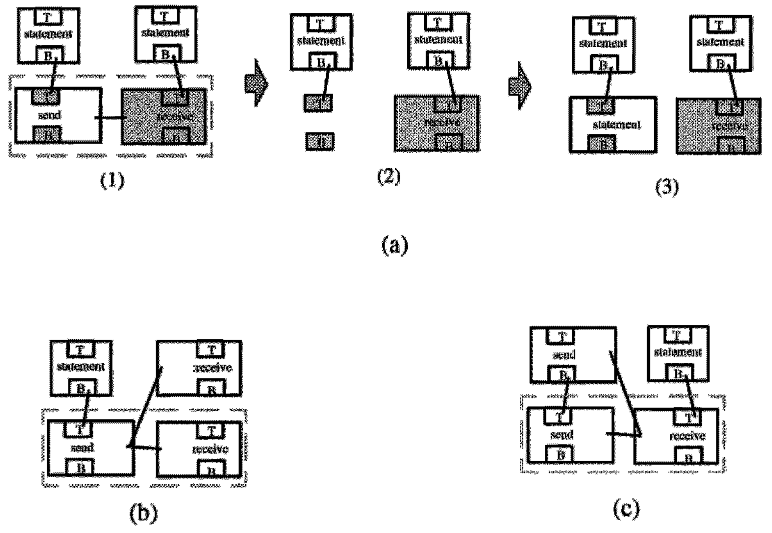


Fig. 2.6. Examples of the R-application

In the above notion of process flow diagrams, a *send* node is allowed to connect to only one *receive* node. We show how such a restriction can be expressed and maintained in the RGG. The solution is simple: we leave the *send* node unmarked in the production. According to the embedding rule, the isomorphic graph in Fig. 2.6(b) is not a redex because the super vertex in the *send* node has an edge that is not inside the isomorphic graph while its isomorphic super vertex in the right graph is unmarked. Therefore, the graph in Fig. 2.6(b) is invalid. On the other hand, we allow a *receive* node to receive data from one or more *send* nodes. To support this, we mark the super vertex of the *receive* node in the production in Fig. 2.5. The graph in Fig. 2.6(c) is valid according to the embedding rule. There is a redex (in the dotted box) in the graph, because the super vertex of *receive* has its isomorphic vertex marked in the right graph of the production, even though it though it has an edge connected outside the isomorphic graph. Therefore, the marking mechanism helps not only in embedding a graph correctly, but also in simplifying the grammar definition.

2.2.3 A Graph Grammar for Process Flow Diagrams

The graph grammar shown in Fig. 2.7 explicitly and precisely depicts the syntax of the PFD language. It consists of a set of productions, and the label <i> identifies Production i.

The L-application defines the language of a grammar. The language is defined by all possible graphs which have only terminal labels and can be derived using L-applications from an initial graph (i.e. λ). The R-application is used to parse a graph. If the graph is eventually transformed to an initial graph after a series of R-applications, the graph is proven to belong to the language. In the sequel, we prove that the R-application can precisely determine the language defined by the L-application for an RGG.

By applying the R-application of the RGG in Fig. 2.7 repeatedly to a specific diagram (i.e. a host graph), we can determine whether the diagram is a process flow diagram. The process of parsing the PFD drawn in Fig. 2.1 is illustrated in Fig. 2.8, where a label in an oval describes a possible R-application order (represented by an alphabetic letter, e.g. c is after a) and the corresponding production (by a numeric figure). The notation d:2 means that the redex of Production 2 is applied after the R-applications a, b, and c have been applied. The R-applications may be applied in different orders but will produce the same result.

In Fig. 8(a), the five sub-graphs in the dotted boxes are possible redexes, which can be applied with Productions <6>, <6>, <2>, <2>, and <2> to produce the graph in Fig. 8(b). Similarly, the graph in Fig. 8(b) can be transformed into the graph in Fig. 8(c), and so on. Finally, the graph is transformed into an initial graph. The original diagram is, therefore, a valid process flow diagram.

The following section presents a formal definition of the reserved graph grammar.

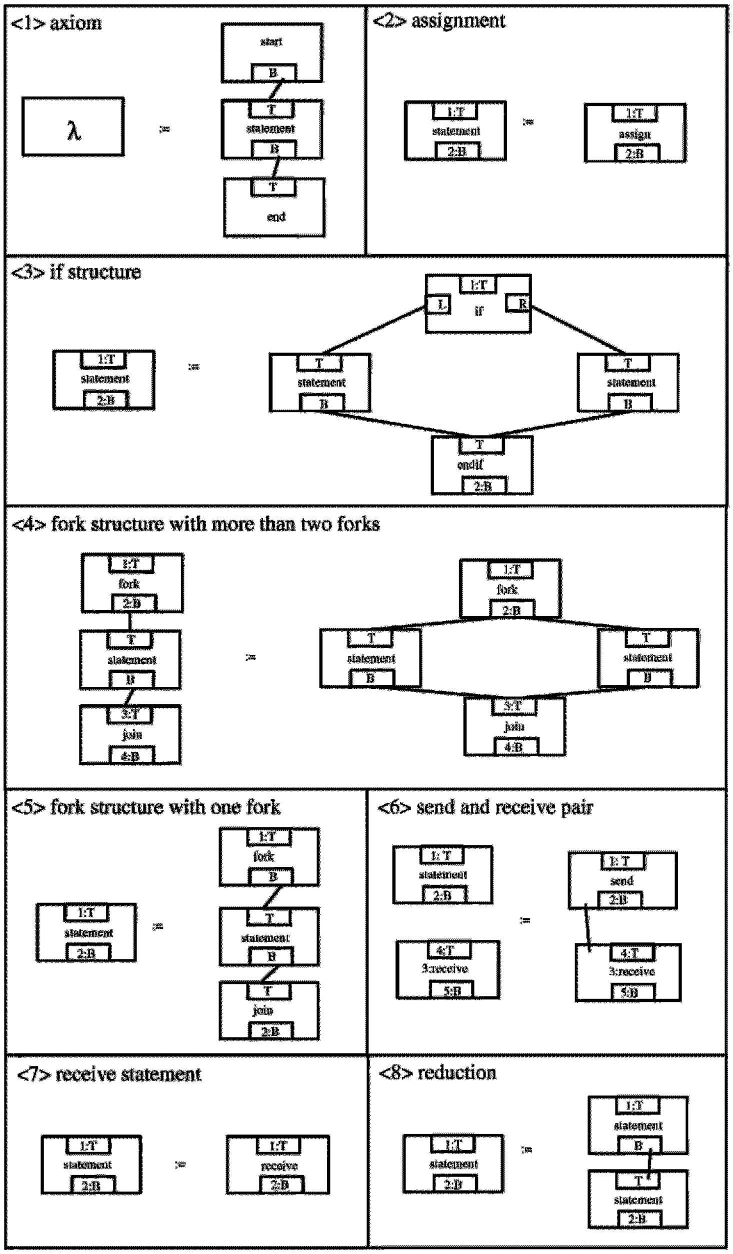


Fig. 2.7. A reserved graph grammar specifying process flow diagrams

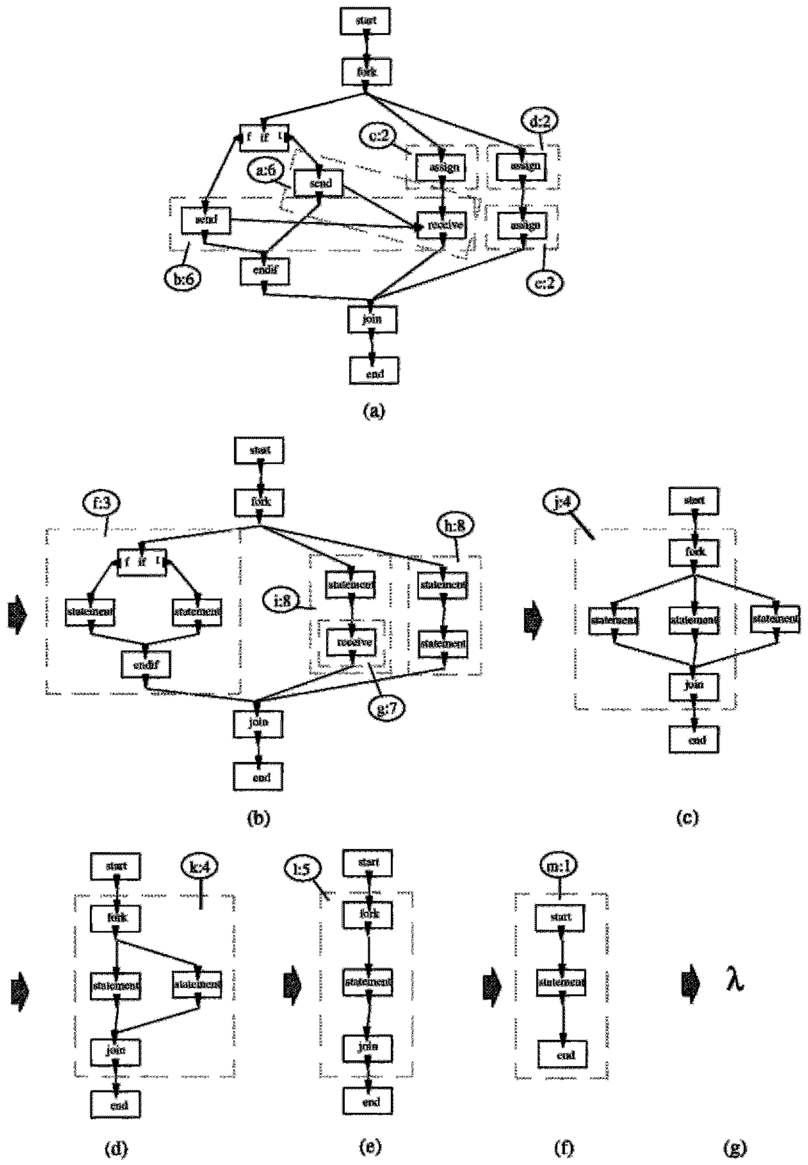


Fig. 2.8. Graph transformations (parsing) when productions are applied

2.3 Formal Definition

2.3.1 Preliminaries

In order to define the reserved graph grammar and its properties, we will first introduce some basic concepts, such as graph element, graph, and isomorphism. We then define the marking mechanism, which allows us to further define a redex and graph transformations including L- and R- applications.

Definition 2.1 $n:=(s, V, l)$ is a node on a label set L, where

- V is a set of vertices,
- $s \in V$ is a super vertex, and
- $l: V \rightarrow L$ is an injective mapping from V to L.

A super vertex contains a set of vertices, and itself is a vertex. A label serves as a type in an RGG. For simplicity, we will use the notations $n.V$ and $n.s$ to represent the corresponding parts of a node n; and this convention is applicable to other definitions.

Definition 2.2 Two nodes n1 and n2 are isomorphic, denoted as $n_1 \approx n_2$, iff

- they are defined over the same label set, and
- $\exists f$ ((f: $n_1.V \rightarrow n_2.V$ is a bijective mapping) $\wedge$ $\forall v \in n_1.V$ $(n_1.l(v)=n_2.l(f(v)))$ $\wedge$ $n_2.s=f(n_1.s)$).

The definition specifies that two nodes are isomorphic if they have the same types of vertices (including super vertices).

Definition 2.3 $G:=(N, E)$ is a graph over a label set L, where

- N is a finite set of nodes over L,
- $E \subseteq N.V \times N.V$, where $N.V= \bigcup_{n \in N} n.V$, is a finite set of edges.

Each edge connects from a vertex of a node to a vertex of another node and is defined by that pair of vertices.

Not all graphs are meaningful. Only certain types of graphs represent meaningful visual sentences. A graph grammar can be used to define those graphs that are valid visual sentences. To specify the graph grammar we need to define the following concepts.

Definition 2.4 A vertex v is said to be *marked*, denoted as mark(v)=m, if it is assigned an integer m called *mark*.

Definition 2.5 G:=(N, E, M) is a marked graph over a label set L, where

- (N, E) is a graph over L, and
- M: $V \to I$ is a bijective mapping, where $V \subseteq N.V$, and I is a set of integers.

A marked graph has unique integers in some of its vertices. Different vertices in a marked graph should have different marks. We use mark(v)=m to indicate that v is assigned an integer m, and mark(v)=null to indicate that v is assigned nothing and said to be unmarked.

Definition 2.6 Two vertices a and b in two different graphs are equivalent, denoted as $a \doteq b$, iff mark(a)=mark(b) and mark(a)≠null.

Definition 2.7 Two graphs G_1 and G_2 are isomorphic, denoted as $G_1 \approx G_2$, iff $\exists f: G_1 \to G_2$ is a bijective mapping such that

- $\forall n \in G_1.N$: $n \approx f(n)$; and
- $\forall e=(v_a, v_b) \in G_1.E$: $f(e)=(f(v_a), f(v_b)) \in G_2.E$.

To apply a production to a graph (called a *host graph*), we need to find a sub-graph in the host graph that matches the right graph (or left graph) of the production. Such a matching sub-graph in the host graph is called a *redex*.

Definition 2.8 A sub-graph X of a graph H is called a *redex* of a marked graph G, denoted as $X \in \text{Redex}(H,G)$, iff $\exists f: G \to X$ is a bijective mapping and under the mapping:

- $X \approx G$; and
- $\forall v \in G.V\ ((\text{mark}(v)=\text{null}) \wedge \forall\ v_1 \in H\ ((e=(f(v),v_1) \in H \vee e=(v_1, f(v)) \in H) \to e \in X)$.

This definition specifies that a sub-graph X of a graph H can be a redex of a marked graph, G, if and only if X is isomorphic to G and every vertex in X that is isomorphic to an unmarked vertex in G should have edges completely inside X. The definition of a redex eliminates the possibility of any dangling edges resulted from a transformation.

A redex is always related to a mapping function and we will not specify the mapping function if this is clear in the context.

Definition 2.9 A *production* p:=(L, R) is a pair of marked graphs over the same label set, where L:=(N_L, E_L, M) and R:=(N_R, E_R, M).

A pair of marked graphs in a production has the same mark set. They are called *left graph* and *right graph* respectively.

When a production is applied to a graph, the graph is said to be *transformed* by the application.

Definition 2.10 Let X be a redex of G in H determined by a bijective mapping $f:G \rightarrow X$, If G and G' are the left and right graphs in a production, then the *transformation* of H to H' after replacing X in H by G' is defined as follows:

1. add G' to H,
2. $\forall v' \in G'.V$ if $\exists v \in G.V$ such that $v \doteq v'$, replace v' with f(v) (called a *reserved node*), then delete v', and
3. delete X from H except the reserved nodes.

The result of H with above operation is H', denoted as H'=Tr(H, G, G', X).The second step ensures that the edges connecting the vertices which are isomorphic to the marked vertices in G are reserved.

Based on the above definition of transformation, the L-application and R-application can be defined as follows.

Definition 2.11 An *L-application* of a production p:=(L, R) to a graph H is a transformation H'=Tr(H, L, R, X), where $X \in Redex(H, L)$, denoted as $H \mapsto^{X} H'$.

Definition 2.12 An *R-application* of a production p:=(L, R) to a host graph H is a transformation H'=Tr(H, R, L, X), where $X \in Redex(H, R)$, denoted as $H \rightarrow^{X} H'$.

3.2 Reserved Graph Grammar and Its Properties

We now define the reserved graph grammar and some of its properties.

Definition 2.13 A *reserved graph grammar* gg is a tuple (A, P, T, N), where A is an initial graph, P a set of graph grammar productions, T a set of terminal labels with $e_l \in T$ (we define all edges to have the same label e_l), and N a set of non-terminal labels. For $\forall p:=(L,R) \in P$ and $\forall l \in T \cup N$:

1. R is non-empty;
2. L and R are over the same label set $T \cup N$;
3. $l \in L_i$ where $L_i \subset \{L_0, ..., L_n\}$ is a global layer set and $L_0 \cap ... \cap L_n = \emptyset$; and
4. L<R with respect to the following order of graphs:

$G<G' \Leftrightarrow \exists i:|G|_i < |G'|_i \wedge \forall j<i: |G|_j=|G'|_j$ with $|G|_k$ defined as $|\{ x \mid x \in G \wedge layer(x)=k\}|$.

The last condition guarantees that a diagram can be parsed in finite steps with the grammar (Rekers and Schürr 1997).

For simplicity, given an RGG gg:=(A, P, T, N), we use the notation $X \in Redex(H)$ to denote $\exists p:=(L,R) \in P \wedge \exists X:(X \in Redex(H, R) \vee X \in Redex(H, L))$, when this is clear in the context.

We denote the sequence of intermediate derivations $H \mapsto^{X1} H_1$, $H_1 \mapsto^{X2} H_2$, ..., $H_{n-1} \mapsto^{Xn} H_n$ as $H \mapsto^{X1} H_1 \mapsto^{X2} \ldots \mapsto^{Xn} H_n$; or simply $H \mapsto^{X1 \ldots Xn} H_n$. We use $H \mapsto^{*} H_n$ to denote $H \mapsto^{X1 \ldots Xn} H_n$, where n may be 0 in which case $H=H_n$ and $H \mapsto H$. This notation is also applicable to the R-application $\to$.

Definition 2.14 Let gg:=(A, P, T, N) be an RGG, its language L is defined by L(gg)={G| A $\mapsto^{*}$ G, where G contains only elements with terminal labels}.

We now prove that the R-application can determine whether a diagram is a language defined by a reserved graph grammar.

Lemma 2.1 Let gg:=(A, P, T, N) be an RGG. $\exists X_1:H \mapsto^{X1} H_1 \Rightarrow \exists X_2:H_1 \to^{X2} H$.

Proof: Let X_1 be a redex determined by a production p:=(L, R). According to the definitions of the RGG and the transformation process, if $\exists X_1:H \to^{X1} H_1$, then H_1 has a redex X_2, which is transformed from X_1 and is determined by R. Hence we have $\exists X_1:H_1 \to^{X2} H'$. But according to the transformation process, we have $H' \approx H$. So $\exists X_2:H_1 \to^{X2} H$.

Lemma 2.2 Let gg:=(A, P, T, N) be an RGG. $\exists X:H \to^{X} H_1 \Rightarrow \exists X':H_1 \mapsto^{X'} H$.

Proof: Similar to Lemma 2.1.

Lemma 2.3 Let gg:=(A, P, T, N) be a graph grammar, if $A \mapsto^{*} G$ then $G \to^{*} A$.

Proof:

$A \mapsto^{*} G \Rightarrow A \mapsto^{X1} G_1 \mapsto^{X2} G_2 \mapsto \ldots \mapsto^{Xn} G \Rightarrow G \to^{Xn'} G_{n-1}, \ldots, \to^{X1'} A$ (Lemma 2.1)

$\Rightarrow \exists G \to^{*} A$.

Similarly we have:

Lemma 2.4 Let gg:=(A, P, T, N) be a graph grammar, if $G \to^{*} A$ then $A \mapsto^{*} G$.

Theorem 2.1 G∈L(gg) iff $\exists\Re:G\rightarrow^{\Re} A$, where $\Re$ is a list of redexes.

Proof: it is straightforward from Lemma 2.3 and Lemma 2.4.

Theorem 2.1 states that R-applications determine exactly the language defined by L-applications. This theorem indicates that if one can find a parsing path (i.e. $\Re$) which transforms a graph to the initial graph, the graph is valid. A recursive algorithm is needed for parsing, which is rather inefficient for parsing a large graph.

2.4 Graph Parsing

Parsing is a process that attempts to reduce a sentence according to a grammar. A reduction (R-application) is performed when a production is applied. Parsing a graph may be more complicated than parsing a piece of text.

2.4.1 A Parsing Algorithm

The process of parsing a graph with a grammar consists of: selecting a production from the grammar and applying an R-application of the production to the graph; this process continues until no productions can be applied (called a *single parsing path*). If the graph has been transformed into the initial graph after R-applications, the graph is valid (i.e. the parsing succeeds); otherwise, the above process is repeated with different selections (i.e. different parsing paths). If all the possibilities have been tried without success, the graph is invalid.

The first stage of any graph parsing algorithm consists of searching in a graph to find a redex of any production. When such a redex is found, the question arises whether the production should be applied or not. The application of one production may inhibit the application of another production and it subsequently causes the entire parsing process to fail. Therefore, every production instance represents a choice point in the algorithm.

Carrying out the above parsing process is time-consuming as it needs to attempt the R-applications for all productions. We have developed a simple parsing algorithm, called *selection-free parsing algorithm (SFPA)*, which only tries one parsing path, as shown in Fig. 2.9. SFPA is effective for an RGG only in the case that, when parsing any graph with SFPA, if one parsing path fails, any other parsing paths will also fail.

```
Parsing(Graph host){
   while(host!=null){
        matched=false;
        for all p∈P
        {
           redex=FindRedexForR(host, p);
           if(redex!=null){
             R-application(host,p, redex);
             matched=true;
           }
        }
        if(matched==false){
          print("invalid");
          exit(0);
        }
     }
}
```

Fig. 2.9. The selection-free parsing algorithm

More formally, only those RGGs with selection-free productions can use SFPA, where the selection-free property for a production set is defined as follows.

Definition 2.15 Graph G is a merger of graph G_1 and graph G_2, if

- G_1 and G_2 are sub-graphs of G,
- $\forall v \in G.V$: $v \in G_1.V \vee v \in G_2.V$, and
- $\forall e \in G.E$: $e \in G_1.E \vee e \in G_2.E$.

Definition 2.16 Let G_1 and G_2 be graphs, merge(G_1, G_2) is a set of mergers of G_1 and G_2.

In the following definition, we will use p.R and p.L to represent the right graph and the left graph of the production p respectively.

Definition 2.17 Let P be a set of productions. P is *selection-free*, if for any $p_1 \in P$, $p_2 \in P$, R_1, R_2, L_1, and L_2 are graphs isomorphic to p_1.R, p_2.R, p_1.L, and p_2.L respectively, and $\forall G \in merge(R_1, R_2) \wedge R_1 \in Redex(G, p_1.R) \wedge R_2 \in Redex(G, p_2.R)$, we have $\exists G_a, G_{ab}, G_b, G_{ba}$: $G_a = Tr(G, p_1.R, p_1.L, R_1) \wedge G_{ab} = Tr(G_a, p_2.R, p_2.L, R_2) \wedge G_b = Tr(G, p_2.R, p_2.L, R_2) \wedge G_{ba} = Tr(G_b, p_1.R, p_1.L, R_1) \wedge G_{ab} \approx G_{ba}$.

The definition specifies that a production set is selection-free if a graph with two redexes corresponding to two productions' right graphs is applied by the two productions in different orders, the resulting graphs are the same. According to this definition, an algorithm for checking whether a reserved graph grammar has a select-free production set can be developed.

To check whether a production set is selection-free, we need to check all the possible combinations of any two production's right graphs. If one combination does not satisfy the definition, the production set is not selection-free. Fig. 2.10 shows examples of the checking process. In Fig. 2.10(1), two copies (enclosed in dashed boxes) of the right graph of Production 6 are merged. According to the embedding rule, different orders of the R-applications to the redexes (i.e. the copies) result in the same graph. Fig. 2.10(2) appears to be merged by the right graphs of Productions 4 and 5, but the embedding rule determines that no redex of Production 5 exists. So the productions satisfy the selection-free condition.

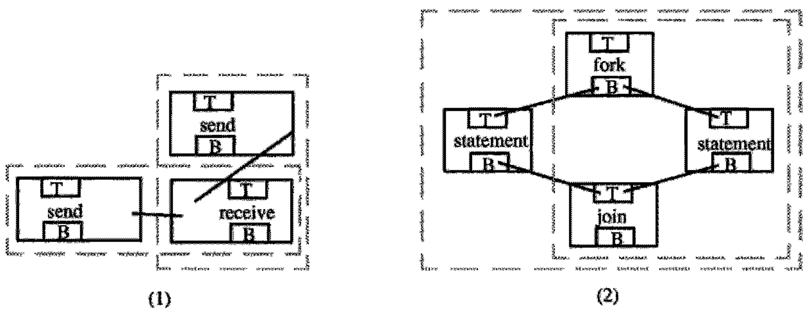


Fig. 2.10. Examples of checking the selection-free condition

The production set of the reserved graph grammar illustrated in Fig. 2.7 is selection-free under the definition, so we can use SFPA to parse any diagrams to check if they are valid process flow diagrams. In the following subsection, we will prove that a reserved graph grammar with a selection-free production set can use SFPA to parse diagrams correctly.

2.4.2 Selection-Free Grammars

The selection-free property of an RGG means that for a valid graph, any selection of an R-application to the graph can lead to a successful parsing. Obviously, a selection-free RGG can use the selection-free parsing algorithm to parse its languages. The selection-free property of a grammar can be formally defined as:

Definition 2.18 Let gg:=(A, P, T, N) be an RGG. If $\forall$(G$\rightarrow$*$G_i$$\rightarrow$*A), for any X$\in$Redex($G_i$), $\exists$G $\rightarrow$* G_i $\rightarrow^X$ $G_{i+1}$$\rightarrow$*A), then gg is said to be *selection-free*.

Definition 2.19 Let gg:=(A, P, T, N) be an RGG. If for any G$\rightarrow$*A, $X_a$$\in$Redex(G) $\wedge$$X_b$$\in$Redex(G) $\wedge\neg$(X_a=X_b)) such that $\exists$(G $\rightarrow^{Xa}$ G_a $\rightarrow^{Xb}$ G_{ab}) $\wedge$ $\exists$(G $\rightarrow^{Xb}$ G_b $\rightarrow^{Xa}$ G_{ba}) $\Rightarrow$$G_{ab}$$\approx$$G_{ba}$, then gg is said to be *order-free*.

The order-free property is similar to the finite Church Rosser property (Brandenburg 1988) but applicable to context-sensitive graph grammars in that productions are applied to sub-graphs rather than to single nodes. For simplicity, if G$\approx$G', we will use G instead G' in the sequel. We now show that if the production set of an RGG is selection-free, the RGG is selection-free.

The following lemma implies that a redex of a graph defined in an order-free graph grammar can be applied with an R-application and the graph can be reduced to the initial graph.

Lemma 2.5 Let a graph grammar gg:=(A, P, T, N) be order-free, if G$\rightarrow$*A $\wedge\exists$X$\in$Redex(G) then $\exists$i:G$\rightarrow$*$G_i$$\rightarrow^X$$G_{i+1}$$\rightarrow$*A.

Proof:

- G$\rightarrow$*A $\wedge\exists$X$\in$Redex(G) $\Rightarrow\exists$G$\rightarrow^{X0}$$G_1$...$\rightarrow^{Xn}$A, where n>0. We have two cases:
- Case 1: X_0=X $\Rightarrow$ $\exists$G$\rightarrow^X$$G_1$$\rightarrow$*A.
- Case 2: $X_0$$\neq$X $\Rightarrow$ $\exists$G$\rightarrow^{X0}$$G_1$$\rightarrow$*A $\wedge$ $\exists$X$\in$Redex(G_1) -- (Definition 2.19).
- This process can continue:
- $\exists$G$\rightarrow^{X0}$$G_1$$\rightarrow^{X1,...,Xm}$$G_m$$\rightarrow$*A$\wedge$ $\exists$X$\in$Redex(G_m)
- where m$\leq$n.
- As n is finite (the property of the layered definition), we have
- $\exists$i$\leq$n: G$\rightarrow$*Gi$\rightarrow^X$$G_{i+1}$$\rightarrow$*A.

Lemma 2.6 presented below implies that a redex can be applied anywhere in the R-application process.

Lemma 2.6 Let a graph grammar gg:=(A, P, T, N) be order-free and $\forall$$G_0$$\rightarrow$*A. If $\exists$X $\in$ Redex(G_0) $\wedge$ $\exists$$G_0$ $\rightarrow$* $G_n$$\rightarrow^X$$G_{n+1}$ then $\exists$$G_0$ $\rightarrow^X$$G_1$'$\rightarrow$*$G_{n+1}$.

Proof:

- $G_0$$\rightarrow$*$G_n$$\rightarrow^X$$G_{n+1}$

- $\Rightarrow \exists G_0 \rightarrow^{X0} G_1 \rightarrow X_1 G_2 \rightarrow \ldots \rightarrow^{Xn-1} G_n \rightarrow^{X} G_{n+1}$
- $\Rightarrow \exists G_{n-1} \rightarrow^{Xn-1} G_n \rightarrow^{X} G_{n+1}$
- $\Rightarrow \exists G_{n-1} \rightarrow^{X} G_n' \rightarrow^{Xn-1} G_{n+1}$ -- (Definition 2.19)
- $\Rightarrow \exists G_{n-2} \rightarrow^{X} G_{n-1}' \rightarrow^{Xn-2} G_0' \rightarrow^{Xn-1} G_{n+1}$
- $\Rightarrow \ldots$
- $\Rightarrow \exists G_0 \rightarrow^{X} G_1' \rightarrow^{X0} G_2' \rightarrow \ldots \rightarrow^{Xn-2} G_n' \rightarrow^{Xn-1} G_{n+1}$
- $\Rightarrow \exists G_0 \rightarrow^{X} G_1' \rightarrow^{*} G_{n+1}$.

Theorem 2.2 If gg:=(A, P, T, N) is order-free, then gg is selection-free.

Proof:

- $G \rightarrow^{*} Gi \rightarrow^{*} A \wedge X \in Redex(G_i)$
- $\Rightarrow \exists G \rightarrow^{*} G_i \rightarrow^{*} G_j \rightarrow^{X} G_{j+1} \rightarrow^{*} A$ -- (Lemma 2.5)
- $\Rightarrow \exists G \rightarrow^{*} G_i \rightarrow^{X} G_{i+1} \rightarrow^{*} G_{j+1} \rightarrow^{*} A$ -- (Lemma 2.6)
- $\Rightarrow \exists G \rightarrow^{*} G_i \rightarrow^{X} G_{i+1} \rightarrow^{*} A$.

Theorem 2.3 Let gg:=(A,P, T, N), if P is selection-free, then gg is order-free and thus selection-free.

Proof:

- Suppose $G \rightarrow^{*} A \wedge X_1 \in Redex(G) \wedge X_2 \in Redex(G)$, we have $\exists p_1 \in P \wedge \exists p_2 \in P$ so that $X_1 \approx p_1.R$ and $X_1 \approx p_1.R$.
- Since P is selection-free and $(X_1 \cup X_2) \subseteq G$, G can be transformed by applying X_1 and X_2 in any order and the resulting graphs are the same.
- The transformation process derives that $\forall G \rightarrow^{*} A$ if $X_1 \in Redex(G) \wedge X_2 \in Redex(G) \wedge \neg(X_1 = X_2)$ then $\exists\ G \rightarrow^{X2} G_1 \rightarrow^{X1} G_2 \wedge \exists G \rightarrow^{X1} G_1' \rightarrow^{X2} G_2$.
- Hence gg is order-free.
- According to Theorem 2.2, gg is selection-free.

Theorem 2.3 says that if the production set of an RGG is selection-free, the RGG is selection-free and can use SFPA to parse its languages.

2.4.3 Parsing Complexity

To study the time complexity of SFPA, we construct an algorithm FindRedexForR(G, p) shown in Fig. 2.11, which is the main part of the SFPA. To explain the algorithm, we first give some definitions.

```
Redex FindRedexForR(host,p)
{
        nodeSequence=findNodeSequence(p.R);
        allCandidates=findAllNodeSequences(host, nodeSequence);
        for all candidate∈allCandidates
        {
                redex=match(candidate, host, p));
                if(redex!=null)
                        return redex;
        }
        return null;
}
```

Fig. 2.11. The algorithm FindRedexForR

Definition 2.20 A node sequence of a graph G is an ordered list of all the nodes in G.

Definition 2.21 Let $L_1=[n_{11}, n_{12}, ..., n_{1k}]$ and $L_2=[n_{21}, ..., n_{2m}]$ be ordered node lists. L_1 is isomorphic to L_2 if $m=k \wedge n_{1i} \approx n_{2i}$ where $i \in \{1,..., m\}$.

Theorem 2.4 The algorithm FindRedexForR(G, p) has $O(|G|^m)$ time complexity, where m is the maximum number of nodes in any right graph of a set of productions.

Proof:

The function findNodeSequence(p.R) finds a node sequence of the right graph of a production p. It lists all the nodes of p.R in a certain order. For a graph grammar, the number of nodes in the right graph of a production is given, so the function takes O(1).

The function findAllNodeSequences(host, nodeSequence) collects all the possible node sequences from the host, each of which is isomorphic to nodeSequence. For a graph G, the number of all possible node sequences, each having m nodes, is k^m, where k is the number of nodes in G. So the time complexity for the function findAllNodeSequences is $O(|G|^m)$.

The function match checks whether a candidate in the host is a redex of the production p, if so, the candidate is returned as a redex, otherwise, a null is returned. The time complexity for the function match(candidate, host, p) is O(m).

As the number of allCandidates is no more than $|G|^m$, the maximum time taken is $O(|G|^m)$.

Theorem 2.5 The time complexity of SFPA is $O(|G|^{m+1})$, where G is a graph to be parsed by SFPA and m is the maximum number of nodes of all the right graphs of productions.

Proof:

Suppose that $T(k)=(2C)^kA_0+(2C)^{k-1}A_1+...+(2C)A_{k-1}+A_k$ is a function and next() is an operation applicable to T(k), where A_i, C and k are integers, and $C>0$, $k\geq 0$, $Ai\geq 0$, $0\leq i\leq k$.

Let $T(k).next(i) = (2C)^k(A_0) + (2C)^{k-1}(A_1) + ... +(2C)^{k-i+1}(A_{i-1}) + (2C)^{k-i}(A_i-1) + (2C)^{k-i-1}(A_{i+1}+C) +...+ (2C)(A_{k-1}+C) + (A_k+C)$ be T(k) after i executions of next() operation, where $A_i>0$, $C>0$, $k\geq 0$, we have

$T(k).next(i) = (2C)^kA_0 + (2C)^{k-1}A_1 +...+ (2C)^{k-i+1}A_{i-1} + (2C)^{k-i}A_i + (2C)^{k-i-1}A_{i+1} +...+ A_k - ((2C)^{k-i} - (2C)^{k-i-1}C -...- (2C)C-C) = T(k) - (2^{k-i}C^{k-i}-2^{k-i-1}C^{k-i}-...-2C^2-C)$

As $(2^{k-i}C^{k-i}-2^{k-i-1}C^{k-i}-...-2C^2-C)\geq (2^{k-i}C^{k-i}-2^{k-i-1}C^{k-i}-...-2C^{k-i}-C^{k-i}) = C^{k-i}(2^{k-i}-2^{k-i-1}-...-2-1) = C^{k-i} >0$.

We have $T(k)>T(k).next(i)$. (1)

This means $\exists n:n\leq T(k)$ such that T(k) will be zero after n executions of its "next" operation.

Let gg:=(A, P, T, N) be a reserved graph grammar and $G\rightarrow^* A$.

A graph G can be mapped to $T(k)=(2C)^kA_0+(2C)^{k-1}A_1+...+(2C)A_{k-1}+A_k$, where $A_i=|G|_i$, k equals to the maximum number of layers, and C is the maximum number of nodes of all the right graphs of productions. We denote G.T(k) as the T(k) that is mapped from G.

Suppose $G\rightarrow^X G'$. According to the definition of the grammar layer and the transformation rules, we have $G<G'$, where $\exists i:|G|_i <|G'|_i \wedge \forall j<i: |G|_j=|G'|_j$ with $|G|_k$ defined as $|\{x|x\in G \wedge layer(x)=k\}|$

This means that in the layer i, the element number of G' is less than the element number of G by $|G|_i-|G'|_i$ elements. In a layer larger than i, the number of additional elements are no more than C. So after the transformation,

$G'.T(k) \le (2C)^k(A_0) + (2C)^{k-1}(A_1) + \ldots + (2C)^{k-i+1}(A_{i-1}) + (2C)^{k-i}(A_i-(|G|_i-|G'|_i)) + (2C)^{k-i-1}(A_{i+1}+C) + \ldots + (2C)(A_{k-1}+C) + (A_k+C) \le G.T(k).next(i)$. So we have $\exists i{:}G'.T(k) \le G.T(k).next(i)$.

For any graph G, $G.T(k) \ge 0$, so according to (1), $G \to^* A$ must finish within G.T(k) steps.

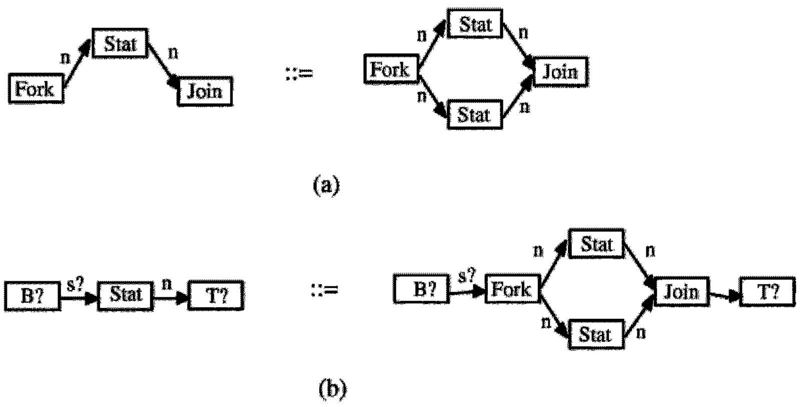
As $G.T(k)=(2C)^kA_0+(2C)^{k-1}A_1+\ldots+(2C)A_{k-1}+Ak \le (2C)^k(A_0+A_1+\ldots+A_k)=(2C)^k|G|$, according to Theorem 2.4, the time complexity of the algorithm SFPA is $(2C)^k|G|{*}O(|G|^m)=O(|G|^{m+1})$.

We now discuss the space complexity of SFPA. We implement an index for each element of a graph. The indices are organized as follows: they are listed in the same array if they refer to the graph elements that have the same label. Thus, a graph is a set of arrays, each of which is a list of elements with the same label. A nodeSequence (in Fig. 2.11) can be implemented as a set of pointers, each pointing to an element of an array. The next node sequence can be found by moving pointers in a proper way, and a candidate of a redex is the pointer set. In this case, the extra space is unnecessary except for the pointers. Thus, SFPA has a linear space complexity.

2.5 Improvements over the Layered Graph Grammar

Fig. 2.13(a) and (b) show two productions of the layered graph grammar for parsing the fork statement, where the elements B? and T? (wildcards) are used as the context elements. For instance, B? means begin, fork, or if, as shown in Fig. 2.13(c). After a transformation, say R-application, the relationships between the new node Stat and the host graph are determined by the B? and T?, which are part of the host graph. New nodes can be embedded into the host graph when they are linked with the matching nodes labeled with B? and T?. Without the wildcards, the number of productions required will be multiplied (Rekers and Schürr 1997).

The productions in Fig. 2.13(a) and (b) lead to ambiguity. For example, if a graph has a redex of the right graph in the production in Fig. 2.13(a), it also has a redex of the right graph in Fig. 2.13(b) because the right graph in Fig. 2.13(a) is a part of the right graph in Fig. 2.13(b). Applications of the productions in LGGs with different redexes may produce different results. A complex algorithm is then needed to ensure that all possible applications of productions are attempted.



B?∈{begin, fork, if}

T?∈{end, assign, fork, join, send, receive, if}

s?∈{n, f, t}

(c)

Fig. 2.13. Productions in a LGG with different embedding mechanisms

A reserved graph grammar can avoid the ambiguity. As a result, its parsing algorithm can be simple and efficient. Therefore, compared with the layered graph grammar (Rekers and Schürr 1997), the reserved graph grammar has the following three major improvements:

- it avoids the use of wildcards,
- it simplifies the specification through an embedding rule, and
- parsing an unambiguous reserved graph grammar can be done in polynomial time.

As discussed earlier, the Reserved Graph Grammars (RGGs) are based on LGGs, and improve over LGGs. Apart from the improvements discussed above, the major differences between the RGG formalism and the LGG formalism are that the former can be implemented more efficiently using the presented parsing algorithm; and that it uses simple embedding rules rather than context elements (as used in the latter) so that grammar specifications are simplified. The following table compares all the discussed grammars with RGGs.

Grammar	**Left-hand side**	**Right-hand side**	**Context**	**Embedding rules**	**Additional restrictions**	**Complexity**
Relational	non-terminal	relational structure	no	yes	explicit vertex order	exponential
1NS-RG	non-terminal	relational structure	no	yes	bounded degree; no non-terminal neighbors	exponential
Boundary NLC Graph	non-terminal	graph	no	yes	bounded degree; no non-terminal neighbors	exponential
Constraint Multiset	non-terminal	multiset	yes	implicit	deterministic	polynomial
Picture Layout	non-terminal	Max two (non-) terminals	1 terminal	implicit	finite set of attribute values	polynomial
Layered Graph	graph	graph	graph	no	layering	exponential
Reserved GG	graph	graph	graph	yes	selection-free	polynomial

The six attributes used to distinguish various grammars in the table are proposed by Rekers and Schürr (1997). They serve a useful purpose in comparing these grammars. Minas (1998) has adapted RGGs to the DiaGen hypergraph environment (Minas and Viehstaedt 1995). The selection-free constraint imposed in RGGs is relaxed to allow more types of hypergraphs to be specified. However, additional information has to be provided in the form of negative application conditions (NACs). A production with a matching left hand side is not applicable if one of its NACs is satisfied. The addition of NACs modifies the original grammar and it is unclear how additional complexity is introduced into the parsing process.

2.6 Summary

This chapter has presented the reserved graph grammar (RGG), which can be used to specify grammars of diagrammatic visual languages. An RGG is a collection of graph rewriting rules represented labeled graphs. It is context-sensitive and its right- and left-graphs can have an arbitrary number of nodes and edges. The grammar uses an enhanced node structure with a marking mechanism in its graph representation. It is this structure that makes an RGG effective in specifying a wide range of visual languages and efficient in parsing a certain class of visual languages. Although the time complexity of the parsing algorithm for a general RGG is exponential, parsing a selection-free reserved graph grammar can be done in polynomial time. The chapter has presented such a polynomial time parsing algorithm and proved its time and space complexities. To ensure a reserved graph grammar to be unambiguous, we also presented a checking criterion and proved its correctness.

There have been some applications of RGGs, for example, for generating a visual language for modeling distributed systems (Zhang and Zhang 1998a), and those to be described in Chapters 4 through 7. A wide range of applications, such as interpreting hand-written mathematical notations (Blostein and Grbavec 1997), have been reported for using layered graph grammars (Blostein and Schürr 1998), upon which RGGs improve.

7 Related Work

Growing interest in visual languages has motivated research in the specification and parsing of multi-dimensional structures. Several specification methods have been proposed and proven to be useful in practical applications. Examples include Web and Array Grammars (Rosenfeld 1976), Positional Grammars (Costaglioga et al. 1993), Relational Grammars (Wittenburg 1992; Ferrucci et al. 1994), Unification Grammars (Wittenburg et al. 1991), Attributed Multiset Grammars (Golin 1991), Constraint Multiset Grammars (Marriott 1994), Layered Graph Grammars (Rekers and Schürr 1997), and Attributed Graph Grammars (Ermel et al. 1999). In this section, we discuss some of the related grammars and compare them with Reserved Graph Grammars.

The relational grammars of Wittenburg (1992) are restricted to relational structures, where relationships of the same type define partial orders. Ferrucci et al. (1994) proposed 1NS-RG grammars, that are adapted from the Boundary NLC graph grammars of Rozenberg and Welzl (1986). The right-hand sides of productions in a 1NS-RG grammar may not contain non-terminals as neighbors in order to guarantee local confluence. Parsing can be done in polynomial time if the generated graphs are all connected and the maximum number of edges at any single vertex is known in advance. This latter restriction also applies to Brandenburg's DNELC graph grammar (1988).

Marriott's constraint multiset grammars (1994) provide context elements. Introducing "not exits" constraints prevent any possible overlap between the right-hand sides of productions, but also make syntax specifications deterministic. Golin's (1991) picture layout grammars allow productions with one non-terminal on the left-hand side and at most two terminals or non-terminals on the right-hand side.

Rekers and Schürr (1997) gave an excellent introduction to context-sensitive graph grammars. They argued that it was difficult for the aforementioned grammars to generate abstract syntax graphs for connected ER diagrams. They proposed a context-sensitive grammar formalism, known as layered graph grammars (LGGs) (Rekers and Schürr 1997), which can specify a wide range of visual languages. The graphical specifications of LGGs are more intuitive and easier to understand than textual grammars. Bottoni et al. (2000) improved the parsing efficiency of the LGG style of graph grammars by detecting conflicts through critical pair analysis.

Attributed Graph Grammars (Ermel et al. 1999) integrate graph transformation rules with Java expressions. The AGG language implements the single pushout algebraic approach (Ehrig et al. 1997) for graph transformation, that differs from the algorithmic approach in LGGs. It combines attributed graph transformation with negative application conditions to allow users to precisely specify a sub-graph that must not be present in order to perform a graph transformation (Habel et al. 1996; Ermel et al. 1999).

Motivated by applications such as modeling and refactoring for object-oriented programming, Drewes et al. (2006) recently proposed adaptive star grammars, which use the meta-model concept to allow multiple nodes to be copied arbitrarily often. The membership problem for a certain subclass of such grammars is shown to be decidable. A parser for adaptive star grammars has also been proposed (Minas 2006), though not implemented at the time of writing this book. Minas (2006) speculates that the parser would show exponential behavior but may in practical applications require polynomial time similar to the parser for hyperedge replacement grammars in DiaGen (Minas 2002).

Chapter 3 Spatial Specification

3.1 Introduction

Rather than expressing sentences in sequences of characters, visual programming languages allow programs to be expressed in visual sentences in a multi-dimensional fashion. As noted by Reker and Schürr (1996), the physical layout and the meaning of a diagram are two important aspects of a visual sentence. A *spatial relations graph (SRG)* specifies spatial relationships between pictorial objects while an *abstract syntax graph (ASG)* provides structural information in a succinct form. Consequently in their approach, a visual sentence is specified through two intermediate graphs. One is geared toward visualization, and the other toward interpretation (Reker and Schürr 1996).

Some researchers specify the interdependency between the concrete syntax and the abstract syntax using graph grammars. For example, a grammatical approach is proposed to maintain a correspondence between the abstract and spatial aspects of VPLs (Reker and Schürr 1996). Aiming at syntax directed layouts, Brandenburg (1995) proposes the *layout graph grammar*, which directly draws a grammatical rule on a plane and determines its spatial configuration according to positions of objects in the plane.

Those approaches explore spatial relationships from the layout perspective without direct contribution to the interpretation of a graph. Due to the visual nature of VPLs, we believe that the spatial information should not only contribute to the representation, but also directly represent structural and semantic requirements over involved objects. For example, a spatial configuration can visually and explicitly hold the information defining an order over a collection of objects (e.g. the left one has a smaller index than the right one). By extending the context-sensitive graph grammar formalism the *Reserved Graph Grammar* (*RGG*) presented in Chapter 2 (Zhang and Zhang 1997), a *Spatial Graph Grammar* (*SGG*) (Kong et al. 2006) introduces spatial relationships into the abstract syntax, and integrates both the spatial and structural specification mechanisms into a single framework. In other words, in addition to defining connectivity among nodes, the formalism is able to specify semantic and structural requirements with spatial information.

This chapter presents a *spatial graph grammar* (*SGG*), which introduces spatial constraints to the abstract syntax. In other words, both the connectivity and spatial relationships construct the pre-condition of a graph transformation. With the extended expressive power, the SGG is capable of intuitively and visually specifying semantic and structural requirements through spatial information rather than through attributes. The SGG is fundamentally different from other graph grammar formalisms by introducing spatial information into the abstract syntax, and can find many applications, such as adaptive layout of multimedia documents to be presented in Chapter 4.

The SGG is enhanced from the *Reserved Graph Grammar (RGG)* (Zhang and Zhang 1997) presented in the previous chapter, with a spatial extension. In particular, spatial relationships are intuitively specified without sacrificing the expressiveness of structural specifications. In summary, tightly integrating spatial and abstract specifications, the SGG can bring the following benefits:

- Correlating spatial configurations with abstract structures, the SGG can generate a syntax-directed layout, and perform an adaptive presentation based on the existing layout;
- The SGG can naturally specify semantic and structural requirements through spatial relationships rather than through attributes or edges; and
- With the help of spatial specifications, the parser of the SGG performs in polynomial time with an improved parsing complexity over its non-spatial predecessor, i.e. the Reserved Graph Grammar.

3.2 The Spatial Graph Grammar Formalism

This section introduces the spatial graph grammar formalism within a generic visual environment supporting spatial programming.

Being inherent in the multiple-dimensional fashion, the SGG incorporates spatial notions into the abstract syntax with nodes and edges. 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. The *size* and *shape* of an object illustrate its internal properties while *spatial relations* express configurations among distinct spatial objects. Based on several well-established models (Clementini et al. 1993; Frank 1996; Hernández et al. 1995), the SGG supports spatial relations in four aspects, i.e. *topology*, *direction*, *distance* and *alignment*. Developers can customize the granularity of spatial relationships upon the application domain, and design a set of graphical notations to visually denote spatial relationships.

3.3 Spatial Relationships and Representations

When considering adding spatial notations to the RGG formalism, we generally aim at

- Retaining the original RGG syntax and semantics, and
- Introducing minimal additional notations that are intuitive for spatial specifications.

We propose five categories of spatial relationships between any two given objects: *direction, distance, topology, alignment,* and *size*. When used to specify desired layout re-arrangement, the relationships can be used to specify changes to be applied during graph transformation. The spatial relationships are currently restricted in a two-dimensional multimedia design space. It is entirely feasible that such relationships and notations can be extended to a three-dimensional space for specifying a virtual reality environment.

All the spatial relationships are defined between two objects, one referred to as the *primary object* and the other as the *reference object*. When discussed in the context of a spatial grammar, media objects are represented and referred to as *nodes* in grammar productions as termed in Chapter 2.

3.3.1 Direction

To represent the relative direction between two nodes, each node is arranged as a 3x3 grid in dotted lines inside the node, as shown in Fig. 3.1. The central region (marked by C) represents the node itself. Surrounding the central region, the eight grid regions represent eight directions: N (North), NE (Northeast), E (East), SE (Southeast), S (South), SW (Southwest), W (West), NW (Northwest), in clockwise order. Each of these directions indicates the relative position of the reference object connected to the current object (primary object). The boundary of the area occupied by the reference object is outlined by dotted lines surrounding the primary object.

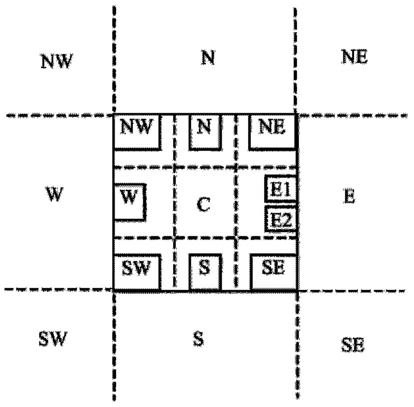


Fig. 3.1. Notation for direction relationships

Each of the eight direction regions may include more than one vertex. The objects that are connected to a primary object through the vertices of the same region are in the same direction. For instance, the East region of the node in Fig. 3.1 has two vertices, E1 and E2, if there are two nodes connected to E1 and E2 from the right side of the present node.

3.3.2 Distance

The distance between two objects' centers measures an important class of spatial relationships. To specify the distance relationship, we prefix a "+" to the vertex label to indicate a long (or increased) distance to the object that the vertex connects to, "-" to indicate a short (decreased) distance, and blank to represent a distance not emphasized (or not changed). Four special cases of distances are treated separately as topological relationships due to their importance in spatial reasoning.

3.3.3 Topology

We can generally define four topological relationships between two nodes: *non-overlapping, overlapping, touching,* and *containing*. Assume that D_x is the set of all the points on an object x, and B_x ($\subseteq D_x$) is the boundary point set of x. Considering a primary object a and a reference object b and $D_a \cap D_b = R$, four topological relationships are defined as the following:

- a is *non-overlapping* with b iff $R = \Phi$;
- a is *overlapping* with b iff $R \neq \Phi$, and further
  - o a is *touching* with b iff $R \subseteq (B_a \cap B_b)$; or
  - o a is *containing* b iff $D_b \subseteq D_a$.

Using a rectangle to represent an object, Fig. 3.2 shows the four types of topological relationships. *Non-overlapping* indicates that there is no common point on both involved objects. *Overlapping* means that there are common points between the two objects. It is represented by dotted lines on the boundary of the overlapped area. We define *touching* and *containing* as two special cases of overlapping. If common points only exist on the boundaries of two objects, the objects are *touching* with each other. The touched part is represented by a dotted line. *Containing* means that all the points on one object belong to the other. In Fig. 3.2, the boundary of an object is totally dotted, indicating that the object is contained in the other object.

3.3.4 Alignment

Two objects may be aligned vertically or horizontally, along the directions of N, S, W or E. In horizontal direction, we define three different horizontal alignment cases for each object, i.e. top alignment, bottom alignment and center alignment, giving a total of 9 different alignment relationships between any two objects. The alignment relationships in vertical direction are similarly defined. The boundary of a node is divided into 12 segments according to the 3x3 grid. A bold segment is used to indicate the alignment relationship. Fig. 3.3 illustrates three most common alignment relationships.

3.3.5 Size

For example, transforming a multimedia document from a desktop Web page to a PDA display may involve size changes of various media objects. To represent the changes, we mark the node's center box with a "+" to indicate that the object is large in size (or zoomed in to become larger), "-" to indicate small in size (or zoomed out to become smaller), and blank to indicate a size not emphasized (or unchanged).

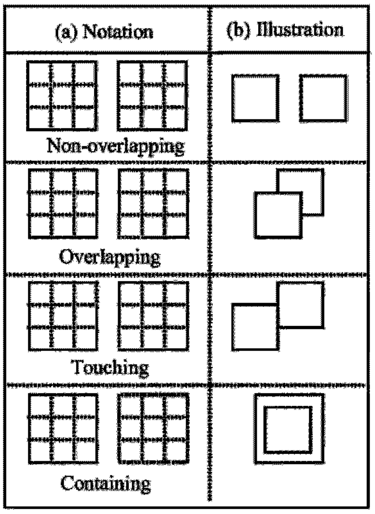


Fig. 3.2. Topological relations

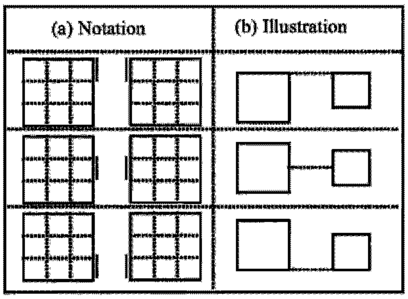


Fig. 3.3. Alignment relations in horizontal direction

3.3.6 Event Driven

As discussed above, the spatial relationships can be used to specify static layout structure or some predetermined structural changes. Similarly, the SWITCH construct in SMIL (W3C 2001) allows different layouts to be applied upon different pre-determined conditions, which are defined through text attributes. Many media players, however, do not provide sufficient support for the specification and run-time evaluation of application-dependent test attributes. On the other hand, the grammatical approach allows great flexibility to associate domain-specific triggering conditions with productions without relying on the media player's capability. There are increasing demands for providing users a sense of focusing, realized by interactively changing the details of certain parts of a multimedia document during viewing. Such a mechanism is called *interactive semantic zooming* (Marriott et al. 2002).

To address the dynamic issues, we classify graph productions into conditional and unconditional ones. A transformation is performed on an unconditional production when a redex is found in the host graph that matches the right graph of the production. A conditional production can only be triggered by a specific event, such as the change of the device capability, the user's interaction etc. Since graph transformations can be performed according to dynamic events, such as user inputs, the appearance of a multimedia document may be adjusted by triggering such conditional productions when the layout structure, user interaction method, and display environment need to be changed. More flexible than SMIL SWITCH, conditional productions are able to handle user interactions, and individually produce local effects.

3.3.7 Syntax-Directed Computations

As described in Chapter 2, the RGG supports syntax-directed computations by associating data and operators to nodes in productions in terms of *attributes* and *actions*. Inherited from the RGG, in SGGs, attributes can supplement graphical and qualitative specifications by providing precise quantitative values. For example, to shorten the distance between two connected nodes by half, we can attach the following action code to the corresponding production:

```
Action(AAMGraph g) {
        int OriginalDistance = Q.left – P.right;
        P.right = P.right + OriginalDistance/4;
        Q.left = Q.left - OriginalDistance/4;
}
```

In summary, a visual representation defines an approximation of layout while attributes and action codes supplement qualitative specifications with precise quantitative information and associated computation.

3.4 Formal Definitions

Entities of real worlds are represented by nodes, which are organized in a two-level hierarchy in the same fashion as the RGG presented in Chapter 2. In a node, each vertex is uniquely identified, and serves as a connecting point of an edge representing a logical connection. The type of an edge is implicitly determined by the labels of the vertices that connect the edge (annotations illustrating types of edges can be attached to the edges though it is not necessary). For example, a node labeled *If* as illustrated in Fig. 3.4 contains three vertices over the vertex label set $\Omega_N = \{T, L, R\}$.

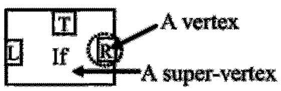


Fig. 3.4. A SGG node

Definition 3.1: Given a vertex label set Ω_N, a *node* is a tuple N = ‹V^N, l^N, $name^N$›, where V^N is a set of vertices, $l^N : V^N \rightarrow \Omega_N$ is the labeling function, and $name^N$ is a node label determining the type of node *N*. Moreover, $\forall\ v_1, v_2 \in V^N$, $l^N(v_1) = l^N(v_2) \Rightarrow v_1 = v_2$.

Functions can be combined using the composition operation denoted by "*o*". For $f{:}R \rightarrow S$ and $g{:}S \rightarrow T$, $g \circ f$ is the function with domain *R* and codomain *T* such that for all $x \in R$, $g \circ f(x) = g(f(x))$.

Definition 3.2: Over two nodes N and N', the *structural isomorphism* g: $V^N \rightarrow V^{N'}$ is a bijective function, which preserves the labels of vertices, i.e. satisfying $l^{N'} \circ g = l^N$. Two nodes are *isomorphic*, denoted as N≈N', iff they have structural isomorphism, and $name^N = name^{N'}$.

This definition determines that two nodes are isomorphic if they have the same types of vertices.

Definition 3.3: Over a set of objects O, a *spatial signature* is a function *f* : O × O → SR, where SR is the set of spatial relationships. *f* maps a pair of nodes to the spatial relationship between them.

Definition 3.4: A *graph* is a tuple G:= ‹N^G, E^G, V^G, s^G, t^G, m^G, g^G›, where N^G is the set of nodes, E^G is the set of edges, V^G is the set of vertices constructing N^G, $s^G{:}E^G \rightarrow V^G$ and $t^G{:}E^G \rightarrow V^G$ are two functions that preserve the two connecting points of an edge, m^G: $V^G \rightarrow N^G$ is a function that maps a vertex to its associated node, g^G is the spatial signature.

Because of the multi-dimensional nature, the embedding problem in visual programming languages, i.e. establishing connections between the surrounding of a replaced sub-graph and a new sub-graph, does not exist in textual languages. The *marking* technique (Zhang et al. 2001a) developed in the RGG (see Chapter 2) addresses the embedding issue. A marked vertex is assigned by a unique integer, and preserves its associated edges connecting from the replaced sub-graph to the outside.

Definition 3.5: mark: V → I is a partial function[1], where V is a set of vertices, and I is a set of integers. A vertex *v* is *marked* iff mark(*v*)↓. Moreover, within a graph G, for $\forall v_1, v_2 \in V^G$, mark(v_1)= mark(v_2) implies $v_1 = v_2$.

[1] A partial function on a set *V* is simply a function whose domain is a subset of *V*. If *f* is a partial function on *V* and $v \in V$, then we write *f(v)*↓ and say that *f(v)* is defined to indicate that *v* is in the domain of *f*; if *v* is not in the domain of *f*, we write *f(v)* ↑ and say that *f(v)* is undefined.

Definition 3.6: Two vertices *a* and *b* in two different graphs are equivalent, denoted as $a \equiv b$, iff mark(*a*) = mark(*b*) and mark(*a*)$\downarrow$.

Definition 3.7: A *marked graph* is a tuple $G := \langle N^G, E^G, V^G, s^G, t^G, m^G, g^G, mark^G \rangle$, the first seven elements are the same as those in Definition 4, and $mark^G$ is a marking function.

Having introduced the basic concepts, this section defines the notion of comparisons between graphs.

Definition 3.8: A *graph morphism* $f: G \rightarrow G'$ defines a pair of functions $\langle f^N: N^G \rightarrow N^{G'}, f^E: E^G \rightarrow E^{G'} \rangle$, where f^N and f^E are bijective functions, and preserve the structural specifications, i.e. satisfying $f^N \circ m^G \circ s^G = m^{G'} \circ s^{G'} \circ f^E$, $s^G = s^{G'} \circ f^E$, $m^G \circ s^G \approx m^{G'} \circ s^{G'} \circ f^E$, $f^N \circ m^G \circ t^G = m^{G'} \circ t^{G'} \circ f^E$, $t^G = t^{G'} \circ f^E$, $m^G \circ t^G \approx m^{G'} \circ t^{G'} \circ f^E$.

Definition 3.9: Over a marked graph G and a graph G', a *spatial morphism* $f: N^G \rightarrow N^{G'}$ is a function that preserves the spatial relationships, i.e. satisfying $\forall n_1, n_2 \in N^G$: $R_2(f(n_1), f(n_2)) \rightarrow R_1(n_1, n_2)$, where R_1 and R_2 are boolean expressions for verifying the spatial relationships $r_1 = g^G(n_1, n_2)$ and $r_2 = g^{G'}(f(n_1), f(n_2))$ correspondingly, g^G and $g^{G'}$ are the spatial signatures of G and G' respectively.

A spatial morphism identifies that a spatial signature can be derived from the spatial relationships of two involved graphs. In the following, we will simply say that the two graphs share a spatial signature.

To apply a grammar rule to a given graph (called a *host graph*), we need to find a sub-graph in the host graph that matches the right graph (or left graph) of the rule. The matched sub-graph is called a *redex*.

Definition 3.10: A sub-graph *X* of a graph *H* is called a *redex* of a marked graph *G*, denoted as $X \in \mathrm{Redex}(H, G)$, iff

1. $f = \langle f^N: N^G \rightarrow N^X, f^E: E^G \rightarrow E^X \rangle$ is a *graph morphism* between *G* and *X*;
2. f^N also serves as a *spatial morphism* between *G* and *X*;
3. $\forall e \in E^H, (\exists n \in N^G, v \in V^G\ (m^H(s^H(e)) \approx f^N(n) \wedge s^H(e) = v \wedge m^G(v) \approx n \wedge mark^G(v)\uparrow) \Rightarrow \exists\, c \in N^G, d \in V^G\ (m^H(t^H(e)) \approx f^N(c) \wedge t^H(e) = d \wedge m^G(d) \approx c)$; or

   $\forall e \in E^H, (\exists n \in N^G, v \in V^G\ (m^H(t^H(e)) \approx f^N(n) \wedge t^H(e) = v \wedge m^G(v) \approx n \wedge mark^G(v)\uparrow) \Rightarrow \exists\, c \in N^G, d \in V^G\ (m^H(s^H(e)) \approx f^N(c) \wedge s^H(e) = d \wedge m^G(d) \approx c)$; and
4. $\forall n_1, n_2 \in N^G, f^N(n_1) = f^N(n_2) \Rightarrow n_1 = n_2$.

This definition specifies that a sub-graph *X* of a graph *H* can be a redex of a marked graph *G* if and only if *X* is isomorphic to *G* in structure, and they share a spatial signature.

The SGG is equipped with a parser that verifies the membership of a visual sentence. By taking the spatial information, the parsing algorithm performs in polynomial time with an improved parsing complexity over its non-spatial predecessor, i.e. the Reserved Graph Grammar.

3.5 Graph Parsing

Having defined the context-sensitive spatial graph grammar formalism, this section presents a parsing algorithm that uses spatial specifications to reduce searching space, and analyzes the time complexity.

3.5.1 A Parsing Algorithm

The parsing process is a sequence of R-applications, which is modeled as recognize-select-execute (Bardohl et al. 1999). Parsing a spatial graph grammar proceeds as follows:

1. Search for a redex of the right graph in the host graph without considering spatial information;
2. Derive spatial relationships between objects in the redex according to the physical drawing of the host graph. Compare the spatial relationships with the spatial signature of the right graph (RHS) of each production;
3. If the redex holds the same spatial signature as the RHS, embed a copy of the left graph into the host graph by replacing the redex. Otherwise, go to Step 1 to search for a new redex.

One or more occurrences of a right graph may exist in the host graph, and the selection will affect the parsing result. Even for the most restricted classes of graph grammars, the membership problem is NP-hard (Rozenberg 1986). Consequently, a parser may not recognize a syntactically correct graph or be inefficient in analyzing a large and complex graph. To allow efficient parsing without backtracking, we are only interested in *confluent* graph grammars. Informally, the *confluence* requires that different orders of applications of productions achieve the same result. Fig. 3.5 illustrates a parsing algorithm for confluent grammars, which only tries one parsing path. In other words, if one parsing path fails, other parsing paths will also fail.

```
Parsing (HostGraph host)
{
  while (host !=NULL)
  {
    matched = false;
    for all $p \in$Productions
    {
      Redex = FindRedexForR(host, p);
      If (Redex!=NULL)
      {
        R-application(host, p, Redex);
        Matched = true;
      }
    }
    if (matched == false)
    {
      print ("Invalid");
      exit(0);
    }
  }
}
```

Fig. 3.5. The parsing algorithm

3.5.2 Object Sequencing

Searching for a redex in the host graph becomes the key to the parsing process when the parser needs not care about the application order. Without an order imposed on objects in the original Reserved Graph Grammar, searching for *m* objects in a host graph *G* runs in $O(m^2|G|^m)$ time as shown in the previous chapter. With the layout information found in the host graph and productions, we should be able to narrow the search space for the parser. Our idea is to sequence the visual objects in each of the host graph and right graphs of grammar productions into an ordered list so that efficient string-matching techniques can be used to find the redex in the host graph.

Fig. 3.6 proposes the algorithm *FindRedexForR* to find a valid redex in a host graph according to a given production. It proceeds as follows:

1. Encode the objects of the host graph into a sequence, called *host sequence*;
2. Encode the objects in the right graph of the production into a sequence, called *pattern sequence*;

```
Redex FindRedexForR(HostGraph G,
Production P)
{
  H=SequenceHostGraph(G);
  R=SequenceRightGraph(P.R);
  Index=SequenceIndex(R);
  redex = match(H, R, Index, P, G);
  return redex;
}
```

Fig. 3.6. The *FindRedexForR* algorithm

3. Search for the pattern sequence in the host sequence as illustrated in Fig. 3.7.

```
Redex match (Sequence H, Sequence R,
Link Index, Production P, HostGraph G)
{ i=1; redex=NULL;
  while (i≤|G|)
  { for all k∈Index[H[i]]
        { UpdateSet(k);
          if (k==|R|)
            redex = VerifyNSObj(P, G);
          if (redex != NULL)
            return redex; }
    i++
  }
  return redex;
}
```

Fig. 3.7. The match algorithm

The first step of the algorithm *FindRedexForR* is to sequence the visual objects of the host graph according to their physical positions. Object *a* is ordered before object *b* (*a*'s index < *b*'s index) if *a* has a larger y-coordinate than *b*. If two objects have the same y-coordinates, their order in the host sequence is determined by their x-coordinates, and the object with a smaller x-coordinate holds a smaller index. We assume that the origin locates at the left-bottom corner of the screen, and the y axis extends to the north while x axis to the east. A unique order is imposed on the visual objects of the host graph according to their physical positions from top to bottom, and from left to right when objects share the same y-coordinate. As mentioned before, shapes of objects are approximated as rectangles, and the coordinates of central points are used to represent the physical positions of objects in the sequencing process.

In the right graphs of productions, spatial relationships are specified through visual notations instead of physical drawings. In order to sequence the right graph, we first generate a directed acyclic graph (DAG) based on the spatial specification in the right graph, and then perform topological sorting on the DAG to derive a unique pattern sequence. In other words, a node in the DAG represents a visual object in the right graph, between which a north-south or west-east relationship is denoted by a directed edge. A DAG denoting north-south/west-east relationship is called V-DAG/H-DAG. In summary, as the second step of *FindRedexForR*, *SequenceRightGraph* first generates a V-DAG and an H-DAG, and then performs topological sorting to obtain the pattern sequence. It proceeds as follows (the following description only illustrates how to generate a V-DAG, and the same principle applies to the generation of an H-DAG):

1. Create nodes for visual objects in the right graph: a node is created to represent each unique visual object. However, no node is introduced for the object which has no spatial relationship with any other objects in the right graph.
2. Inter-connect nodes: in the right graph, if an object is defined locating south-west, south, or south-east to another object, a directed edge is introduced to connect the two corresponding nodes from north to south.
3. Perform topological sorting: insert into the pattern sequence a visual object represented by a V-DAG node that has no incoming edge. If there is more than one node without incoming edges, insert the leftmost object by traversing the H-DAG. After insertion, delete the node and its associated edges in the V-DAG. Iterate this step until the V-DAG becomes empty.

In the right graph, a visual object is a *non-spatial object* if it has no spatial relationship with other objects; otherwise, it is a *spatial object*. A non-spatial object will not appear in the pattern sequence. In other words, we ignore such an object in the process of searching for a pattern sequence in the host sequence, and process it later in the *VerifyNSObj* procedure as shown in Fig. 3.7.

Lemma 3.1: There exists a unique order on spatial objects in the right graph of a production P=(*L*,*R*) iff $\forall o_1, o_2 \in N^R$, $(o_1 \nrightarrow^{V} o_2$ and $o_2 \nrightarrow^{V} o_1) \Rightarrow (o_1 \rightarrow^{H} o_2$ or $o_2 \rightarrow^{H} o_1)$, where $o_i \nrightarrow^{V} o_j$ and $o_i \nrightarrow^{H} o_j$ denote no path between o_i and o_j in the V-DAG and H-DAG respectively, and $o_i \rightarrow^{V} o_j$ and $o_i \rightarrow^{H} o_j$ indicate a path between o_i and o_j in the V-DAG and H-DAG respectively.

Proof:

(1) Given that there is a unique order on spatial objects, we assume that $\forall o_1, o_2$: o_1's index < o_2's index:

Case 1: There is a path from o_1 to o_2 in the V-DAG, i.e. $o_1 \rightarrow^{V} o_2$, which contradicts with the condition $o_1 \nrightarrow^{V} o_2$.

Case 2: There is no path from o_1 to o_2 in the V-DAG, i.e. $o_1 \nrightarrow^{V} o_2$. According to the assumption, it follows that $o_2 \nrightarrow^{V} o_1$. The condition of $o_1 \nrightarrow^{V} o_2$ and $o_2 \nrightarrow^{V} o_1$ indicates that we cannot determine the order of o_1 and o_2 in the V-DAG. Based on the third step of *SequenceRightGraph* and the assumption, it follows that $o_1 \rightarrow^{H} o_2$.

Therefore, $\forall o_1, o_2 \in N^R$, $(o_1 \nrightarrow^{V} o_2$ and $o_2 \nrightarrow^{V} o_1) \Rightarrow (o_1 \rightarrow^{H} o_2$ or $o_2 \rightarrow^{H} o_1)$.

(2) Assume $\forall o_1, o_2$, $(o_1 \nrightarrow^{V} o_2$ and $o_2 \nrightarrow^{V} o_1) \Rightarrow (o_1 \rightarrow^{H} o_2$ or $o_2 \rightarrow^{H} o_1)$. We prove that there is a unique order on spatial objects.

- If $o_1 \rightarrow^{V} o_2$, o_1's index < o_2's index;
- If $o_2 \rightarrow^{V} o_1$, o_2's index < o_1's index; otherwise,
- Since $(o_1 \nrightarrow^{V} o_2$ and $o_2 \nrightarrow^{V} o_1)$ is satisfied, either $(o_1 \rightarrow^{H} o_2)$ or $(o_2 \rightarrow^{H} o_1)$ is true. It follows that there is an order between o_2 and o_1.

Therefore, there exists a unique order on spatial objects. ■

If two objects' order cannot be determined on their available spatial information in a right graph, we supplement additional spatial specifications to ensure a unique sequence.

By sequencing host graphs and productions, the problem of searching for a sub-graph in a host graph becomes that of searching for a pattern sequence in a host sequence. With a pattern sequence containing *m* objects, (*m-1*) sets are required to record sub-sequences of the host sequence. In particular, each sub-sequence in the k^{th} ($k<m$) set satisfies the following two requirements: (1) it matches the first *k* objects of the pattern sequence; (2) a morphism can be found between the first *k* objects of the pattern sequence

and their corresponding objects in the sub-sequence. Initially, all sets are empty and a new sub-sequence is obtained by extending it with an appropriate object. Furthermore, each object of the host sequence maintains a set of pointers, which point to sub-sequences including the object. From the first object of the host sequence, the ***match*** algorithm as shown in Fig. 3.7 proceeds as follows:

1. Assume that the class of the current visual object a (we treat the type of a node as its class) takes the i^{th} position in the host sequence and the k^{th} position in the pattern sequence. A new collection of sub-sequences is obtained by extending all sub-sequences in the $(k\text{-}1)^{th}$ set with object a. We need to exclude a sub-sequence from the collection if no morphism exists between the first k objects of the pattern sequences and their corresponding objects in this sub-sequence. Since all sub-sequences in the $(k\text{-}1)^{th}$ set have been verified that a morphism exists, we only need to check the morphism between the k^{th} object and any j^{th} ($j<k$) object of the pattern sequence and their corresponding objects in the sub-sequence of the host sequence. In other words, if no morphism exists between the k^{th} object and the j^{th} object of the pattern sequence and their corresponding objects a and b in a sub-sequence, we eliminate that sub-sequence from the collection. After verifying all the previous $(i\text{-}1)$ objects, the remaining sub-sequences satisfy the above two requirements, and are added to the k^{th} set.
2. Move up to the next object in the host sequence, and go to Step 1.
3. Whenever a pattern sequence is found in the host sequence, the non-spatial objects excluded from the pattern sequence are searched for in the remaining objects of the host sequence. Identify a morphism of the structural relationships defined in the right graph among non-spatial objects and spatial objects with those presented in the host graph. If the non-spatial objects are found in the host sequence and their morphism is confirmed, a redex is found and a graph transformation is performed.

By ordering visual objects according to their spatial relationships, we can reduce the searching space, and thus perform an efficient matching. As illustrated in Fig. 3.6, the function *SequenceHostGraph* generates a host sequence, and *SequenceRightGraph* obtains a pattern sequence. As an example, Fig. 3.8(a) presents a host graph and Fig. 3.8(b) illustrates a production, where subscripts are used to distinguish visual objects of the same class. The corresponding host sequence and pattern sequence are demonstrated in Fig. 3.8(c) and (d) respectively. Moreover, the function

SequenceIndex (see Fig. 3.6) is used to calculate the positions of the visual classes in the pattern sequence. Since a visual class may take more than one position, we use set to represent indices of classes. In the above example, the *If* class holds the first position, and thus *Index(If)*={1}; the *Statement* class occurs twice with *Index(Statement)*={3,2}; and finally *Index(Endif)*={4}, as shown in Fig. 3.8(e).

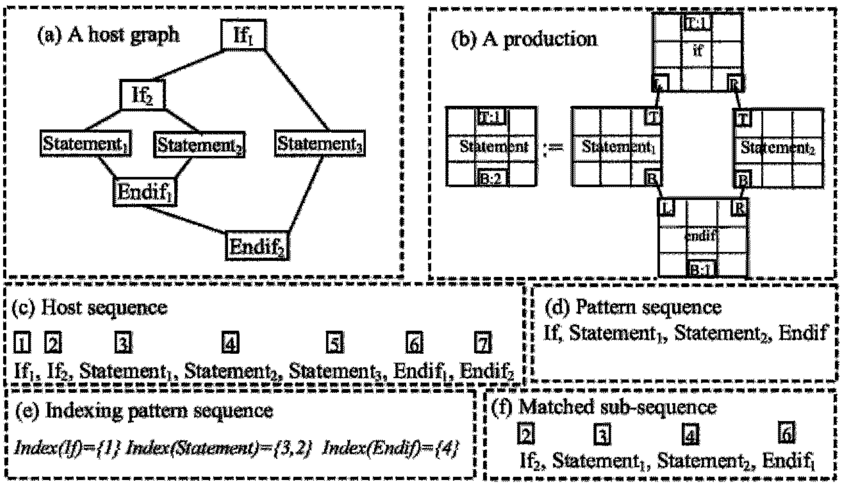


Fig. 3.8. Sequencing host graph and production

3.5.3 A Sequencing Example

Fig. 3.9 traces the execution of *match* for the example in Fig. 3.8. Each object in the host sequence needs to be inspected once before the redex has been found or until all the objects have been inspected.

Initially, all sets are empty. Since the first object of the host sequence can be mapped to the first object of the pattern sequence, a sub-sequence containing only one object is obtained at the 1st iteration. Similarly, a subsequence containing the second object of the host sequence, which is mapped to the first object of the pattern sequence, is generated and inserted into Set 1 at the 2nd iteration.

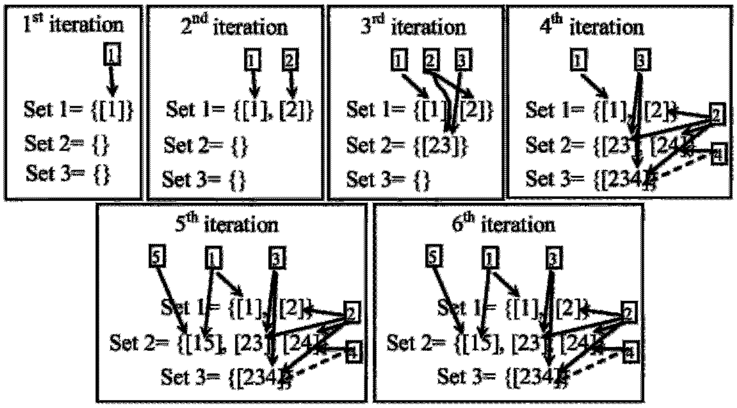


Fig. 3.9. An execution trace of *FindRedexForR* on the example of Fig. 3.8

At the 3rd iteration, sub-sequences in Set 1 are extended by appending the 3rd object of the host sequence. A morphism exists between the first two objects of the pattern sequence and the 2nd and 3rd objects of the host sequence, and thus the sub-sequence [23] is inserted into Set 2. On the other hand, though the sub-sequence [13] matches the first two objects of the pattern sequence, the structural verification has failed: the production requires a structural relationship between the two objects, which does not exist in the host graph.

At the 4th iteration, the 4th object can be mapped to either the 3rd or 2nd object of the pattern sequence. A new sub-sequence [234] is obtained by appending the 4th object to the sub-sequence in Set 2, and it is added to Set 3 since a morphism can be confirmed. Another two sub-sequences [14] and [24] are generated by appending the 4th object to sub-sequences in Set 1. Since no morphism is found between the first two objects of the pattern sequence and the 1st and 4th objects of the host sequence, the sub-sequence [14] is excluded. The 4th object of the host sequence can be mapped to either 3rd or 2nd object of the pattern sequence. Therefore, a dotted edge is used to point to the sub-sequence, in which this object takes the 3rd position, and a solid edge to the other sub-sequence.

Similarly, the sub-sequence [15] is added to Set 2 after processing the 5th object of the host sequence. At the sixth iteration, a pattern sequence, which meets both structural and spatial configurations as shown in Fig. 3.8(f), is found in the host sequence. Since there are only spatial objects in the above example, *FindRedexForR* terminates and a redex is found.

3.6 Complexity Analysis

We now proceed to analyze the parsing complexity of the presented spatial graph grammar formalism. In the host sequence of a host graph, a sub-sequence matching the pattern sequence of a production is called a *quasi-redex*. If the non-pattern objects in the production find their matches in the host graph, the non-pattern objects in the host graph together with the quasi-redex construct a redex.

Theorem 3.1: The time complexity of searching for a redex in SGGs is $O((m+n)^2.\frac{|G|^m}{m!}.|G|^n)$, where *m* and *n* are the maximal numbers of pattern and non-pattern objects in the right graphs of all the productions in the grammar.

Proof: With *m* pattern objects and *n* non-pattern objects of a production, there are $O(\frac{|G|^m}{m!})$ sequences for a quasi-redex and $O(|G|^n)$ sequences for non-pattern objects. Each quasi-redex requires $O(m^2+n^2+mn)$ time to identify a morphism. Therefore, the time of searching for a redex takes $O((m^2+n^2+mn).\frac{|G|^m}{m!}.|G|^n)$ ($\leq O((m+n)^2.\frac{|G|^m}{m!}.|G|^n)$). ■

In particular, we discuss three special cases:

1. *m=0*: No spatial information is specified in the productions, and the SGG degenerates to the RGG. The time complexity is $O(n^2.|G|^n)$, equal to that of the RGG.
2. *n=0:* No non-pattern object is specified in the productions, and the time complexity is $O(m^2.\frac{|G|^m}{m!})$.
3. *m≠0, n≠0*: The productions include both pattern and non-pattern objects, and the time complexity is $O((m+n)^2.\frac{|G|^m}{m!}.|G|^n)$ ($\leq O((m+n)^2.|G|^{(m+n)})$).

Therefore, searching for a redex in the SGG is always more efficient than that in the RGG.

Theorem 3.2: Given a host graph *G* with a grammar, the time complexity of the proposed algorithm searching for a redex is $O(m^2|G|^2+k(m+n)^2|G|^n)$, *m* and *n* are the maximal number of pattern and non-pattern objects in the

right graphs of all the productions in the grammar, and (*k-1*) is the number of quasi-redexes processed before the first redex is found.

Proof: The function *SequenceRightGraph* converts the right graph of a production into a DAG, and finds the longest path of the DAG to represent the pattern sequence. Since there are $O((m+n)^2)$ relationships (i.e. edges), *SequenceRightGraph* runs in $O((m+n)^2)$ time.

The function *SequenceHostGraph* generates a unique sequence from a host graph, and takes $O(|G|\lg|G|)$ to sort the y-coordinates and x-coordinates.

The function *match* searches for a redex in the host graph. It proceeds as follows: 1) look for a quasi-redex in the host sequence; 2) then identify non-pattern objects in the host graph. The i^{th} object in the host sequence is paired with each j^{th} object ($j<i$). We check whether there exists a morphism between each of such pairs and its corresponding pair in the right graph. Since an object can be matched to m positions at most in the pattern sequence, the time complexity of verification is $O(mi) \times O(m)$, i.e $O(m^2i)$. Since there are $|G|$ objects, the time complexity of searching for a quasi-redex is $O(\sum_{i=1}^{|G|} m^2 i) = O(m^2|G|^2)$. Once a quasi-redex is found, we need to identify non-pattern objects in the host graph, which takes $O((n+m)^2)$ (the time of verifying whether there exists a morphism between visual objects in the right graph of a production and their occurrences in the host graph) by $O(|G|^n)$ (the number of sequences of non-pattern objects), i.e. $O((m+n)^2|G|^n)$. Since there are totally (k-1) quasi-redexes before the first redex is found, the time complexity of the second step is $O(k(m+n)^2|G|^n)$. Therefore, the time complexity of *match* is $O(m^2|G|^2+k(m+n)^2|G|^n)$.

Consequently, the time complexity of *FindRedexForR* is $O(m^2|G|^2+k(m+n)^2|G|^n)$. ■

Theorem 3.3: **The time complexity of the parsing algorithm for a graph G is $O(m^2|G|^3+k(m+n)^2|G|^{n+1})$, *m* and *n* are the maximal number of pattern and non-pattern objects in the right graphs of all the productions, and (*k-1*) is the number of quasi-redexes processed before the first redex is found.**

Proof: We have proven that the total time complexity of *FindRedexForR* is $O(m^2|G|^2+k(m+n)^2|G|^n)$. Since $G \longrightarrow^* A$ must finish within G.T(p) steps, where $G.T(p) = (2C)^p|G|$ (full proof for this step can be found in the proof for the RGG's parsing complexity in Chapter 2), the time complexity of the parsing algorithm is $(2C)^p|G| * O(m^2|G|^2+k(m+n)^2|G|^n) = O(m^2|G|^3+k(m+n)^2|G|^{n+1})$. ■

Therefore, due to the additional spatial information available to the parser, parsing spatial graph grammars is generally faster than parsing non-spatial reserved graph grammars.

3.7 Summary

Physical layout and abstract structure are two aspects of a graph. This chapter has presented a spatial graph grammar formalism, which introduces spatial relationship into the abstract syntax as language constructs. In order to automatically verify structural properties of graphs defined through a graph grammar, a parser is indispensable.

This chapter has described in details a deterministic parsing algorithm for the SGG. The parser runs in polynomial time with confluent grammars. Briefly, the parser sequences objects in the right graph of a grammatical rule according to some spatial criterion (e.g. the direction). Then, the same criterion is applied to sequencing objects in a given graph to be parsed. Consequently, sub-graph matching can be performed by searching for the common sub-string of two generated sequences, which satisfies the connectivity requirements. Taking advantage of the spatial information, the SGG parser improves the complexity over its non-spatial predecessor, i.e. the RGG. To avoid backtracking, the parsing algorithm requires that grammatical rules need to be locally confluent.

3.8 Related Work

Rekers and Schürr (1996) classify the spatial relations graph (SRG) and abstract syntax graph (ASR). The former is geared towards visualization, and the latter towards interpretation. A grammatical approach is proposed to build the interdependencies between SRGs and ASRs.

Brandenburg (1995) presents a layout graph grammar consisting of an underlying context-free graph grammar and of layout specifications. Spatial relationships are derived according to the drawing of productions. A desirable layout is achieved by satisfying those constraints. One serious drawback of the approach is that grids and planar graphs cannot be captured by context-free graph grammars (Brandenburg 1995).

Those formalisms explore spatial relationships merely from the layout perspective. On the other hand, in the SGG, spatial relationships directly contribute to the interpretation of a graph, and are considered as language constructs like nodes and edges. Furthermore, with the extended expressive power, the SGG is also capable of generating a syntax-directed layout through a sequence of graph transformations, in which the spatial information of the post-condition specifies a desirable layout among involved objects.

Picture description formalisms, e.g. the *constraint multiset grammar* (*CMG*) (Marriott 1994), provide a high level framework for the definition of visual programming languages. An incremental bottom-up parsing algorithm is proposed to verify the membership for cycle-free CMGs. The algorithm encodes a CMG as a logic program by adding an extra attribute to each token which represents the multiset of terminal tokens.

Many proposed graph grammars, such as the NLG graph grammar (Engelfriet and Rozenberg 1997), 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 (1988) investigates the complexity of node rewriting graph grammars, and concludes the character distinguishing polynomial time graph grammars. Those parsing algorithms are based on context-free graph grammars.

Being a context-sensitive graph grammar, the *Layered Graph Grammar* (*LGG*) is equipped with an exponential parsing algorithm (Rekers and Schürr 1997). Instead of exhaustive search, the parser uses a *breath-first search algorithm* such that possible sub-derivation are constructed and extended in parallel instead of re-computing them multiple times. Filters are used to discard useless sub-derivation as soon as possible.

Extending the LGG, Bottoni et al. (2000) proposes the *Contextual Layered Graph Grammar* (*CLGG*), 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 (Plump 1993). 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 (Bottoni et al. 2000).

Chapter 4 Multimedia Authoring and Presentation

4.1 Introduction

With the rapid advance of the Internet and Web technology, an increasing amount of graphs and media contents are delivered on the Web. On-line multimedia presentations, such as news, need to be constantly updated. The content and the presentation structure of an on-line multimedia presentation may also be frequently updated. At the client side, there are various kinds of viewing conditions, such as varying screen size, style preference, and different device capabilities. For example, consider a diagram representing an organizational structure on the Web that may be of considerable complexity occupying a large screen space, and thus may be unsuitable for small displays (Marriott et al. 2002). Thus, if the diagram is to be viewed on the screen of a mobile device, such as a PDA (Personal Digital Assistant), the original diagram layout may not be appropriate. Another example is a news Web site, which generally needs to be constantly updated with the incoming news items. Such a site may have to adapt itself frequently to the changing space and style requirements for different news categories. The ability of dynamically adapting its layout would be highly desirable. There are also increasing demands for accessing on-line multimedia documents from mobile devices such as PDAs

With the current document markup languages such as HTML and WML, the layout of a Web page is relatively static and fixed (Borning et al. 2000). When the user's requirement or the device capability is changed, the layout may become unsatisfied. The reason is that such markup languages do not provide any mechanism powerful enough for specifications to be adaptable to the changing context. Though SMIL (W3C 2001) and CSS (W3C 2004b) provide more flexible markups for multiple alternative layouts, the markups provide absolute layout functionality, rather than adaptive to the user's intention or the existing layout. Therefore, a meta-level design mechanism capable of adapting multimedia presentations in response to the dynamic changes in information content is highly desirable. We need a sound but practical formalism that supports automatic adaptation to the change of media contents, display environments, and the user's intention.

To illustrate the concept of multimedia adaptation that we perceive, we use Ishizaki's schematic diagram of a process between content creation and information reception (Ishizaki 2003) as depicted in Fig. 4.1. The design system should be able to adapt itself to the changes in information content and in individual users' intentions. As mobile devices provide an increasing proportion of on-line content accesses, we argue that a multimedia authoring system should support an additional type of context changes – i.e. adaptation to the change of device capabilities. In other words, the designer of a multimedia system needs to be able to specify how the presentation would evolve based on the change of environments (e.g. from a desktop screen to a mobile display panel), user's intention (e.g. zooming in or out), and information content (e.g. news update).

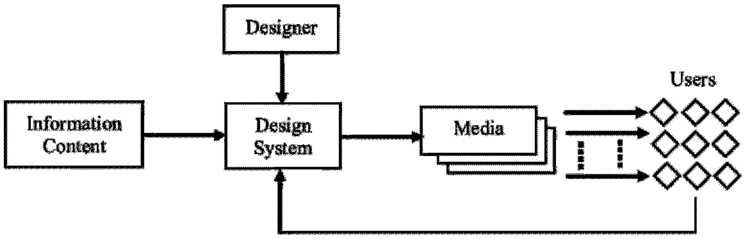


Fig. 4.1. Multimedia presentation design and delivery process

This chapter presents a visual language approach, specifically a spatial graph grammar, for adaptive multimedia authoring and presentation. It focuses on the issues and techniques for size adaptation and style adaptation in response to the change of device requirements and user's interactions. The approach is highly intuitive yet also sound in theory. The central theme of this chapter is to demonstrate how to use a graph grammar formalism to visually specify and support automatic transformation and adaptation of multimedia presentations. The approach has two major advantages due to its meta-tool capability: a graphical authoring tool can be automatically generated by a visual language generator, such as VisPro to be described in Chapter 8; and the generated authoring tool can be used by novices who have no computing knowledge.

The chapter is organized as the following. Section 4.2 introduces the spatial grammatical representation and specification and focuses on adaptation techniques to support size and style changes of multimedia presentations. Sections 4.3 and 4.4 demonstrate size and style adaptations by going through two real-world examples. Section 4.5 presents a system architecture implementing the grammatical approach. Section 4.6 summarizes the chapter. Finally, relevant materials for further reading are reviewed in Section 4.7.

4.2 Adaptation to Context Changes

As discussed at the beginning of the chapter, context changes may be due to the change of information contents such as a traffic monitoring system, device capabilities such as from a desktop screen to a PDA panel, or the viewer's intention. A graph grammar based approach is able to adjust the appearance to different displaying environments.

This section outlines automatic adaptation of the size and style of a multimedia presentation in response to any of these changes, though other aspects of adaptation may also be supported by the graph-grammar approach. Detailed example adaptive presentations will be discussed in Sections 4.3 and 4.4.

4.2.1 The Marking Scheme

As described in Chapters 2 and 3, the marking mechanism plays a central role in the specification and parsing of reserved and spatial graph grammars. To apply the marking technique to multimedia adaptation, we will demonstrate the power of spatial graph grammars. Consider a simple example in graphical presentation: a vertical layout provides a different visual perception and requires a different screen (usually smaller) estate from a horizontal layout, as shown in Fig. 4.2. This is one of the most common issues in graphical design and can be effectively applied to transforming Web graphics to suit small screen mobile devices. Fig. 4.3 depicts the rewriting rule (production) for this required transformation.

Since there may be multiple nodes chained in the same direction, we mark the vertices on the both ends of the two nodes by attaching unique integers to the vertex labels (i.e. "N:1" and "S:2"). This means that during transformation, the edges connected to both ends will be reserved. The direction change from horizontal to vertical is reflected in the positions of the vertices, as explained in Section 4.2.3. The edges between the nodes will be shortened after transformation, as specified by "-S" and "-N".

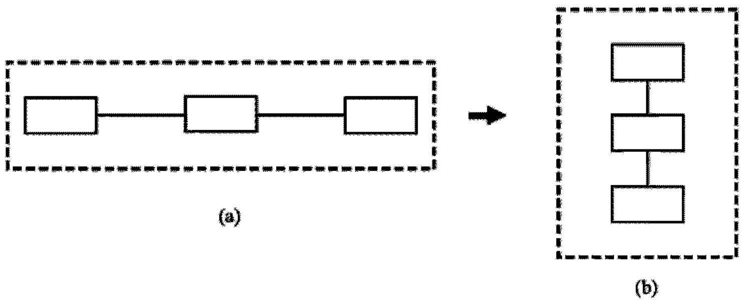


Fig. 4.2. Horizontal layout (a) transformed to vertical layout (b)

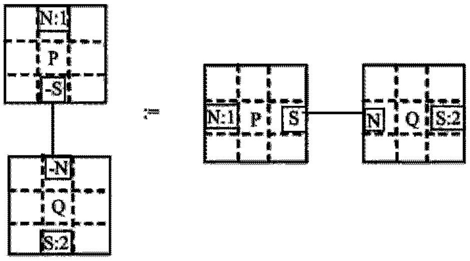


Fig. 4.3. A production for the transformation in Fig. 4.2

4.2.2 Size Adaptation

The most typical application of size adaptation is for Web display layout to be reduced to suit mobile devices. The simplest solution to the problem of the limited screen size is *linear scaling* (or *normal zooming*), but this is often not the best way. A more elaborate technique is *differential scaling*, in which different components of a document are scaled differently. *Differential scaling* is effective in compressing white spaces. For example, rather than performing a linear scaling, each white space is compressed, while the box sizes are maintained (Marriott et al. 2002), as illustrated in the simple example in Fig. 4.4. To specify such a transformation, we can use distance relationships as shown in Fig. 4.5.

To represent the change of a node size, we use "+" in the node's center box to indicate that the node will become larger (or zoomed in as discussed below) in the transformation, "-" for smaller (zoomed out), and blank for unchanged size.

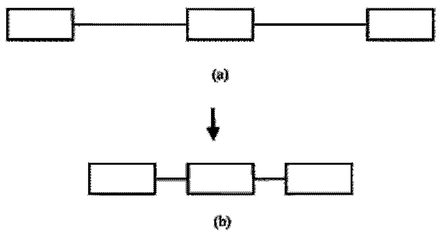


Fig. 4.4. An example of differential scaling

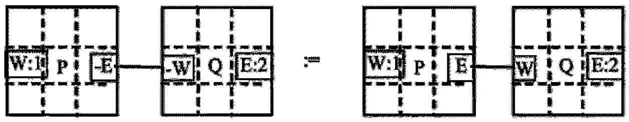


Fig. 4.5. Grammar specification of differential scaling in Fig. 4.4

4.2.3 Style Adaptation

To suit different display spaces and devices, the layout of individual media objects and that of the entire document may need to be adapted. One of such adaptation techniques is known as *alternative layout*. Fig. 4.6 illustrates a typical example of alternative layout. Originally object *B* is on the right of object *A*. After transformation, as in Fig. 4.6(b) and Fig. 4.7(b), object *B* is at the bottom of object *A*, thus the locations of vertices *E* and *W* have also changed.

Another type of multimedia style adaptation is called *semantic zooming* (Marriott et al. 2002). For varying interest in detail, an adapted layout may initially show one level of details. It allows the viewer to zoom in hierarchically, while adapting the layout level of each individual component or group of components to the available screen size or to the viewer's preference. For example, we may need to enlarge one part, in which the user is particularly interested, while compressing unrelated parts, as illustrated in Fig. 4.6. We need to look into the detail of object *A* first, so we may view the details of *A* and *B* separately. Fig. 4.6 illustrates the combined effects of alternative layout and semantic zooming. Fig. 4.7 depicts the snapshots of using grammatical rules to achieve the style transformation from (a) to (b), including the reduced size of B, and shortened distance between A and B in (c).

In some systems (Muchaluat et al. 1998), the above viewing technique is called *fisheye* view. More commonly accepted concept of fisheye views refers to the geometric distortion technique when highlighting a focused area of a large display (Sakar and Brown 1994). Geometric distortion enlarges the focused area while proportionally reducing other areas depending on their distances to the focused area. Hyperbolic trees (Herman et al. 2000) offer another similar viewing technique, widely used for Web browsing.

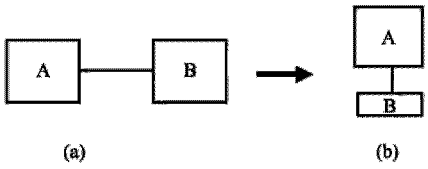


Fig. 4.6. Semantic zooming with an alternative layout

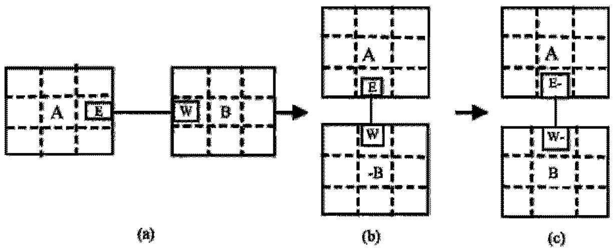


Fig. 4.7. Application of semantic zooming and distance rules to achieve the effect in Fig. 4.6

4.3 Example 1: Adapting Sizes for PDA Displays

This section and the next section focus on two detailed examples of size and style adaptations through grammatical specifications and graph transformations. This section describes how to transform a desktop Web page to several small pieces for mobile Web browsers – an example of size adaptation.

4.3.1. Original Web and Resulting PDA Presentations

Fig. 4.8 shows the popular NASA home page, whose size and layout may be adapted to suit small screen Web browsers. We will transform this page into the WML format to be displayed on PDAs. The XML description for the above Web page is as the following:

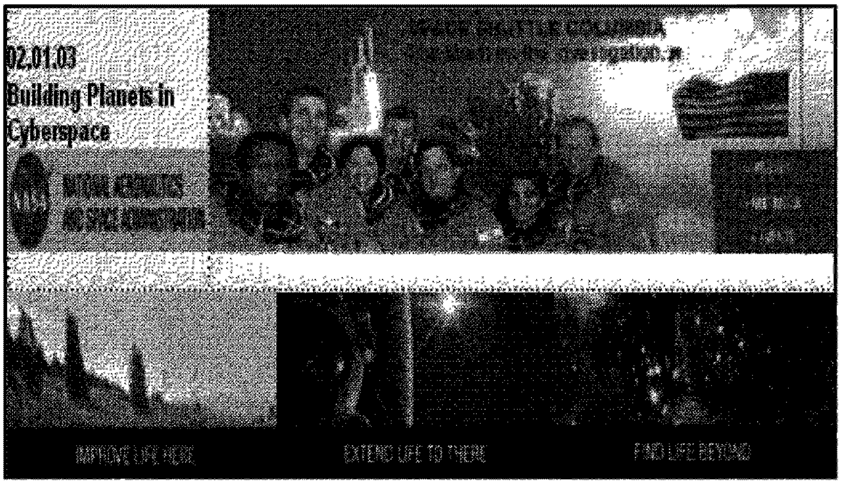


Fig. 4.8. The original NASA home page

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<page>
   <section1>
      <block1>
         <Logo>
            <pic>
               <id> nasa </id>
               <source> ./images/nasa.bmp </source>
            </pic>
         </Logo>
         <text>
            02.01.03 Building Planets in Cyberspace
         </text>
      </block1>
      <block2>
         <theme>
            <pic>
               <id> shuttle </id>
               <source> ./images/shuttle.bmp </source>
            </pic>
         </theme>
         <button>
            <link>
               <pic>
                  <id> missions </id>
```

```
                <source> ./images/missions.gif </source>
              </pic>
              <href>
                  http://www.nasa.gov/missions/current/
              </href>
            </link>
              ......
          </button>
        </block2>
    </section1>
    <section2>
        <pic>
          <id> improve life </id>
          <source> ./images/improvelife.bmp </source>
        </pic>
        ......
    </section2>
</page>
```

Assume the desirable outcome as illustrated in Fig. 4.9. We divide the original Web page into four small pages based on the four images, and copy the top-left heading information and top-right hyperlinks to all the small pages. As a result, each small page contains three parts: the top part contains date and title (tagged "Text") and NASA logo ("Logo"), the middle part is an image ("Theme" or "Picture") and the bottom part contains three hyperlinks ("Link").

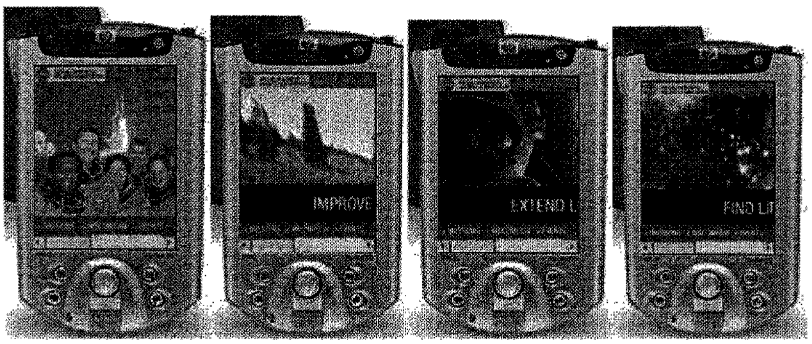


Fig. 4.9. Resulting presentation as four pages on a PDA

The output tree structure is translated into a WML document. Each page or a single interaction between a user agent and a user is known as a *card*. One advantage of this arrangement is that multiple screens can be downloaded to a client in a single retrieval and vice versa. Our task is simply to transform the XML description into several cards, each to be displayed as a PDA page. The following is part of the WML document for the PDA presentation in Fig. 4.9, where "card" represents a separate page:

```
<wml>
   <card id="section1" Title="nasa">
      <p>
         <img src="./images/nasa.jpg" alt="earth"/>
         02.01.03 Building Planets in Cyberspace
      </p>
      <p>
         <img src="./images/shuttle.bmp" alt="shuttle"/>
      </p>
      <p>
         <img src="./images/missions.bmp" alt="shuttle"/>
         <img src="./images/multimedia.bmp" alt="shuttle"/>
         <img src="./images/events.bmp" alt="shuttle"/>
      </p>
   </card>
   <card id="improve" Title="improve life">
      <p>
         <img src="./images/nasa.jpg" alt="earth"/>
         02.01.03 Building Planets in Cyberspace
      </p>
      <p>
         <img src="./images/improvelife.bmp" alt="improve life"/>
      </p>
      <p>
         <img src="./images/missions.bmp" alt="shuttle"/>
         <img src="./images/multimedia.bmp" alt="shuttle"/>
         <img src="./images/events.bmp" alt="shuttle"/>
      </p>
   <card>
   ........
</wml>
```

4.3.2 Structural Transformation

Each Web page is a multimedia document that has a layout constructed by many media objects. We start by analyzing the logical structure (automatically generated as a tree) and desired layout and adaptive properties of the given Web page. In the Spatial Graph Grammar, each object is presented by a node. A Web page in XML is a tree structure whose elements can be grouped hierarchically, as shown in Fig. 4.10 for the given example. To convert the tree to a more structured arrangement suitable for transformation, we need to introduce the concepts of logical nodes and grouping. The tree contains several logical nodes (LNs) such as *Page, Section1, Section2*, etc. As the root, *Page* contains two *Section* nodes. *Section1* contains two *Block* nodes. *Block1* contains *Logo* and *Text*, and *Block2* contains *Theme* and *LN Button*. *LN Button* includes a number of *Link* nodes. *Section2* has a number of child nodes, called *Pictures*. Such hierarchical relationships can be automatically derived from the XML document and used to generate the data structure in Fig. 4.11.

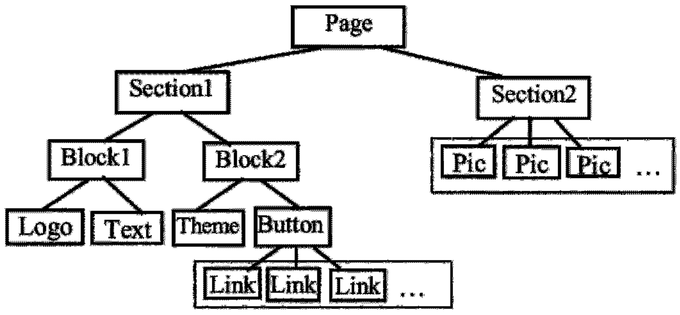


Fig. 4.10. Tree structure of the Web page

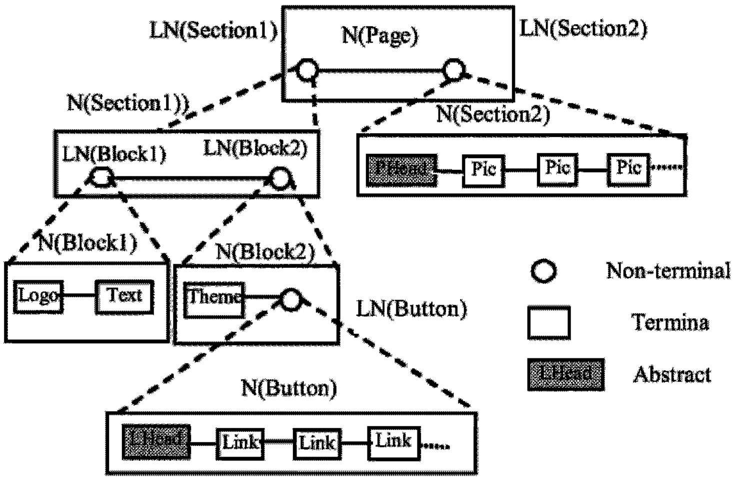


Fig. 4.11. A hierarchical data structure

We use an abstract node to head a group that has many objects of a single type. Such a group header has a generic set of attributes applicable to the whole group. Each group member inherits from its parents' attributes such as vertices with spatial information. This arrangement improves the presentation efficiency. Using the concepts of groups and *LNs*, we only need to consider spatial relations of a node with its parent, child and sibling nodes (i.e. direct relatives). For example, we will consider the relationship between the siblings *LN Block1* and *LN Block2*, but not the relationships between the children of *N(Block1)* (i.e. *Logo, Text*) and those of *N(Block2)* (i.e. *Theme*, *N(Button)*). Combining the spatial information from Fig. 4.8

and above logical and hierarchical information from Fig. 4.11, the host graph in Fig. 4.12(a) can be automatically generated to be processed by the spatial graph grammar. The application of the SGG generates the new layout structure in Fig. 4.12(b) for PDA presentations, as explained in the next subsection.

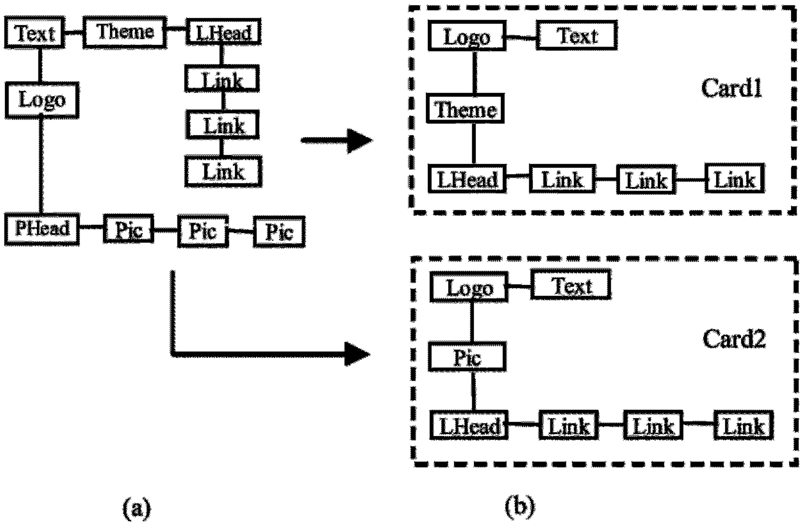


Fig. 4.12. (a) Host graph of the original structure (b) The resulting layout structure

4.3.3 Grammatical Specification

In order to perform the desired transformation, we define a set of productions as illustrated in Fig. 4.13. There are two right graphs for some productions. The right graph not enclosed in a dashed box participates in syntactical parsing, and, together with the left graph, will be called a *syntax production* or simply *S* in the following description. The right graph enclosed in a dashed box is used for the layout transformation, and, together with the left graph, will be called a *layout production* or simply *L*. A set of *L* productions generates a new layout either from an existing layout or from logical relationships between media objects.

Syntax Productions

Syntax production <1> (or simply S<1>) expresses the initial state. If a parsing eventually reaches the state λ (initial state), it is regarded as successful (Zhang et al. 2001a).

S<2> illustrates that such a page (NASA Homepage) consists of *Card* and *PHead*, and *Card* is on the top of *PHead*. S<3> abstracts a *Card* from *Section1*.

S<4> specifies that *Section1* contains two blocks, and *Block1* is side by side with *Block2*. The vertex in gray color in a node means that it is marked and will be reserved during parsing. For example, the vertex labeled *P* is marked, and will stay unchanged after parsing.

S<5> specifies that *Block1* consists of *Text* and *Logo*, and *Text* is directly on the top of *Logo*. The vertices labeled *P* and *D* are marked.

S<6> indicates that *Block2* includes *Theme* and *LHead*. *LHead* is a *Group Header* in the *Link* structure, and used to inherit the attributes from its parents. If the *Link* structure contains many members, using *LHead* will significantly improve the efficiency of the graphical presentation. To represent the containing relationship between *Theme* and *LHead*, we use dotted boundary in *LHead*, and connect the two nodes' central grids.

S<7> specifies that the *Link* structure consists of several terminal nodes of *Link*, stacked on top of each other.

S<8> and S<9> indicate that *Section2* includes several *Pictures*. In S<8>, *PHead* and *Card* can be reduced to *PHead*. *Card* is an intermediate node and can be abstracted from *Picture* (*Pic* for short) by using S<9>. We can apply S<9> continuously until no terminal node exists.

The R-application in the SGG is a parsing process, which in general consists of: selecting a production from the grammar and applying an R-application of the production to the host graph, and the process continues until no productions can be applied. If the host graph is transformed into an initial graph λ, the parsing process is successful and the host graph belongs to the language defined by the graph grammar. We first use S<9> and S<8> to reduce the *Picture* structure to *PHead*. S<7> is used to reduce the *Link* structure to *LHead*. S<6> is then used to reduce *LHead* and *Theme* to *Block2*, and S<5> to reduce *Logo* and *Text* to *Block1*. Then we use S<4> to obtain *Section1*. Finally, S<2> reduces *Card* and *PHead* to *Page* and S<1> to λ, and thus the parsing process is successful.

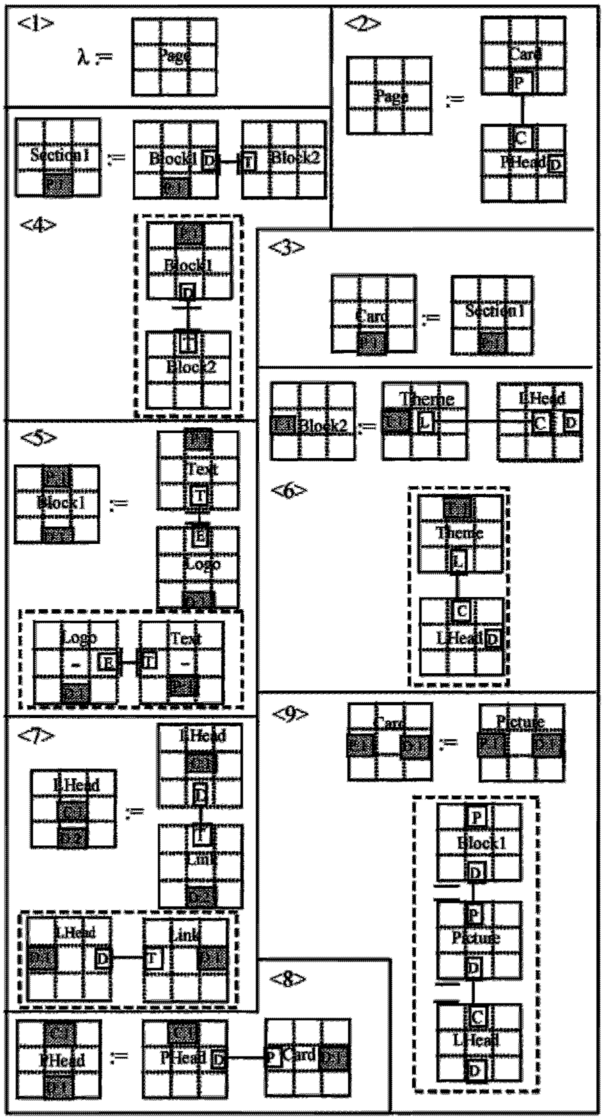


Fig. 4.13. Productions for the transformation from the presentation in Fig. 4.8 to the one in Fig. 4.9

Layout Productions

Based on the above syntax productions for parsing the host graph, we add several extended productions enclosed in dotted boxes called layout productions for transforming the presentation in Fig. 4.8 to the one in Fig. 4.9. The layout productions are thus an additive set to the syntax productions. Combining these two sets of productions, we can generate the desirable layout.

Layout production <4> (or simply L<4>) transforms *Block1* and *Block2* from the horizontal relationship to vertical relationship with *Block1* on top of *Block2*.

L<5> transforms *Text* and *Logo* from a vertically touching relationship to a horizontally touching relationship.

L<6> specifies how to transform two objects from a containing relationship to a vertical relationship. Before the transformation, *Theme* contains *LHead*. After the transformation, *Theme* is on the top of *LHead*.

L<7> transforms a sequence of *Links* from vertically touching relationships to horizontally touching relationships that is repeatedly applied.

In L<9>, when *Picture* with left and right vertices finds a match, it is converted to a *Block1-Picture-LHead* structure, whose three nodes are vertically aligned along the left edges.

We first parse the host graph in Fig. 4.12(a) to λ. During parsing, a stack is used to record the sequence of the productions being used. Then from λ, the original parsing tree is retrieved. At each step, the corresponding layout productions are popped from the stack to perform layout transformations. For example, when *Card* with its southern vertex is matched, S<3> is used to generate *Section1*. Then, we use L<4> to obtain a new layout in which *Block1* and *Block2* hold a vertical relationship. For *Block1*, L<5> is used to derive a horizontal relationship between *Logo* and *Text*. Using L<6>, *Theme* is moved to the top of *LPHead*. L<7> is used to obtain the horizontal *Link* structure. Now we obtain the first PDA page, represented as *Card1* in Fig. 4.12(b). L<9> is used to expand *Card* to the *Block1-Picture-LHead* structure. *Logo* and *Text* are then generated using L<5> and the *Link* structure generated using L<7>. We therefore obtain the second PDA page (marked *Card2* in Fig. 4.12(b)). The third and fourth pages, also of the *Card2* structure, are generated in the same fashion. The layout in Fig. 4.12(b) can be automatically transformed to the final layout illustrated in Fig. 4.9.

4.4 Example 2: Adapting Presentation Styles

This section provides a further example, on the adaptation of presentation styles.

4.4.1 A Presentation Style

As an example for style adaptation, consider an art museum that organizes its multimedia documents in a pre-determined logical structure, as shown in Fig. 4.14. At one exhibition season, the museum would like to display the documents on the Web as displayed in Fig. 4.15 with the following presentation organization. The page consists of a menu bar of various hyperlinks on the left side for the whole museum, hyperlinks to all the curatorial departments on the top, and the collection highlights occupying the main page area. The highlights of each museum department consist of a number of well-known art works (i.e. pictures). Assuming the pictures need to be displayed with 3 in each row, six pictures of the selected "Painting and Sculpture Highlights" are displayed in 2 rows by 3 columns in the main area.

4.4.2 Grammatical Specification

The logical structure of Fig. 4.14 is regarded as a host graph, which is used to dictate the presentation layout according to a grammar specification. The complete set of graph grammar production rules that meet the requirements of the presentation style of Fig. 4.15 is listed in Fig. 4.16. The document mainly consists of two composite objects, *Sections* and *Content*, which participate in Production <2> in Fig. 4.16.

The *Sections* object consists of a number of hyperlinks, which enable the user to navigate other museum documents from the same page. The hyperlinks are organized hierarchically. A link at level *i* may include several links at level *(i+1)*. The links at the same level are aligned to the left, and level *(i+1)* links are indented from its level *i* links. Production <3> abstracts a terminal node called *Link* to a non-terminal node *Section*. Production <4> dictates how to reduce two links (represented by two *Section's*) while establishing their spatial relations – vertically aligned and touched with each other. Production <5> is for reducing the last *Section* node, which is characterized by an unmarked *N* vertex in *Section*. Production <6> demonstrates how to attach hyperlinks (upgraded from *Link* to

Section) at level *(i+1)* to a hyperlink at level *i* with right indentation (realized by a partially touched relationship as shown in the right graph of the production).

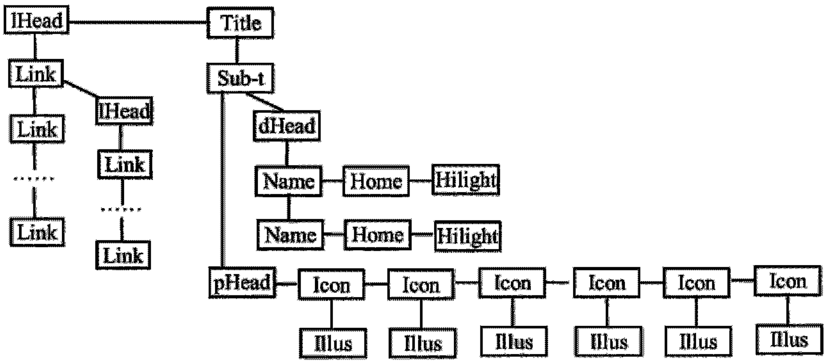
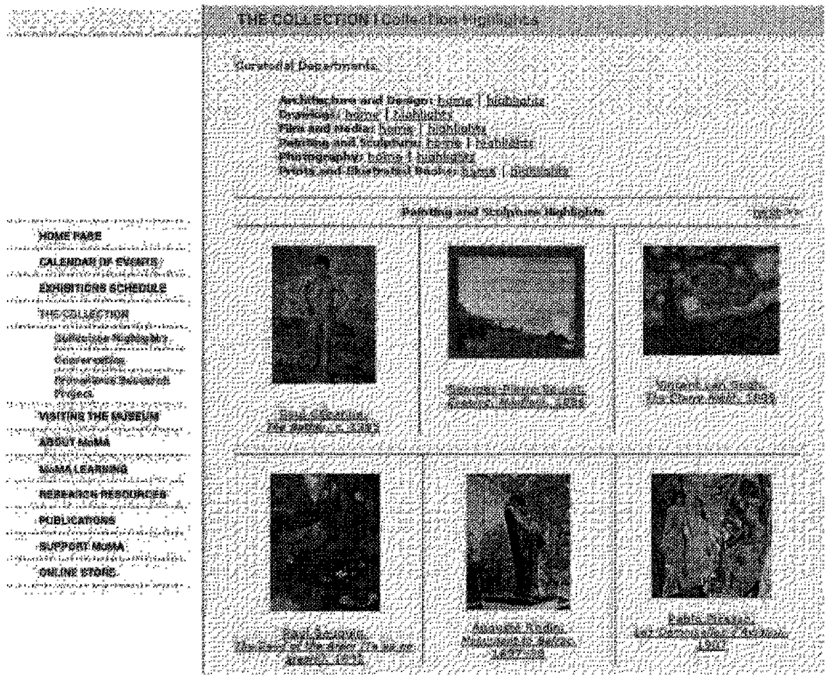


Fig. 4.14. Host graph of the Museum multimedia document


THE COLLECTION | Collection Highlights

Curatorial Departments

Architecture and Design: home | highlights
Drawings: home | highlights
Film and media: home | highlights
Painting and Sculpture: home | highlights
Photography: home | highlights
Prints and Illustrated Books: home | highlights

Painting and Sculpture Highlights

HOME PAGE
CALENDAR OF EVENTS
EXHIBITIONS SCHEDULE
THE COLLECTION
Collection Highlights
Conservation
Provenance Research Project
VISITING THE MUSEUM
ABOUT MoMA
MoMA LEARNING
RESEARCH RESOURCES
PUBLICATIONS
SUPPORT MoMA
ONLINE STORE

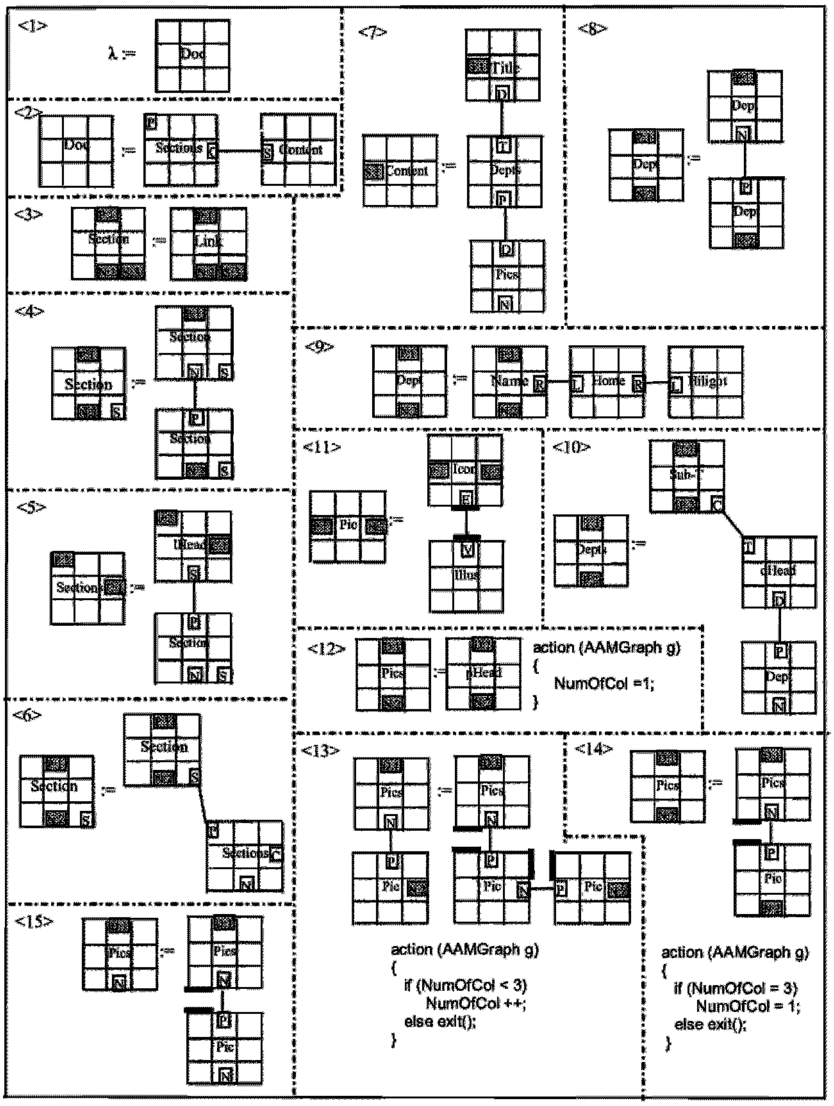
Paul Cézanne, The Bather, c. 1885
Georges-Pierre Seurat, Evening, Honfleur, 1886
Vincent van Gogh, The Starry Night, 1889
Paul Gauguin, The Seed of the Areoi (Te aa no areois), 1892
Auguste Rodin, Monument to Balzac, 1897–98
Pablo Picasso, Les Demoiselles d'Avignon, 1907

Fig. 4.15. A museum multimedia presentation

The right part of *Context* consists of three objects, *Title*, *Depts* and *Pics* as specified in Production <7>. The *Title* object presents the title of the document, and is placed above the two other composite objects. The *Depts* object, representing a list of departments in the museum, consists of multiple *Dept* objects to be aligned to the left and vertically touched with each other. As shown in Production <9>, each *Dept* object consists of three primitive objects, *Name*, *Home* and *Hilight*, which are to be touched with each other and aligned horizontally. Production <8> reduces two *Dept* objects into one, and Production <10> provides the syntax of a *Depts* object.

A *Pic* object represents a picture displayed in the main area and has an *Icon* and an *Illus* objects as specified in Production <11>. The *Icon* object is placed above the *Illus* object, and both are aligned to the center. Representing the highlights of a department to be displayed in the main area, a *Pics* object is abstracted from a *pHead* object in Production <12>, and consists of multiple *Pic* objects to be laid out according to Productions <13> and <14>. By applying Production <13> repeatedly to reduce two *Pic* objects into one *Pic* object, a sequence of *Pic* objects is generated.

Since it is required that three *Pic* objects are displayed in each row, we need to use *action* codes to specify the constraint. An action code associated with a production is like a Java exception-handling method, used to specify the semantics of the production and to provide additional control information to the parser. We introduce a global variable, called *NumOfCol*, to record the current number of the *Pic* objects in the currently row. Initially, *NumOfCol* is set to 1 in Production <12>. Every successful application of Production <13> increases *NumOfCol* by 1 until its value reaches 3. Production <14> is applied only when *NumOfCol* is equal to 3, indicating that there are already three *Pic* objects in a row. However, the last row may contain less than 3 *Pic* objects. Production <15> handles such a special case. Apart from the action code (Production <15> has no action code), the only difference between Production <14> and <15> is that the *N* vertex is marked in <14> and unmarked in <15>.



```
action (AAMGraph g)
{
    NumOfCol =1;
}
```

```
action (AAMGraph g)
{
  if (NumOfCol < 3)
      NumOfCol ++;
  else exit();
}
```

```
action (AAMGraph g)
{
  if (NumOfCol = 3)
      NumOfCol = 1;
  else exit();
}
```

Fig. 4.16. Graph grammar definition of both the document structure in Fig. 4.14 and presentation in Fig. 4.15

4.4.3 Adapting to An Alternative Style

Assume in another exhibition season, the museum home page will be presented in an alternative layout as illustrated in Fig. 4.17, where the main area has a different arrangement. It would be time-consuming to manually adjust the layout of each page for a large number of similar pages and also error-prune. Fortunately, the spatial graph grammar provides an adaptive approach to document presentation since the system can select an appropriate set of productions and automatically generate a desirable layout when the context is changed. To support the above alternative presentation style, what is needed is simply a subset of new productions that will replace Productions <11> to <15>, as listed in Fig. 4.18. By applying the alternative productions, the document not only displays two *Pic* objects each row, but also interleaves the *Icon* and *Illus* objects. Other presentation styles could also be easily adapted by modifying the relevant part of the grammar. A typical example is when the width of the viewing device is too narrow to fit three *Pic* objects, the layout can be adjusted to two *Pic* objects or even one in each row.

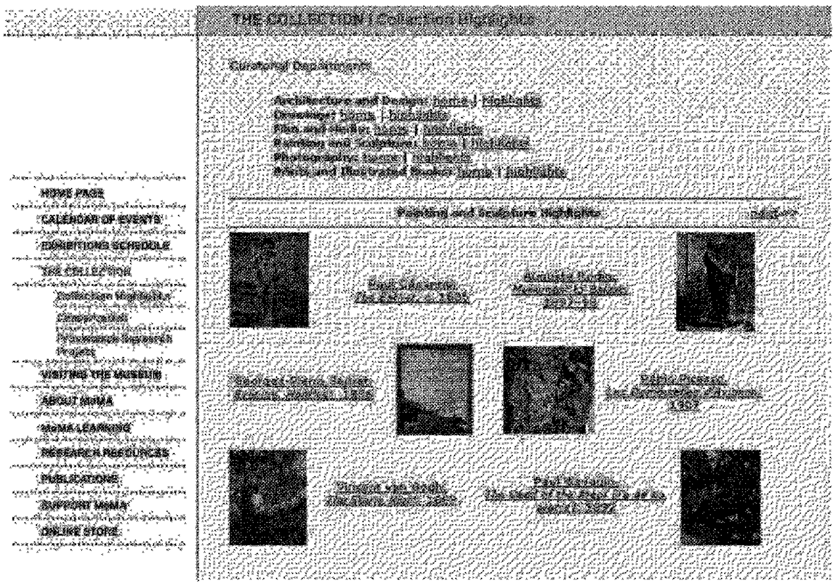

THE COLLECTION | Collection Highlights
Curatorial Departments
Architecture and Design: home | highlights
Drawings: home | highlights
Film and Media: home | highlights
Painting and Sculpture: home | highlights
Photography: home | highlights
Prints and Illustrated Books: home | highlights
Painting and Sculpture Highlights
HOME PAGE
CALENDAR OF EVENTS
EXHIBITIONS SCHEDULE
THE COLLECTION
Collection Highlights
VISITING THE MUSEUM
ABOUT MoMA
MoMA LEARNING
RESEARCH RESOURCES
PUBLICATIONS
SUPPORT MoMA
ONLINE STORE

Fig. 4.17. An alternative presentation style

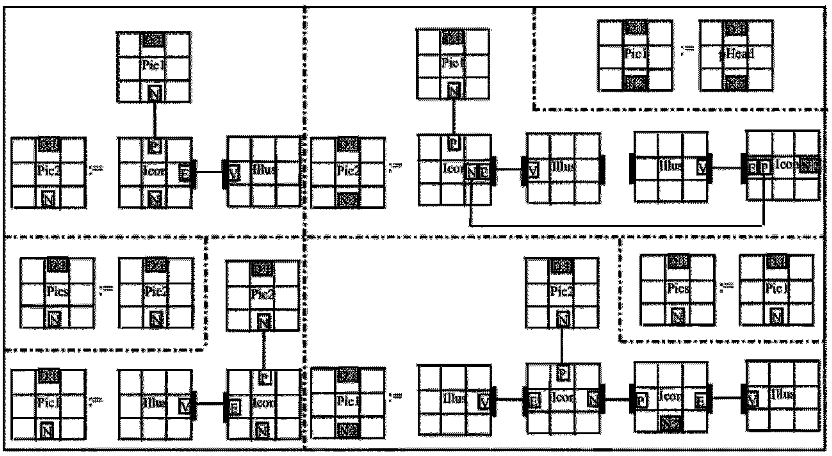


Fig. 4.18. Revised subset of productions (replacing Productions 11-14 in Fig. 4.16) for generating the alternative presentation style of Fig. 4.17

4.5 System Architecture and Implementation

At the system level, the grammatical approach described above is realized by four modules as shown in Fig. 4.19: event encoding, event listener, production authoring and parser. The event-encoding module lets the user describe the events, to which the grammar should be sensitive. The event listener dynamically monitors the system to see if any changes have occurred due to the content update or user interactions. Upon the user's inputs and messages retrieved from the event database by the event listener, the parser performs the corresponding graph transformation according to the predefined graph grammar.

The production authoring module provides a tool to define a graph grammar according to the desired document layout and its dynamic behavior. The grammar dictates how to construct a multimedia document layout through various types of media objects as described in the previous sections. A production not only specifies how to construct composite objects, but also how the constructs look like and adapt to dynamic changes.

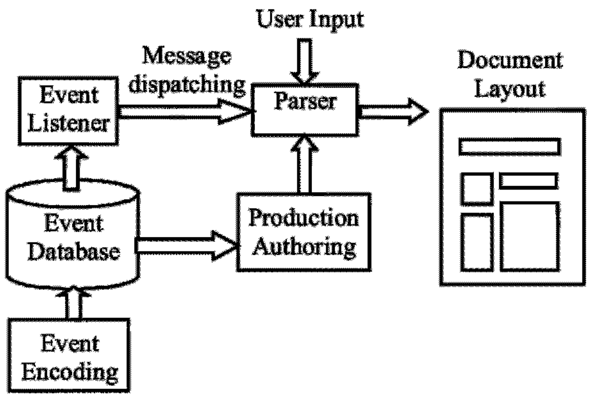


Fig. 4.19. A system architecture

The parser validates the structure of a host graph, and automatically generates a parsing tree, which reflects the hierarchical structure. Also, the layout is adjusted according to the spatial specifications, which are integrated with the structural specifications. For example, when the user modifies the font sizes, or device characteristics, a message is dispatched to the parser, and a conditional (even-driven) production may be triggered to perform a graph transformation. The positions and styles of objects are adjusted according to the spatial specifications in the grammar. During the process of graph transformation, some objects may collectively construct a composite object, which is treated as one entity whose position change in the later layout process will not affect the spatial relationships among its internal objects.

When defining grammar productions for graph layout where edges represent only geometric relations, we allow only one relation between any pair of nodes. Such relationships can be efficiently handled by the original RGG formalisms. The graph grammar formalism with spatial specification mechanisms is sufficiently expressive in specifying multiple connectivity and complex presentation structures.

The system implementation is based on a visual language generation framework. The framework is essentially a meta-tool for automatic generation of visual specification tools (Zhang et al. 2001a), with which different multimedia authoring and presentation languages can be automatically generated according to varied requirements specified through spatial graph grammars. For the example in Section 4.3, we can specify original document structure through syntax productions as shown in a snapshot of Fig.

4.20 (i.e. production authoring module in Fig. 4.19, events are not used in this example). The production without spatial specification is slightly different from the full version of Fig. 4.13. The meta-tool then automatically generates the language environment with a graphical editor and a parser. A user can then use the graphical editor to draw an application document structure and provide desired texts and attributes, as illustrated in the snapshot in Fig. 4.21. When the compiler is triggered from the menu on top of the editor, the resulting document structure is visualized (Fig. 4.22) and WML document listed in Section 5.1 is generated.

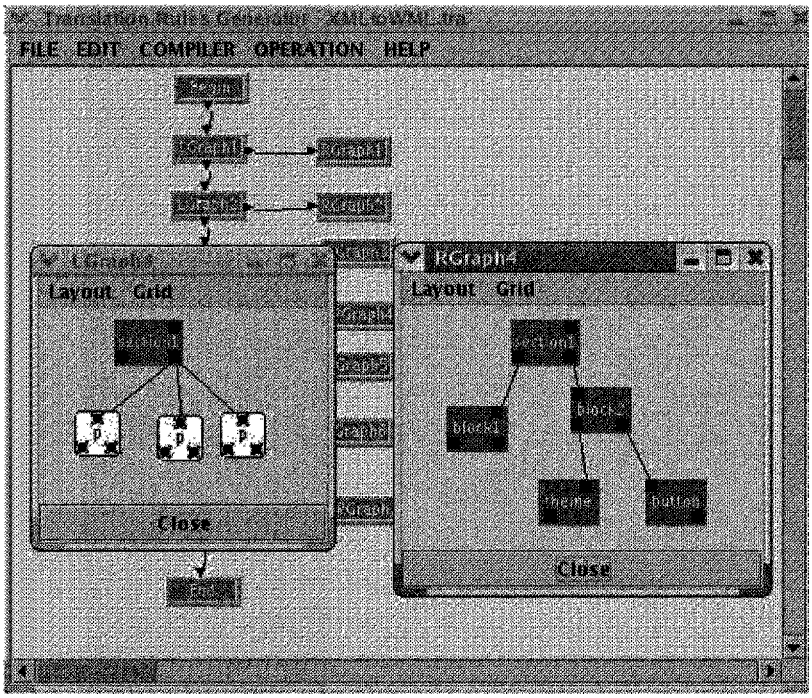
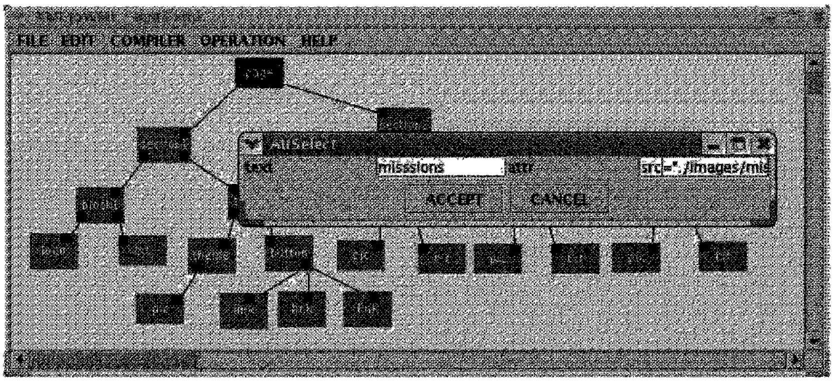


Fig. 4.20. Specifying productions using Rule Generator



text missions attr src="./images/mis

Fig. 4.21. Specifying original document structure using the generated graphical editor

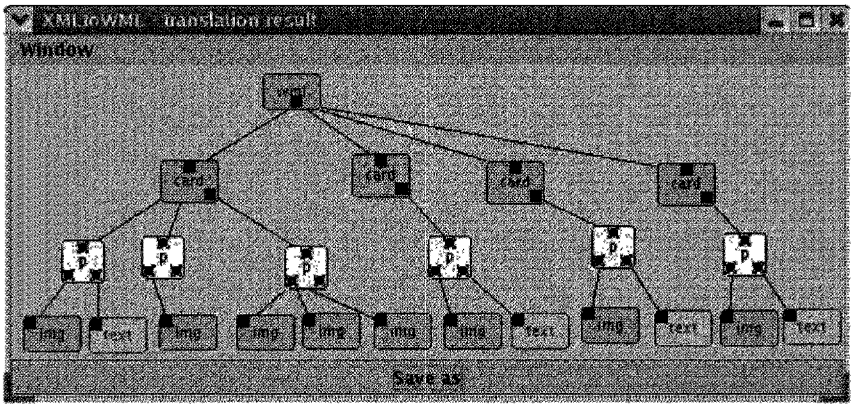


Fig. 4.22. Automatically translated document structure

4.6 Summary

The grammatical approach is promising in providing a powerful mechanism to represent the layout structure graphically and to perform an online validation and adaptation through an automatically generated parser. This chapter has presented the concept of applying graph grammars to the transformation of multimedia presentations to achieve automatic adaptation to the change of media contents, different layout requirements and user interactions. Such transformations usually involve location change, differential scaling and semantic zooming. To graphically represent these three types of changes, we have proposed the notation of grid nodes, and three rules: location rule, zooming rule and distance rule. We use the spatial graph grammar formalism to explicitly describe the syntax of Web application layouts and transformation methods.

The graph transformation tool can be considered an authoring language generator, i.e. a meta-tool, that can generate any authoring tool environment or re-generate a modified tool whenever needed. A multimedia author without any knowledge of graph grammars or design rules will be able to use the generated authoring tool to make adaptive presentations by drawing graphical structures. Syntax check and design validation are then automatically performed by the authoring tool. A graph layout can be transformed according to the defined grammar or run-time events such as a user interaction.

As mentioned at the beginning of the chapter, the Synchronized Multimedia Integration Language (SMIL) (Bulterman and Rutledge 2004; W3C 2001) allows control over which media elements, and where and when the media elements are to appear in a multimedia presentation. Though the SMIL is flexible to support multiple alternative layouts, there are several fundamental differences between the SMIL and our graph-grammar-based approach. Firstly, rather than providing absolute layout positioning as in the SMIL, a graph grammar defines the desirable layout adaptive to the existing layout or the user's intention. Secondly, when a media element is deleted or inserted satisfying predefined structural constraints, an updated representation can be automatically generated in the grammatical approach through parsing but this is not possible with the SMIL. Thirdly, the position of a media element in the SMIL is defined relative to the size of the element's parent geometry. The SGG cannot only define a representation the same way through attributes and action codes, but also specify the position of one element relative to another through graphical notations.

Cascading Style Sheets (CSS) (W3C 2004b) define how to display Web documents, including specification of fonts, background, foreground, and so on. They allow both the author and reader to provide rules that specify various attributes of a Web document. Multiple style definitions will cascade into one according to some conflict resolving rules. The layout mechanism of both the CSS and SMIL works on a predefined specification. Only through a transformation language, such as XSLT (W3C 1999), may CSS and SMIL allow the layout mechanism to work conditionally on a previous layout or a spatial property. The graph grammar approach, however, allows the new layout to be generated based on the previous layout as well as on a specification.

4.7 Related Work

There have been a number of systems and approaches for the authoring and presentation of multimedia systems (Prabhakaran 2000). Among knowledge-based approaches, Comet (Feiner and McKeown 1993) and WIP (Andre et al. 1993) employ some forms of rule-based mechanisms to represent the graphical design knowledge. The rules control the search of all possible solutions and determine an appropriate solution. One of the most challenging issues in these systems is how to specify the control mechanism.

Vazirgiannis *et al.* (1999) proposed a spatio-temporal composition model, and indexing schemes for efficient querying in such a spatio-temporal coordinate system (Vazirgiannis *et al.* 1998). The model translates spatial and temporal relationships among multimedia objects into minimal and uniform expressions, and allows authors to specify an object's spatial features either as absolute coordinates or in relation to other objects. Algorithms and tools have been developed to transform relative data into absolute coordinates, and to verify the integrity of spatial and temporal relationships. The model does not address its adaptability to the changing space and layout requirements. Based on the nested context model (NCM) (Casanova et al. 1991), HyperProp (Soares et al. 2000) emphasizes the importance of document logical structuring. It supports event-based spatial synchronization and behavior specification, but offers no explicit specification of document layout and spatial adaptation. Temporal aspects are also investigated by Guan *et al.* (1998) in their model of Distributed Object Composition Petri Net (DOCPN) that facilitates the synchronization of multimedia presentation in a distributed computing environment.
In a dynamic interface, the attributes of elements are defined in terms of other elements and attributes of the viewing environment: information links indicate a (semantic) connection between two pieces of information, which can belong to different information domains, an information view is a collection of correlated objects displayed together to help the user to perform some activities on the objects (Bjork et al. 2000). Interactivity allows the display to be dynamically adapted to the user's requirement. Borning *et al.* (2000) present a system architecture in which both the author and the

viewer can impose page layout constraints. The final appearance of a Web page is thus the result of negotiation between the author and the viewer. Marriott *et al.* (2002) extends Scalable Vector Graphics (SVG) with constraint-based specification. Such an extension supports client-side adaptation of documents to different viewing conditions. These approaches do not offer visual specifications and their layout solutions rely on constraint solvers.

In dynamic authoring, "authoring" refers to creating the content for any kind of presentation or document (Myers 1998). Dynamic authoring advocates that capture-based systems should support flexible hypertext structures generated by linking through interactive operations (Pimental et al. 2000). Some user interface toolkits uses the approach of recognition and mediation by constructing a library of reusable error correction, or mediation, that can provide structured support for resolving ambiguity at the input event level (Mankoff et al. 2000).

The work presented in this chapter was influenced by that of Weitzman and Wittenburg (1994). Weitzman and Wittenburg (1998) applied a graph grammar formalism – Relation Grammar, to the automatic presentation of multimedia documents. The grammar governs the structure of the document. One or more parsing trees, each of which represents an independent presentation, are derived through a parser. Then, a syntax-directed translation is made on the tree. The final layout is created by a constraint solver following the translation. In this approach, relational grammar functions as a mapping from a representation of one style of multimedia documents to the forms that specify how to realize the media objects. Inspired by the work of Weitzman and Wittenburg, Cruz and Lucas (1997) developed a visual querying and presentation system called Delannay^{MM}, but grammars are not used in this system (Cruz et al. 1997).

Another area of research is graph drawing (Di Battista et al. 1999). Six *et al.* (2000) proposed post-processing techniques (after some major graph layout process), called refinement, for effective graph drawing. The techniques can significantly improve the quality of orthogonal drawings by reducing a graph's area, bends, crossings, and total edge length (Di Battista et al. 1999). In a graphical layout, maintaining a consistent view by automatically beautifying the display is desirable (Minas and Viehstaedt 1993). This chapter discussed a grammatical approach rather than an algorithmic approach to the graph layout problems addressed by Six *et al* (2000). Zhang *et al.* (2002) presented an approach to combining the RGG formalism with constraint rules to support automatic layout of orthogonal graphs.

Research has been done in the graph grammar support for Web information transformations. To support automatic layout of flowcharts, recently Zhang *et al.* (2001c) presents a visual approach to XML document design and transformation, which uses RGGs (Zhang et al. 2001b) to define the XML syntax and to specify the transformation between different XML formats. The details will be covered in the next chapter.

Chapter 5 Data Interoperation

5.1 Introduction

As more research disciplines and social sectors are becoming computerized, an increasing amount of observational data and documentation are digitized. To support interchange of these digital materials, encoding standards such as XML (eXtended Markup Language) (W3C 2004a) have been proposed for digital document markup. A document structure is realized by a set of element tags that can be used to delimit data items in a document of the specific domain. If all the delimiting tags are properly placed in a document, the document is said to be *well-formed*. However, whether an XML document is valid is determined by the Document Type Definition (DTD), a formal grammar for specifying the structure and permissible values of XML documents. The content of the document elements and their markups can be specified and validated by a schema language, such as the XML Schema (Thompson et al. 2000), RELAX (Makoto 2000), or SOX (Davidson 1999). The tag set of a specific domain is called the "vocabulary" of that domain. People of the same domain could use the same basic syntax, parsers, and assisting tools of the vocabulary. This opens a way for different types of document structures to be created to facilitate communications for various professional domains. Digital data is thus not only represented but also defined in different languages. To reuse and exchange digital information, two levels of information translation need to be addressed, i.e. *data translation* for data instances of different formats, and *schema translation* between different schemas.

A naive way to translate data between different formats is writing a specific translator for each pair of formats. Writing such a program is typically a non-trivial task, and is often complicated by the manipulation of data sources (Milo and Zohar 1998). The extensible styling language (XSL, www.w3.org/TR/2003/WD-xsl11-20031217/) is proposed precisely for translating from one XML representation to another as well as for styling. XSL specifies and supports the transformation of an input document to another structure, and describes how to present transformed information. XSL specifications are, however, hard to express intuitively in a linear textual form even though the structure of a XML vocabulary is essentially a tree structure. Transformations using XSL must be created manually on a case-by-case basis. Furthermore, writing an effective XSL code requires some degree of programming skills and good understanding of XML's working principle. Therefore, the current XML technology has limited user population. A more general framework can be based on a common data model (CDM) (Sheth and Larson 1990) to which the source/target data is mapped, and a common translation language that enables the specification and customization of the translation task. This would facilitate new translations, but still require considerable programming effort whenever a new translation is to be defined (Abiteboul et al. 1997).

On the other hand, meta-model based techniques (Atzeni and Torlone 1995; Bowers and Delcambre 2002; Torlone and Atzeni 2001) concentrate on both data and schema translation. A meta-model is a higher-level abstraction of data representation than models. Meta-model based approaches provide a uniform representation for various levels of abstractions and enable data and schema translation declaratively. However, most of the existing approaches are based on textual languages, which are not as intuitive as graph based techniques.

It is natural to represent the structure of any digital artifact and associated schema graphically so that specifications and translations can be performed intuitively. Aiming at providing user-friendly means for people of various communities to use and exchange digital artifacts, we explore the power of graphical visualization and automatic language generation, with a sound underlying theory. The generated language environments can automatically verify the syntactical structure of any constructed digital artifacts and, when translation specifications are provided, automatically translate a source artifact expressed in one encoding language or schema to its equivalent in another language or schema.

In the remaining part of the chapter, Section 5.2 will first present a vision of an interoperable framework based on multi-level abstractions of digital artifacts. Section 5.3 focuses a real document to show the interoperation at the instance level by translating it into a different format. Section 5.4 presents both model and schema levels of specifications using graph grammars, followed by the implementations of model management operators in Section 5.5. Section 5.6 finally summarizes the chapter.

5.2 A Hierarchical Interoperable Framework

To support full interoperability across heterogeneous platforms, vocabularies, and representation schemas, we propose a hierarchical interoperable framework for not only digital information interchange but also for interoperation of information across different disciplines.

We consider four levels of abstraction: meta-models, models, schemas, and data instances, as illustrated in Fig. 5.1. At the bottom level, *instances* contain concrete digital data, such as NCBI XML and HTML documents. At the second level, a *schema* defines the structure of instances. Different schemas are defined by different formalisms, called *models* at the third level. On top of these three levels, we use a *meta-model* to define multiple data models, using the basic abstractions for defining models. Each level of the architecture can be viewed as an instantiation of the level above. More specifically, models are particular instantiations of the abstractions defined by a higher-level model, i.e. meta-model, schemas are instantiations of a model, and instance-level data are instantiations of a schema. This means that there is a general relationship between any two consecutive levels: a level of information can be specified and encoded by its upper level in the form of an encoding language (for example, XML documents specified by DTDs or XML Schemas). The lower the level of abstraction, the more concrete and easier it is to comprehend. In the following, we will explain an approach in details at the levels of instance, schema, and model.

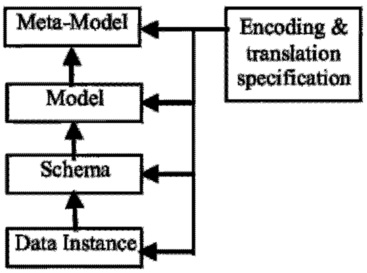


Fig. 5.1. Four levels of abstraction

The presented approach at the instance (data/documents), schema (language rules), and model (meta) levels will serve two purposes: supporting graphical construction of new documents and automatic generation of documents upon validation; and supporting automatic translation of existing documents from one encoding format to another.

Using the RGG as a meta-model, Fig. 5.2 shows an example of hierarchical abstraction levels of meta-model, model, schema, and instance data, together with their correspondences to the formalisms in the RGG. At the top level, nodes and edges in the RGG represent meta-elements and relationships respectively. At any of the lower levels, a node denotes an element of a model, a schema or an instance. Labeled with different names, nodes have different semantics in different contexts and levels of abstraction, e.g. an element of Schema, a PCDATA of DTD, or a tag of a document instance. An edge defines a relationship between two nodes, such as element-element relationship and element-attribute relationship. Each RGG production denotes relationships among elements. A whole set of productions consisting of meta-elements defines a model, which in turn defines a schema with elements. A schema consisting of tag names defines an instance. As a meta-model, the RGG enables users to define a model as a RGG, i.e. a set of graph transformation productions. A specific schema is defined by a RGG, as a host graph of the RGG at the model level in terms of primitives. A digital document is denoted by a document tree of the RGG from the schema level.

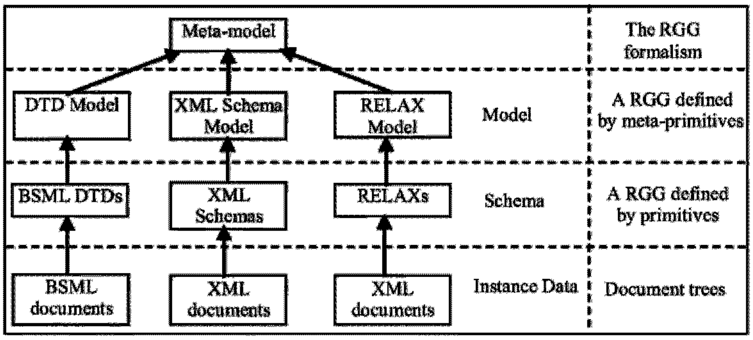


Fig. 5.2. Hierarchical levels of abstraction of data representation

Fig. 5.3 shows the system architecture, which consists of three major modules, the visual object generator (VOG), the rule generator, and the visual modeling environment (VME), which is automatically generated. In the following, the term *primitive* refers to a component of a model, and the term *meta-primitive* refers to a component of a meta-model.

VOG is used to define meta-primitives. It provides a generic approach for users to introduce a new meta-primitive suitable for the meta-model when a new model needs a special construct that has no counterpart in the meta-model. This approach is considered by Atzeni and Torlone (1995) as "asymptotically" complete in due to its support for generality and variety of meta-models.

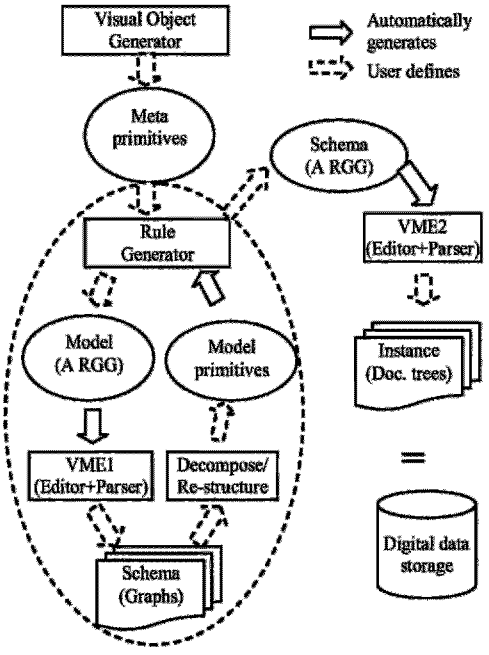


Fig. 5.3. Architecture of the hierarchical framework

The rule generator is a visual environment for users to define graph transformation rules. With user-defined meta-primitives or primitives, a user may construct a graph grammar to define a model or a schema according to different types of model/schema components. Supplied with user-defined rules, the rule generator compiles and automatically generates a visual modeling environment (VME).

Each VME consists of a graph editor and a parser for the pre-defined graph grammar, based on which the VME is generated. The graph editor provides users with guidance on how to draw a host graph to represent a schema (VME1 in Fig. 5.3) or a data instance (VME2), and prompts errors whenever the syntax is violated. It is also able to perform syntax-directed computations, specified with individual transformation rules.

The above hierarchical framework allows users to perform the following steps to specify and transform digital artifacts:

1. use VOG to define meta-primitives that are able to capture the main primitives adopted by different schema languages describing structured data (XML documents in particular) for digital archiving or interchange;
2. use these primitives to define a model or a schema, which is effectively a specific RGG, enabling automatic generation of VME (VME1 or VME2 depending on the level of abstraction); and
3. further define, decompose or re-structure schemas and transformation rules graphically within the generated VME1 to make model primitives, and then return to Step 2; or define source data instances graphically within the generated VME2 and automatically apply the rules.

As shown in Fig. 5.3, Step 2 and the first half of Step 3 could form an infinity loop, which simulates the recursive definition in the real world. For example, the XML Schema is used to define XML documents, while it is an XML document itself, i.e. schema is an instance of its own. Detailed visual model management operators and their implementations in RGGs for schema interoperation can found in Song et al. (2004a; 2004b) and Song (2006).

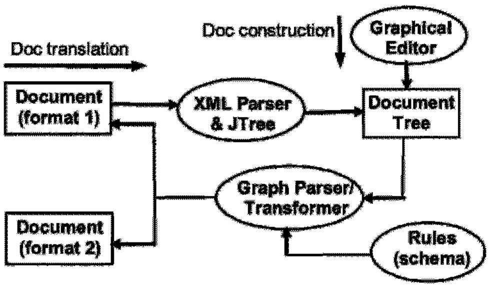


Fig. 5.4. Meta-tool approach to specification and translation at the document level

5.3 Interoperation at Instance Level

As illustrated in Fig. 5.4, a document author can use the graphical document editor to construct a document of the desired structure by drawing a document graph (an XML document structure is essentially a tree). Or if a document already exists, it could be converted into a graphical representation by an XML parser and Java JTree. The document tree, either drawn or generated from an existing document, is graphically parsed and executed based on the graph transformation theory described in Chapter 2, so that the document structure is validated and textual version generated. If needed, the document can be translated into a different vocabulary.

The underlying mechanism supporting the interoperability at multiple levels is a graphical encoding language generation toolset, adapted from the visual language generation environment, VisPro, to be discussed in Chapter 8. The toolset consists of a visual object tool, a visual rule tool, and a language-generation engine. A person in charge of data integration (called data integrator) can use the visual object tool to define the visual notations for nodes (typically representing tags) and links (parent-child relationships) and provide attribute types of the nodes. The visual rule tool inherits the visual notations defined in the visual object tool, so that rules can be constructed using nodes and links. The data integrator can then use the visual production tool to define the XML language vocabularies and their relationships, and specify translation rules for a target XML language. According to the specified rules, the language-generation engine could automatically generate a graph parser/transformer and a graphical document editor. A scientist (information provider) would be able to use the generated tools to create formatted data without the need to know the formatting language. In summary, the major advantages of this approach are the following:

- The graphical representation of digital information structures and the transformation process is easier to comprehend than the textual form, as it could reflect one's mental image of the structure of a digital artifact (e.g. a document structure).
- Automatically generated by a visual language generator, the graphical editor and translation tool can be rapidly updated upon an information provider's needs. The generated graphical editor can perform syntax-directed computations and syntactic checking of any constructed digital artifact, and if so desired, automatic translation to a different vocabulary and format.
- Separation of concerns of the design and construction of digital artifacts using XML-based languages, so that only a small proportion of specialists (as data integrators) would need to understand the encoding standard and languages while the large majority of scientists (as information providers) need not concern about detailed encoding formats.

In the following we go through a detailed example to demonstrate the ideas of graphical representation and transformation of digital artifacts through a NCBI XML (www.ncbi.nlm.nih.gov/IEB/ToolBox/XML/) document, and its translation into BSML (www.bsml.org/overview/default.asp).

5.3.1 Source and Target Documents

The example document in both NCBI XML and BSML representations is downloaded from the Web site http://www.kaiseryang.com/bio/db/NCBI2BSML.html. To save the space, we have shortened the document by removing many tag entries that have the same structures and replacing such entries with simple comments as shown in the following NCBI XML representation.

```
<?xml version="1.0"?>
<!-- <!DOCTYPE Seq-entry PUBLIC "-//NCBI//NCBI Seqset/EN"
"http://www.ncbi.nih.gov/dtd/NCBI_Seqset.dtd"> -->
<Seq-entry>
 <Seq-entry_set>
  <Bioseq-set>
   ......
   <Bioseq-set_descr>
    <Seq-descr>
     <Seqdesc>
      <Seqdesc_comment>[1] suggests .......</Seqdesc_comment>
     </Seqdesc>
     <Seqdesc>
      <Seqdesc_update-date>
       <Date>
        <Date_std>
         <Date-std>
          <Date-std_year>1995</Date-std_year>
          <Date-std_month>1</Date-std_month>
          <Date-std_day>6</Date-std_day>
         </Date-std>
        </Date_std>
       </Date>
      </Seqdesc_update-date>
     </Seqdesc>
     <Seqdesc>...inc. source, create date, and pub ...</Seqdesc>
    </Seq-descr>
   </Bioseq-set_descr>
   ......
  </Bioseq-set>
 </Seq-entry_set>
</Seq-entry>
```

NCBI (National Center for Biotechnology Information) has previously been using a language called *Abstract Syntax Notation 1* (ASN.1) for describing and exchanging information, in a similar fashion as using XML-based languages. NCBI ASN.1 support modular specification by allowing information sharing and reuse. This means that a single tag could mean two different things in different contexts. It also allows the same name used across different structures, while XML requires all tag names (non-attributes) to be unique across the DTD. Due to these different language requirements, an NCBI XML document directly translated from ASN.1 is inevitably verbose, with extensive tags, that can be observed in the above document. Fig. 5.5 shows a screen dump of the prototype called Biotrans when the input NCBI XML document was converted to a JTree.

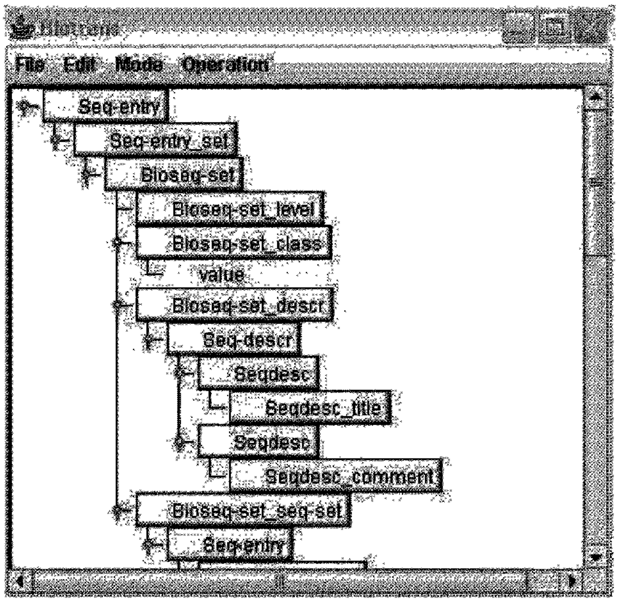


Fig. 5.5. Input NCBI XML document converted to JTree

Assume that we wish to translate any documents written in NCBI XML to their BSML representation, which will be much more concise. For the above digital document, its BSML equivalent is as follows

```
<?xml version="1.0" encoding="UTF-8"?>
 <Bsml>
  <Definitions>
   <Sequences>
    <Sequence title="HUMINSR" ic-acckey="M10051" id="G186439" comment="Human
insulin receptor mRNA, complete cds." representation="raw" molecule="rna" length="4723"
strand="ss">
     <Attribute name="version" content="M10051.1"/>
     <Attribute name="comment" content="[1] suggests that ......."/>
     <Attribute name="update-date" content="1995 1 6"/>
     ...Attributes for "create-date", "source", "pub", and "keywords"......
     </Sequence>
   </Sequences>
  </Definitions>
 </Bsml>
```

We first translate the textual NCBI XML document according to its tag structure into a graphical form (essentially a tree) whose syntax is suitable for grammar interpretation. Although the example is for translating from NCBI XML to BSML, the principle that we will be demonstrating equally applies to the translation between any two XML-based documents, and to more complex and varying document structures.

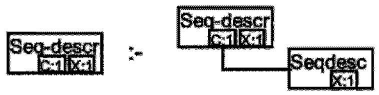
5.3.2 Specifying Structures and Translation Rules

We need to devise a formal grammar (i.e. "Rules" as shown in Fig. 5.4) for the underlying XML-based language that is able to understand and interpret any source documents written in the language (e.g. NCBI XML). To support the automatic transformation of the source document to a target document written in another XML-based language (e.g. BSML), we also need a transformation mechanism. Since all documents are represented graphically as trees, it would be most natural to use a graph parser and a graph transformer to perform the above two tasks. As the sound foundation of visual programming languages, graph grammars would serve well the desired graph parsing and transformation (Rozenberg 1997).

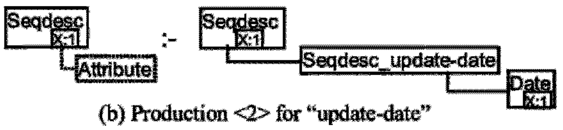
We adapt the RGG formalism to specify XML-based document structure and transformation. As shown in the BSML listing in Section 5.3.1, there are two types of attributes in this target language: the ones enclosed in a tag (e.g. "update-date" and "comments") and the ones represented as tags themselves (e.g. "Attribute"). To distinguish these two types of attributes in the final target document, we represent the former type as the nodes connected directly to the super-vertex of the tag node, and use a vertex X to designate the connecting point of all the latter type of child tags.

The following simple example explains the graph transformation approach. Fig. 5.6 illustrates three productions specifying the NCBI XML structure. They define "Seq-descr" as the parent tag of "Seqdesc" Fig. 5.6(a), and two child tags of "Seqdesc", one for "update-date" in Fig. 5.6(b) and the other for "comment" in Fig. 5.6(c).

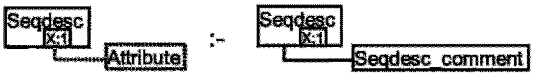
Fig. 5.7 demonstrates the transformation process of a sub-graph involving the use of the above three productions. In Fig. 5.7(a), the sub-graph isomorphic to the right graph of Production <2> and that isomorphic to the right graph of Production <3> are redexes, enclosed in two dashed boxes. After applying Productions <2> and <3> to the example sub-graph, the newly transformed sub-graph is depicted in Fig. 5.7(b). Vertex C in the right graph of Production <1> (Fig. 5.6) is *marked* by a unique number, indicating that its corresponding isomorphic vertex in the new sub-graph can be connected to multiple nodes, so that Production <1> can be applied separately to the child tags "Seqdesc" of "Seq-descr". The first application of Production <1> results in the new sub-graph in Fig. 5.7(c) and the second application generates the final sub-graph with two "Attributes" connected to "Seq-descr" as in Fig. 5.7(d). Consider the step in Fig. 5.7(b), the transformation deletes the redex in the dotted box that matches the right graph of Production <1>, while keeping the vertices, i.e. C and X, that are marked in the production. Then the left graph of Production <1> is embedded into the host graph, as shown in Fig. 5.7(c), with the "Seq-descr" node connecting to an "Attribute" node via vertex X.



(a) Production <1> for "Seqdesc"



(b) Production <2> for "update-date"



(c) Production <3> for "comment"

Fig. 5.6. Three productions to explain graph transformation

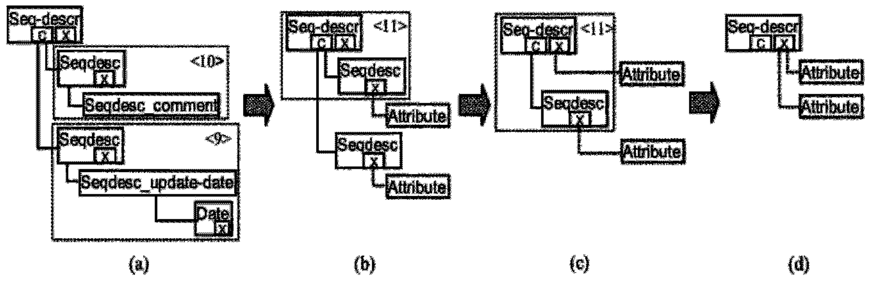


Fig. 5.7. Demonstration of graph transformation process

The syntax of NCBI XML is defined by the sub-graphs without dotted lines; and the mechanism for translating the NCBI XML syntax to BSML is defined by the sub-graphs including dotted lines. We will refer to the former rules as the *grammar*, and the latter, extended from the grammar, as the *translator*. The grammar serves the validation and generation of documents of the defined NCBI XML structure, while the translator allows documents of the defined structure to be validated and also translated into documents of the BSML structure.

A production may contain elements that serve as instructions for creating result tree fragment. The instructions are executed when the production is applied. For example, to convert the "update-date" format from three separate strings for year, month, and date to a single string format as required by BSML, we will attach the following action code to Production <2>in Fig. 5.6:

```
Action(AAMGraph g) {
   Date.value = Date-std_year + Date-std_month + Date-std_date;
   }
```

apart from other operations that may be required for this particular production.

Fig. 5.8 is a screen dump of Biotrans after the transformation rules were entered and defined. Fig 5.9 shows the output result after translation.

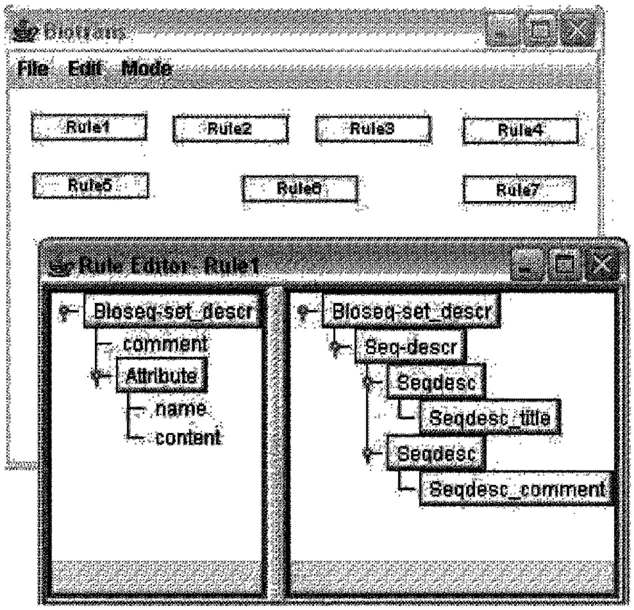


Fig. 5.8. Transformation rules defined in Biotrans

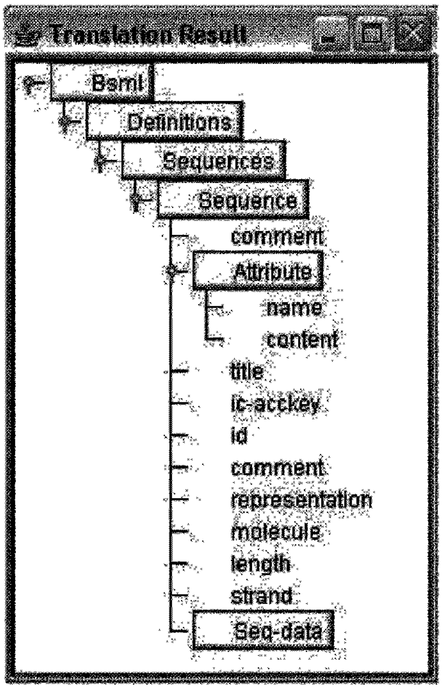


Fig. 5.9. Output display of Biotrans after automatic translation

5.3.3 Automatic Validation and Translation

We now discuss how graph transformation and associated syntax-directed computations using the RGG formalism can be performed to achieve the validation and, if desired, text processing of any input XML document. We define the productions by taking advantage of the tree structure of any XML-based document so that the document can be parsed (transformed) efficiently from tree leaves to the root.

The parsing algorithm for XML-based documents is different from that of the RGG, because matching a right graph to a host graph is a tree-to-tree match rather than graph-to-graph match in the RGG. The tree-to-tree match can be performed much faster when using appropriate data structure and algorithm. The parsing process proceeds in two steps:

- Search in the host graph for a redex of the right graph;
- Embed a copy of the left graph (i.e. the root of the right graph) into the host graph by replacing the redex.

Repeat Steps 1 and 2 until the host graph is empty or no more matches for any productions.

Different from the process of validation, in which the parser finally presents a parsing result, i.e. valid or not, the translation process produces a new graph from the input host graph according to the translator productions. Slightly different and extended from the original RGG parser, the translator mechanism performs the embedding process in a copy of host graph and adjusts the copy to a new graph as the result.

For example, by applying the grammar (excluding the sub-graphs with the dotted lines) repeatedly to a specific diagram, we can determine whether the diagram represents a valid NCBI XML document. Similarly, by repeatedly applying the translator (including all the sub-graphs in dotted lines), a NCBI XML document can be transformed into a more concise BSML document. The productions may be applied in different orders but will produce the same result. To take advantage of the tree structure of any XML documents, we define the production rules in such a way that efficient parsing and translation can be performed from tree leaves to the root. The extended part (from the grammar) in the translator would not affect the transformation efficiency since its sub-graphs could all be regarded as terminals.

5.4 Model and Schema Specifications

This section presents the model and schema specifications, and shows the basic process of data/model management by identifying meta-primitives of a meta-model, defining a model, constructing a schema, and drawing an instance.

5.4.1 Identifying Meta Primitives

All the constructs used in most known schema models and formalisms for expressing digital data fall into a rather limited set of categories. Therefore a meta-model can be defined in terms of a basic set of meta-primitives, corresponding to these categories.

In considering the above fact, our example meta-model includes the meta-primitives limited to the following: object type, ordered sequence, unordered sequence, choice, cardinality, key, and foreign key. Obviously the meta-primitives are not complete for defining all the types of data models, so the names and number of nodes may change when the meta-model needs to be extended to subsume newly introduced models. To ease the process of extending the meta-model, we can provide the VOG for users to specify new meta-primitives. A relationship between meta-primitives is defined by an edge, whose semantics is determined by the nodes it connects. The set of example meta-primitives includes eight types of nodes (meta-primitives) as shown in Fig. 5.10.

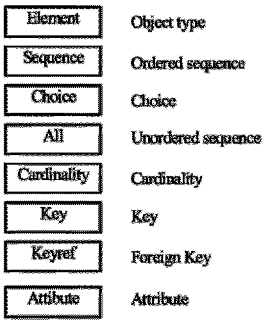


Fig. 5.10. Meta primitives of the meta-model

5.4.2 Defining a Model

As stated in Section 5.2, a model that can be used to specify a schema (*scheme model* for short) is a specification for the corresponding schema. This section uses the meta-primitives in Fig. 5.10 to define an example model for a subset of the XML schemas, which includes most of XML Schemas and DTDs. In addition, the rule generator provides a convenient way for users to define and add more rules to specify a complete set of schemas for the structure of XML documents. Fig. 5.11 shows the model defined by a graph grammar, which consists of 14 rules, and the *ith* production is marked with <*i*>.

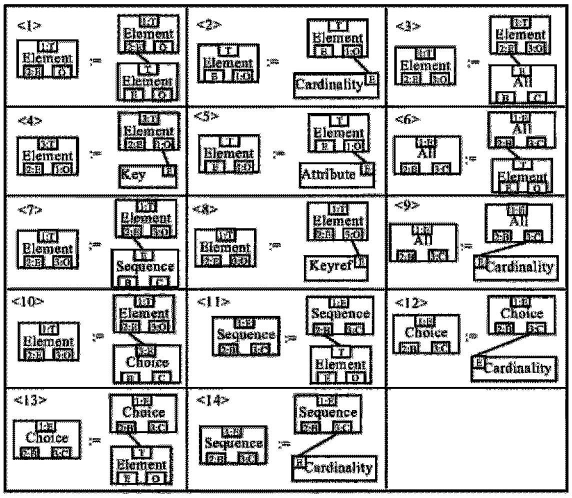


Fig. 5.11. A subset of schemas defined by the meta-model

The model defines relationships among **Elements** and **Attributes** etc. Rules <1> and <5> declare that an element can enclose arbitrary number of other **Elements** and **Attributes** respectively. Rule <2> defines that an **Element** may have a **Cardinality** constraint. Rules <4> and <8> define that an **Element** may have an arbitrary number of **Keys** or **Foreign Keys**. Rules <7>, <11>, and <14> define that an **Element** may have a **Sequence**, a **Sequence** may include any number of **Elements**, and a **Sequence** may have a **Cardinality** constraint, respectively. Rules <10>, <12>, and <13> define the same relationships among **Elements, Cardinalities**, and **Choices**. Rules <3>, <6>, and <9> define the relationships among **Elements, Cardinalities**, and **All**.

5.4.3 Constructing a Schema

After the rule generator parses the rules that define the model, a VME is automatically generated. Using the VME, the user draws and customizes a host graph, i.e. an instance of the model, to define a schema. Then with the user's intervention, the VME transforms the host graph into a set of rules suitable for the system to define the structure of data instances. During the procedure users could interact with the system to adjust the schema, i.e. the procedure is semi-automatic. To construct a schema using VME, users should follow the three steps described in the following three subsections.

5.4.4 Drawing an Instance Data

Under the syntax guidance of VME, the user draws a host graph in the generated VME to represent the structure of the expected schema. Fig. 5.12 shows an example of the host graph conforming to the rules described in Fig. 5.11. The host graph defines the structure of the schema, in which the name of each node denotes the type of the node, such as Element and Sequence. The host graph does not however define the instance of the type, such as the name of an element or range of the cardinality, which is domain specific. The framework cannot determine the value of each node automatically. Therefore the host graph does not completely define the structure of a document, and rest of the job has to be tackled by the user, i.e. to customize the host graph in the VME.

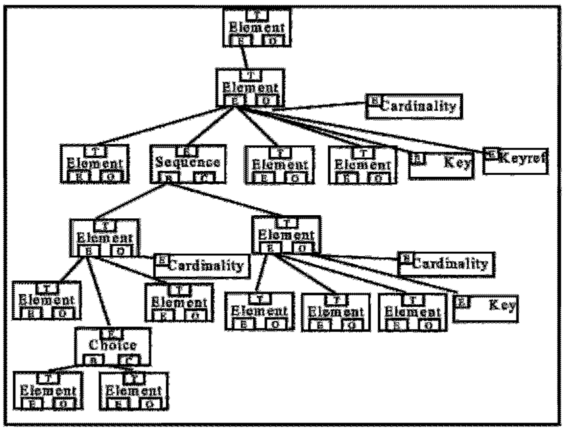


Fig. 5.12. An example schema conforming to the rules in Fig. 5.11

5.4.5 Customizing the Host Graph

To customize the host graph means to specify the parameters of each node according to the domain requirement, such as the name of an element, the range of the cardinality, and so on. This could be done very easily with the generated VME by editing each node. Fig. 5.13 shows the user-defined domain specific schema based on the example in Fig. 5.12. The name of the node Sequence is the same as that in Fig. 5.12, but the meaning is different, the one in Fig. 5.12 defines the type of the node, while the Sequence in Fig. 5.13 represents the instance itself. The DTD defines a Sequence using a comma "," between every two elements.

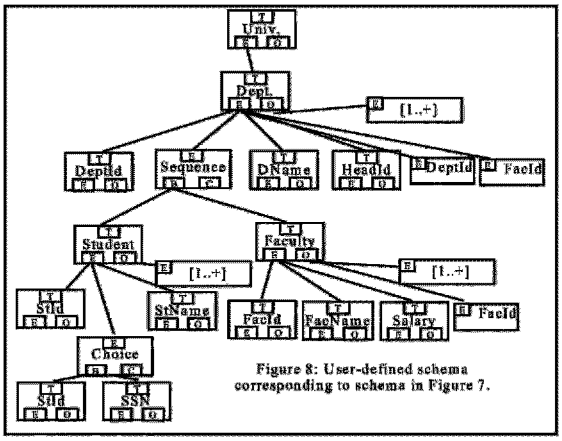


Fig. 5.13. User-defined schema corresponding to the schema in Fig. 5.12

5.4.6 Adapting the Rules

After being customized in the domain context, the host graph represents the structure of a schema. From now on we refer to the host graph as *schema graph*. Before the rule generator processes a schema graph, it needs to be adapted because of the following.

1. Although we can simply make the schema graph a right graph of a rule, we still need a left graph, because each rule of a graph grammar consists of a right graph and a left graph.
2. As some nodes in the host graph carry structural semantics, they should be converted to the corresponding notations in the RGG for the parser to recognize.
3. The number of nodes in the right graph influences the performance of the parsing process, whose time complexity is partially determined by the maximum number of right graph as discussed in Chapter 2. The larger the right graph is, the more costly the parsing process is. So a large schema graph should be broken into smaller ones to improve the parsing performance while kept correctness.
4. In order to further construct rules for data translation based on the schema, the schema graph needs to be decomposed.

The VME automatically adapts the schema according to the following principles.

- Remove those nodes that carry structural semantics, such as cardinality and attribute, and add appropriate attributes to the nodes associated with the removed nodes.
- Trim vertices of each super-vertex, i.e. remove those vertices not used in the schema, such as the vertex links to the cardinality node.
- Break any schema trees of more than three levels to smaller trees to improve the parsing performance.
- For each sub-graph of the schema graph, add the left graph to construct a rule.

The user then adapts the automatically generated rules in the rule generator. The rule generator could in turn parse the rules and generate a new VME for the user to draw instance documents. Fig. 5.14 shows the resulting rules generated from the user-defined schema in Fig. 5.13. In the generated VME, the user could draw and parse host graphs, which are the instances of the schema.

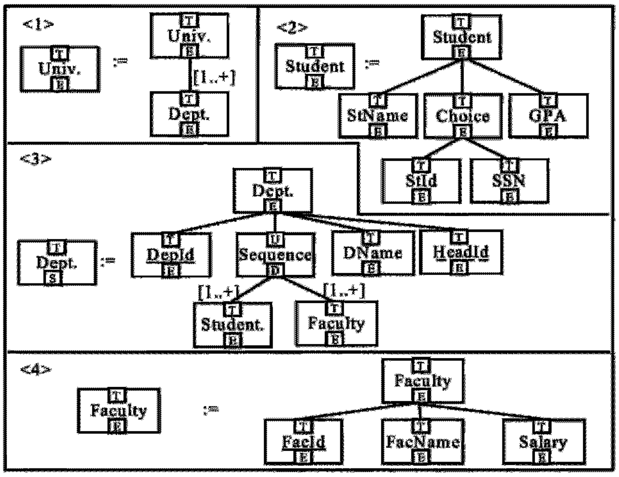


Fig. 5.14. Rules generated from user-defined schema in Fig. 5.13

5.4.7 Drawing an Instance

The VME generated by the rule generator from a schema graph enables users to visually define any instance documents conforming to the schema.

Fig. 5.15 illustrates an example document conforming to the schema rules defined in Fig. 5.14. In the VME, a user can draw any host graph, which defines the structure of an instance document and conforms to the rules used to generate the VME, i.e. the schema. So far, the host graph does not have concrete data except the structure. The framework cannot determine what the user wants data to represent; therefore the user has to customize the data instance in the structure.

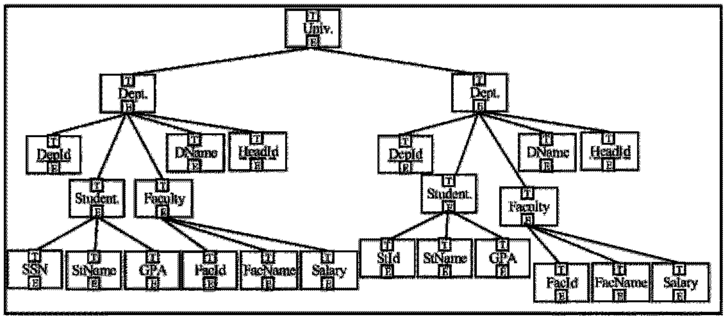


Fig. 5.15. User-defined document structure corresponding to schema in Fig. 5.14

In the host graph, the name of each node is a markup in a markup language, which denotes the data type of the node and could enclose an instance value. For example, StName in Fig. 5.15 is of data type "string", and could have any string as a value, such as "Lawrence". For simplicity, Fig. 5.15 does not show the detailed value of each node. After the user instantiates the structure, the document is completed.

Recall the process of constructing a schema, the process of drawing an instance follows a uniform procedure except that the host graph of an instance needs not to be adapted for further definition.

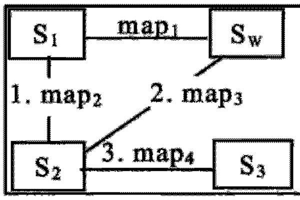
5.5 Model Management Operators

Comparing to data instances, models are far more complex to interoperate as little help can be found in query languages. Model management systems are proposed to ease the programming for model operations (Bernstein 2003). A model management system consists of a set of operators which represent the generic operations on input models. As described by Bernstein (2003), the main model management operators are defined as follows:

- Match – takes two models as input and returns a mapping between them.
- Compose – takes a mapping between models A and B and a mapping between models B and C, and returns a mapping between A and C.
- Diff – takes a model A and mapping between A and some model B, and returns the sub-model of A that does not participate in the mapping.
- ModelGen – takes a model A, and returns a new model B that expresses A in a different representation (i.e. data model).
- Merge – takes two models A and B and a mapping between them, and returns the union C of A and B along with mappings between C and A, and C and B.

These operators are applied to models and mappings as a whole, rather than to their individual elements. The operators are generic in the sense that they can be utilized for different kinds of models and scenarios.

Consider a typical example of building a data warehouse (Bernstein 2003). Given a mapping *map*$_1$ from a data source S_1 to a data warehouse S_W, we wish to map a second source S_2 to S_W, where S_2 is similar to S_1 (Fig. 5.16). First we call Match(S_1, S_2) to obtain a mapping *map*$_2$ between S_1 and S_2, that shows which elements of S_2 are the same as those of S_1. Second, we call Compose(*map*$_1$, *map*$_2$) to obtain a mapping *map*$_3$ between S_2 and S_W, which returns the mapping between S_W and the objects of S_2 corresponding to the objects of S_1. To map the remaining objects of S_2 to S_W, we call Diff(S_2, *map*$_3$) to find the sub-model S_3 of S_2 that is not mapped by *map*$_3$ to S_W, and *map*$_4$ to identify the corresponding objects between S_2 and S_3. We can then call other operators to generate a warehouse schema for S_3 and merge it into S_W. Comparing to programming the whole system for all the individual interoperation requirements, the model management process reduces considerable programming effort by composing generic operators.



Given: S_1, S_2, map$_1$, S_W
1. map$_2$ = Match(S_1,S_2)
2. map$_3$ = Compose(map$_1$,map$_2$)
3. <S_3, map$_4$> = Diff(S_2, map$_3$)

Fig. 5.16. Using model management to help generate a data warehouse

5.5.1 Hierarchical Operations

Various data models and mappings are specified by different syntaxes, which are mostly defined in natural languages in spite of some formal attempts.

The visual model management approach provides a formal visual representation of data models and mappings defined by the RGG as inputs of model management operators. It exploits graph grammars in defining the syntax of data models. The parser would detect any syntax violation of input data models and mappings. The grammatical approach also sets a foundation for defining various model management operators by graph transformation. Inputs to an operator are viewed as a set of host graphs compliant to the predefined abstract syntax.

Model management operators can be specified at two levels, i.e. *specific operator* and *generalized operator*. A specific operator is a low level description of an operator on a specific input, and presents users a concrete image of the expected output and interface for tuning the result. A specific operator is automatically generated on specific inputs through a generalized operator that is at a high level abstraction, and can be applied to general inputs. The generalized operator graphically describes the algorithm used to transform the input to output of the operator, i.e. the algorithm is performed through a set of graph transformation rules. Since most model management operators require operations on mappings, i.e. results of the match operator, a generalized operator cannot produce perfect result without human intervention. But at a high level of abstraction, a generalized operator is hard to be adapted on specific inputs and is therefore necessary to cooperate with a customizable specific operator.

The two-level hierarchy of operators defines two levels of system-user interactions, i.e. *design level* and *operation level*. At the design level, experts of model management and graph transformation describe the algorithm of an operator by graph transformation rules, i.e. generalized operator. At the operation level, users, such as DBAs, perform metadata-intensive management tasks by adjusting and executing specific operators, which are generated automatically from generalized operators (the process will be described in details in Section 5.5.3).

Fig. 5.17 shows an overview of the visual model management system, which embeds a set of predefined generalized operators. Users compose the operators by scripts or command line to construct metadata applications. According to the generalized operator the system generates a set of specific rules as an interface to accept user's customization. During each step of execution, users may adjust the customizable specific operators to obtain desired output rather than adjusting output directly which could be error-prone. After specific operators are parsed, a visual environment is generated, which produces final results of the operator.

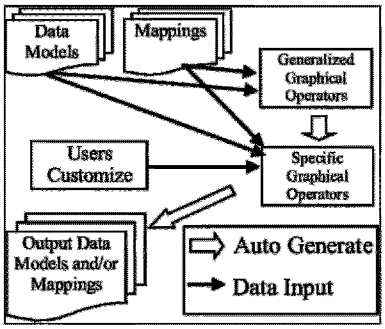

Data Models
Mappings
Generalized Graphical Operators
Users Customize
Specific Graphical Operators
Output Data Models and/or Mappings
Auto Generate
Data Input

Fig. 5.17. Operation Architecture

The two-level hierarchy of graphical operators presents flexibility and clear-cut between two types of model management users. Generalized operators can be applied to host graph directly. Generalized operators operate on the type of nodes, e.g. mapping element, reference element and so on. Therefore, generalized operators define an overall transformation, i.e. any elements with the same type have the same transformation action according to the rules. However if some specific elements, such as a reference element *book*, need to be changed, a generalized operator cannot help. One has to count on the specific operator. Since it is hard to draw a specific operator for each host graph (or a set of host graphs), one can apply generalized operators to specific host graph to generate a specific operator, which can be customized afterwards. The translation is totally different from the original RGG translation, as it generates another set of rules.

5.5.2 Graphical Representation of Models and Mappings

A data model contains a set of objects and various relations between the objects. An object could be an entity in ER models or an element in XML schemas, and a relation could be an "is-a" or "has-a" relation. Each object has an identity and type, and each relation has properties denoting its semantics, such as the min and max cardinality.

We represent a data model, e.g. ER model, by a host graph in terms of a directed node-edge diagram. A node represents an object, and an edge denotes a relation. A node has a name and a type corresponding to the object the node represents. The attribute of an edge defines the relation between the two connected objects.

Graphs of a data model should be compliant to the syntax of that model. For example, two entities of an ER model cannot be connected directly. Such a syntax is defined by graph grammar rules. With these rules, one can easily draw models under the syntax guidance of the RGG toolset.

A mapping, Map$_{AB}$, defines how models A and B are related (Pottinger and Bernstein 2003) as shown in Fig. 5.18. Many proposals use graphical metaphors to represent schema mappings like in Rondo (Melnik et al. 2003) and Clio (Miller et al. 2001). These mappings are shown to the user as sets of lines connecting the elements of two schemas. This kind of representation is simple but not as powerful as SQL view (Madhavan and Halevy 2003) or that of Bernstein (2003). SQL view is not a generic representation for mappings among heterogeneous data sources, such as XML schemas. On the other hand, mappings are structured instead of flat bidirectional, and hard to be described by simple two-way correspondences. The mapping structure described by Bernstein (2003) is an appropriate compromise, being generic yet powerful for describing mappings.

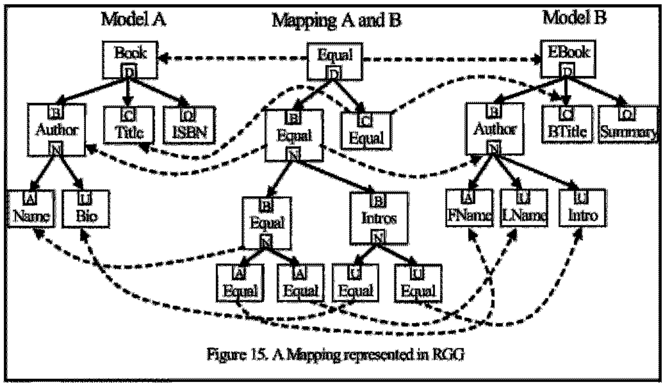


Fig. 5.18. A mapping represented in RGG

We represent mappings as special data models. A mapping has only one relationship type, i.e. has-a relationship, and three element types, i.e. *mapping element*, *reference element* and *helper element*. A *mapping element* specifies how two referenced models' elements are related, such as equality, or similarity, such as node Equal in Fig. 5.18. A *reference element* serves a reference to the element of two corresponding models, such as those nodes of Model A in Fig. 5.18. The relationship between a mapping element and a reference element is denoted by a dashed line. A *helper element* is a make up element to represent extra semantics of a mapping. For example, Intros is a helper element indicating that Bio and Intro can be composed together to form a detailed and official description of Author as shown in Fig. 5.18.

The syntax of mappings is defined by a graph grammar in Fig. 5.19, which includes five production rules. The first production shows that the initial state of the mapping is a mapping element. Production <2> shows that each mapping element can be connected to and has the has-a relationship with more than one mapping element since vertex F is marked. Productions <3> and <5> define the relationship between the helper and mapping elements, i.e. they may have the has-a relationship in either direction. The fact that a mapping element can have multiple reference elements is specified in Production <4>, and the relation between a mapping element and a reference element is denoted by a dashed edge.

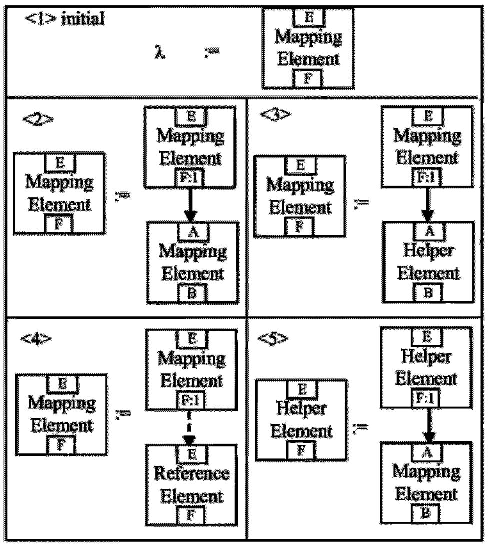


Fig. 5.19. Mapping as a model defined by a Graph Grammar

5.5.3 Implementing Operators by Graph Transformation

Model management operators take data models and mappings as input and generate another set of data models and mappings as output, and are described by a set of graph transformation rules. This section illustrates the graphical representation of two operators, Merge and ModelGen. The same principle applies to other operators.

5.5.4 Merge Operator

Merge takes three inputs, i.e. model A, model B, and a mapping between A and B, and returns the union model C of A and B along with mappings between C and A, and between C and B (Bernstein 2003). The input of merge is $S = (A, B, M_{AB})$, which consists of three graphs representing model A, model B, and the mapping between A and B. After applying merge to S, output T consists of five graphs, i.e. $T = (A, B, C, M_1, M_2)$, where A, B are copies of input graphs, C represents the output union model, M_1 and M_2 represent mappings between C and A, and between C and B.

The semantics of merge can be briefly described as follows: The output of merge is a model that retains all non-duplicated information in A, B, and Map_{AB}; it collapses the redundant information declared by Map_{AB}.

Fig. 5.20 shows a set of graph transformation rules for merging models A and B as defined in Fig. 5.19. Each production rule shows what the result of merge should be. Production <1> defines that root nodes of input models, Book and Ebook, will produce an output data model with a root node Book, and two mappings. Productions <2> and <3> are similar to Production <1>, and copy the referenced node to the output and set a correspondence between the output and input models to form two output mappings. Production <4> merges the structured mappings by defining a new structure in the output model with the nodes referenced by the mapping element and constructing two mappings from elements in the input models to the constructed elements in the output models. Production <5> shows the transformation with a helper element (Intros in this case), and is similar to Production <4>. Productions <6> and <7> copy the input elements that have no reference in the input mapping to the output and establish a mapping between the original element and the copy.

Comparing to an operator algorithm, the graph transformation rules intuitively and explicitly specify what the result should be, and therefore a user with little domain knowledge can manipulate the rules to meet the specific requirements. For example, if one wants to use EBook rather than Book as the root of the output data model, he/she can change the node Book in the left graph of Production <1> to EBook.

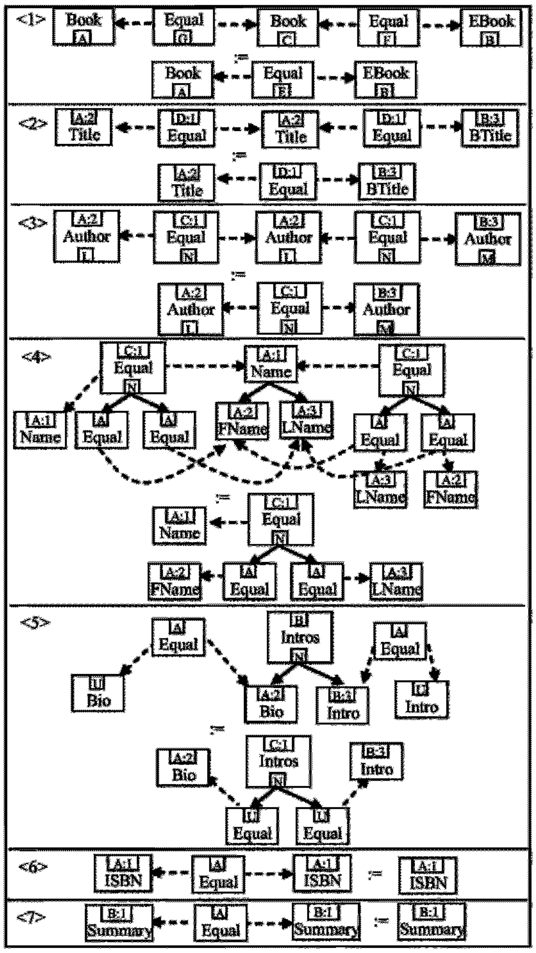


Fig. 5.20. Merge operator defined by graph transformation rules

5.5.5 ModelGen Operator

ModelGen takes a model A as input and returns a new model B based on a mapping between A and B (Bernstein 2003). In the visual model management approach, ModelGen takes input $S = (A, M_{AB})$, where A is a model, M_{AB} is a mapping, and output is $T = (B)$. ModelGen transforms from input graph S to T by applying a set of transformation rules P, i.e. $T = A\ (S, P)$.

For the input (A, Map_{AB}) in the example of Fig. 5.18, the ModelGen is described by the graph transformation rules in Fig. 5.21. Productions <1>, <2>, and <3> show that the result of a one-to-one correspondence is copied directly from the reference elements of model B in the mapping. In Productions <4> and <5>, the reference elements in model A are mapped to the elements in B via a complex structure of mapping elements or helper elements. For example, Production <4> produces new elements by duplicating reference elements of the mapping, e.g. LName and FName.

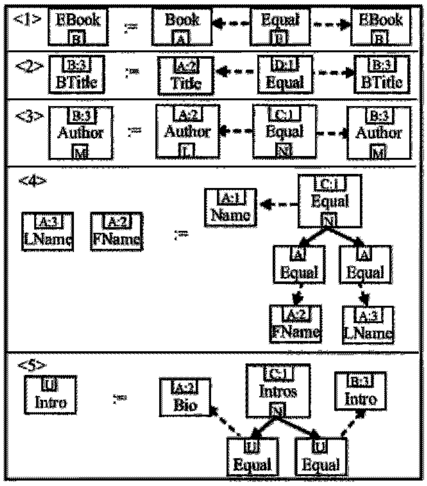


Fig. 5.21. ModelGen by graph transformation rules

ModelGen on the input *(A, M_{AB})* does not produce model B accurately. It cannot produce element summary of the original model B, because the input S *(A, M_{AB})* has no such element. To maintain a high fidelity of the output model one can add summary to the left graph, so that the parser will produce the element missing in the output model.

As shown above, two operators, Merge and ModelGen, are defined by transformation rules on specific inputs. It is easy and feasible for users to specify the specific transformation rules on small-scale inputs, but not for large data models. We therefore should automate the process of defining rules for specific inputs by exploiting traditional algorithms, or generalizing the specific graphical operators, as discussed in the following subsection.

5.5.6 Generalization of Operators

This subsection describes the concept of operator generalization by going through the merge operator. Based on mappings, generalized graph transformation rules visually describe the algorithms for the corresponding operators at a level higher than specific operators. Ideally if we could define all the detailed algorithms of model operators by graph transformation rules, model management could be an automatic and visualized process. Due to the *ad hoc* nature, however, generalized operators still need to be customized for specific inputs, for example the ModelGen in Fig. 5.21 needs to add summary to Production <1> for an accurate output.

Therefore generalization aims at describing algorithms of operators by graph transformation and when applied to a specific input, the parser generates the corresponding specific operators, which are customizable. The approach shown in Fig. 5.17 could be fully interactive and also visualized.

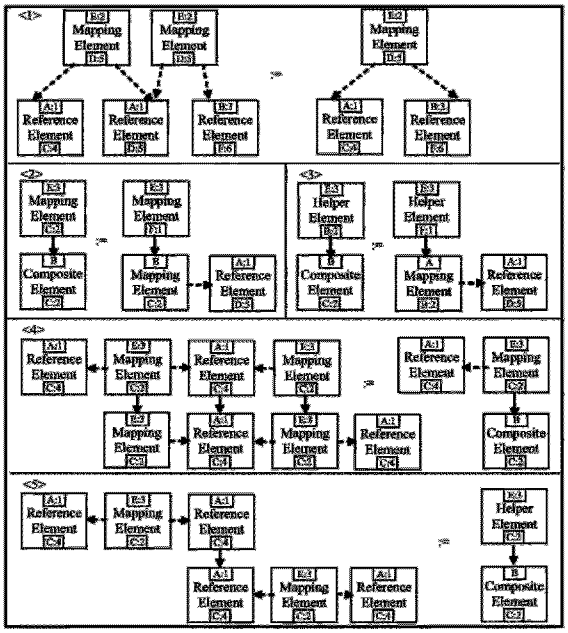


Fig. 5.22. Generalized graph transformation rules for merge operator

For example, merge could be generalized as in Fig. 5.22, which defines five transformation rules. Unlike the merge algorithm, the transformation rules can be customized on the input. Generalized operators do not resolve conflicts, which are to be solved by specific operators. Production <1> merges an elementary mapping, i.e. one to one correspondence as Productions <1>, <2>, and <3> in Fig. 5.20. The output consists of two mappings and one data model together with input elements. In the middle of the left graph of Production <1>, the reference element of the output model is a copy of one of the mapped input elements, the element in model A in this case. The remaining two output reference elements are copies of the corresponding input elements. Two mapping elements on top are output mappings, which map the middle reference element to the left and right reference elements. Production <2>, together with Production <4>, merges the structured mapping elements, such as the equal element of Production <4> in Fig. 5.20. The merge is achieved by making the mapping element and the related reference element a composite element and then extracting the reference element to form the output elements in Production <4>. Similarly Productions <3> and <5> transform the structured helper elements by composing them in <3> and then extracting in <5>.

When the rules are applied to a host graph, the parser matches the nodes in the host graph to the nodes of the same type in the right graph. For example equal in Fig. 5.18 is a mapping element of Fig. 5.22. Because the rules are based on the graph grammar in Fig. 5.20, they can be applied to any host graphs conforming to the grammar.

Similarly, the ModelGen operator in Fig. 5.22 can be generalized and the generic model management visualized. But users cannot customize generalized operators like they do with specific operators. As shown in Fig. 5.17, the two types of operators play complementary roles to provide a visual, generic, and customizable model management environment.

5.5.7 A Parsing Example

This section describes the transformation process of merging input data models and mapping illustrated in Fig. 5.18. The corresponding merge operator is defined in Fig. 5.20. The output includes models A and B (i.e. copies of input), output model C, and mappings Map_{AC} and Map_{BC}.

The first redex found is that of Production <6> in Fig. 5.20, i.e. ISBN of model A. The parser copies ISBN of the model and connects it to the mapping element Equal. A redex of Production <7> is found in the second step, which merges Summary element of model B.

Production <5> is applied in the third step, which merges mappings with helper elements. The helper element Intros and two connected mapping elements, Bio and Intro, are moved to the output model. Two mapping elements in Map_{AC} and Map_{BC} are connected to Bio and Intro respectively. In the forth and fifth steps, the parser applies Productions <4> and <3> respectively.

Due to the space limit, only the last step is shown in Fig. 5.23, when Production <1> is applied. The mapping between Book and EBook is found as a redex and replaced with two mappings. After the application of this rule, the output model C and two mappings between C and A, B are finally produced.

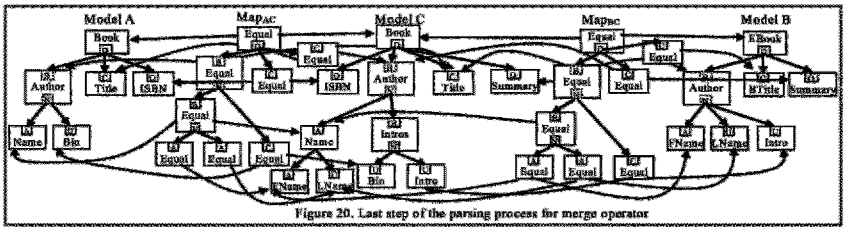


Fig. 5.23. The last step of the parsing process for merge operator

5.6 Summary

This chapter has presented a graphical methodology for specification and interoperation based on the RGG formalism. Once the document structure of an XML dialect and its conversion to an alternative dialect structure are specified through grammar rules and translation rules, a visual transformation environment with a graphical editor, a parser and a translator is automatically generated. XML documents conforming to the former dialect can then be graphically created using the graphical editor, and if desirable, automatically translated into the target dialect. At the schema and meta-model levels, interoperation can be achieved through high level model management operators implemented using the RGG transformation rules. Visual language generation mechanism, to be discussed in Chapter 8, helps handling different levels of operators, i.e. from generalized operators to specific operators, to ease the user's effort.

5.7 Related Work

Using visual programming and visual language approaches to work with XML has mostly been focused on visual query languages, such as VQL (Vadaparty et al. 1993), XML-GL (Ceri et al. 1999), and Xing (Erwig 2000). For example, Xing is a visual language for querying and transforming XML data. It achieves XML transformation and restructuring using some rules that combine the patterns of queries and results returned by queries. It uses nested boxes to represent XML data such what each element tag is written on top of a box while all the attributes of the element are enclosed in the box. Such representation is essentially textual, supplemented by hierarchical boxes surrounding the text.

On the other end of the spectrum, VRDL (Visual Repository Definition Language) (Minas and Shklar 1996) aims at harnessing the massive amount of data on the Web by imposing a logical structure visually and encapsulating chunks of original information into metadata entities. Similar to our approach, VRDL is generated by a visual language generator (Minas and Viehstaedt 1993) and thus supports graphical and syntactic editing. The visual notation used in VRDL is based on and adapted from the Nassi-Shneiderman diagram.

Graph based techniques, such as hyper-graph data model (HDM), and Telos (Mylopoulos et al. 1990), construct schema transformation operators, perform inter-model transformations, and define inter-model links in terms of graph. The notations used by these techniques are defined in a natural language, and thus the ability of automated analysis and transformation of graph-based models is limited. Therefore most of the operations on these data models have to be specified in textual languages. For example, HDM requires considerable effort to specify the transformation rules and links in an appropriative textual language, and Bowers and Delcambre used RDF (Lassila and Swick 1999) to represent model-based information translation (Bowers and Delcambre 2002).

Of the general textual approaches to the automatic translation of XML documents, Xtra (Su et al. 2001) aims at automatic transformations between XML documents by discovering a sequence of transformation operations from the source and target DTD trees. The operations are used to generate an XSL script, which can then be applied to source XML documents to transform them to XML documents conforming to the target DTD. The user needs to be familiar with the XML technology to be able to use the Xtra tool. A similar method is that of Leinonen (Kuikka et al. 2002; Leinonen 2003) which offers semi-automatic transformation between two XML structures based on the input of both source and target DTDs and user-provided element mappings. Other textual approaches include those used in ARTEMIS (Castano and Antonellis 1999) and Clio (Yan et al. 2001), both based on relational schema mappings. Due to the flat nature of relational schemas, such systems could not properly process hierarchical XML schemas. Also based on database schemas, TransScm (Milo and Zohar 1998) uses rules to match similar document components for most common cases, and allows the user to customize the rules for more complex cases. Other approaches include the use of declarative languages (Cluet et al. 1998) and wrappers (Chen and Jamil 2003; Shui and Wong 2003). These approaches offer neither automatic structure validation, nor means for visual representation and specification.

As a visual programming environment built upon the Circus transformation language (Vion-Dury et al. 2002) for XML document transformation, VXT (Pietriga et al. 2001) allows users to construct tree-maps to represent nested document structures. Similar to the approach in this chapter, its transformation is specified by a set of rules, but different in that VXT rules are at the same level of abstraction as XSLT and relies on the transformation power of Circus (Vion-Dury et al. 2002) rather than graph grammars. Additional notations are introduced for constructing the rules by linking the tree elements, which would potentially clutter the display space.

Though model management is a relatively new research area, its promising and exciting potential has attracted much attention and made significant advances in several aspects since it was first proposed (Bernstein et al. 2000). In the transformation perspective, according to Bézivin et al. (2003), model management may be considered the 3rd generation, with text scripts like the *awk* Unix command being the first generation and tree scripts like XSLT being the second.

Various systems for model management have been presented. Cupid (Madhavan et al. 2001; Madhavan and Halevy 2003) and Clio (Miller et al. 2001) match two models and output the mapping between them, i.e. performing the match operator. Merge has been a hot spot in database research area for a long time. Buneman *et al.* (1992) described a theoretical foundation of merge. In the context of generic model management, there are various implementations of the operator, such as Pottinger's approach, which presents the operator based on the BDK algorithm (Pottinger and Bernstein 2003), and data integration project Clio (Miller et al. 2001) that is based on a query language specific to databases or XML schemas. Most of the approaches concentrate on parts of generic model management.

Rondo (Melnik et al. 2003) is the first complete prototype of the generic model management system, which defines the key conceptual structure of models, mappings, and selectors. Melnik et al presented an algorithm for the merge operator as an example, and applied it to XML schemas and SQL views. Rondo represents mapping between two data models by a set of correspondences, rather than by a model. Comparing to the interactive and customizable approach in this chapter, Rondo is like a black box to users and presents no intuitive interface for users to customize.

Model management is also combined with peer-to-peer computing technology (Bernstein et al. 2002) and further used as an infrastructure for future Web data representation, notably the semantic Web (Halevy et al. 2003a). Piazza (Halevy et al. 2003b) offers a language for mediating between data sources over the semantic Web. Piazza describes mapping by an adapted query language and has more sophisticated mechanism to retrieve complex data from RDF and XML documents. The appropriate mapping language is derived from XQuery and is complicated for a Web page designer to map some Web pages to others. Users or designers have to resolve conflicts manually. The complex query language could potentially hinder the deployment of the Piazza system.

Using graphs to represent and manage data models is not new, and there are many proposals based on graph grammars. Rekers and Schürr (1997) presented an ER data model specified by layered graph grammars. Schürr (1994) also proposed the Triple Graph Grammar to represent and support the specification of interdependencies between graph-like data structures. Different from the approach presented in this chapter, the TGG specifies the translation from input graphs to output graphs in a generic fashion and does not consider mappings between input graphs. We explicitly define the relationships between input graphs that represent data models, and construct graph transformation rules for each operator based on the mappings such that the operators are customizable.

Jahnke and Zundorf (1998) presented *varlet*, a database reverse engineering environment based on triple graph grammars. The varlet environment supports the analysis of legacy database systems, translation of any relational schema into a conceptual object-oriented schema. More recent work of Wermelinger and Fiadeiro (2002) focuses on software architecture reconfiguration using an algebraic approach, i.e. category theory. Consistency of model evolution based on real-time UML is further investigated by Engels et al. (2002). These graph transformation based approaches address specific aspects of model management.

Chapter 6 Software Architecture Design

6.1 Introduction

Software architecture and design (Shaw and Garlan 1995) are usually modeled and represented by informal diagrams, such as architecture diagrams and UML diagrams. While these graphic notations are easy to understand and are convenient to use, they are not amendable to automated verification and transformation. The developer has to rely on personal experience to discover errors and inconsistencies in the architecture/design diagrams. She also has to manually transform an architecture/design diagram while needed. These processes are tedious and error-prone. This chapter presents an approach that abstracts UML class diagrams and architecture styles into graph grammars. These grammars enable a high level of abstraction for the general organization of a class of software architectures, and form a basis for various analysis and transformations. In this approach, software verification is performed through a syntax analyzer. Architecture transformation is achieved by applying predefined transformation rules. In general, the presented approach facilitates the following aspects:

- Graphs are used to specify software by distinguishing individual components and their relationships. Using graph grammars as design policies, the presented approach provides a powerful mechanism for syntactic checking and verification, which are not supported by most current tools.
- In addition to software design and verification, the presented approach facilitates a high level of software reuse by supporting the composition of design patterns, and uses graph rewriting techniques in assisting the transformation of software architectures and in reusing the existing products.

The rest of this chapter is organized as follows. Section 6.2 outlines the approach. Section 6.3 illustrates how to verify software design using a graph grammar. Section 6.4 demonstrates the support for the composition of design patterns. Section 6.5 shows the mechanism for software architecture transformation. Section 6.7 summarizes the chapter.

6.2 Designing Architectural Styles

Having introduced the Reserved Graph Grammar in Chapter 2, we explain the definition from building blocks, i.e. components and connectors, to high level specifications, i.e. architectural styles. An architectural style specifies the constraints on configurations of architectural elements (Mehta and Medvidovic 2003).

Architectural styles fall in two broad categories (Garlan and Allen 1994). The *idioms and patterns* refer to global organizational structures, i.e. pipe-filter style. The *reference models* include system organizations that prescribe specific configurations of components and interactions for specific application areas. A common pattern can be easily communicated and understood among a broad group of people while a reference model can be more efficient in a specific domain by enforcing domain-specific constraints. The following description illustrates the specification of a set of common patterns guiding the composition of a real-time system, and a reference model can be prescribed following the same principle.

6.2.1 Components and Connectors

The building blocks of an architectural description language are: 1) components, 2) connectors, and 3) architectural configurations (Medvidovic and Taylor 2000). Components denote various units of computation, and connectors model interactions among components. A configuration representing an architectural structure is specified by connecting components and connectors. In the grammatical approach, nodes are used to represent components, and edges model connectors between components (Kong et al. 2006). A graph grammar "glues" various components and connectors into a meaningful architecture.

As described in Chapter 2, a node in the RGG is organized in a 2-level hierarchy. The name of a node denotes the type of a component, while vertices are used to define *ports*[1] referring to the provided and required

[1] ADLs may differ in the terminology of ports. For example, an interface in UniCon (Shaw et al. 1995) is a *player*.

functionalities. Connectors used to model interactions among components are specified in the same fashion as components (Medvidovic and Taylor 2000). Vertices in a connector node denote roles of communication protocols.

6.2.2 Architectural Styles

Having a vocabulary of architectural elements, i.e. component and connector types, a graph grammar specifies an architectural style, which denotes constraints on configurations of architectural elements (Métayer 1998). A graph grammar is made up of a set of productions, and each production illustrates the composition of sub-systems from the right graph to the left graph. All possible inter-connections between individual components need to be defined in the graph grammar. Any legitimate connection can be derived from a sequence of applications of grammar rules. Conversely, an un-expected link signals a violation on the graph grammar. Therefore, parsing a host graph representing an architecture can validate the structural integrity. The parsing process is a sequence of R-applications, which is modeled as recognize-select-execute as explained in Chapter 2. This process is continued until no production can be applied. If the host graph is eventually transformed into an initial graph, the parsing process is successful and the host graph is considered to represent a valid architecture satisfying the structural requirements enforced by the graph grammar.

The pipe-filter style is made up of pipes and filters. A filter having a set of input and output ports reads streams of data on its input and produces streams of data on its output. A pipe having a source role and a sink one transforms information from the output of one filter to the input of another one. Building on pipe and filters, an architecture committing to the pipe-filter style must respect some structural constraints, e.g. a source role needs to attach to an output port and a sink role to an input port.

In order to investigate whether a user-defined architecture observes configuration constraints of a style, a graphical representation of an architecture friendly to end-users needs to be automatically transformed into a node-edge diagram, suitable for the RGG graph transformation engine to analyze the structural integrity. In general, a node in the RGG indicates a component (a connector) and the vertices within the node represent ports (roles). For example, a pipe is represented by a node labeled *Pipe*, which has two vertices named *Sink* and *SRC* denoting the sink and source roles respectively (the vertices should be named with a clear meaning and vertex labels within the same node are distinct from each other). In order to denote a filter with an arbitrary number of ports, we represent a filter by a graph instead of a single node as the following:

- A node labeled *Filter* with two vertices *I* and *O* represents the filter component.
- Nodes labeled *I_Port* with two vertices *P* and *F* indicate the input ports within a filter component. An edge connecting the *F* vertex of an *I-Port* node and an *I* vertex of a *Filter* node indicates the belonging relationship between an input port and a filter. The other vertex *P* attached to the *Sink* vertex of a *Pipe* node represents a data flow.
- The output ports are processed in the same fashion as the input ports.

Fig. 6.1(a) shows an architecture of the pipe-filter style. According to the above principle, a corresponding internal graphical representation used by a RGG parser is automatically generated as shown in Fig. 6.1(b).

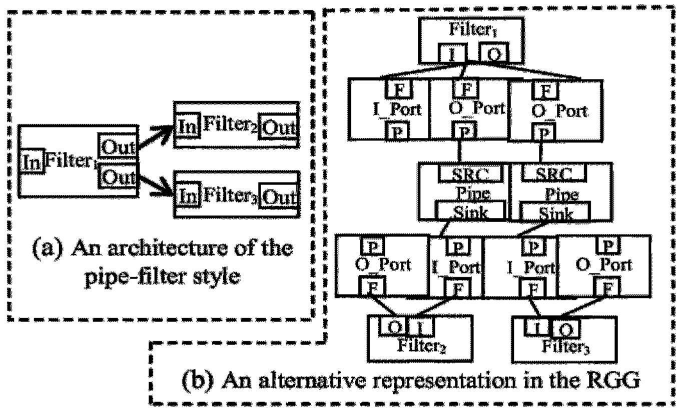


Fig. 6.1. Graphical representations of a software architecture

The necessary connectivity among components is stated through the right graph of a production, i.e. the application condition, which a host graph must fulfill. Application conditions, however, cannot express the condition which cannot be present for a production to be applicable. Therefore, the *negative application condition* (Ermel et al. 1999) is introduced. For example, Production 1 in Fig. 6.2 uses the negative application condition, expressed by a rectangle crossed by a line, to define the condition that an initial graph is derived from a single component *Filter* without any other component.

Fig. 6.2 gives a grammatical specification prescribing a pipe-filter style. In particular, Production 1 demonstrates that an initial graph denoted by λ is abstracted from a *Filter* node. Production 2 abstracts two filters into one filter. Production 3 states a data flow between a pair of pipes.

The client-server style has two types of components, i.e. the server and the client. The server is able to provide services to clients. A dispatcher being a connector (represented by a node labeled *Dis* as shown in Fig. 6.3) serves to dispatch requested services to appropriate clients. Fig. 6.3 presents the rules constructing a class of architectures committing the client–server style.

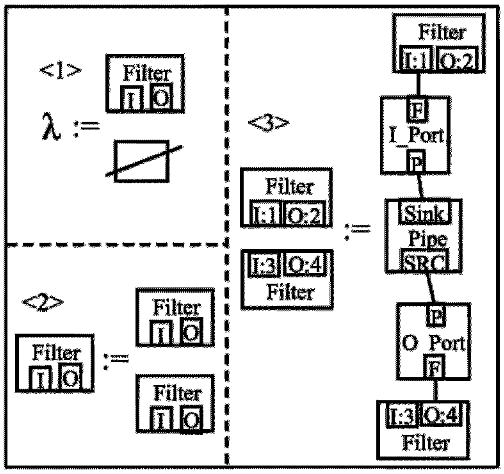


Fig. 6.2. A graph grammar defining the pipe-filter style

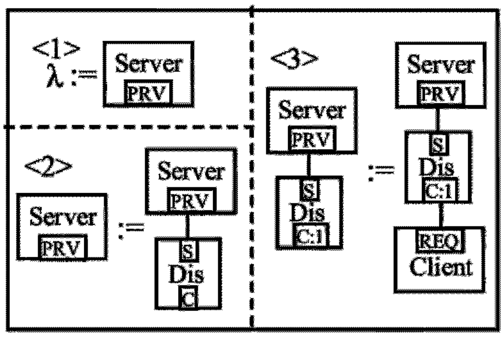


Fig. 6.3. A grammatical specification prescribing a server-client style

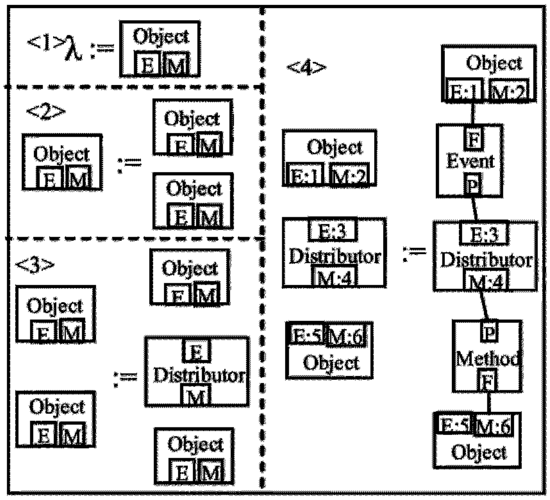


Fig. 6.4. A graph grammar defining the event-based style

Components in an event-based style, represented by *Object* nodes, interact with each other through event broadcast, i.e. the occurrence of an event can invocate methods in components. The connector *distributor* takes announced events and transforms them into method invocations. Since an *Object* can be associated to an arbitrary number of events and methods, we represent an object through a graph in the same fashion as a filter. Fig. 6.4 illustrates a graph grammar defining the structural properties shared by architectures within the event-based style.

6.3 Designing an Architecture

Supporting a set of general architectural styles, a graph transformation engine within a generated visual language environment can validate user defined architectures against architectural styles. The following section goes through a toll gate example to explain the designing of a system using multiple general architectural styles.

6.3.1 Toll-Gates

In a road traffic pricing system, drivers of authorized vehicles are charged at toll gates automatically. The tolls are placed at special lanes called *green* *lanes*. A driver has to install a device (called an *ezpay*) inside his/her vehicle's windshield in order to pass a green lane. The registration of an authorized vehicle having an *ezpay* includes owner's personal data (such as name, date of birth, driver license number, bank account number and vehicle registration number).

Each toll gate has a sensor that reads *ezpay*. The information read is stored by the system and used to debit the respective accounts. When an authorized vehicle passes through a green lane, a green light is turned on, and the amount being debited is displayed. If an unauthorized vehicle passes through it, a yellow light is turned on and a camera takes a photo of the vehicle's license plate.

6.3.2 Designing a Toll-gate system

Based on a set of general architectural style defined through graph grammars, a visual architecting environment can be automatically generated. In the architecting environment, users without any knowledge of graph grammars can define software architectures by drawing graphs. The structural integrity of such graphs can be validated by a graph grammar parser. In the environment, users can design a system or sub-system by choosing an appropriate style and customize components and connectors inherited from a vocabulary within the style.

A toll-gate system is made up of a database system and several gates. The database stores customs' information, which can be updated and retrieved upon requests coming from the gates. Such a communication model is implemented as the sub-system$_1$ in Fig. 6.5(a) using a server-client style. The node labeled *DB* inherited from the component type *Server* represents a database, and nodes labeled *Gate* inherited from the type *Client* denote toll gates. The node *Dis* represents a connector sending requests to the database and dispatching replies to appropriate toll gates.

A toll gate needs to scan the identification of an arriving vehicle. The sub-system$_2$ applies the event-based style to implement the interactions between toll gates and vehicles (The node *Gate* is a common component in the sub-system$_1$ and sub-system$_2$, and it is inherited from both the *Client* type in the client-server style and the *Object* type in the event-based style). In particular, the vertex named *Arrive* in the *Car* node is an event port invoking the scanning operation denoted by the *Scan* vertex in the *Gate* node.

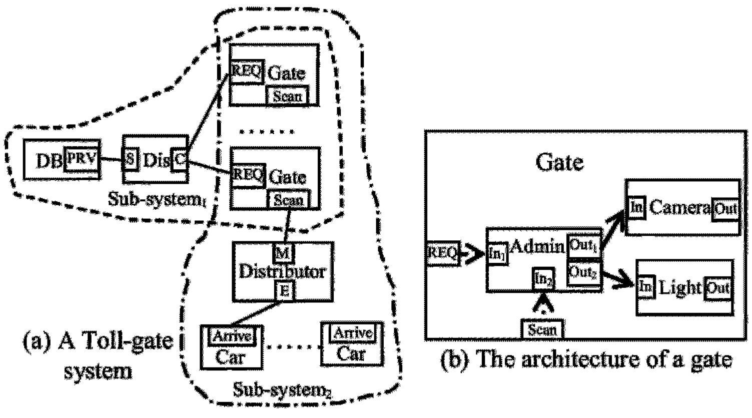


Fig. 6.5. An architecture of the Toll-gate system

We can further elaborate the design of the toll gate using a pipe-filter style as shown in Fig. 6.5(b). Three components constructing a gate are inherited from the filter type, and are denoted by nodes *Admin*, *Camera* and *Light* representing an administrator, a camera and a signal light respectively. Directed edges in the Fig. 6.5(b) denote pipes connecting different filters.

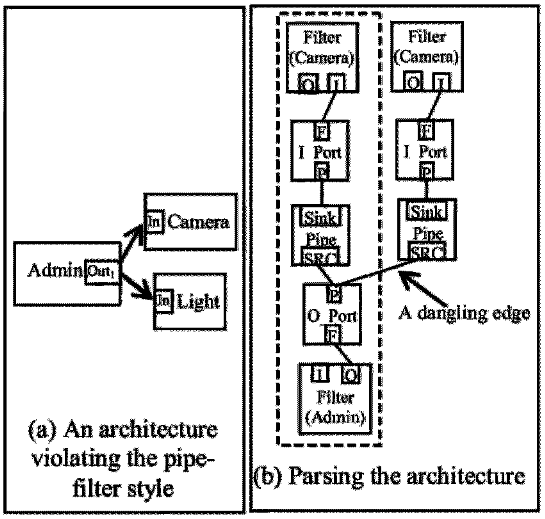


Fig. 6.6. Checking an architecture

Based on graph grammars specifying a set of architectural styles, users can implement sub-systems with an appropriate style and incrementally glue sub-systems into a complete architecture. The structural integrity of each sub-system can be verified through the parser. For example, Fig. 6.6(a) shows an architecture violating the pipe-filter style, which requires that at most one pipe connects an output port to an input port between a pair of filters. The violation can be detected by the RGG parser. If an unmarked vertex in the right graph of a production matches a vertex *v* in the redex of a host graph, then all edges connecting to *v* have to be completely inside the redex. According to this above embedding rule, an isomorphic graph (surrounded by a dotted rectangle in Fig. 6.6(b)) matching the right graph of Production 3 in Fig. 6.2 is not a valid redex due to the *dangling edge* (Rozenberg 1997) as shown in Fig. 6.6(b).

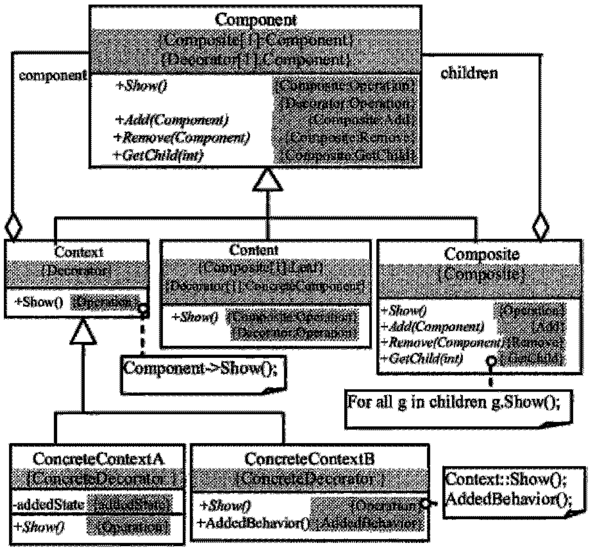
6.4. UML Class Diagram Verification

In this section, we first use an example to illustrate how to represent a class diagram using a RGG diagram. We then define a graph grammar for the UML class diagram. A parser can verify some properties of the design. In the next section, we show how this graph grammar can help visualizing design pattern applications and compositions in their class diagrams.

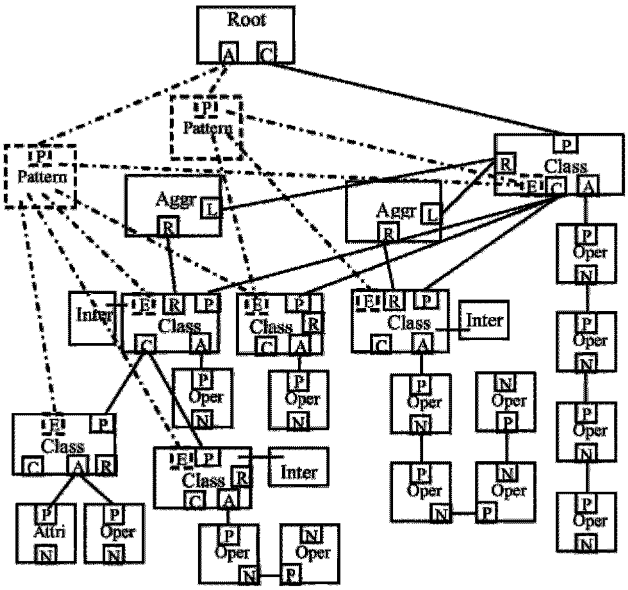
6.4.1 Class Diagrams

Class diagram, one of the most popular diagrams in UML, visually models the static structure of a system in term of classes and relationships between classes (Booch et al. 1999). In order to verify the structure of a class diagram, we translate the class diagram (Fig. 6.7(a)) into a node-edge format (Fig. 6.7(b)), on which the RGG parser operates.

In the class diagram, classes are represented by compartmentalized rectangles. In a node-edge diagram, a node labeled *Class* denotes the first compartment containing the class name. A set of nodes labeled *Attri* represents attributes in the second compartment. Nodes are sequenced by linking two adjacent attributes in the same order as displayed in the compartment, and the sequence is attached to a class by linking the first *Attri* node with the *Class* node. Operations in the third compartment are processed in the same fashion as attributes by replacing *Attri* with *Oper* nodes.



(a) A class diagram



(b) The corresponding RGG diagram

Fig. 6.7. A class diagram and its corresponding RGG diagram

Associations denoted by straight or diagonal lines in UML carry information about relationships between classes. In a node-edge diagram, a node labeled *Asso* is used to symbolize an association. A line connecting an *Asso* to a *Class* node holds the association. Associations may be named. In order to indicate the direction in which the name should be read, the vertex labeled *R* inside an *Asso* node is connected to the *Class* node designated by the verbal construct, and the vertex labeled *L* to the other *Class* node. On the other hand, if the order is unimportant, we ignore the difference between *R* and *L*. *Aggregation* and *Composition*, two special types of associations, are translated in the same way as associations.

In UML, the *generalization* specifies a hierarchical relationship between a general description and a specific description. In the node-edge representation, a line, which links from the vertex labeled *c* in a *Class* node to the vertex labeled *p* in the other *Class* node, designates the generalization relationship from the former class to the latter. In other words, the vertex labeled *c* indicates the general class, and the vertex labeled *p* denotes the specific class accordingly.

We introduce a new node to the node-edge representation, namely *root*, without a counterpart in the class diagram. The *root* is connected to any *Class* node representing a class without a super-class. The introduction of the *root* node is to facilitate the parser to verify the structure of a node-edge diagram.

A graph grammar abstracts the essence of structures. It, however, is not suitable to convey precise information visually. We store specific information into attributes. For example, association names are recorded in attributes associated with *Asso* nodes. Those values of attributes can be retrieved and evaluated in the parsing process.

Fig. 6.7(a) illustrates a class diagram and Fig. 6.7(b) presents its corresponding node-edge diagram recognizable by its RGG. The shaded texts in Fig. 6.7(a) represent an extension to UML with pattern names, and the dotted rectangles in Fig. 6.7(b) correspond to the extended UML (Dong and Zhang 2003). We will discuss the pattern aspects in Section 6.5.

A graph grammar can be viewed as a style that any valid graph should hold, i.e. any possible inter-connection between entities must be specified in the grammar. Each production specifies the relationships between local entities. Combining all the productions together, a RGG grammar defines the way of constructing a valid class diagram through different entities represented by nodes with different types.

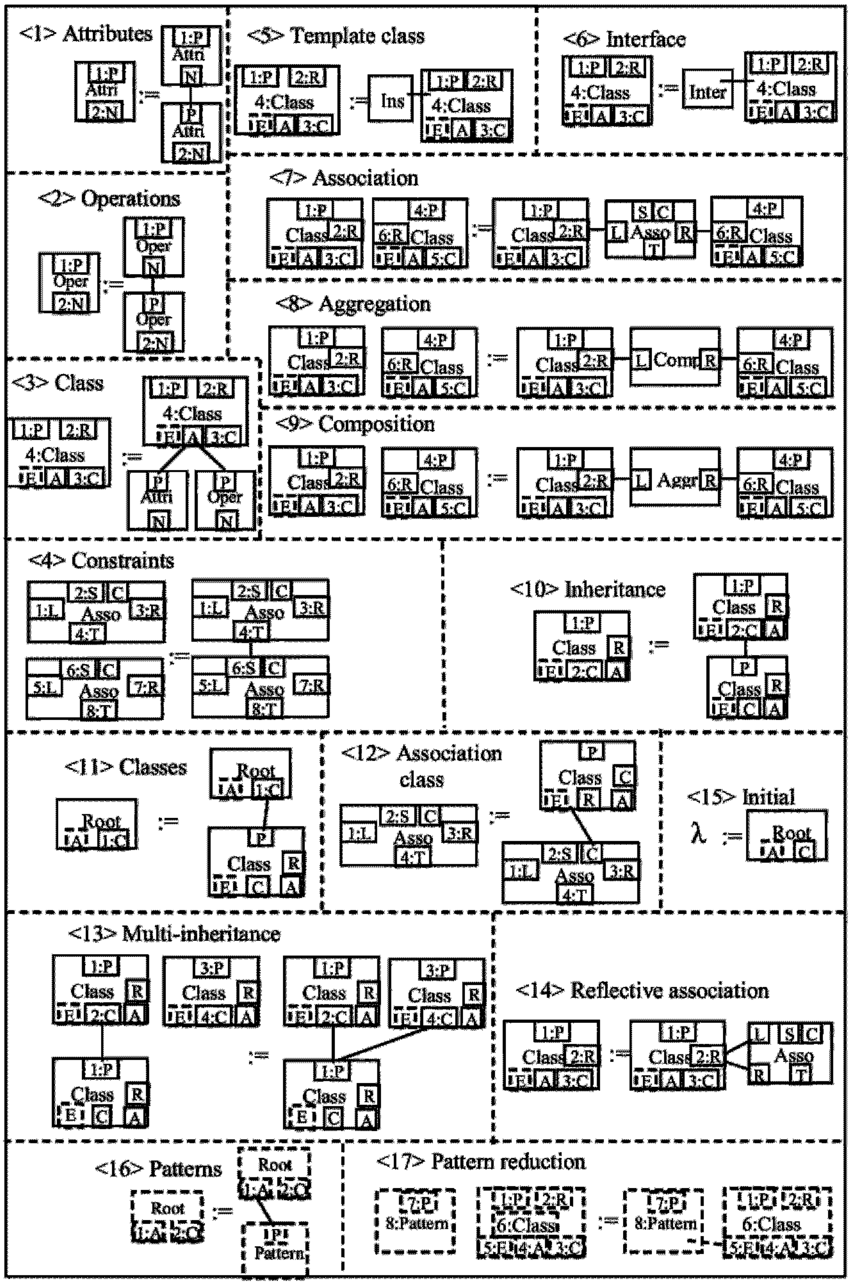


Fig. 6.8. The graph grammar for class diagram

Fig. 6.8 presents the RGG defining class diagrams. Production 1 reduces two attributes into one node, which is treated as one entity in later applications. Repetitive applications of Production 1 reduce attributes of the same class to one entity, which can be applied by Production 3 later. Productions 1 and 2 serve to reduce a sequence of attributes and operations. Production 3 specifies the class structure by attaching sequences of operations and attributes to a *Class* node. Production 4 defines the constraints between associations. Production 5 specifies the template class, followed by the production presenting the interface. Productions 7, 12 and 14 define associations, and Productions 8 and 9 specify aggregation and composition respectively. Productions 10 and 13 specify generalization. Production 15 represents the initial state. The nodes and vertices in dotted rectangles define pattern-extended class diagrams, which will be explained in Section 6.5.

6.4.2 Automatic Verification

There are already some tools supporting the general syntactic checking on class diagrams. However, they are not capable of performing specific verification. For example, multi-inheritance may cause ambiguity, it is desirable to prohibit it when modeling software written in conventional programming languages. Each production specifies a local structure. By "gluing" separate structures together, repetitive applications of various productions can generate a complete structure. A graph specifying a structure is invalid if it breaks at least one relationship specified in any production. For example, Production 6 in Fig. 6.8 defines that one interface can only attach to one class. If an interface is designed to be related to more than one class, a parser can indicate a violation of Production 6.

The following example illustrates how to verify inheritance relationships between classes. In Fig. 6.8, Production 10 defines the scenario of single inheritance, and Production 13 specifies that of multi-inheritance. Since any valid relationship between components can be eventually derived from a graph grammar, removing Production 13 implicitly denies multi-inheritance. In the right graph of Production 10, the edge indicates an inheritance relationship between the classes. According to the marking mechanism explained in Chapter 2, the unmarked vertex *p* in the bottom class node representing a sub-class requires that any class can only inherit from one class. On the other hand, the marked vertex *c* in the top class node representing a super-class defines that one super-class can have more than one sub-class, which does not contradict with single inheritance. If the multi-inheritance as illustrated in Fig. 6.9(a) occurs, the application of Production 10 causes a dangling edge (Rozenberg 1997), which is not allowed in the RGG formalism. Considering the scenario where one class has more than one sub-classes, a successful application is shown in Fig. 6.9(b).

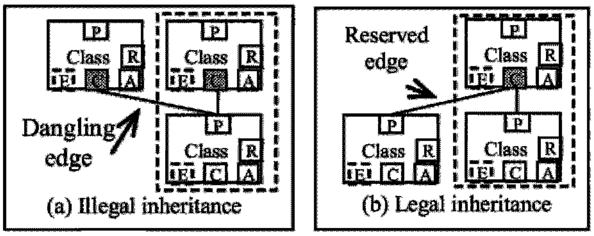


Fig. 6.9. Inheritance verification

6.5 Design Pattern Visualization

UML (Booch et al. 1999) provides a set of notations to demonstrate different aspects of software systems. However, it is still not expressive enough for some particular problems, such as visualizing design pattern applications and compositions (Dong and Zhang 2003). In this section, we use the RGG formalism to visualize design patterns in their class diagrams.

Design patterns (Gamma et al. 1995) document good solutions to recurring problems in a particular context, and their compositions (Keller and Schauer 1998) are usually modeled using UML. When a design pattern is applied or composed with other patterns, the pattern-related information may be lost because UML does not track this information. Thus, it is hard for a designer to identify a design pattern when it is applied or composed. The benefits of design patterns are compromised because the designers cannot communicate with each other in terms of the design patterns they use when the design patterns are applied or composed. Several graphic notations have been proposed to explicitly represent pattern-related information in UML class diagrams (Dong and Zhang 2003). While all these solutions need to attach additional symbols and/or text, they all suffer scalability problem when the software design becomes very large. We propose a solution that can dynamically visualize pattern-related information based on the RGG. As shown in Fig. 6.8, we introduce a new type of nodes, called *pattern*, which denotes a specific pattern, and pattern-related information is expressed by linking a pattern node with its associated class nodes. Fig. 6.7(b) presents the corresponding node-edge diagram by highlighting the newly introduced nodes and edges with dotted lines.

A syntactic analyzer can dynamically collect separate pieces of information, and reconstruct them into an entity. In the process of parsing, the sequence of applications of Production 17 in Fig. 6.8 collects all classes belonging to the same pattern together. For example, if the user clicks the *composite* class in Fig. 6.7(a), the *component* class, *content* class and *composite* class, which belong to the Composite pattern, are highlighted. Therefore, there is no need to attach any additional information on the original class diagrams.

6.6 Software Architecture Transformation

The architectures of software systems are usually not fixed. With the changes of requirements and contexts, software architecture may be transformed into a new configuration. Furthermore, a high-level software architecture needs to be refined into detailed architecture (Moriconi et al. 1995) in software development. This transformation process can be tedious and error-prone without tool support. This section illustrates the automated transformation of software architecture between different styles. Graph rewriting provides a device for reusing existing products by performing a transformation.

A software architecture style defined through a RGG characterizes some common properties shared by a class of architectures. To satisfy new requirements and reuse current designs, an architecture with one style needs to evolve into another with a more appropriate style in the new contexts. In general, software architecture transformation proceeds in two steps: a) verify the style of an architecture; b) transform an architecture from one style to another style.

Assume that a system is originally implemented in a client-server style, only consisting of one server storing all data. In order to retrieve data, clients must send requests to and receive responds from the server. This communication pattern is abstracted into a graph grammar shown in Fig. 6.10(a), and an architecture with that style is illustrated in Fig. 6.10(b).

With the increase of the amount of data and communication, one server may not be able to bear clients' requests. On possible solution is to distribute data to different servers. Therefore, we need to transform the current style to a more advanced one. We divide servers into control server and data server. A system can only contain one control server, but may have several data servers. A client sends requests to the control server, which forwards them to an appropriate data server. Then, the data server directly replies to the client. Such a communication pattern is defined in Fig. 6.10(c), which is achieved through the transformation rule in Fig. 6.10(d).

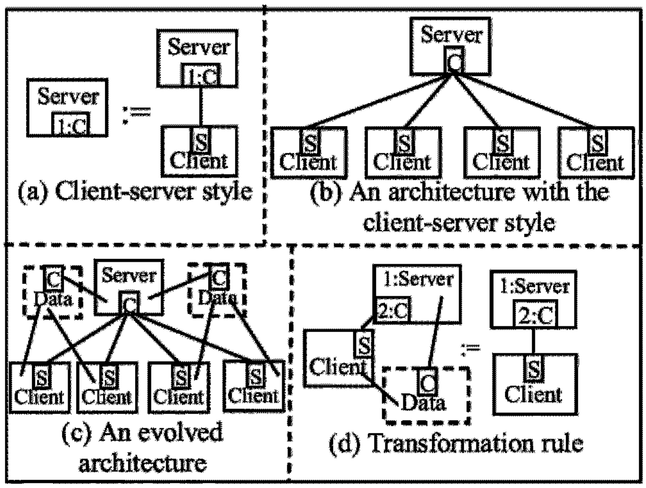


Fig. 6.10. Architectural transformation

We go through another example to illustrate the architecture transformation. A simple pipe and filter system without feedback is shown in Fig. 6.11(a), where a circle represents a task and a directed edge indicates a stream between tasks. Correspondingly, a node labeled *Str/Task* simulates a stream/task in the node-edge representation. An edge connecting the *R/L* vertex in a *Str* node to the *I/O* vertex in a *Task* node expresses an incoming/outgoing stream. Fig. 6.11(c) illustrates the node-edge representation for the system shown in Fig. 6.11(a). Productions defined in Fig. 6.11(b) abstract the communication pattern in pipe and filter systems without feedback. By allowing an edge between *Task* nodes, which designates a feedback between tasks, the transformation rule given in Fig. 6.11(d) transforms a system without a feedback to one with feedback. Fig. 6.11(e), where the dotted edges represent feedbacks, illustrates the system with feedback after we apply the transformation rule to the example in Fig. 6.11(a).

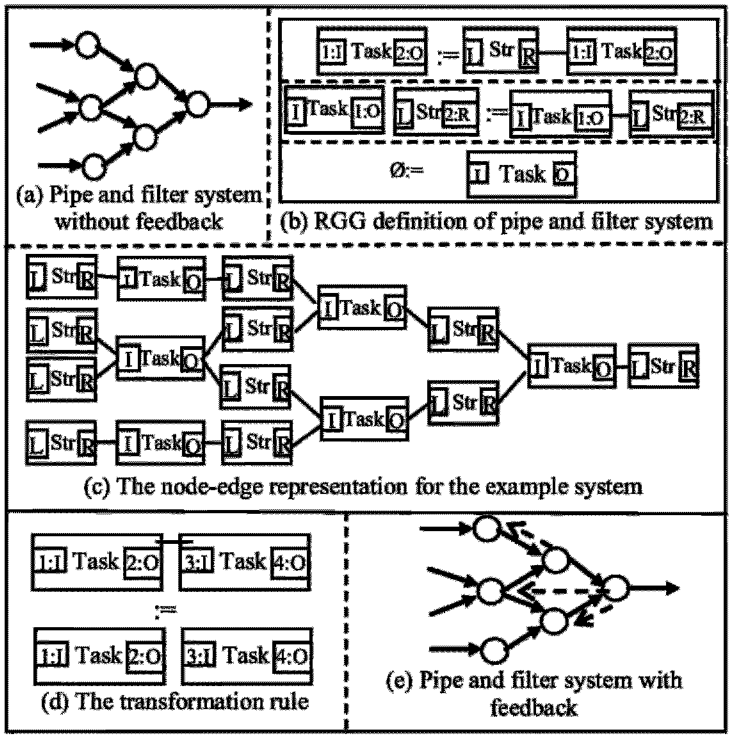

(a) Pipe and filter system without feedback
(b) RGG definition of pipe and filter system
(c) The node-edge representation for the example system
(d) The transformation rule
(e) Pipe and filter system with feedback
Task
Str
Ø:=
:=

Fig. 6.11. Pipe and filter system

6.7 Summary

Based on a graph grammar formalism, this chapter has presented an approach for software architecture definition, verification and transformation. Through this approach, UML notations can be easily translated to the graphical notations adopted by the RGG formalism. The conformity makes the approach consistent with current design tools. In general, the approach can provide the following benefits:

1. **Consistent**: It expresses software architectures in terms of "box and line" drawings (Allen and Garlan 1994), which meets the common practice of software engineers (Métayer 1998).

2. **Scalable**: The underlying graph grammar formalism is applicable to various classes of graphs. It is easy to accommodate new compo-

nents by extending the graph schema and revising corresponding grammar rules, and thus support software reuse.

3. **Automatic**: Automatically generated by a visual language generator like VisPro (see Chapter 8), a transformation tool is capable of syntactic checking of software architectures. Automatic transformation from one architecture style to another assists software engineers to reuse existing products in the new application contexts.

Using the attributes in the RGG, we can express semantic information of software architecture and design. In addition to syntactic checking and transformation, semantic analysis and transformation need to be investigated.

The grammatical approach is promising in providing an intuitive yet formal method to the specification of software architectures. Graph grammars are used to define architectural styles, which impose constraints on the interaction among components. With the well-established theoretical foundation, the grammatical approach can automatically validate the structural integrity and reveal the hierarchy of a user-defined software architecture through parsing.

The graph transformation tool can be considered an authoring language generator, which can generate a specific design environment whenever needed. A software engineer without any knowledge of graph grammars will be able to use the generated environment to design software architectures by drawing graphical structures. Syntax check can be automatically performed within the environment.

6.8 Related Work

Many ADLs, such as Wright (Allen and Garlan 1997) and Rapide (Luckham et al. 1995), 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 (Baresi et al. 2003). 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 above approach can overcome the problem by automatically generating a style-specific design environment. Then, users without any graph grammar knowledge can directly specify and manipulate software architectures in terms of box-and-line drawings. Based on well-established theoretical foundations, a graph transformation engine underlying the environment can verify the structure of user-defined architectures. Therefore, the graph grammar approach is visual yet formal.

The Unified Modeling Language (UML) (Booch et al. 1999) provides a family of design notations to model various aspects of systems. Being a general purpose modeling language, the UML has also been applied to defining software architectures. Medvidovic *et. al.* (2002) systematically presents two strategies to model software architectures in the UML. Focusing on the usability of concepts, Garlan *et. al.* (2002) proposes several alternatives to map concepts from ADLs to the UML. Both works exhaustively discuss the strength and weakness of each method. Emphasizing on the analysis and validation of designed models, Baresi *et. al.* (2003) uses the UML notations to specify the static aspect of structural styles paired with graph transformation for modeling dynamic reconfiguration. Selonen and Xu (2003) apply UML profiles to software architecture design process and software architecture description. 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.

In general, class diagrams provide a declarative approach to defining instances of an architectural style. On the other hand, the Reserved Graph Grammar proposes a grammatical approach to specifying architectures in a constructive and incremental fashion. Though the declarative approach is easier to understand, the constructive and incremental method is more suitable for analysis. Furthermore, parsing an architecture can reveal the hierarchical structure of the architecture.

Some researchers apply the typed graph approach to define architectural styles. For example, Wermelinger and Fiadeiro (2002) use typed graphs to specify all possible connections between components. Briefly, a typed graph $<G, t>$ is a graph G equipped with a morphism $t: G \rightarrow TG$, where TG is a fixed graph, i.e. the type graph (Corradini et al. 1996). 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 (1995) apply the diagrammatic syntax to express software architectures, and use it for pattern matching. Their work focuses on exploiting the composition of software architectures. Taentzer *et. al.* (1998) uses 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 to the structure of the distributed system itself. This work emphasizes on modeling dynamic changes of distributed systems rather than specifications of structural composition.

Métayer (1998) presents 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, the approach presented in this chapter is more expressive in specifying software architectures by allowing arbitrary graphs in both left and right graphs of a production. Furthermore, the methodology is supported by a set of tools. Using the Reserved graph grammar as the underlying formalism, a visual language generator can automatically generate an application-specific design environment, as shown in Chapter 8. The environment is able to verify the structural properties of software architectures in terms of graphs.

Radermacher (1999) discussed graph transformation tools supporting the construction of an application conforming to a design pattern, which is specified through graph queries and graph rewriting rules. A prototype can be generated by the PROGRES environment (Schürr et al. 1999). Since the presented approach conforms to UML, it has a broader acceptance and application scope than the above tools.

Based on the theoretic foundation of term rewriting systems, Inverardi and Wolf (1995) apply the *Chemical Abstract Machine* (*CHAM*) (Berry and Boudol 1992) model to the architectural description and analysis. Briefly, software systems are viewed as chemicals whose reactions are controlled by explicitly stated rules. Wermelinger (1998) further proposes two CHAMs, i.e. the creation CHAM and the evolution CHAM, to define the architectural style and the reconfiguration respectively.

Karsai *et. al.* (2003) propose 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 program 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 a rule-based paradigm, i.e. a RGG grammar, to define a language. Supported by a formal basis of graph grammars, the rule-based specification is more suitable for reasoning and verification.

The approach described in this chapter is also inspired by the development of the Aesop system, the effort of adapting the principles and technology of generic software development environment to provide style-specific architectural support (Garlan et al. 1994). The Aesop system 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.

Chapter 7 Visual Web Engineering

7.1 Introduction

The development of Web sites with complex interconnections of large number of Web pages so far has been largely an ad hoc process. There has been no commonly accepted methodology, which supports ease of design, navigation, and maintenance of sophisticated Web sites. As the number of Web sites is increasing in an exponential order, with the huge information space provided by the Web, users become increasingly confused when they navigate a growing number of Web sites; finding the right information also takes longer time. The problems are partially due to the unstructured nature of the current organization of Web sites. For example, in most of the existing Web browsers, the process of jumping from one location to another could easily confuse the user. The main reason for this is that the user does not know the current context of space with respect to the overall information space.

Attempts have been made to develop tools and facilities to support Web site construction, although most of these tools are designed only for one stage of Web design, navigation, and maintenance.

Designing a *good* Web page is considerably easier nowadays. There are many guidelines describing in details the so-called good design of Web pages (Conger and Mason 1998). Siegel (1997) classifies three generations of sites ranging from default backgrounds with wall-to-wall text used in the first generation through the second generation of visual treatments such as menus and Web maps. The third generation of Web sites allows users to pursue paths designed for their needs and interests. We are now in the third generation and moving towards a more personalized multimedia capable Web. Hinton (1998) further discusses how an organization could maintain and design its Web resources with such paths. More and more organizations are embracing the idea of personalized Web site for different types of users (Maarek and Shaul 1997). Personalized paths designed for different individuals would enable an organization to tailor its priority and services for the individuals according to their values to the organization.

A variety of tools, including navigational tools such as browsers and lenses (Huang et al. 1998a; Muchaluat et al. 1998), and history lists (Frecon and Smith 1998), have been developed to assist users to overcome the problem of finding information in the unstructured Web space. WebOFDAV (Huang et al. 1998a) also tries to help the user to visually navigate the Web by displaying a sequence of small visual frames corresponding to user's focuses of attention. Yet, these approaches do not solve the navigation problems through structured Web design since Web design and navigation are not integrated and problems in each domain are tackled separately.

In order to improve the design and navigation of WWW, a well-designed Web structure is desirable. Better tools are needed that enforce structure in the design phase, while supporting fully integrated maintenance and navigation capabilities. A complicated Web system can be made more structured and navigated more easily through graphical visualization and graphical interactions. More importantly, maintaining a uniform view throughout the design, navigation, and maintenance cycle can reduce considerable development effort and enhance the navigation efficiency. The goal of this chapter is to propose an integrated view throughout the Web development cycle. The major advantages of this approach are the following:

- A visual approach to constructing and navigating Web sites is easier to comprehend than the textual form. A novice user without any programming experience would find the visual approach intuitive if the visual representation could reflect one's mental image of a Web structure.
- Automatically generated by a visual language generator, the graphical Web construction and navigation tool can be rapidly prototyped and enhanced to meet the end-user's needs. The generated tool is a syntax-directed visual editor that is capable of syntactic checking of any constructed Web graph. The Web site design and navigation share the same graph formalism so that the user's mental map is preserved.
- A Web site can be maintained using a site visualization tool that shows the site in the same graphical format as in the design stage. The full integration of the design tool with a Web site visualization tool also allows a user to construct new Web sites through reverse engineering based on some existing site structures and contents.

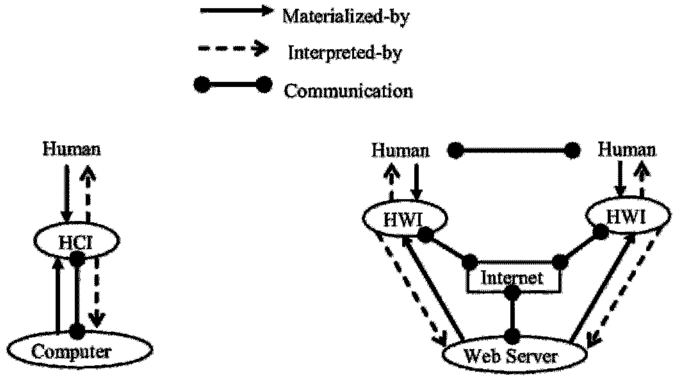
7.2 The Human-Web Interface

This section proposes a uniform view of the design, maintenance, and navigation of the Web, which we call the *Human-Web Interface* (HWI).

To compare HWI with traditional HCI (human-computer interface), we consider the following three aspects: the device for which the interface is suitable and designed, main functionality of the interface, and the target of the communication that the interface facilitates.

- **Device**: A HWI could be installed not only on a computer, but also on a PDA (portable digital appliance), a mobile phone, or a television set. In the latter case, the HWI needs no Web design function and thus would not be equipped with a graph editor and Web site generation engine (as described later in this chapter). The display could also be much more simplified. Taking a PDA for example, the display may only include clickable texts and running texts for navigation and browsing, possibly with a voice interface as in WebViews (Freire et al. 2001).
- **Functionality**: The major role of a HWI is to act as a window to the world while a HCI could just be for a standalone computer. Therefore, the main objective of a HWI is to facilitate information gathering and retrieval while that of a HCI is to facilitate operations on a computer.
- **Communication target**: Related to the above difference, the communication target of a HWI is human while that of a HCI is machine. The human-to-human communication through HWIs may be direct, such as in a Web-based net-meeting, and indirect as in usual Web browsing. To support indirect human-to-human communications in various professional domains, we need commonly understandable and agreed communication protocols. The XML standard (W3C 2004a) has been motivated precisely for this reason.

As illustrated in Fig. 7.1(a), in the traditional human-computer interaction, the human user's intention is materialized through the HCI and interpreted/executed by the computer, which in turn outputs results through the HCI to be interpreted by the user. Fig. 7.1(b) shows the human-to-human communication model, realized indirectly through human-Web interfaces, which communicate to the Web server via the Internet. A full human-to-human communication path is described as: a human user's intention can be materialized on a HWI, which is then interpreted by the Web server according to the predefined HWI syntax and semantics (i.e. Web graph grammar discussed later); another HWI on the other end materializes and presents the user's intention according to the Web server, and the other human user interprets and understands what is presented on the HWI.



(a) Human-Computer interaction model (b) Human-Web interaction model

Fig. 7.1. Human-to-computer communication in HCI and human-to-human communication in HWIs

Both HCI and HWI aim at enhancing the usability and thus the user's productivity. They may therefore be developed based on similar conceptual architectures. The Model-View-Controller paradigm, or MVC for short, has been successfully used to build user interfaces in Smalltalk (Krasner and Pope 1988). As one of the earliest successful object-oriented programming languages, Smalltalk supports construction of new interactive systems based on existing ones. MVC consists of three main objects as shown in Fig. 7.2: *Model, View* and *Controller*. *Model* represents the application semantics, and its screen presentation is managed by *View*. *Controller* defines the way in which the user-interface reacts to user inputs.

Based on the MVC paradigm, we propose a HWI framework as shown in Fig. 7.3, where "Graph Editor and Navigator" corresponds to *Input* in MVC, "Web Browser" corresponds to *Display*, "Filters" and "Display Markup" correspond to *View*, "Customizer" and "HWI Engine" correspond to *Controller*, and "Web Graph Grammar" and "XML Database" correspond to *Model*. The framework consists of the support for three major activities: Web site design, navigation and browsing, and maintenance and updating. The front-end of the user interface consists of a *Graph Editor and Navigator* (GEN) for Web site construction and navigation that is capable of automatic graph layout, and a Web browser that could be

Netscape or Internet Explorer. This combined front-end forms the human-web interface (HWI).

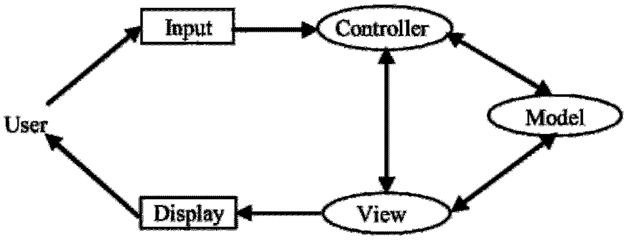


Fig. 7.2. The Model-View-Controller paradigm

The Web designer uses the *Graph Editor* of GEN to design and construct Web sites as graphs to be transformed and processed by the *HWI Engine.* The Engine performs grammatical check of the constructed graphs according to the predefined Web graph grammar, transforms the validated graphs into either XML documents or inter-related HTML files, and generates an internal data structure for debugging and maintenance purposes. If the generic document structure is desirable, XML document structures will be generated and stored in the XML database. The HWI engine is able to transform from one XML to another, or from an XML description to an HTML display format according to the predefined transformation grammar (Zhang et al. 2001c).

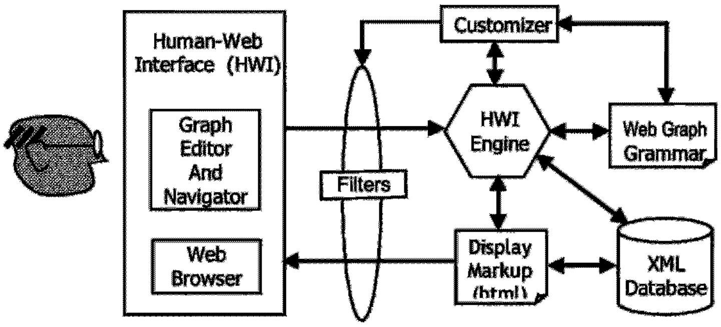


Fig. 7.3. The HWI framework

The HWIT framework supports several modes of displaying, including Level view, Domain view, Category view, Pattern view, and Constraint view. These views are implemented by different *filters* that are also shown in Fig. 7.3. These views and associated filters are described in Section 7.5.4. Filtering rules can be defined on various structures, including graph structure, Web context, and document structure (Huang et al. 1998b). Other conditions may be defined to facilitate more specialized filters. Web designers can design or customize their own filters to suit their specific application purposes. The *Customizer* allows a Web designer or webmaster to define other desired filtering criteria, integrity conditions suited for maintenance, and syntax-directed operations associated with the Web graph grammar. For example, the user may define an integrity condition through the Customizer that no page should belong to more than one group. Web designers may also customize their designs, such as the use of graphical notations, the way in which the site will be navigated, etc, to suit the needs of domain-specific applications.

7.3 Using the HWI Tool

The HWI framework provides a Web designer with a uniform graphical view for the effective design and maintenance of Web sites, and allows users to navigate the Web site graphically by direct manipulation and information filtering as desired. The Web designer designs and generates a Web site by drawing the Web graph that conceptually represents the site structure. Navigation and maintenance of Web sites are performed on the same Web graph by the user.

During design or navigation, the user can click on any graph node to enter directly into the page symbolized by the node without going through all the intermediate pages. This direct access method via a Web graph is much more efficient than linear access method in conventional browsers. The grammatical and structural organization of a Web site allows various (system or user-defined) integrity conditions for the site to be checked and any violation or inconsistency to be reported in a systematic fashion. For example, any Web pages that are orphaned by the deletion of some other pages should be detected.

We have designed an experimental tool, called *HWIT* (Human-Web Interface Tool), that realizes the above HWI functionality through visual Web programming and Web visualization. A designer or a user would be able to view any Web site from different angles, using HWIT's filtering capabilities, in a structured and personalized manner. HWIT also accepts filters that are defined and specified by users using the Customizer. The tool not only allows the user to easily navigate and explore the Web, but also assists the designer to design and maintain better-structured Web sites. Fig. 7.4 depicts a snapshot of the HWI when navigating Kang Zhang's research home page on the Internet Explorer (IE) by a simple click on the "research" node in the graph on the navigator. The Web page shown on the IE on the right-hand side represents the "research" page in the navigation window on the left-hand side. HWIT's "Preference" dialog allows the user to choose a preferred navigational browser from various options.

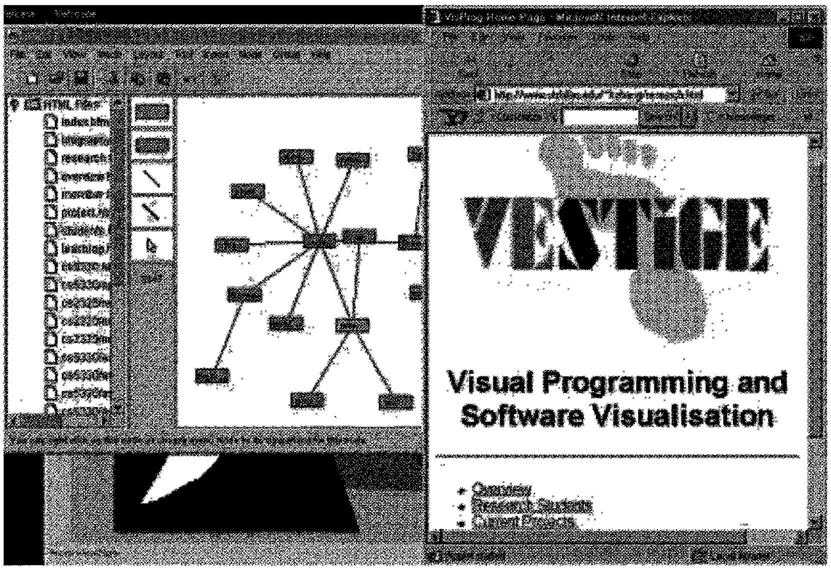


Fig. 7.4. Navigation on a Web graph in HWIT

7.4 Graphical Programming for Web Design

Visual structures and relationships are much easier to reason about than similar linguistically described structures. This is why designs in many application domains have been conducted on graphical representations. Using visual programming techniques to graphically design Web sites and Web pages will obviously enable more visual artists and other non-computing professionals to develop their own Web sites easily. The main philosophy behind the HWI framework is its consistent visual approach to Web design, navigation, and maintenance. This section introduces the concept of Web graphs and their notations, and describes the support for multi-versioning and reverse Web engineering through graph visualization.

7.4.1 Web Graphs and Design Notations

A graph G (N, E) consists of a finite set N whose members are called *nodes* and a finite set E whose members are called *edges*. An edge is an ordered pair of nodes in N. A node of a graph G1 can itself be another graph G2, which is called a *sub-graph* of G1. The properties of a graph may be inherited by its sub-graphs. We regard the organization of a Web site of any size as a graph, known as *Web graph*. A node in a Web graph represents a Web page, and an edge represents a link from one page to another. The World Wide Web is certainly the largest Web graph that is expanding all the time. For scalability and convenience of design and navigation, we define a special class of nodes, called *group*. A group represents a set of pages that are connected to a common parent page, and share the same set of attributes.

The *distance* between a pair of nodes, node A and node B, is defined as the number of intermediate nodes along the shortest path between A and B (including B). A sub-graph of graph G consisting of a node A and all such nodes in G that have a distance of N or shorter from A is called *A's level-N sub-graph of G*, or simply *A's level-N sub-graph*.

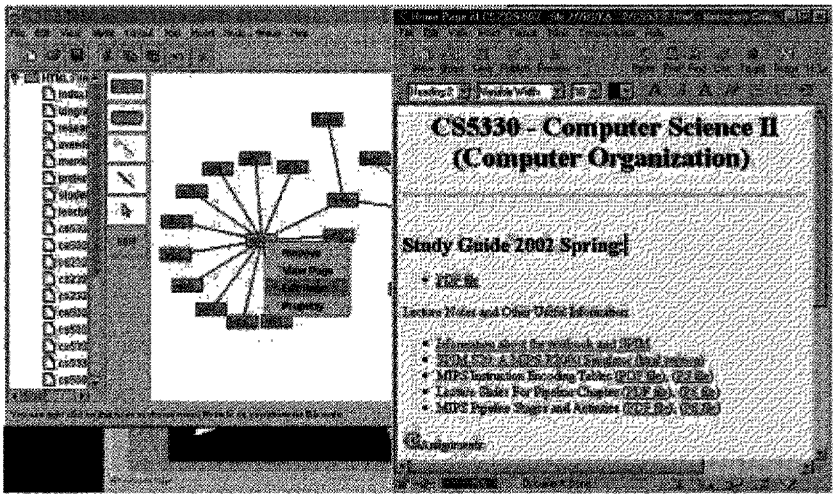
A graph class provides the general common properties that dictate whether certain operations are applicable to the corresponding graph objects. A Web graph can be constructed using a combination of tools: a graphical editor for constructing a Web graph at the high level and a Web page tool for constructing Web pages at the lower level. This is demonstrated in Fig. 7.5, that captures a snapshot during the design of the "CS5330" home page using the Netscape Composer (launched from the "CS5330" node within the HWIT Graph Editor). The graphical editor supports two-dimensional construction of Web graphs with direct manipulation.

HWIT uses a small number of simple notations, as shown on the left hand side of the screen in Fig. 7.4 and Fig. 7.5, to design and visualize the components of a Web graph.

The rectangle denotes a Node that represents a Web page. The label is used to identify the node and the page.

The round-cornered rectangle denotes a Group, representing a group of Web pages that are combined together either due to their commonality or for the brevity of viewing. It is like a Web template or class, which can be used to generate similarly structured-pages. A Group also has a label that identifies a specific class of pages. This notation also enforces the consistency in the pages belonging to one Group.

The thin arrow denotes an Edge that represents a Web hyperlink. This is the most common link seen in Web pages.



CS5330 - Computer Science II (Computer Organization)

Study Guide 2002 Spring

- PDF file

Lecture Notes and Other Useful Information

- [illegible]
- [illegible]
- MIPS Instruction Encoding Tables (PDF file), (PS file)
- Lecture Slides For Pipeline Chapter (PDF file), (PS file)
- MIPS Pipeline Stages and Activities (PDF file), (PS file)

Assignments

Fig. 7.5. HWIT Graph Editor and its connection to a Web page editor

The thick arrow is called a Gedge, short for Group Edge, which represents an edge coming out of or entering a Group. The difference between an Edge and a Gedge is that a Gedge connects to a Group and thus refers to all the Nodes belonging to the Group (some kind of inheritance). For example, if there is a Gedge connecting a Group A to a Node D and Node B is a member of A, then B is also connected to D. More importantly, the Nodes connected by Gedges to a Group share a common set of characteristics and attributes. This is useful in generating consistent look-and-feel pages.

The broken-line arrow (not shown in the figures) is called a Hedge, short for Hidden Edge. It may be defined as either a connection between pages of different domains, or as a connection between a collapsed node and its neighboring node after a filtering effect. When generated automatically by HWIT a Hedge indicates the existence of a connection (in form of hyperlink) between collapsed nodes. The designer may use Hedge at the design stage. In this case, a Hedge denotes either an Edge or a Gedge between two nodes and the designer has not yet made a decision in an early stage of design.

Each constructed Web graph is syntactically verified against the Web graph grammar that is defined according to the reserved graph grammar (RGG) defined for Web graphs.

7.4.2 Graph Operations

At the design level, the graph-oriented operations for editing, navigation, and maintenance are categorized into the following groups:

- *Graph editing*: this category is for the construction and editing of a Web graph, whose nodes are associated with Web pages or are lower level Web graphs, i.e. sub-graphs;
- *Sub-graph generation*: these operations derive sub-graphs according to some classification or filtering criteria. Sub-graphs are easier to navigate and maintain than a full graph. Categorizing sub-graphs removes unrelated information in the display and during processing. This also allows the user to record and replay a series of browsing steps, expressed in a sub-graph.
- *Query*: these operations provide information about the graph, such as the number of nodes in a graph, whether an edge exists between two nodes, and how a node is reached from another node. Such information is useful for the Web design and maintenance.
- *Time and version control*: graphs may change over a period and may reach a particular state at a predetermined time. For example, a Web designer may set a time when a page or a sub-graph should be activated or disabled. A graph from an early state may be partially reused and incrementally updated for a later graph. Graph states are represented through versioning.
- *Viewing*: these operations include applying various filters in selecting certain categories of sub-Webs, graph updating due to a change of any graph element, and collapsing of a sub-graph into a graph node for reducing clustering. To support scalable visualization, HWIT allows a number of nodes to be collapsed into a single node, which is called a *super-node*, graphically, it is represented as a Group. This operation achieves the zoom-in effect. A super-node can be expanded to allow its contents to be viewed or modified. The expansion achieves zoom-out. The zoom-in and zoom-out effects help designers to view their designs easily. Collapsing and expanding could be performed in various levels or depths.

7.4.3 Web Graph Grammar

The Web graph grammar used in HWIT is based on the *reserved graph grammars (RGG)* presented in Chapter 2. The main advantages of using the RGG formalism include its expressiveness and efficiency in parsing.

In HWIT, the Web graph grammar illustrated in Fig. 7.6 is used to parse and interpret a created graph and perform related graph operations. The correctness of a partially or fully constructed Web graph is checked according to the Web graph grammar. HWIT maintains referential integrity on the design to avoid broken links on the actual pages.

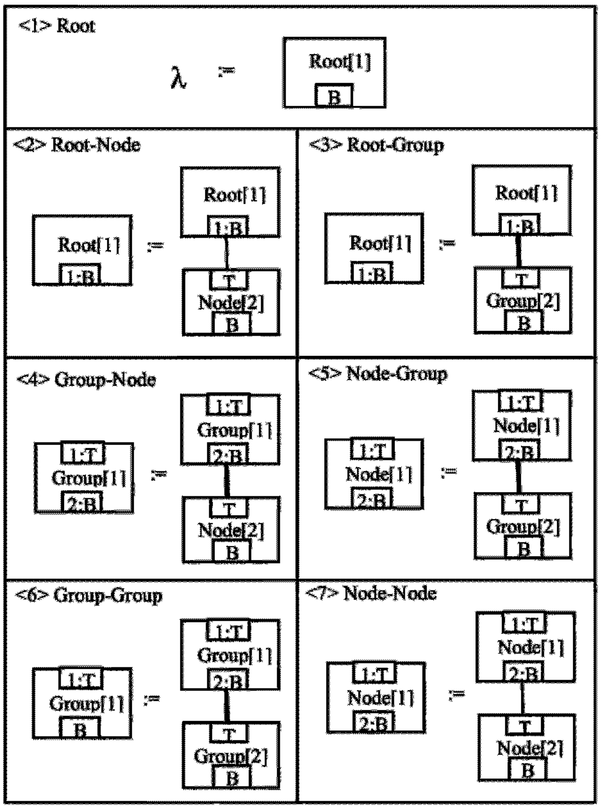


Fig. 7.6. Web graph grammar

As shown in Fig. 7.6, the Web graph grammar is simple and has seven production rules. Parsing a Web graph takes two phases: syntax parsing and semantics parsing. Syntax parsing is to check whether the graph is valid. If a graph is eventually transformed into an initial graph (i.e. λ) by the graph rewriting rules, it is valid. Semantics parsing is to produce a result or perform actions by executing a graph. The result is meaningful only when the graph is valid. The syntax and semantics are specified in the same set of graph rewriting rules. The graph transformation process checks the syntax of the graph and translates it into the textual specification in XML or HTML and internal formats at the same time.

Associated with each rewriting rule, an action code performs a syntax-directed computation over the attributes of a redex when the production is applied. For example, a simple action included in any production of the Web grammar can be written as follows:

```
action(AAMGraph g){
Attributes attributes = g.getAttributes();
(Property) attributes(2).get("URL");
}
```

The action takes a graph *g* as its input. *g* has a matching redex isomorphic to the right graph of the production. To facilitate the access of attributes in the redex, an array referring to required attributes is first produced through the method *g.getAttributes()*. The array member *attributes(2)*, for example, refers to the attributes of the node which has *A_ref = 2*. The URL of this node is obtained so that the related page can be displayed.

7.4.4 Information Filtering

The HWI framework supports several modes of displaying, including Level view, Domain view, Category view, Pattern view, and Constraint view. The Level view allows the user to choose the level of Web page pointers to display, i.e. a given level of linked pages in the Web graph relative to a given node. The Domain view shows the pages of a given application domain. If the Web designer has classified all the pages according to some application criteria, the user can choose Category view to see a given class of pages. The Pattern view allows the viewer to see some common patterns in a Web graph. Finally, the Constraint view shows all the pages that satisfy a given set of constraints.

The views described above are implemented by different filters that are described below. Web designers can design or customize their own filters to suit their specific application purposes.

- **Level filter.** If the user selects a particular node, say node A in a Web graph, and specifies a level N, the system will display A's level-N sub-graph. The Level filter makes use of hierarchical nature of the Web information space to set the levels. Only information of the given level will be shown on the view. "Collapse" and "expand" operations use the level filter to decide whether a particular page will be shown.
- **Domain filter.** The domain-related pages can be displayed if the user provides several keywords of an application domain.
- **Category filter.** The Category filter allows user to view the Web pages belonging to certain categories. The user could choose categories to be viewed from a list of categories available. This list of categories is gathered when the user adds new Web pages into the Web graph and categorizes them by filling the property form for each page.
- **Pattern filter.** Groups of Web pages with certain patterns can be filtered and shown in the view. A typical use of this filter is when the user attempts to remove redundant pages by comparing similar sub-graph patterns. The filter accepts a sub-graph (perhaps generated using another filter) from the user and finds the matching patterns to display. The matching process can be slow for a large Web graph.
- **Constraint filter.** Web pages that satisfy certain constraints can be displayed in the view. Examples of constraints are file size, file attributes, file's creation dates, etc. For instance, the user may wish to view all the pages under a certain file size.

Filtering rules can be defined on various structures, including graph structure, Web context, and document structure. Other conditions may be defined to facilitate more specialized filters. Web designers can design or customize their own filters to suit their specific application purposes.

7.4.5 Support for Multi-version Web Sites

Web designers are under increasing pressure to produce updated Web sites. The conventional approach to creating and modifying a Web site is to create every single page and make changes on the copy of the source code of the page. Problems arise when the main frames of the pages are almost the same while changes are needed only for part of the original documents. To allow efficient creation and modification of changing Web sites, the idea of multi-version Web sites has been proposed (Wadge and

Yildirim 1997). In a multi-version Web site, a generic source page acts as a template for other documents in the same site. The generic source page represents the common part of other documents and is used as the index page of the site. Each of the other documents can be considered a version of the site. Every request from a client is associated with a version label that is interpreted by a CGI or Java Servlet program on the Web server to point to an appropriate version of the document. This version of document is then retrieved and loaded into the template of the generic source page to be displayed. Multi-versioning is also useful when different languages or different representations are needed on a single Web site.

Fig. 7.7 depicts a possible organization of a multi-version stock market site, in which each stock market source page (a version such as that of "New York") share a generic information page. In this case, different graph objects carry different meanings under different contexts, the semantics of a graph operation will depend on the context defined by the Web designer. Graph operations can be implemented according to their contexts but all provide the same interface to the designer.

HWIT supports the concept of associative queries for multi-version Web sites. The basic idea is that all the objects in a generic source page are categorized into three hierarchical classes: root, node and leaf, and they form a hierarchical graph. HWIT uses a data structure called *virtual version tables* (VVTs) to organize different versions of a document and facilitate the retrieval of appropriate documents.

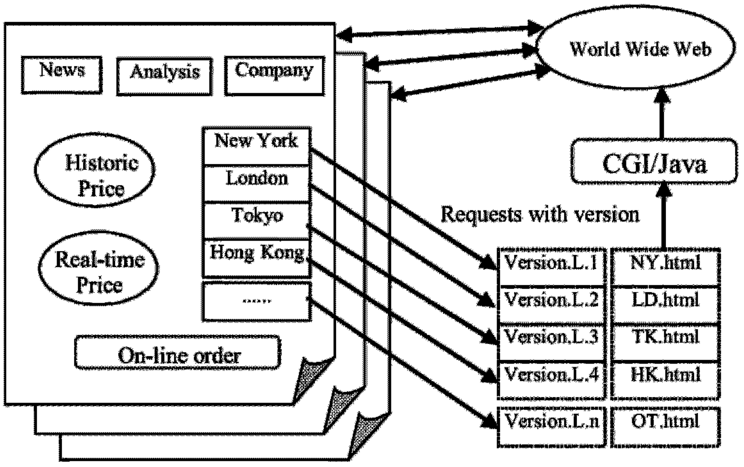


Fig. 7.7. A multi-version Web site

The title page is considered the root class. Node classes include HREF links, includes, headings, and other node classes. Leaf classes are disjoint objects such as graphic files and audio files. A version label is assigned as an attribute to the root class and objects in each leaf class when submitting a request for a specific version. A hierarchically structured graph is created when the version document is generated. Information retrieval is achieved by querying the graph of the version through VVTs. More details of the associative query approach can be found in Zhang and Zhang (2000).

7.5 Web Reuse Through Reverse Engineering

One of the major advantages of integrating Web design and navigation features in the HWIT framework is the capability of reusing existing Web graphs of pre-developed Web sites for the generation of new Web sites. Web graphs generated in HWIT provide not only the visualization for viewing and navigation of Web sites, but also the graphical user interface for direct manipulation of Web data, attributes and relational structures. We call the process, the generation of new Web sites through the direct visualization and editing of existing Web graphs, *reverse Web engineering*.

7.5.1 Reverse Engineering Approach

In HWIT, all Web graphs are commonly presented as node-like diagrams. To reduce the life cycle of Web engineering, HWIT provides facilities for reverse engineering of Web sites. We can use the HWIT editing environment to quickly migrate the Web graph of an old Web site into the Web graph of a new Web site. We can then visualize the new Web graph on a graphical user interface to create (edit) new Web data (sites).

A Web designer could use HWIT to display an existing Web graph as the primitive Web graph of a new Web site, and modify the attributes and structures of this Web graph through direct visual editing, and then create the new site as well as the graph-migration by writing back (output) the modified Web graph into a new Web site. We adapt the techniques of Huang et al. (1998b) for reverse Web engineering. Fig. 7.8 is a transition diagram showing three phases of the reverse Web engineering process. The process allows any levels of details in a Web site infrastructure to be preserved and effectively reused.

We use an incremental visualization technique to navigate the entire Web site as we assume that the corresponding Web graph of the Web site is very large that is unable to be displayed entirely in any type of the available screen with a limited number of pixies. The navigation of the Web graph uses a sequence of *logical display frames*. These frames maintain the user's orientation for Web exploration. This method also reduces the cognitive effort required to recognize the change of views. This is done by connecting successive displays in the *logical display frame* of the Web graph and by smoothly swapping the displays via animation as shown in Fig. 7.9.

The underlying structure of the hypermedia system on the Web can be regarded as a huge digraph with nodes and edges representing URLs (or anchors) and links respectively. For the purpose of visualization and backtracking during navigation, it is more convenient to make no distinction between the directions of hyperlinks. We will treat a Web graph as undirected. Furthermore, we assume that the Web graph **G** we dealing with is a huge, connected, undirected graph.

To display a given Web site as a graph, we need an effective and efficient graph layout algorithm. Among many graph drawing algorithms, force-directed algorithms (Di Battista et al. 1999; Eades 1984; Huang et al. 1998b) are very popular; they are easy to understand, and the results of layouts can be good. HWIT uses a force-directed algorithm to draw existing Web graphs for visualization. A *force-directed* algorithm views a graph as a system of bodies with forces acting between the bodies. The bodies are represented nodes in the graph, and the forces are relationships between the nodes in a graph and determine the geometrical positions of the nodes. A force-directed algorithm aims to compute a position for each body such that the sum of the forces applied on each body is locally minimized.

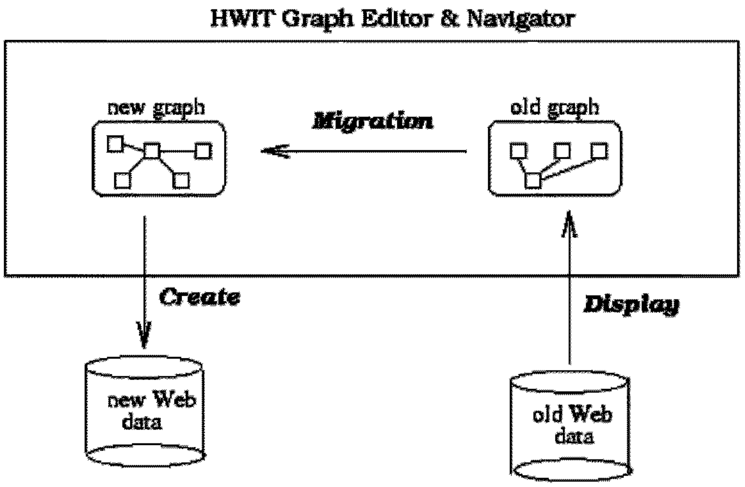


Fig. 7.8. A transition diagram showing three phases of Web reverse engineering

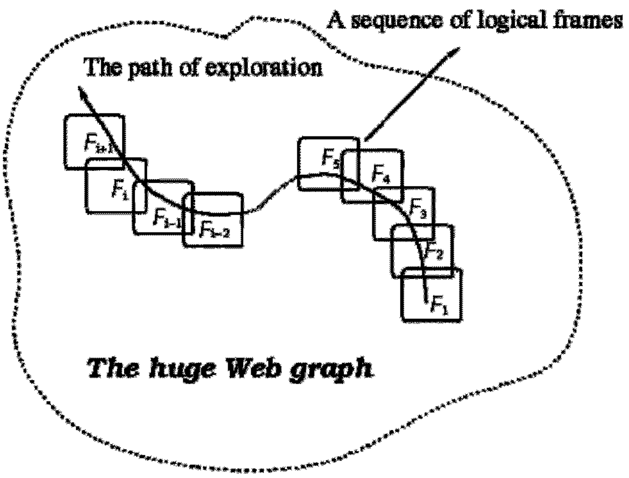


Fig. 7.9. An exploratory visualization model

One of the most popular force-directed algorithms is called the *spring algorithm* (Eades 1984). The original spring model uses a combination of *spring* and *gravitational* forces. The spring force is based on Hooke's law springs, and the strength of the springs varies. The gravitational force follows the Newton's universal law of gravitation, except that attraction is replaced by repulsion. Edges are modeled as springs, and nodes are particles that repel each other. This is illustrated in Fig. 7.10.

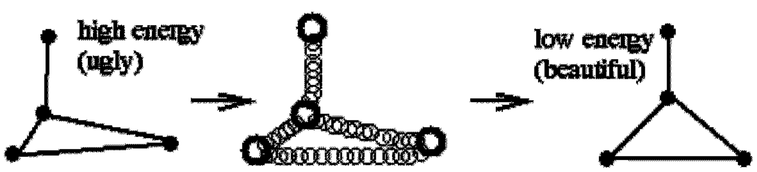


Fig. 7.10. Spring model represents nodes as steel rings and edges as steel springs, and then finds a drawing with a minimal energy

7.5.2 Web Visualization Algorithm

A Web graph being visualized is treated as a rooted tree, which consists of a set of *focus nodes*, each surrounded by the nodes linked to it. A focus node is usually the center (or previous center) of the user's attention when navigating and viewing the Web graph.

Suppose **G=(V, E)** is a connected Web graph, **v** is a node. A node **u** is called a *neighbor* of **v** if there is an edge between **u** and **v**. The *neighborhood tree* of **v** is defined as the subgraph of **T(v)=(N(v), L(v))** of **G**, where **N(v)** consists of the neighbors of **v** and **v** itself, **L(v)** consists of the edges between **v** and its neighbors.

Given a chain $\mathbf{Q=\{v_1,v_2,...,v_s\}}$ of nodes, a *logical display frame (LDF)*, **F = (T,Q)** consists of a spanning tree **T** of the union of the neighborhood trees of these nodes and **Q**; the nodes $v_1,v_2,...,v_s$ are called the *focus nodes* of **F**.

We proceed by visualizing a sequence $\mathbf{F_1=(T_1,\ Q_1),\ F_2=(T_2,Q_2)}$ ... of logical display frames. To limit the size of these frames, we assume a upper bound **B** on the number of focus nodes in them, that is, $|\mathbf{Q_i}|<=\mathbf{B}$ (i=1,2,...), where $|\mathbf{Q_i}|$ denotes the number of nodes in $\mathbf{Q_i}$.

To obtain $\mathbf{F_{i+1}}$ from $\mathbf{F_i}$, the user selects a non-focus node **u** in $\mathbf{F_i}$ with a mouse click, if **u** has more than one neighbors, then it becomes a new focus node of $\mathbf{F_{i+1}}$; Now if $|\mathbf{Q_i}|=\mathbf{B}$, we will delete an old focus node **v** from $\mathbf{Q_i}$, that is, the chain of focus nodes in $\mathbf{F_{i+1}}$ is $\mathbf{Q_{i+1}=Q_i}+\{u\}-\{v\}$. We choose **v** to be one of the focus nodes in $\mathbf{Q_i}$ whose graph-theoretical distance from **u** is the largest in $\mathbf{F_i}$. Using these focus nodes, we can easily calculate $\mathbf{F_{i+1}}$.

A *drawing* **D** of a Web graph **G = (V, E)** consists of a location for each node $v \in V$ and a route for each edge $e \in E$. A visualization of the sequence $\mathbf{F_1 = (T_1,\ Q_1),\ F_2 = (T_2,\ Q_2)}$, ... of display frames consists of a drawing $\mathbf{D_i}$ of each graph $\mathbf{T_i}$. Drawing sequences occur in many interactive systems which handle relational information. Most such systems suffer from the "mental map" problem: a small logical change in the graph results in a large change in relative positions of nodes in the drawing. The *mental map* problem is addressed by using animation or "in-betweening" along with a force-directed layout algorithm (as discussed below) to preserve the *mental map* between drawings.

Each drawing $\mathbf{D}_i$ is a "spring drawing", that is, it is calculated using the force-directed algorithm. The "in-betweening" technique aims to achieve the twin goals of good layout and the preservation of the mental map. The in-betweening consists of a sequence $\mathbf{D}^0_i$, $\mathbf{D}^1_i$, $\mathbf{D}^2_i$, ... $\mathbf{D}_i$ of drawings of $\mathbf{T}_i$ called *screens*. They are computed by the *modified spring algorithm* (Huang et al. 1998b), based on the original spring algorithm (Eades 1984).

The modified spring algorithm seeks an equilibrium configuration of the forces for the current display frame $\mathbf{F}_i$. That is, a drawing in which the total force $f(v)$ on each node $\mathbf{v}$ is zero. When a logical transformation occurs from view $\mathbf{F}_i$ to another view $\mathbf{F}_{i+1}$, the current equilibrium configuration of the forces is broken by deleting and adding nodes. Thus, the modified spring algorithm moves nodes toward the next equilibrium configuration of forces for the new display frame $\mathbf{F}_{i+1}$. From one screen $\mathbf{D}^j_i$ to the next screen $\mathbf{D}^{j+1}_i$ (each animation loop), the animation model computes the total force $f(v)$ for every node $\mathbf{v}$ in $\mathbf{F}_i$ (except the history nodes) and moves each node $\mathbf{v}$ a small amount proportional to the magnitude of $f(v)$ in the direction of $f(v)$. Each $\mathbf{D}^{j+1}_i$ has energy a little lower than that of $\mathbf{D}^j_i$. The movement stops when the user makes another logical transformation, or when the system reaches equilibrium.

In order to address the specific criteria of incremental drawing and clearly distinguish the focus nodes and their neighborhoods, we have extended the spring model by adding some extra forces among the neighboring nodes surrounding the focus nodes. Suppose that $\mathbf{F}_i$=($\mathbf{G}_i$, $\mathbf{Q}_i$) is the display frame being visualized, where $\mathbf{G}_i$=($\mathbf{V}_i$, $\mathbf{E}_i$) is the Web graph consisting of a vertex set $\mathbf{V}_i$ and edge set $\mathbf{E}_i$, and $\mathbf{Q}_i$ is the set of focus nodes. More precisely, the force applied on node $\mathbf{v}$ is:

$$f(v) = \sum_{u \in N(v)} f_{uv} + \sum_{u \in Vi} g_{uv} + \sum_{u \in Qi} h_{uv}$$

where $\mathbf{f}_{uv}$ is the force exerted on v by the spring between $\mathbf{u}$ and $\mathbf{v}$, and $\mathbf{g}_{uv}$ and $\mathbf{h}_{uv}$ are the gravitational repulsions exerted on v by one of the other nodes u in the graph. This extended spring model aims at satisfying the following three important aesthetics:

The spring force between adjacent nodes ensures that the distance between adjacent nodes **u** and **v** is approximately equal to zero length.

- The gravitational force ensures that nodes are not too close to each other.
- The extra gravitational force aims to minimize the overlaps among the neighborhoods in the display frame and to keep the focus nodes along a straight line.

7.5.3 An Example

To illustrate how the reverse Web engineering works, a simple example is presented. Fig. 7.11 and Fig. 7.12 demonstrate two different phases, *display* and *creation*, of the reverse engineering process. Fig. 7.11 shows that a Web site is being selected and the corresponding Web graph is read into the HWIT Navigator for visualization. This is achieved by selecting the root page from the pop-up file management window. The layout of the Web graph is animated when the extended spring model is applied. The user may manually drag nodes and edges in the graph during animation to adjust the graph into a user-desired layout. The user can also use this Web graph to navigate the Web site for finding information he/she needs.

After a desired layout is reached, the user can switch from the navigation mode to the editing mode in HWIT, and modify the attributes and structure associated with the original graph as shown in Fig. 7.12, and then create the new Web site by simply save this modified graph into a newly named Web site. Fig. 7.12 is a snapshot showing that a sub-graph has been added into the bottom of the original Web graph and the properties of node CS6366 has been changed in the editing mode of HWIT. The user can then create a new Web site (the third phase of the reverse engineering process in Fig. 7.8) by simply clicking on the "convert back" button to write back the modified properties of Web graph into a newly named Web site.

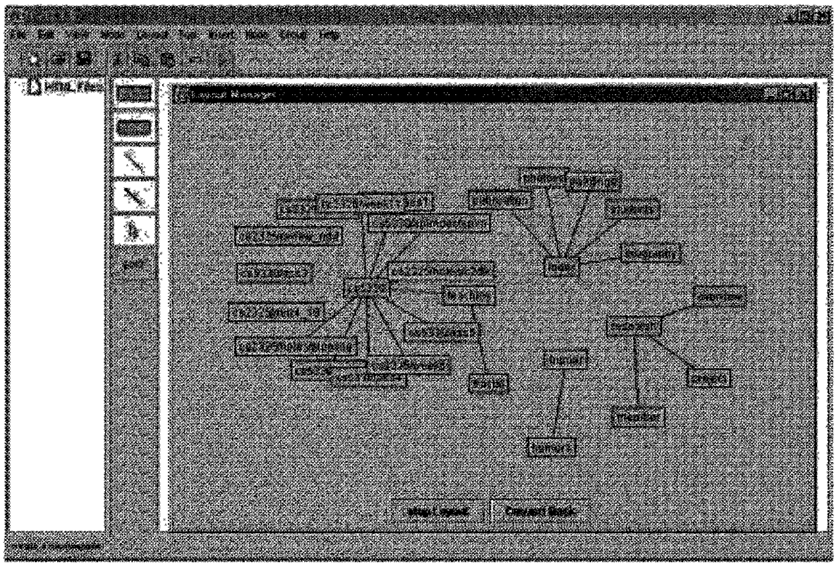


Fig. 7.11. The first phase of reverse engineering that displays the Web graph of an existing site in the HWIT Navigator

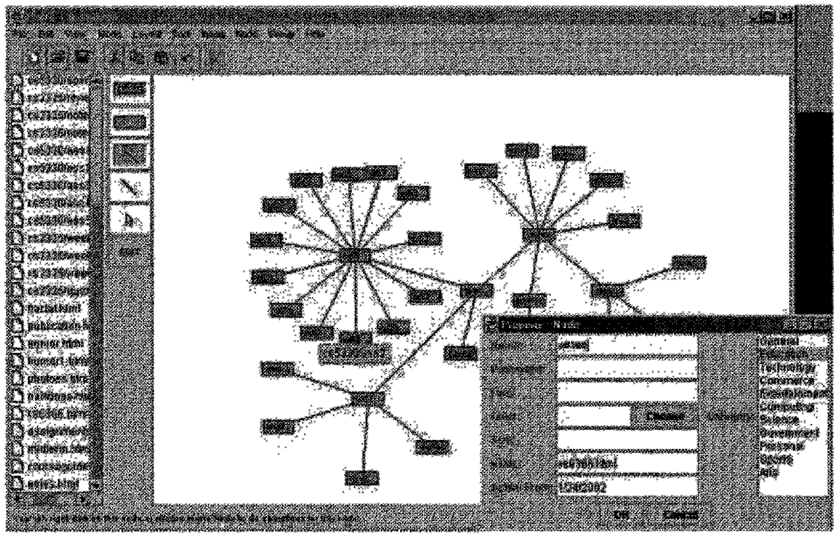


Fig. 7.12. Modifying the structure and properties of the existing Web graph and save the changes to create a new Web site

7.6 Summary

This chapter has presented a visual framework to Web site design, navigation, and maintenance. It advocates the integration of the tools for all activities, ranging from Web page and Web site design, navigation and browsing, to Web system maintenance, while preserving the same mental map for both the Web designer and the Web user throughout these activities. The presented approach is a step closer towards narrowing the gap between Web designers and users (Nakayama et al. 2000).

A prototype of HWIT has been implemented in Java, which is capable of generating Web sites from Web graphs drawn on the HWIT Graph Editor. The Web re-engineering and reverse Web engineering capability in HWIT allows previously developed Web sites to be visualized and re-developed graphically. The modified spring algorithm can be adapted to support more personalized and pleasant viewing during navigation and maintenance.

This work has opened up many more opportunities for further investigation. Possible future work includes the following.

- Security features can be built into the framework so that different groups of people may access different parts of a Web site.
- Empirical studies need also be conducted in order to evaluate the usability of HWIT in real world applications.

7.7 Related Work

To aid Web navigation and maintenance with a sense of orientation, researchers have proposed "site mapping" methods for constructing a structured geometrical map for one Web site (Maarek and Shaul 1997) or for comparing the structures and contents of different Web sites (Liu et al. 2002). They can guide the user through a limited region of the Web. Other approaches define the entire WWW as a graph and then navigate the graph (Anupam et al. 2000; Huang et al. 1998a). Yet, these systems do not support the integration of Web site design, navigation, and maintenance.

Various tools and methodologies have been developed or proposed in the past few years. Most of the tools assist, in one way or another, different areas of WWW development, mainly aiming at improving navigation. For example, Fisheye-View Graphical Browser (Muchaluat et al. 1998) adopts fisheye view filtering strategies (Sarkar and Brown 1994) to allow logical management of documents with nested compositions. This browser degrades dramatically as the number of nodes increases.

WebMap (Doemel 1994) shows a 2D graphical relationship between pages. Small circles depict pages whereas links are coloured to indicate the status of destination documents. Users can visualize the document space without having to visit all documents since WebMap implements an exploratory approach to gather the documents as a batch job. However, the whole process is time-consuming and resource intensive.

PadPrints (Bederson et al. 1998) is a zooming Web browser within a multi-scale graphical environment. It displays multiple pages at a time and a large zoomable information surface depicts the links between the pages. The current page is clearly shown as it is larger than other pages. The system only enhances information browsing among different documents.

WebML (Ceri et al. 2000) uses a model-based approach to Web site development. In WebML, a structural model expresses the site's data content using commonly accepted modeling languages such as UML; hypertext model describes the contents and structure of the site's pages; presentation model dictates how the pages are presented with a layout specification; finally, personalization model allows group-based or individual-based content categorization.

Other tools use software engineering methodology to approach Web development. For example, WOOM (Coda et al. 1998; Klapsing et al. 2001) and XWMF (Gómez et al. 2001) use object-based formal metadata model for designing Web structures expressed as directed acyclic graphs (DAGs). The emphasis is on the high level design for interoperable exchange and reasoning about the Web data. The OO-H (Object-Oriented Hypermedia) method uses UML-like conceptual modeling to specify navigation and presentation features. They do not address the important issue of integrated view or offer the capability of reverse engineering.

Chapter 8 Visual Language Generation

8.1 Introduction

Implementing a visual programming language (VPL) is much harder than implementing a textual programming language (Myers 1990). VPLs are usually embedded and tightly integrated within visual environments. Consequently, they are often characterized by the attributes of the environments (Goldberg et al. 1994). The VPL implementation involves the implementation of a whole programming environment with a user interface which supports developing programs using a visual language. Notice that VPL interfaces are not the same as graphical user interfaces (GUIs) nor are they just for visualization. Traditional GUI development toolkits are inadequate for the creation of VPLs because they do not support syntactic and semantic specifications of visual programming. The graphical user interface of a visual language relates to the language's syntax and semantics. The interaction (dialogue) between the interface, the syntax and the semantics must be maintained. Implementing a VPL interface and its support for syntactic and semantic specifications of visual programming suffers from a problem common to all large, complex software systems, i.e. the generation is difficult and time consuming. The remaining part of this section will address the typical problems of implementing VPLs.

8.1.2 Why Automatic Generation?

Repetitive efforts have been spent on developing various domain-oriented VPLs, due to their specialised requirements and inseparatable development processes. In a visual programming environment, users must be able to interactively construct and manipulate expressions in the visual language. The graphical requirements of a visual language include defining the visual elements of the language and the graphical relationships that must be maintained when these elements are connected together. The editing operations themselves are event-driven, and appropriate interpretations of mouse and keyboard events must be provided. Algorithms must be provided for graphically editing these elements. The solutions to these graphical requirements are intricate and inherently difficult to implement. The underlying data structures are complex, containing information about the visual representation, logical connectivity, domain knowledge, etc. They make it difficult to parse an edited diagram with a general parsing algorithm. Existing solutions to solving the data structure problem tend to be so specialized that they apply only to a single visual language.

As a textual programming language construction tool, Lex/Yacc divides the process of language creation into two steps: lexicon definition and grammar specification. The created lexical and grammar analyzers are combined together to serve as a language parser. In particular, its rules (i.e. grammar) can be associated with actions written in C, so that a wide range of textual languages can be specified. The fundamental reason that no VPL generation tools can be as effective as Lex/Yacc is that no design model has been able to completely separate the processes of visual elements creation, visual editing, and syntax and semantics specifications. Therefore, it is difficult to integrate independently developed functional components into a single VPE. Existing tools that aim at supporting different aspects of VPL generation, e.g., for user interface generation, and for parsing, are not able to cooperate to generate VPLs. The generation of every new visual language requires a re-development of the whole machinery.

Another problem with many VPL generation tools is that their underlying graph formalisms are not expressive enough to describe many types of visual languages or not efficient enough to parse various types of visual programs. The multi-dimensionality of visual languages makes it difficult to build formal grammars and compilers for them. While text strings only allow concatenation before or after a character, visual languages allow multiple concatenation options between its visual elements. Attempts at developing visual grammars using textual grammars as models have had limited success; many visual language formalisms cannot be specified and parsed effectively and efficiently with existing grammars. We have addressed this problem by developing the Reserved Graph Grammar formalism, and its spatial counterpart, the Spatial Graph Grammar formalism, as discussed in Chapters 2 and 3.

Tools and formalisms have been created for automatically generating visual languages. Most of them are specialized in certain aspects of visual language generation, e.g., user interface or grammar formalism. For example, Pietriga (2005) presented a toolkit called ZVTM based on the layers of virtual spaces, zoomable cameras, and glyphs. ZVTM aims at promoting and development of user-interface aspects of visual programming environments by easing the creation of structured graphical editors. Others provide support for producing a complete visual language environment with limited capabilities. For example, VPE generation tools based on grammar formalisms usually generate visual editors automatically through their grammars. This is the easiest way to produce a visual editor, but generated visual editors are often not user-friendly and the functionalities are limited (Costagliola et al. 1995). Yet, the formalisms which can generate powerful visual editors do not provide an adequate mechanism to support syntactic and semantic specifications (Brown 1997) or are inefficient in parsing (Rekers and Schürr 1996).

8.1.3 A Generic Multi-Level Approach

To avoid the re-development, we need to find a proper representation of the data structure and a generic model, which are able to decouple the components of a visual programming environment. The approach to be presented is to view a target or domain-oriented VPE as a specific instance of a generic VPE such that the techniques applicable to the generic VPE can also apply to the target VPE and functionalities common to the VPEs need not be re-developed. This approach enables the design of a generic visual framework that can be customized into any target VPEs. Such a customization process is realised by a set of visual specification tools in a similar fashion as by Lex/Yacc in generating textual languages. The main characteristics of this work include

- a high level design model that supports a generic but customizable framework with decoupled functional modules;
- a set of customization and specification tools which are visual tools supporting direct manipulation; and
- an underlying graphical formalism that can express and parse a wide range of visual languages effectively and efficiently.

This chapter presents the design of a generic visual programming environment which has a multi-level tool structure. It addresses the issues in developing a design model that supports the development of a VPE by dividing the whole development process into several independent stages. The model offers several decoupled functional modules, each supporting an independent development stage. This makes it possible to develop an effective generic system for the generation and reuse of a wide range of

VPLs. The chapter describes a toolset called VisPro which provides a similar mechanism as lex/yacc in the process of constructing VPLs. It is very easy to use, since the tools in VisPro are meta visual languages. To formally represent VPLs, VisPro uses reserved graph grammars presented in Chapter 2 to express a wide range of diagrammatic VPLs. A graph grammar in VisPro is a set of graph rewriting rules associated with actions written in Java. The target language compilers for a large class of diagrams can be automatically generated in polynomial time by VisPro according to the grammar specifications. Moreover, a set of language components (i.e. visual objects) can be created through direct manipulation and a visual editor can be produced according to control specifications. A visual programming environment integrating the visual editor and the compiler is then created. Therefore, VisPro provides a high level support for VL developers to rapidly construct a wide variety of domain-oriented diagrammatic VPLs. It can easily create both the user interface and the underlying language.

The chapter focuses on the design, construction, and application of VisPro. It is organized as the following. Section 8.2 summarizes the design criteria for a generic VPE, followed by a detailed discussion of the VisPro design model that meets the criteria in Section 8.3. Section 8.4 describes the design of the VisPro toolset which consists of a set of decoupled functional modules. Section 8.5 presents an application of the VisPro system in generating a visual distributed modeling language. The chapter is summarized by Section 8.6.

8.2 Design Criteria for VPEs

A generic VPE can be viewed as a collection of visual and textual specification tools, which are themselves visual languages and/or textual languages. A program for generating a domain-specific VPE is a combination of specifications written in a set of hierarchically organized languages. Such a complex environment needs a careful design. We regard the following three aspects as the key to the successful design of a generic VPE.

- **Heterogeneous programming**: the VPE should support heterogeneous visual programming, where various visual languages and textual languages at different levels of abstraction can work together to specify real world applications.
- **Hierarchical structure**: with the support of various languages and programming paradigms, the VPE should have a well-designed

mechanism that organizes and coordinates the languages in an effective and efficient manner.

- **Design model**: to increase the reusability of existing visual languages and various language components and simplify the generation of domain-specific VPEs, the generic VPE should be designed as several decoupled modules which can be developed independently with possibly different formalisms.

The following sections discuss these criteria in more details and how a generic VPE is designed against the criteria.

8.2.1 Heterogeneous Visual Programming

The argument for supporting heterogeneous visual programming is based on the following considerations:

- **High Level Programming**: Visual languages do not usually support the entire programming process (Burnett 1994). A typical class of visual languages is designed to be used for visual manipulation at a high level and to combine low level application components which could be written in text languages. Examples of this approach include the object-oriented visual programming system HI-VISUAL (Hirakawa et al. 1991) and the Web service composition language ZenFlow (Martinez et al. 2005).
- **Low Level Programming**: Another class of VPLs allows programming only at the lowest level. This type of languages have all the capabilities needed to express the fine-grain logic in a program, such as conditions and repetitions. But they do not have the facilities to organize portions of the program into modules. Most VPLs of this nature are intended for specific problem solving. They provide a number of primitives for their particular domains, thereby keeping most programs small enough to avoid the need for user-defined abstractions. One example is the NoPump system for interactive graphics (Wilde and Lewis 1990). For such low level VPLs, a high level organization mechanism could enhance their usability in large scale applications.
- **Independent Development**: If visual languages are independently developed to suit different application purposes, they are usually unrelated to each other. It is difficult, or impossible, to make these visual languages work together to solve a complicated problem. The most plausible way is through a high level protocol, such as a formatted information transfer system under the OS level. However,

such a protocol is usually inefficient and error-prone. A framework for creating hybrid visual programming environments is, therefore, desirable.

8.2.2 Hierarchical Structure

Schefstrom and van den Broek (1993) proposed a model that organizes tools used in the software engineering life cycle (Salis et al. 1995). A sophisticated application may be specified or modeled by more than one software tool in a coordinated fashion. These tools may work at different levels of the software development process, but may interact at the same level. The relationships among the tools in a programming environment can be seen as a multi-level tool structure which supports the following concepts, as illustrated in Fig. 8.1.

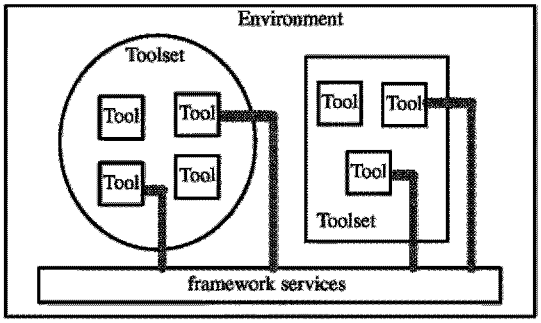


Fig. 8.1. Multi-level tool structure

A *service* is the smallest functional unit of interest to a developer. A *tool* is a strongly related and clearly delimited set of services that support a particular job, such as a diagram editor. Similarly, a *toolset* has a set of tools that show strong internal cohesion and low external coupling. The set of tools work together to cover part of the development process such as a compiler, its associated syntax directed editor, and debugger. An *environment* is a group of toolsets. A *framework* is a set of software modules that are related to several tools and are typically well-documented and supported.

As the scope of the support entity increases from a single tool to a large environment, the cohesion among its components will inevitably decrease. At the same time, the coupling of the components may also decrease, or at least not increase. The primary reason for this is that, as the support scope widens, the range of support activities diversifies. Software development planning, for example, uses toolsets different from those for software construction (programming, integration, and testing).

A sophisticated programming environment, such as a software engineering environment, may have a set of visual or textual languages. With the multi-level tool structure, the languages may be seen as tools in the environment.

8.3 Design Model

To design a visual programming environment, one needs to consider the language's syntax and semantics, and the visual interface. For supporting the generation of a wide range of VPLs, we aim to maximize the reuseability of the language components with the following considerations:

- different modules of a VPE should be designed and implemented separately, and
- improvement of one module should have little impact on other modules.

To ease maintenance, modification, and reuse of a VPE, interactions between different modules of the VPE should be clearly specified. This also simplifies a VPE's generation by dividing the VPE into several decoupled modules, and allows different formalisms to be developed into individual modules.

8.3.1 The MVC Framework

A popular model for the user interface construction is the Model-View-Controller (MVC) (Krasner and Pope 1988) framework that has been successfully used in Smalltalk-80. MVC consists of three types of objects: *Model, View* and *Controller*. *Model* represents the logical structure of an application, whose screen presentation is *View*. *View* requests data from *Model* and handles all the graphical tasks. *Controller* defines the way in which a user-interface reacts to user inputs. The standard interaction cycle is that the user provides some input action and *Controller* responds by invoking an appropriate operation in *Model*. *Model* then carries out the prescribed operation, possibly changes its state, and broadcasts the changes to all its dependent *Views*. Each *View* can query *Model* for its state and update its display, if necessary.

There are some implementations of MVC which effectively decouple the relations between different objects and enhance the reusability (McWhirter and Nutt 1994; Minas and Viehstaedt 1995). They are, however, mainly for constructing user-interfaces with windows applications, rather than for constructing visual languages. For constructing VPLs, more detailed specifications and tools are needed for declaring and specifying the required interations between the system modules.

Based on MVC, the DV-Centro framework (Brown 1997) aims at supporting visual language development, as shown in Fig. 8.2. It uses the Supervisor-Agent pattern to specify the interactions between the modules in the framework.

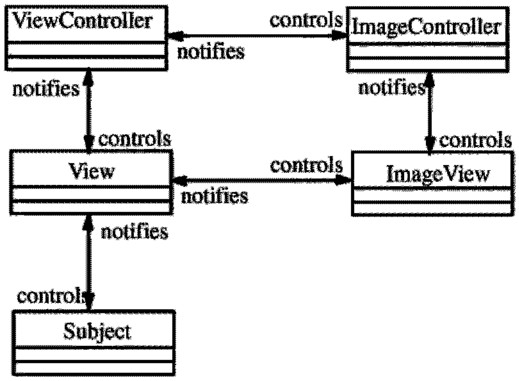


Fig. 8.2. The DV-Centro framework

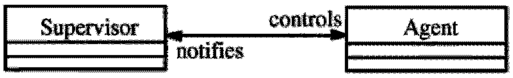


Fig. 8.3. Supervisor-Agent pattern

The Supervisor-Agent pattern (Fig. 8.3) assumes that *Supervisor* must be able to control *Agent*'s behavior, while *Agent* is independent of *Supervisor*, except that it may notify *Supervisor* in a predefined protocol. Since a Supervisor-Agent pattern indicates a one direction dependency, i.e. the design of *Supervisor* depends on that of *Agent*, the DV-Centro framework reduces the number of dependent relationships in a general MVC model.

For example, *ImageView* in Fig. 8.2 is independent from other modules, so that it can be developed as a standalone tool.

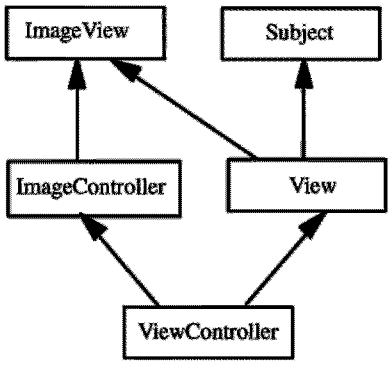


Fig. 8.4. Dependency relationships

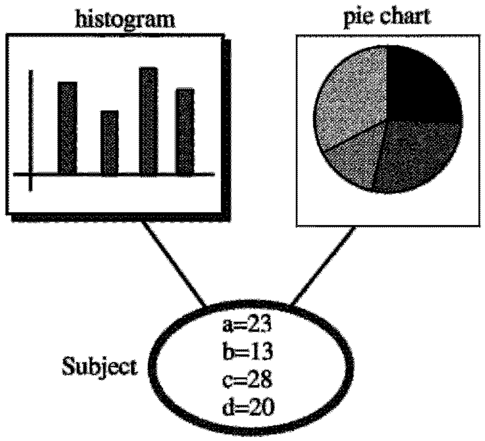


Fig. 8.5. A *Subject* with two versions of *View*

There are, however, other dependency relationships (as shown in Fig. 8.4) which have impact on the design and implementation of various modules. For example, *View* depends on *Subject*, which means that it must be designed after the design of *Subject*. Any change of *Subject* may affect *View*. On the other hand, as *Subject* is more application-specific than other modules, the relationships between *View* and *Subject* should be reversed. For example, Fig. 8.5 shows a model having a *Subject* with two versions of *View*. The model contains some data values, and the views define a histogram and a pie chart. It communicates with its views when its values change, and the views communicate with the model to access these values. With the DV-Centro framework, the histogram and the pie chart have to be designed according to the specifications of the model. But we believe that the design of the histogram and pie chart should be independent of the model so that they are general enough to be predefined in a toolset. The best solution is that *Subject* and *View* are designed to be independent of each other so that a subject can use any suitable views without changing itself and the views.

8.3.2 An Ideal Design Model

The dependence of *ViewController* on *ImageController* and *View* implies that the high level control depends on its low level implementation (e.g. *ImageController*). It is, however, desirable that any improvement on the low level facilities will have no impact on the high level specification. So a model that removes the dependency relationship between *ViewController* and *ImageController* is more flexible.

Considering the model in Fig. 8.4 where the link between *ViewController* and *ImageController* has been removed, we find that *View* becomes a key module because it relates to almost all the other modules. To allow *ViewController, Subject,* and *ImageController/ImageView* to be designed and implemented independently, we propose to reverse the relationship between *View* and *ImageView* so that these modules depend only on *View*. The resulting improved framework is shown in Fig. 8.6, where *View* serves as an interactive protocol between different modules.

To represent this framework with the MVC notations, we redefine it as an ideal design model as shown in Fig. 8.7, where *View* corresponds to *ImageController* and *ImageView* in Fig. 8.6, *Model* corresponds to *Subject, Controller* corresponds to *ViewController*, and *Protocol* replaces the previous *View*. This new design model confines the dependencies between the functional modules such that each module can be developed independently.

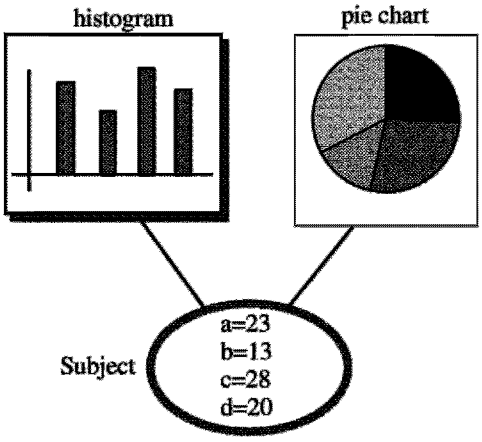


Fig. 8.6. An improved framework with Supervisor-Agent pattern

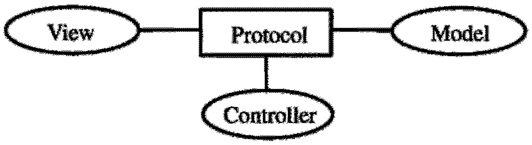


Fig. 8.7. An ideal design model

8.3.3 The VisPro Design Model

The VisPro design model needs a protocol to define the interaction between its functional modules. The protocol is designed as a combination of an ***abstract diagram*** and a ***concept space***, as shown in Fig. 8.8. An abstract diagram represents a common internal data structure that may be used to display diagrams in various formats, such as Nassi-Shneiderman diagrams and flowcharts. In an abstract diagram, which can be considered a kind of entity-relationship diagram, directions, distances, data and control flows, joins, contacts, etc, can all be represented as relations between entities. A concept space is a set of specifications for a group of objects that share some common characteristics. If we view a concept space as a lexicon of a visual language, an abstract diagram provides the sentence structure with which the words of the lexicon can be described as visual sentences by associating each word with an entity or a relation of the abstract diagram. The sentences are constructed through direct manipulation by the user on the screen (*View*) and controlled by *Controller*. By providing a high level description of domain concepts in the form of a concept space, *Model* can interpret the visual sentences. The VisPro design model specifies the roles of *Model, View* and *Controller*, and how they interact with each other in the design model.

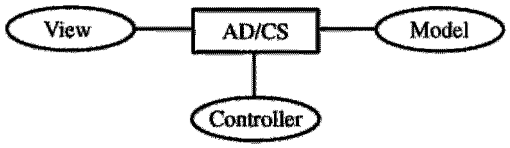


Fig. 8.8. The VisPro design model

- **User Interaction Control:** *View* consists of visual objects which can be manipulated directly on the screen. For example, a user may move the mouse onto a visual object and click the left button to trigger an action. When a visual object receives a user input, *View* sends the visual object to *Controller*, which interprets the input and sends back a control command indicating what *View* should do next. For example, *Controller* may instruct *View* to pop up a menu to allow the user to act further.
- **Diagram Creation:** A graph consisting of a set of visual objects can be created on a visual editor controlled by the *Controller*. Once the graph is constructed, its abstract diagram with domain concepts is created. The mapping relationships between an abstract diagram and a graph implies that the abstract diagram provides a logical interface understood by all the VPE modules, and any modification to the abstract diagram will be reflected on the graph on the screen. A visual editor itself can be a visual object in *View*.
- **Parsing:** *Model* receives parsing demands from *Controller* and performs corresponding transformations and computations on abstract diagrams.
- **Layout and Animation:** If an abstract diagram is associated with visual concepts, the parsing algorithm can perform graph layout and animation by operating on the visual concepts. This is because the appearance of a visual object may be changed dynamically when its visual attributes are modified through the corresponding concepts.

In summary, as long as a domain concept space is provided, each module can be designed independently and used with other modules by sharing an abstract diagram and some domain concepts. The following are detailed descriptions of abstract diagrams and concept space.

8.4 The VisPro Toolset

Based on the above design model, we have developed a generic VPE and a set of visual programming tools for generating domain-oriented VPEs (Zhang and Zhang 1998). The generic VPE can be customized to any domain VPEs once the domain specifications are provided through these tools. Fig. 8.9 shows the generation process, which is supported by the following three tools:

1. *visual object generator* that is used to specify visual objects with desired appearances to be used in the target visual language,
2. *rule specification generator* that is used to provide the parsing specification for the target visual language in the form of graph re-writing rules, and
3. *control specification generator* that is used to specify the control commands for each generated visual object manipulated in a visual editor, which is to be automatically generated.

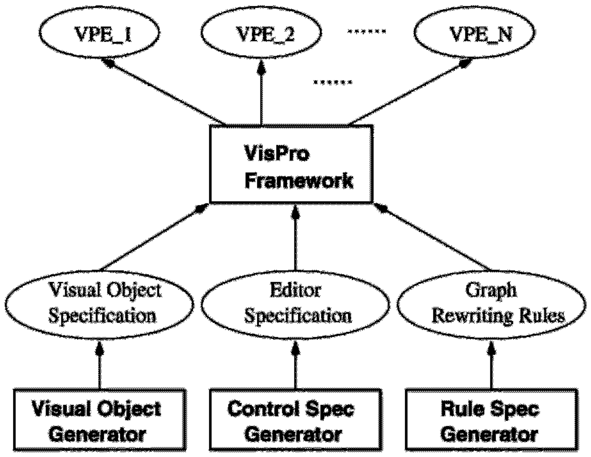


Fig. 8.9. Constructing VPEs with VisPro

In VisPro, the object-oriented language Java serves as a low level specification tool for details which may not be effectively or accurately specified in these visual specification tools. This arrangement allows users to precisely construct effective visual programming environments.

The tools are meta visual programming languages that are used to specify domain VPEs through direct manipulation. First, the Visual Object Generator is used to construct visual objects - it creates the appearance of each visual object, and attaches a specification of its behaviour produced by other tools, or another visual program as its logical function. The user then uses the Control Specification Generator to specify the behaviours of constructed visual objects. The specifications will define and automatically generate a visual editor for the target visual language. Finally, with the Rule Specification Generator, the user can describe the grammar of the visual language in forms of graph rewriting rules (Zhang and Zhang 1997). The rules can be specified as either graphical productions or textual ones written in Java as action codes. Having obtained all the required specifications, the generic VPE becomes customized to the desired domain VPE that integrates the target visual language editor and compiler.

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

8.4.1 A Case Study

In the following sections, we will explain the functions and the use of each specification tool by demonstrating the construction of a simple visual programming environment called *summation*. More sophisticated VPEs can be similarly built through the same process but with more interactions, which will be the subject of Section 8.5. Fig. 8.10 is a snapshot taken during the use of the generated *summation*. Using *summation*, one can sum up integers and visualize results. It has three visual classes: numbers, pluses, and scalers. A number stores an integer which can be entered through the keyboard. A plus receives integers from two numbers and produces their summation, also as an integer, which can be stored as a number, sent to another plus, or sent to a scaler for visualization. A scaler visualizes an integer in a vertical bar. In Fig. 8.10, the maximal (100) and minimal (0) values of the scaler can be changed by entering new values through the keyboard. During the program execution, the displays of the numbers and scalers are changed according to the values sent to them. The following sections introduce the specification tools and explain how *summation* can be created using these tools.

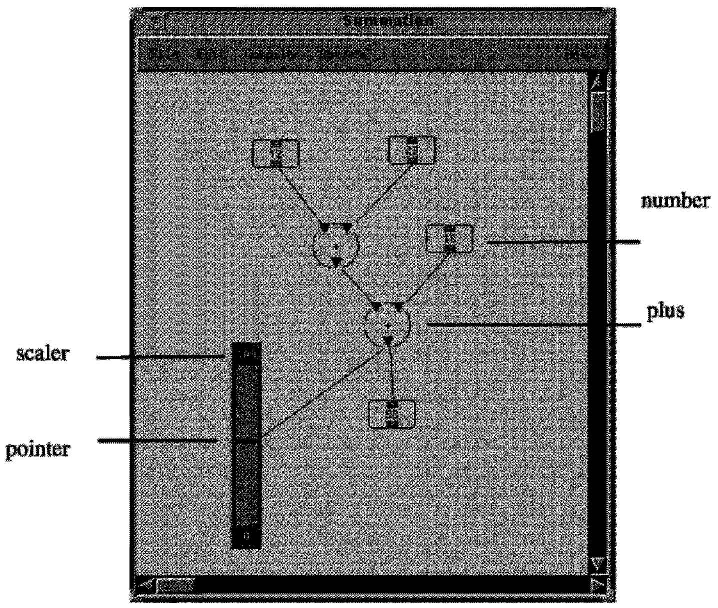
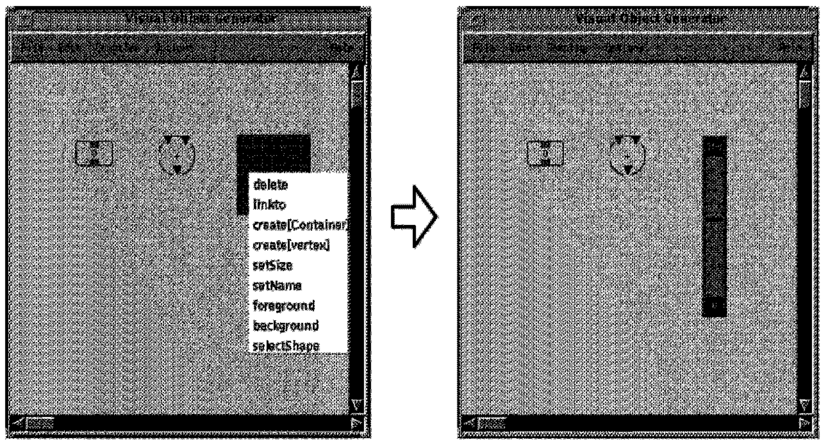


Fig. 8.10. A snapshot of the summation VPE

8.4.2 Visual Object Generator

In the VisPro framework, the Visual Object (VO) generator generates a set of visual classes to suit any special-purpose visual language by editing the predefined visual objects called *VO prototypes*. We call such a process *customization*. A constructed visual object is not just an image. It can be manipulated and may be a composite graph, whose components can be manipulated independently.

A VO prototype is customizable. Fig. 8.11(a) shows two visual classes which have been customized, and a VO prototype which is a black box in its original form. One can edit the VO prototype by triggering the editing commands attached to the box, i.e. by clicking the right mouse button when the cursor is over the box. Fig. 8.11(a) shows the menu commands of the box. For example, by selecting the command create[node] in the menu items, a node can be created in the box. A sub-graph or a node has its own commands which can be popped up in a menu and used for editing the sub-graph to obtain a desired shape and color.



(a) (b)

Fig. 8.11. Snapshots of visual object construction

To construct a scaler, for example, a command called selectShape can be triggered from the pop-up menu. This command opens a dialog box which contains a set of graph patterns. If a scaler is selected, the black box will be reshaped to a scaler as shown in Fig. 8.11(b). The dimension and color of the scaler can be edited and also labeled if necessary.

Fig. 8.11(b) shows three nodes that have been created: number, plus, and scaler. One may notice that the scaler has a fixed pointer. According to the semantics, the pointer should be created dynamically using the mouse during program editing, and a scaler can have more than one pointer at a time. This is done in the VO generator by specifying its construction style as "dynamic" (by selecting the menu item construction in the command menu).

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

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

An edge class can also be created in the VO generator. The edge named flow-to used in summation is defined as shown in Fig. 8.12, where two little filled rectangles are supposed to be replaced by two nodes in an application when the edge is used. The VO prototype of an edge can be customized by changing its shape, color and label through the menu items selectShape, selectColor and setLabel respectively.

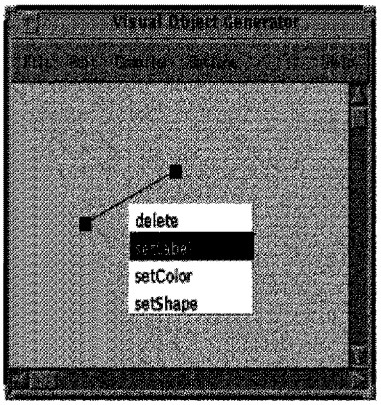


Fig. 8.12. Generating an edge class with the VO generator

A diagram workbench prototype can be customized to a workbench for a specific VPL with a set of node classes and edge classes. This is obtained via the control specification. A workbench can be accessed, e.g. opened, through its icon. Fig. 8.13 shows an icon for the *summation* workbench, which can be created in another diagram workbench as a special icon.

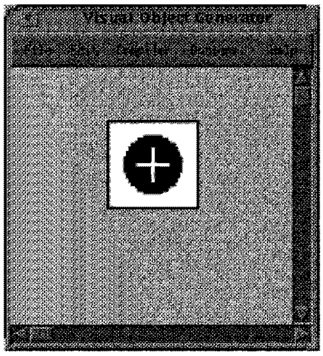


Fig. 8.13. An icon for summation

8.4.3 Control Specification Generator

8.4.3.1 Object-Oriented Editing Commands

The process of editing a graph can be considered consisting of a number of steps, each being an execution of a command on the graph. Usually in a visual editor, commands and visual objects are independent of each other. Execution of a command is the selection of both the command and its target graph. In the object-oriented formalism, a graph is an object which encapsulates a set of related commands.

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

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

8.4.3.2 Command Specification

Fig. 8.14 shows a visual sentence which specifies a part of control in *summation*. The visual objects handled in the CS generator include edge objects, node objects and command objects. An edge object, e.g. an ellipse in Fig. 8.14, is an instance of the edge class defined in the target VPL. Its value is the label type of an edge class, i.e. flow-to, which can be entered or modified through the keyboard. A node object, drawn as an unfilled rectangle, is an instance of a node class. It can be edited to form a *super node* which is embedded with some other nodes. For example, the node labeled plus represents the plus class of *summation*. A command object, i.e. a gray box in Fig. 8.14, represents an editing command, and its value (i.e., delete, link, etc.) can also be entered through the keyboard.

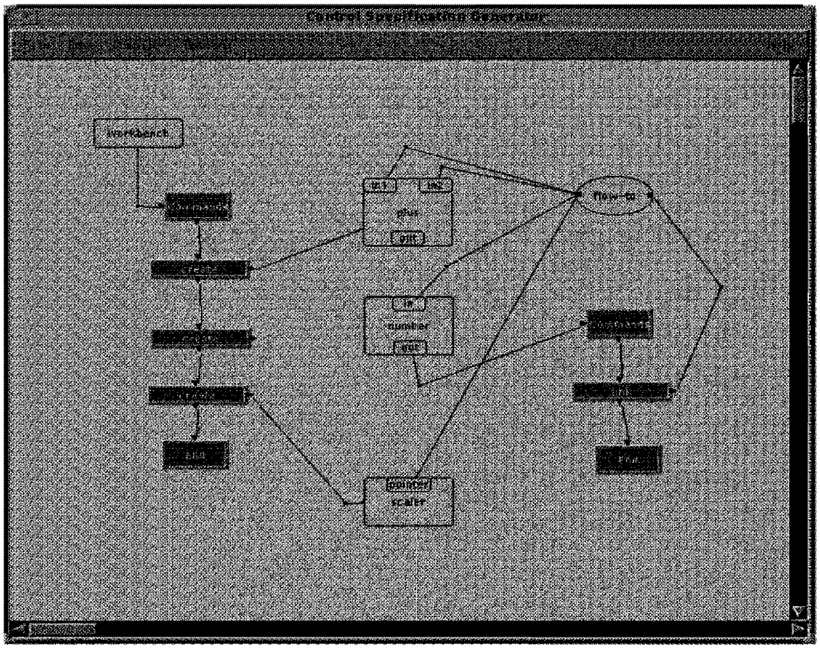


Fig. 8.14. An example of command specifications

In Fig. 8.14, the node object labeled Workbench represents the visual editor for *summation*. "Workbench" is a reserved word in the CS generator. The Workbench node links to a command list which includes three create commands. A create specification can be generated by interpreting the link between a command object and a node. A create command node linking a plus node, for example, will be interpreted to produce a command specification create create[plus] plus, where create[plus] is the name of the command menu item.

The number node object is a super node that has two embedded nodes in and out. The out node has one command link, which links to an edge object labeled flow-to. The flow-to edge object links to the nodes in1, in2, in, and pointer. This indicates that an out can link to an in1, in2, in, or a pointer in *summation*.

The control specification diagram in Fig. 8.14 will produce a list of control specifications as the following:

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

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

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

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

The specifications

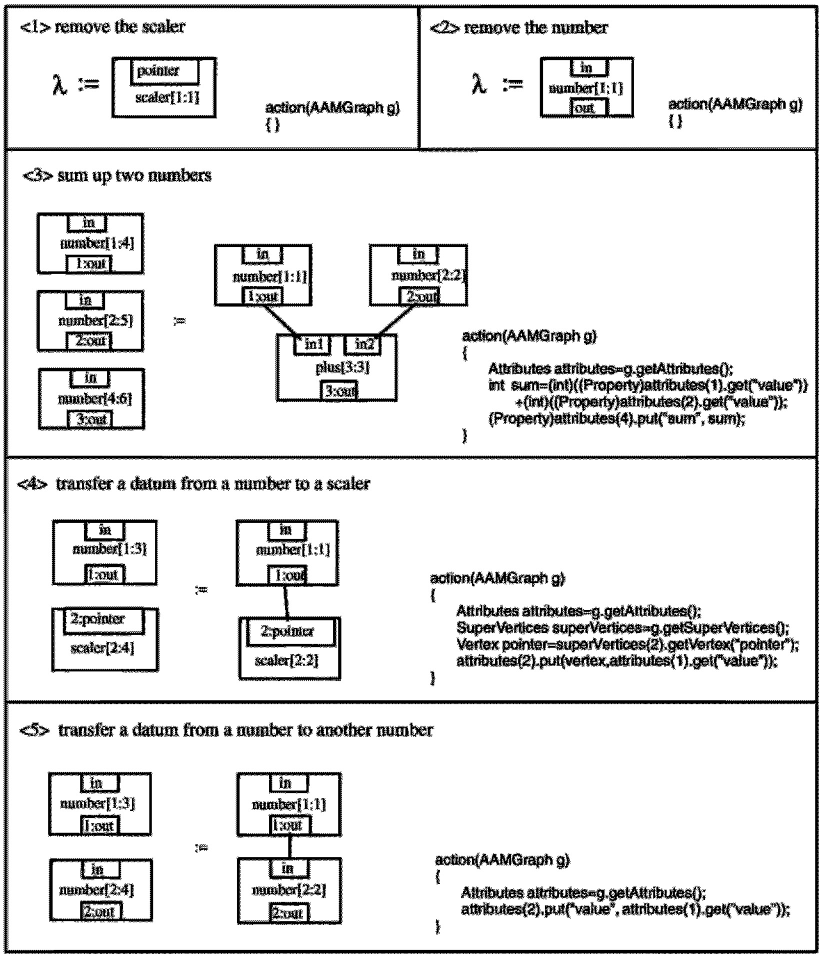
```
VE number.out flow-to plus.in1
 VE number.out flow-to plus.in2
 VE number.out flow-to number.in
 VE number.out flow-to scaler.point.
```

describe that the number.out may be linked to visual objects instantiated from plus.in1, plus.in2, number.in, and scaler.point visual classes, where number.out represents an out node in a number super node. The number.out cannot link to other node classes which are not provided in the specifications.

The control specification diagram in Fig. 8.14 can be extended to specify the complete behavior of *summation*. Thus, the CS generator provides an intuitive and easy way to produce the control specification for a visual editor.

8.4.4 Rule Specification Generator

Fig. 8.15 shows a reserved graph grammar for *summation*. The grammar completely describes the syntax of a valid *summation* diagram. For example,



```
action(AAMGraph g)
{}
```

```
action(AAMGraph g)
{}
```

```
action(AAMGraph g)
{
    Attributes attributes=g.getAttributes();
    int  sum=(int)((Property)attributes(1).get("value"))
          +(int)((Property)attributes(2).get("value"));
    (Property)attributes(4).put("sum", sum);
}
```

```
action(AAMGraph g)
{
    Attributes attributes=g.getAttributes();
    SuperVertices superVertices=g.getSuperVertices();
    Vertex pointer=superVertices(2).getVertex("pointer");
    attributes(2).put(vertex,attributes(1).get("value"));
}
```

```
action(AAMGraph g)
{
    Attributes attributes=g.getAttributes();
    attributes(2).put("value", attributes(1).get("value"));
}
```

Fig. 8.15. A reserved graph grammar for summation

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

The graph grammar also specifies the semantic aspects. Fig. 8.16 shows a valid *summation* diagram. According to the semantics, sub-graph (1) must be interpreted first by applying the grammar rule of Production ⟨3⟩. Sub-graph (2) should be done next, followed by sub-graph (3). Otherwise, a wrong result will be produced. For example, if sub-graph (2) is interpreted first, since its numbers do not have the correct values (from sub-graph (1)), the result of sub-graph (2) will be incorrect. Such an order of applications is not allowed according to the graph rewriting system which dictates that a rule can be applied to an unmarked visual object only if all of the object's edges are matched by the rule.

A detailed description and formal treatment of reserved graph grammars and their parsing complexity can be found in Chapter 2.

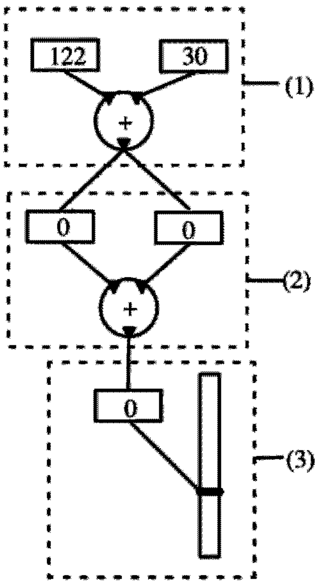


Fig. 8.16. Application of the grammar

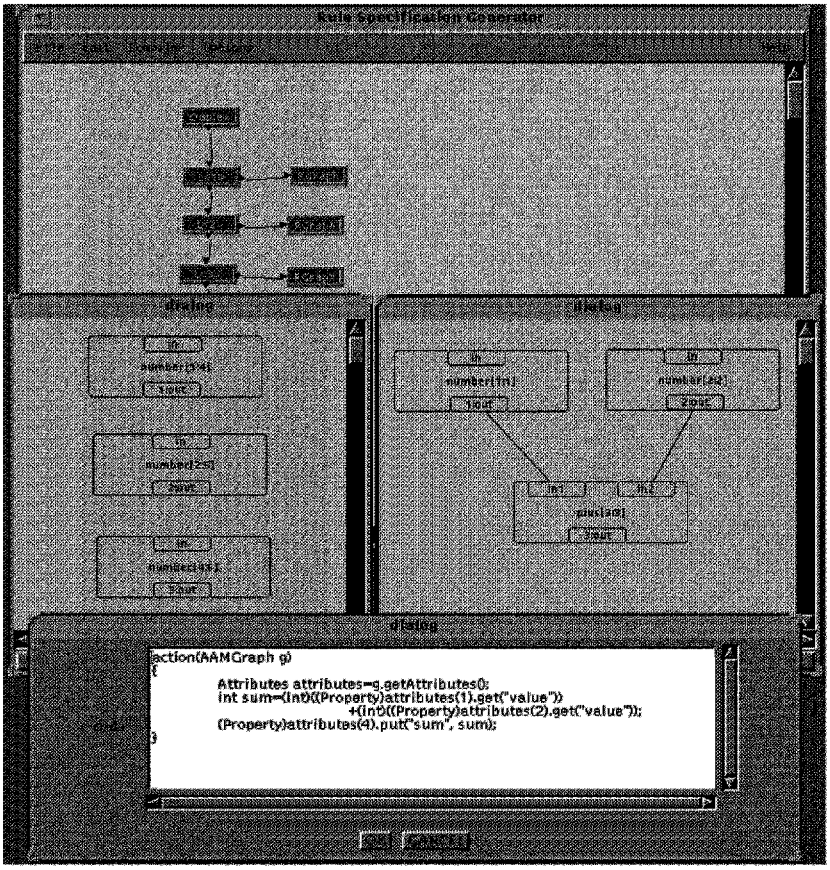
Parsing a diagram takes two phases: syntax parsing and semantics parsing. Syntax parsing is to check whether the diagram is valid. If a diagram is eventually transformed into an initial graph (i.e. λ) by the graph rewriting rules, it is valid. Semantics parsing is to produce a result from a diagram. The result is meaningful only when the diagram is valid. In a translation process, say from a diagram to a textual specification, the syntax and semantics can usually be specified in the same set of graph rewriting rules. In this case, the graph transformation process checks the syntax and translates a graph into a textual specification at the same time. For an executable diagram, this is not always the case. The syntax and semantics specifications of a Petri net visual language, for example, should be specified separately. This is because a Petri net can be executed repeatedly, while the syntax checking must be done in finite steps. For *summation*, the syntax and semantics can be specified with the same set of graph rewriting rules, as shown in Fig. 8.15.

As mentioned in earlier chapters, an action code performs syntax-directed computation over the attributes of a redex (a sub-graph of the program which is isomorphic to the right graph of a production) when the production is applied. The actions of the graph rewriting rules of *summation* are listed in Fig. 8.15. With the actions, the desired results can be produced after the graph transformation. For example, the action of Production ⟨3⟩ is as follows.

```
action (AAMGraph g)
{
  Attributes attributes=g.getAttributes();
  int sum=(int)((Property)attributes(1).get(``value"))
            +(int)((Property)attributes(2).get(``value"));
      (Property)attributes(3).put(``sum", sum)
  }
```

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

The Rule Specification (RS) generator facilitates the rule specification. Fig. 8.17 is a snapshot when using the RS generator, where two kinds of nodes (left graph node and right graph node) are used to represent the left graph and the right graph of a production. For example, the node labeled L⟨3⟩ is a left graph node for Production ⟨3⟩. The node labeled Duplex is the head of the rule list. It indicates that the rules are applied in a duplex mode such that a production is created by linking a left graph node and a right graph node. Each graph node has a sub-editor for defining a graph in the node. In addition, a textual editor workbench can be triggered from the right graph to be used for writing action codes.



```
action(AAMGraph g)
{
        Attributes attributes=g.getAttributes();
        int sum=(int)((Property)attributes(1).get("value"))
                        +(int)((Property)attributes(2).get("value"));
        (Property)attributes(4).put("sum", sum);
}
```

Fig. 8.17. A snapshot of the rule editor

Fig. 8.17 shows a snapshot of creating Production ⟨3⟩ of the *summation* graph grammar, where two windows are opened for editing the left graph and the right graph of the production respectively. Also there is a textual window for editing the action. Thus, the RS generator provides visual editors for specifying the graph rewriting rules, and a textual editor for specifying actions. The RS generator can compose complete graph rewriting rules automatically by interpreting the connected editors.

8.4.5 Implementation

The VisPro architecture includes seven functional modules (Zhang 1997), as shown in Fig. 8.18:

- The configuration interpreter receives the configuration specification, and transfers the lexicon definition of the specification to the user interface and the grammar specification to the parsing module.
- The user interface controls the interaction between users and the VisPro tools.
- The underlying structure manages the diagrams which are being edited.
- During parsing, the logical structure module creates and manages the logical graph converted from the underlying structure of a diagram.
- The parsing module is designed to parse the logical structure of a diagram using the reserved graph grammar formalism.
- The documentation module automatically records edited diagrams and parsing results.
- The actions module collects actions for each VPL from grammar specifications. The collected actions are represented as a Java program and dynamically linked to the parsing module during execution.

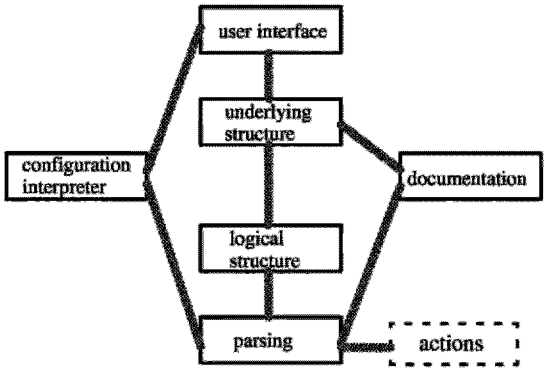


Fig. 8.18. The VisPro architecture

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

8.5 A Case Study: Generating A Distributed Programming Environment

This section demonstrates the application of VisPro in generating a distributed programming environment, called *PEDS*. It describes the features of PEDS and then shows how the PEDS environment is generated using VisPro.

8.5.1 PEDS

In a heterogeneous distributed system, processors and software resources available are of different types. It is often difficult for a user to interface cooperative processes which are implemented with different software resources and located on different processors (Grimshaw et al. 1994; Shatz 1993). Unfortunately, there are few systems that are aimed at providing shared processing power in a distributed environment, while taking into account the utilization of software resources of the environment.

The programming environment for distributed systems, or PEDS, has the following important features.

- It consists of a set of tools (visual languages) which can cooperate with each other to solve complicated distributed problems.
- It supports developing distributed program graphically, so that resources sharing and mapping can be visually specified. Moreover, different graph formalisms, such as control flow graphs and Petri Nets, can work consistently in a single environment.
- A distributed program is divided into several local processes, which may be located on multiple physical machines. Local processes can be written using different tools based on existing software resources of the distributed system such as compilers and program libraries.
- A user can have the freedom of control over the mapping of processes to processors. With a high-level graphical notation, a user can specify the processor assignments completely, partially, or leave it entirely up to the environment.

Based on a distributed graph model (Cao et al. 1995), PEDS can be used to implement a wide range of distributed programs. Each distributed program is modeled as a set of related diagrams. The components of a diagram are implemented with existing software resources of the distributed system. The construction process is independent of any specific distributed system, and a constructed program can be mapped onto different configurations by a flexible mapping facility.

Graph Modeling of Distributed Applications

When designing distributed programs, programmers commonly draw informal directed graphs showing distributed structures (Zhang et al. 1999). These graphs abstract away the details of the nodes being designed and concentrate on their interactions. The advantage of this is that the programmer can specify the distributed structure without concern about the internal working of each node.

PEDS uses a graph abstraction method to represent distributed programs. It divides a distributed program into several local processes (LPs) and defines their interactions. Each LP can be allocated on a processor in a distributed system. A LP is characterized by the fact that all work initiated in it is, mostly, limited to its sphere of control; it essentially executes independently of other LPs except for specific points in its processing when it needs to interact with other LPs.

The process of creating a distributed program is then divided into two steps: drawing an overall graph and creating corresponding LPs. A graph-based visual distributed programming language can help realize this process (Chan et al. 2005). With the visual programming language, we separate the specification of LPs from that of synchronization and communication, and express synchronization and communication directly (but abstractly) using graphs.

A LP is defined as a graph node, which can have a set of input and output ports. With these ports, a graph illustrates the interaction among LPs. As LPs are located on distributed processors, the interactive behavior describes the message-passing mechanism that is performed between distributed processors. A processor can send messages directly only to a subset of the processors with which it is directly connected. Its directly connected processors are called its neighbors. For communication with non-neighbors, a routing algorithm is needed. Routing is the term used to describe the decision procedure by which a processor selects one of its neighbors to send a message to an ultimate destination.

Programming Tools

In PEDS, various distributed tools are used to support implementing the mentioned functionalities. They can cooperate with each other to create sophisticated distributed programs. Such tools, which will be called *workbenches*, include:

- *High Level Process Flow Diagram workbench (HPFD workbench)* is a process flow diagram providing a high level control structure over a set of processes, whose details can be specified in other workbenches.
- *High Level Petri Net workbench (HLPN workbench)* is a modeling tool for specifying the high level behavior of a task using Petri net. Each of its transitions can be connected to a workbench, with which the transition specification can be provided.
- *Java workbench* provides a platform for editing and compiling Java programs.
- *Supporting workbench* is used to specify a set of available software resources and their relationships for mapping processes to processors.
- *Net workbench* is for specifying the configuration of processors and their interconnecting network (e.g. a distributed system).
- *Distributed workbench* (Fig. 8.19) is the top level working environment that is used to configure all the other workbenches to form an integrated distributed application.

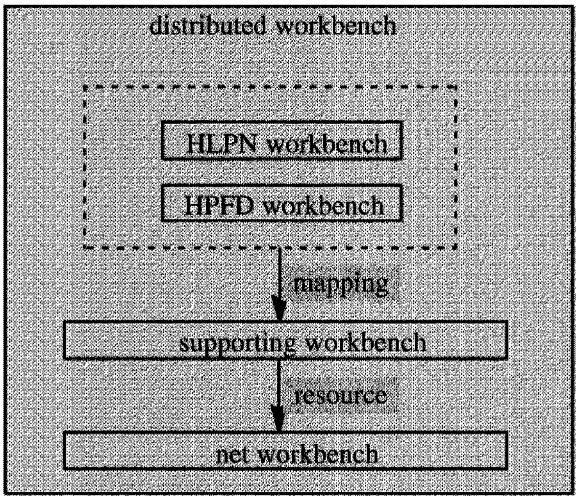


Fig. 8.19. Distributed workbench

A distributed program modeled with a set of HLPN workbenches and HPFD workbenches is linked to a supporting workbench for resource mapping, which in turn links to a net workbench for finding proper set of processors. PEDS is, therefore, a hierarchical programming environment supporting multiple programming paradigms.

8.5.2 Generation of PEDS Using VisPro

This section focuses on how to construct the PEDS hierarchical environment and implement interactions between different sub-visual languages using the VisPro system.

Hierarchical Environment

An icon in the PEDS interface represents a window, called a *workbench*, which can be opened and operated upon and can include child icons. The main window in PEDS is an icon window, called management-win, where various icons can be created and managed. Apart from the management-win, there are other windows: HLPN workbench, HPFD workbench, Java workbench, supporting workbench, net workbench and configuration workbench.

In an icon window, each workbench represents a program (or a specification). For example, when one wants to create a Petri net, he/she can create an HLPN-icon, (i.e., an icon for HLPN window). By opening the HLPN-icon, one can create a Petri net. The HLPN-icon, thus, uniquely represents the created Petri net.

Fig. 8.20 shows the hierarchical programming environment of PEDS. It has a main icon window in which some child icons have been created. A child icon window and a HLPN workbench are also shown in the figure. They can be triggered from icons in the main window.

Generating such a hierarchical programming environment is easy in the VisPro system. First, one can create icon classes in the VO generator for each of the workbenches. Then the control specifications for the icons can be created, which include

```
ND-has-node management-win management-icon
ND-has-node management-win HPFD-icon
ND-has-node management-win HLPN-icon
ND-has-node management-win Java-icon
ND-has-node management-win supporting-icon
```

ND-has-node management-win net-icon
ND-has-node management-win configure-icon

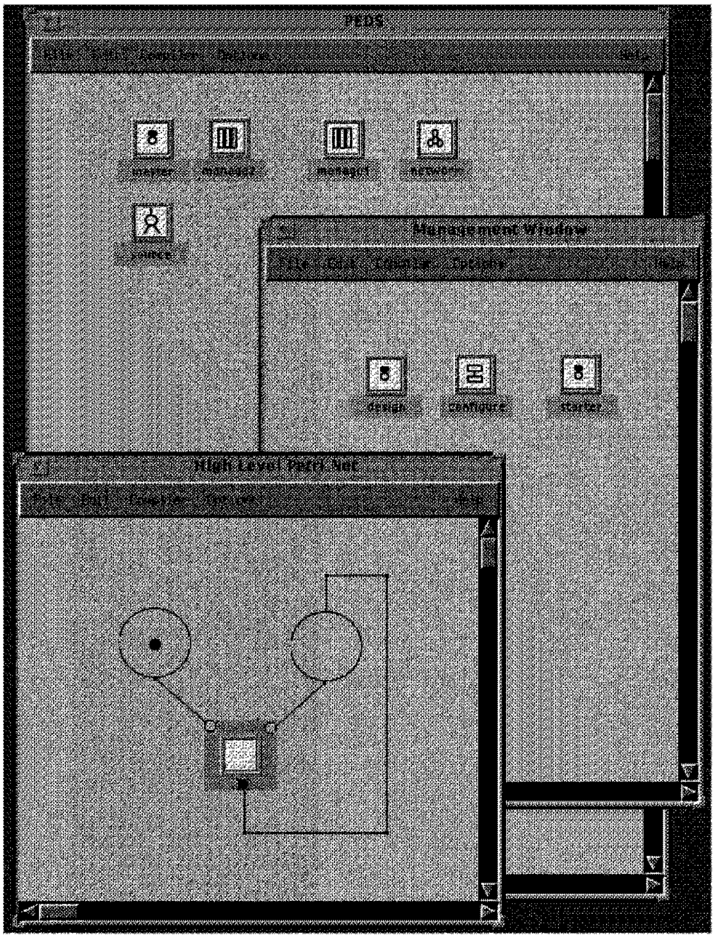


Fig. 8.20. PEDS hierarchical programming environment

It indicates that seven classes of icons, namely, management-icon, HLPN-icon, HPFD-icon, Java-icon, supporting-icon, net-icon, and configure-icon, are created in a management-win to include the seven icons. Each of the child icons created in the management-win may open a corresponding window (i.e. workbench). This can be specified as:

ND-has-workbench management-icon management-win
ND-has-workbench HLPN-icon HLPN-win

...

It indicates that seven classes of icons, namely, management-icon, HLPN-icon, HPFD-icon, Java-icon, supporting-icon, net-icon, and configure-icon, are created in a management-win to include the seven icons. Each of the child icons created in the management-win may open a corresponding window (i.e. workbench). This can be specified as:

```
ND-has-workbench management-icon management-win
ND-has-workbench HLPN-icon HLPN-win
...
```

The first item specifies that one can open a management-win on a management-icon object. Similarly, the second item indicates that a HLPN-win can be opened through an HLPN-icon, where HLPN-win is the workbench class for HLPN. In addition, commands should be added to each of the icon classes, such as an open operation. For example:

```
open open-management management-win
```

specifies a menu item named open-management which can be triggered to open a management-win.

The VisPro framework is able to create a command system over each of the icons according to the control specification. Icons and their commands provide a mechanism for hierarchically specifying distributed programs. For a hierarchical programming environment, interactions between graph nodes and workbenches should also be created.

Construction of Interactions

A workbench *w1* may be used to specify a sub-task of another workbench *w* by linking *w1*'s own icon with a node of *w*. As an example, we use a HLPN workbench to illustrate the interaction between different specification levels. Fig. 8.21 shows the visual objects used in a HLPN workbench. Normal objects in a Petri net are transitions and places. An annotation object can specify the annotation for a visual object by linking to the visual object. To specify the data transfer mechanism, we construct two classes: *input port* and *output port*. An input port can be used in a transition to specify the input information (i.e., name, type, etc.). When an input port used independently, it represents an input from outside and is called a global input port. Similarly, we have output ports and global output ports. Annotations can be given to an input port or an output port to specify its type and name. The reference object in a transition is used to link to another workbench for specifying the transition details.

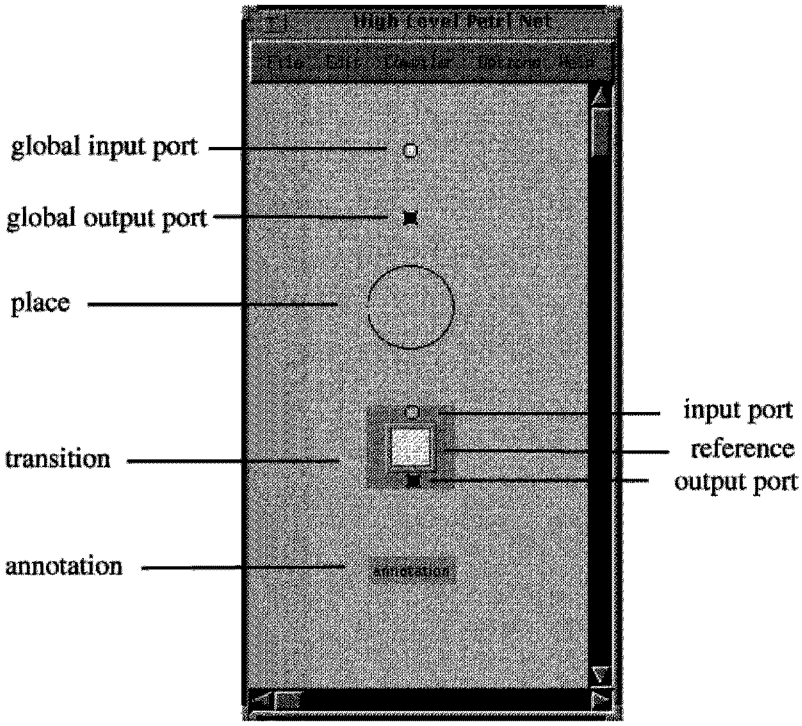


Fig. 8.21. Visual objects in a HLPN workbench

Fig. 8.22 shows two HLPN workbenches, where workbench (b) is used to specify a transition in workbench (a). To transfer data correctly between two workbenches, each port of the transition is annotated. In workbench (b), two global input ports, annotated as `int count` and `int work-id`, have the same annotations as the input ports of the transition. Input ports with the same annotations are taken as the same port. Thus, data accepted in the port annotated as `int count` in workbench (a) is accepted in the port annotated as `int count` in workbench (b). How data is transferred in output ports can be specified in a similar fashion.

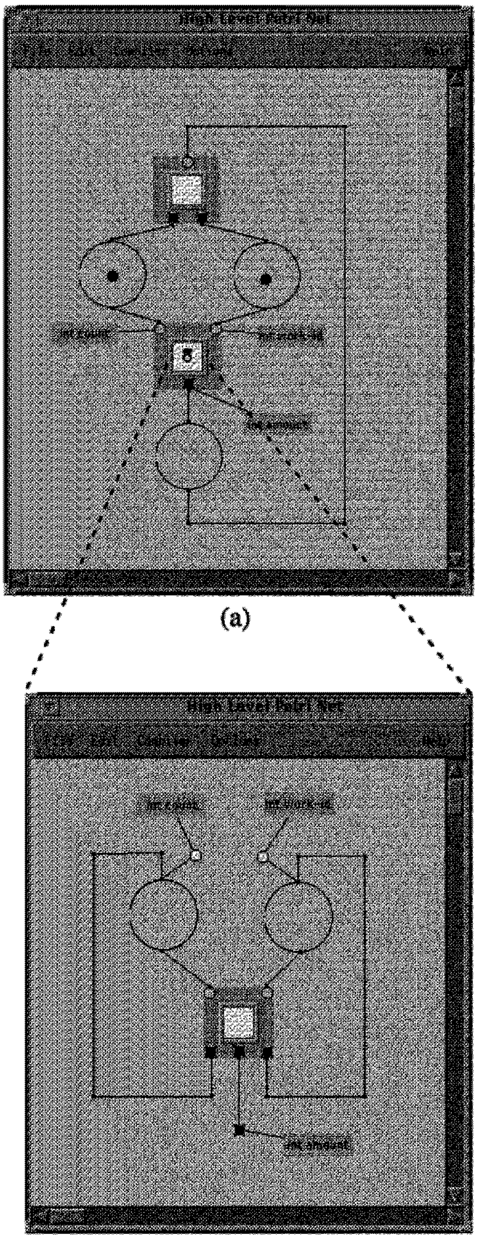


Fig. 8.22. Hierarchical specifications

To implement the data transfer mechanism, we associate a transition with an *Attribute* concept if the transition refers to another workbench such that its ports are specified as attributes in the concept. For example, the input port labeled int count is specified as attribute("int count", parameter), where parameter is data transferred to the port. In the sub-workbench, a global input port will check the *Attribute* concept and access the parameter it needs. In this way, data are transferred between different workbenches properly.

After the user has specified the hierarchical programming environment of PEDS and the interactions between the workbenches, the VisPro framework is customized to PEDS.

This section has discussed how to generate PEDS using the VisPro system. The management of the environment is described in the control specification. On the other hand, interactions between different workbenches can be specified in the annotations and with the *Attributes* concept. PEDS, a hierarchical programming environment with multiple paradigms, thus, is created by customizing the VisPro framework through specifications.

8.6 Summary

This chapter has presented a generic visual language generation environment with a hierarchical specification structure and multiple programming paradigms. To ease the development of VPLs, we have proposed a VisPro design model that divides any VPE into independent functional modules and defines a protocol supporting the interaction between the modules. The VisPro toolset with its framework has been developed based on the above design model, which can be used to generate diagrammatic VPLs. The toolset consists of three specification tools, each of which facilitates one aspect of the construction of VPLs in the VisPro framework. These tools are visual languages themselves so that the target language properties and the domain specifications can be visually described by direct manipulation.

The VPL construction process using the VisPro toolset is similar to the textual language construction process using Lex/Yacc. The process can be described as customization, i.e. the VisPro framework can be customized to any target visual language with a set of domain specifications provided through the tools. The VisPro framework and the specification tools together provide a complete support for the VPL generation. They can be used to generate a wide range of visual programming environments easily.

8.7 Related Work

Researchers have developed some high level tools to ease the implementation of visual languages and visual programming environments. These tools can greatly reduce the effort of developing visual languages, although they focus mainly on the construction of user interface aspects of VPLs. Example tools include SIL-ICON, VLCC, Escalante, Glide, SPARGEN, DiaGen, PROGRES, and Fujaba, as described below.

Early tools such as SIL-ICON (Chang et al. 1989) and VLCC (Costagliola et al. 1995) use parameterizable frameworks to support VPL generation. They are easy to use since generating target VPLs is simply done by customizing the predefined frameworks through domain specifications. SIL-ICON (Chang et al. 1989) has a complete functionality for the construction of icon-based visual languages. The SIL-ICON compiler is based on the generalized icon theory and thus is limited to iconic VPLs. VLCC (Costagliola et al. 1995) assists the user with tools for defining a language's syntax, semantics, and graphical objects. It produces an integrated environment with an editor and a compiler for the defined language. VLCC uses positional grammars as the underlying theory and pure images as single-level visual objects.

SPARGEN (Golin and Magliery 1993) is a visual language compiler generator. Its generated parser supports additional action routines written in C++, thus allows complicated actions to be specified in the form of rules. SPARGEN does not support the generation of a visual programming environment.

Escalante supports the construction of applications for visual languages that are based on object-relationship abstractions (McWhirter and Nutt 1994). It provides mechanisms for iterative design, rapid prototyping and generation of visual language applications within an integrated environment.

Glide (Kleyn 1995) provides a BNF-like language for specifying the logical structure and the user interface of a VPL. The user specifies a graph data structure, associates graphical attributes to the data structure, and then desribes a set of permissible changes to the data structure. Glide constructs a VPL based on the specified data structure. It can also reason about the VPL through its logic programming rules. Since the Glide grammar is used for creating the underlying data structure in the form of links between nodes, it is unsuited for specifying the syntax of a VPL grammar.

DiaGen (Minas and Viehstaedt 1995; Minas 2002) is a tool for producing diagram editors, which can be used to construct visual programs. DiaGen uses hypergraphs to model various types of diagrams, with a hyperedge graph grammar that can be parsed efficiently. It recognizes syntactic corrections of graphs during the editing process. DiaGen supports both freehand and syntax-directed editing modes, which provides a flexibility for users.

PROGRES (Rekers and Schürr 1996; Schürr et al. 95; Rekers and Schürr 1997) is a strongly typed multi-paradigm language with a well-defined context-free syntax, type checking rules and semantics. The graph rewriting rules in PROGRES provides a powerful formalism for graph transformations and are particularly suitable for specifying semantics of VPLs whose underlying structures are node-edge graphs. PROGRES can generate both programming environments and parsing algorithms. It does not use any existing programming language (e.g. C, C++, or Java) to specify the actions of its rules directly. Instead, it uses a simple textual language which is a part of the system. PROGRES uses layered graph grammars to specify VPLs. The parsing algorithm of layered graph grammars requires exponential time, as analyzed in Chapter 2.

The *Attributed Graph Grammar system* (AGG) (Ermel et al. 1999) is a visual programming environment based on a hybrid programming language, i.e. the AGG language, that integrates graph transformation rules with Java. An AGG program consists of a set of production rules attributed by Java expressions so that the standard Java library can be used to compute objects' attributes.

VLPEG (Ferrucci et al. 2001) can automatically generate a visual language environment consisting of a graphical editor, and a lexical, syntactic and semantic analyzer. It is based on the context-free Symbol Relation Grammar model (Ferrucci et al. 1996), that can specify the relationships among the symbols in a visual sentence at a higher level of abstraction than that in RGGs. Similar to VisPro, VLPEG supports incremental development of visual programming languages through rapid prototyping.

Fujaba (2005) provides a UML-like CASE tool for round-trip engineering. It supports code generation from class diagrams, activity diagrams, state diagrams, and collaboration diagrams, as well as reverse engineering from Java code to UML diagrams (Maier and Zündorf 2003). Graph transformation in Fujaba is used to specify the behavioral aspects of modeling systems through *story diagrams* (Fischer et al. 1998), a graph rewriting language that combines several UML diagrams.

Chapter 9 Conclusions and Future Perspective

9.1 Conclusions

The book has described visual programming languages and their applications in several important domains. We started with a general introduction to the concepts of visual languages and presented a theory behind such languages, i.e. a context-sensitive graph grammar formalism, known as the reserved graph grammar (RGG). The RGG was based but improved on the LGG (Rekers and Schürr 1997). By keeping the layer decomposition mechanism of the LGG to terminate parsing in finite steps, the RGG uses a marking technique with an embedding rule to solve the embedding problem. The rule ensures that the application of a production in the graph rewriting process would not create dangling edges. By ensuring selection-free productions in the RGG, sometimes called *confluent graph grammars* elsewhere, a *selection-free parsing algorithm (*SFPA) attempts only one parsing path and thus achieves a polynomial time complexity. It is difficult to estimate how many types of diagrams could be specified by selection-free productions. Having applied RGGs to many different application domains, we have not come across a diagram formalism that cannot be specified by selection-free productions.

By extending the RGG with spatial specification capability and a more efficient parser, we obtained the spatial graph grammar (SGG) formalism. The SGG formalism was motivated due to the graphical nature of visual languages - the spatial information not only contributes to the representation, but also intuitively conveys structural and semantic constraints. For example, an order over a set of objects can be directly specified according to their spatial locations (e.g. the left object has a smaller index than the right). In a SGG, both the connectivity and spatial relationships construct the pre-condition of a graph transformation. This sets a solid foundation for visual and spatial reasoning, with a great potential for practical applications such as geographical information systems.

The subsequent four chapters have described the applications of visual languages and graph grammars in multimedia authoring and presentation, data interoperation, software engineering, and Web development.

The diversity of the multimedia devices and the advance of multimedia technology demand multimedia presentations to be intelligently adaptive to different viewing contexts. With a graph grammar based authoring approach, a grammar functions as a mapping from a presentation of a style to a physical layout. The syntactical definition in the grammar captures the structure of any multimedia document and the parser performs automatic validation on the document. Rather than assigning every object with an absolute co-ordinate value, spatial graph grammars specify how the document looks like by defining spatial relations in the grammar. Automatic presentation layout could also be performed through the parsing process.

To support data interoperation at different levels of abstraction, we have proposed a framework with a uniform graphical representation of data instances and models. Operations on data instances and models could also be represented visually using a set of graph transformation rules. The framework graphically defines the syntax and translation rules for data instances, and high-level operators for schemas and models. The framework also provides an intuitive interface for users to customize the operators. The presented concepts represent a step forward to automatic data interoperation through generic visual operators, reusable graphical interfaces, and customizable declarative rules.

A language with a simple and well-understood syntax, possibly with a graphical representation, benefits the communication of software designs between different designers. Using graphs to interpret graphs, graph transformation, which offers a promising formal approach to modeling architectural evolutions and, dynamic behaviors of software systems. We have presented a grammatical approach to the specification and evolution of software architectures. The validation of structural integrity through syntactic checking is made possible by the underlying grammar. Moreover, such an approach is open and interoperable with other formal approaches. For example, by viewing evolution as transitions of system states, the model checking technique applied in the large and realistic systems (Baresi et al. 2003) can be incorporated.

Web engineering and development, including design, search, navigation, and maintenance, has been a central focus of the Internet technology. Graphical support for Web engineering that hides the coding of mark-up languages would expose and ease the activities of Web development to the general public. We advocate a uniform graphical view throughout the design, navigation, reuse, and maintenance cycle of Web site development. The structured design supported and validated by the Web graph grammar would benefit the search and navigation, and thus promote the semantic Web. It also enhances the reusability and maintainability of many legacy Web systems.

Fundamental to the power and applicability of visual languages in many application domains is the generative technology. The major advantages of the grammatical approach with the generative technology can be summarized as the following:

- The meta-tool capability allows any domain-specific visual languages (DSVLs) to be automatically generated according to the specification of the domain characteristics. A DSVL can thus be readily created, modified and enhanced whenever the domain needs arise. Thus an iterative and incremental process is supported.
- Any non-disposable software prototype conforming to the specifications can be visually constructed through direct manipulation by a domain programmer who needs not to know the detailed specification. Once the prototype is confirmed to meet the domain requirements, the full scale prototype program can be generated.
- Verification is naturally supported. The generated DSVL environment includes a syntax-directed visual editor that is capable of syntactic checking and function verification of any prototype constructed in the DSVL.

9.2 Future Perspective

Visual programming languages have been investigated for nearly 25 years as overviewed in Chapter 1. Visual language theories and their applications have been significantly advanced over the 25 years. As discussed in Chapter 1, the general concept of visual languages has been used long before the invention of the visual languages for computer programming, i.e. visual programming languages (VPLs). The trend is to move back to the more general sense of visual languages for various types of applications, rather than merely for the programming purpose. Visual modeling has entered the main stream of software engineering, largely due to the success of UML.

Considering the future of diagrammatical visual programming languages, three major hurdles need to be overcome as summarized below.

- **The inefficient compilation due to high dimensional search and match of sub-graphs.** More effective graph grammars with efficient parsers need to be developed, which may be the most challenging issue for all VPL researchers. The expressiveness of a graph grammar appears to always work against the efficiency of the parser. Spatial specification assisting narrowing down the search space represents an interesting attempt to gain both efficiency and expressiveness.
- **The lack of effective yet intuitive programming metaphors supported by efficient user interactions.** Research in human factors in visual programming has been at its early stage. Commonly accepted visual metaphors may be used as visual language primitives to support user-friendly interactions. Empirical studies on the effectiveness and usability of such visual metaphors need to be conducted. This is largely an interdisciplinary research involving not only computer science, but also semiotics, cognition science, and ergonomics.
- **Less scalable than textual programming due to the more use of the screen estate.** To meet the new challenges multi-dimensional visual programming, particularly in distributed and collaborative computing environments, we need investigate visual programming in the large (Chang et al. 1999). This is also related to the effective use of visual metaphors. The capability of hierarchical yet graphical specifications through grammar rules is indispensable in a scalable visual programming environment. One solution is to introduce hierarchical views (Pietriga et al. 2001). Another is to apply the grammar replacement process to support graph expansion and shrinking, which would however limit to context-free graph grammars.

Research in VPLs will continue in these directions, together with other issues such as testing, tracing, and comprehension. Also, more work needs to be done to enrich the grammar formalism and enhance the event-handling capability. We describe some specific future projects that are worth pursuing.

Designing dynamic user interfaces to support interactive communication for new applications, especially mobile devices, is a research and design problem. In dynamic capture, access, and authoring of Web and multimedia presentations, the attributes of some media elements may be defined in terms of those of other media elements, or in relation to the attributes of the viewing environment. Attributed grammars that are capable of adapting to multiple simultaneous changes offer a promising solution.

We may consider a visual language for multimedia authoring and presentation as a multidimensional language that has objects of different media types as its primitives and spatial and temporal operators as its operators (Chang 2000). A future direction of investigation is to add the time dimension to the design of multimedia presentations. Temporal specifications determine the sequence of presentation. Allen presented some common temporal relations such as during, before, meet relations (Allen 1983), which are potentially adaptable to the grammar as conditional attributes. Extensive research has been devoted to the temporal aspects of multimedia authoring and presentations. We plan to investigate the combined use of temporal and spatial specifications and explore the full power of the grammatical approach.

Writing production rules and their action codes for any graph grammar is not an easy task, even for a design expert with computer science training, since it requires a good command of the grammar formalism. It has been the author's goal to partially automate the production authoring tool to create part of the rules upon layout requirements. Using an induction engine is one solution that can simplify the grammar construction process. Often, samples of a visual language are processed to automatically construct a graph grammar, at least in part (Ates et al. 2006). The induced graph grammars can be further modified by the user.

Most grammar induction methods are based on inductive logic programming (ILP) which represents data in first-order logic. An induction process is performed on the logic to produce a learned set, or generalization, of rules on the data. Inducing context-free graph grammars using ILP is a promising approach. ILP-based induction systems, such as SubdueGL (Jonyer 2003) and its successor (Kukluk et al. 2006), have produced some encouraging results. Automatic induction of context-sensitive graph grammars such as RGGs is a challenging research direction.

The ability of inducing RGGs will not only ease the grammar writing efforts in various applications such as those discussed in this book, but also empower more advanced applications. For example, in model-driven architectures, we can define the behavioral semantics of state diagrams grammatically. Instead of manually designing a graph grammar to formalize the state hierarchy, we will be able to automatically derive the graph grammar from a state machine. In order to interpret state transitions of simple and composite states, graph grammars automatically induced from state machines could be used to validate, recognize and generate active state configurations. Such a grammar induction mechanism and its validation and recognition capability would facilitate the automated design of model-driven architectures.

Apart from software engineering, visual languages and their underlying grammar formalisms will continue to be applied to more application domains, including the deployment of senor networks, scientific modeling and simulation, round-trip engineering, data interoperation, resource management, digital design, and pattern recognition.

Bibliography

Abiteboul S, Cluet S, Milo T (1997) Correspondence and Translation for Heterogeneous Data. *Proc. Int. Conf. on Data Theory (ICDT)*

Allen JF (1983) Maintaining Knowledge about Temporal Intervals. *Communications of the ACM.* 26:832 – 843

Allen R, Garlan D (1994) Formalizing Architectural Connection. *Proc. 16th Int. Conf. on Software Engineering.* pp 71-80

Allen R, Garlan D (1997) A Formal Basis for Architectural Connection. *ACM Transaction on Software Engineering and Methodology* 6:213-249

Ananda A, Srinivasan B (1991) *Distributed Computing Systems: Concepts and Structures.* IEEE CS Press, Los Alamitos, California

Andre E, Finkler W, Graf W, Rist T, Schauder A, Wahlster W (1993) WIP: The Automatic Synthesis of Multimodal Presentations. In: MayBury M (Ed) *Intelligent Multimedia Interface.* AAAI Press/MIT Press, Cambridge, MA, pp 75-93

Anupam V, Freire J, Kumar B, Lieuwen D (2000) Automating Web Navigation with the WebVCR. *Proc. 9th Int. World Wide Web Conf..* Amsterdam, Netherlands

Ates K, Kukluk J, Holder L, Cook D, Zhang K (2006) Graph Grammar Induction on Structural Data for Visual Programming. *Proc. 18th IEEE Int. Conf. on Tools with Artificial Intelligence (ICTAI'06).* Washington D.C., USA

Atzeni P, Torlone R (1995) Schema Translation Between Heterogeneous Data Models in a Lattice Framework. *Proc.6th IFIP TC-2 Working Conf. on Data Semantics (DS-6).* Atlanta, Georgia

Bardohl R, Taentzer G, Minas M, Schürr A (1999) 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, World Scientific, pp 105-180

Baresi L, Heckel R, Thöne S, Varró D (2003) Modeling and Validation of Service-Oriented Architectures: Application vs. Style. *Proc. ESEC/FSE'03.* pp 68-77

Bederson B, Hollan J, Steward J, Vick D, Ring L, Grose E, Forsythe C

(1998) A Zooming Web Browser. In: Ratner, Grose, Forsythe (Eds) *Human Factors in Web Development.* Lawrence Erlbaum Assoc. pp 255-266

Bernstein PA (2003) Applying Model Management to Classical Meta Data Problems. *Proc. 2003 CIDR Conf.*. Asilomar, CA, pp 209-220

Bernstein PA, Halevy A, Pottinger RA (2000) A Vision for Management of Complex Models. *SIGMOD Record* 29:55-63

Bernstein PA, Giunchiglia F, Kementsietsidis A, Serafini JML, Zaihrayeu I (2002) Data Management for Peer-to-Peer Computing: A Vision, *Proc. 5th Int. Workshop on the Web and Databases*. Madison, Wisconsin, pp 89-94

Berry G, Boudol G (1992) The Chemical Abstract Machine. *Theoretical Computer Science* 96:217-248

Bézivin J, Breton E, Dupé G, Valduriez P (2003) The ATL Transformation-based Model Management Framework. Research Report No.03.08, Université de Nantes, Sep. 2003

Bjork S, Redstrom J, Ljungstrand P, Holmquist LE (2000) PowerView - Using Information Links and Information Views to Navigate and Visualize Information on Small Displays. *Proc. HUC'2000*. pp 46-62

Blostein D, Grbavec A (1997) Recognition of Mathematical Notation. In: Bunke H, Wang P (Eds) *Handbook of Character Recognition and Document Image Analysis*. World Scientific, pp 557-582

Blostein D, Schürr A (1998) Visual Modeling and Programming with Graph Transformations. *Tutorial at 14th IEEE Symposium on Visual Languages*. Halifax, Canada

Booch G, Rumbaugh J, Jacobson I (1999) *The Unified Modeling Language User Guide*. Addison-Wesley.

Borning A, Lin RK, K. Marriott K (2000) Constraint-based Document Layout for the Web. *Multimedia Systems* 8:177-189

Bottoni P, Grau A (2004) A Suite of Metamodels as a Basis for a Classification of Visual Languages. *Proc. 2004 IEEE Symposium on Visual Languages*. Rome, Italy, Sep. 27-30, pp 83-90

Bottoni P, Levialdi S (2005) Resource-Based Models of Visual Interaction: Understanding Errors. *Proc. 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*. Dallas, USA, Sep. 20-24, pp 137-144

Bottoni P, Taentzer G, Schürr A (2000) Efficient Parsing of Visual Languages Based on Critical Pair Analysis and Contextual Layered Graph Transformation. *Proc. 2000 IEEE Symposium on Visual Languages*. Seattle, USA, pp 59-60

Bowers S, Delcambre L (2002) A Generic Representation for Exploiting Model-Based Information. *ETAI Journal*, 6

Bradbury JS, Dingel J (2003) Evaluating and Improving the Automatic Analysis of Implicit Invocation Systems. *Proc. ESEC/FSE'03*. pp 78-87

Brandenburg FJ (1988) On Polynomial Time Graph Grammars. *Proc. 5th Conf. on Theoretical Aspects of Computer Science*. LNCS 294, Springer-Verlag, pp 227-236

Brandenburg FJ (1995) Designing Graph Drawings by Layout Graph Grammars. *Proc. Int. Workshop on Graph Drawing (GD'94)*. LNCS 894, Springer, pp 416-428

Brown PC (1997) Satisfying the Graphical Requirements of Visual Languages in the DV-Centro Framework. *Proc. 13th IEEE Symposium on Visual Languages*. Capri, Italy, Sep. 23-26, pp 84-91

Bulterman DCA, Rutledge L (2004) *SMIL 2.0 - Interactive Multimedia for Web and Mobile Devices*. Springer

Buneman P, Davidson SB, Kosky A (1992) Theoretical Aspects of Schema Merging. *Proc. 3rd Int. Conf. on Extending Database Technology*. Vienna, Austria, Mar. 1992. pp 152-167

Bunke H, Haller B (1989) A Parser for Context Free Plex Grammars. *Proc. 15th Int. Workshop on Graph-Theoretic Concepts in Computer Science*. LNCS 411, pp 136-150

Burnett MM (1994) Seven Programming Language Issues. In: Burnett MM, Goldberg A, Lewis T (Eds) *Visual Object-Oriented Programming*. Manning Publishing Co., pp 161-181

Burnett MM (1999) Visual Programming. In: Webster JG (Ed) *Encyclopedia of Eletrical and Eletronics Engineering*. John Wiley & Sons Inc., New York

Burnett MM (2006) Visual Language Research Bibliography. http://www.cs.orst.edu/~burnett/vpl.html

Burnett MM, Goldberg A, Lewis T (1995) (Eds) *Visual Object-Oriented Programming: Concepts and Environments*. Prentice-Hall/Manning

Burnett MM, Gottfried (1998) Graphical Definitions: Expanding Spreadsheet Languages Through Direct Manipulation and Gestures. *ACM Transactions on Computer-Human Interaction* 5:1-33

Cao J, Fernando F, Zhang K (1995) Dig: A Graph-Based Construct for Programming Distributed Systems. *Proc. 2nd Int. Conf. on High Performance Computing*. New Delhi, pp 417-422

Casanova M, Tucherman L, Lima M, Rodriguez N, Soares L (1991) The Nested Context Model for Hyperdocuments. *Proc. Hypertext*. San Antonio, USA, pp 193-201

Castano S, Antonellis VD (1999) A Schema Analysis and Reconciliation Tool Environment for Heterogeneous Databases. *Proc. Int. Database Engineering and Appplication Symposium*. pp 53-62

Ceri S, Comai S, Damiani E, Fraternali P, Paraboschi S, Tancia L (1999) XML-GL: A Graphical Language for Querying and Representing XML Documents. *Proc. 8th Int. World Wide Web Conf.*.

Ceri S, Fraternali P, Bongio A (2000) Web Modeling Language (WebML): A Modeling Language for Designing Web Sites. *Proc. 9th Int. World Wide Web Conf.*. Amsterdam, Netherlands

Chan F, Cao J, Chan ATS, Zhang K (2005) Visual Programming Support for Graph-Oriented Parallel/Distributed Processing, *Software – Practice and Experience* 35:1409-1439

Chang SK (1971) Picture Processing Grammar and Its Applications. *Information Sciences* 3:121-148

Chang SK (1987) Visual Languages: A Tutorial and Survey. *IEEE Software* 4:29-39

Chang SK (2000) *Multimedia Software Engineering*. Kluwer Academic Publishers

Chang SK, Burnett MM, Levialdi S, Marriott K, Pfeiffer JJ, Tanimoto SL (1999) The Future of Visual Languages. *Proc. 15th IEEE Symposium on Visual Languages*. Tokyo, Japan, pp 58-61

Chang SK, Ichikawa T, Ligomenides PA (1986) (Eds) *Visual Languages*. Plenum, New York

Chang SK, Tauber MJ, Yu B, Yu JS (1989) A Visual Language Compiler. *IEEE Trans. on Software Engineering* 15:506-525

Chang SK, Tortora G, Yu B, Guercio A (1987) Icon Purity – Towards a Formal Definition of Icons. *Int. Journal of Pattern Recognition and Artificial Intelligence* 1:377-392

Chen L, Jamil HM (2003) On Using Remote User Defined Functions as Wrappers for Biological Database Interoperability. *Int. J. Cooperative Info. Systems* 12: 161-195

Chok S, Marriott K (1995) Parsing Visual Languages. *Proc. 18th Australasian Computer Science Conference*. Glenolg, South Australia, pp 90-98

Clementini E, Felice PD, Oosterom PV (1993) A Small Set of Formal Topological Relationships for End-User Interaction. *Proc. 3rd Int. Symp. on Advances in Spatial Databases*. pp 277-295

Cluet S, Delobel C, Simeon J, Smaga K (1998) Your Mediators Need Data Conversion! *ACM SIGMOD*. pp 177-188

Coda F, Ghezzi C, Vigna G, Garzotto F (1998) Towards a Software Engineering Approach to Web Site Development. *Proc. 9th Int. Workshop on Software Specification and Design*. IEEE Press

Conger SA, R. O. Mason RO (1998) Planning and Designing Effective Web Sites. *Course Technology*. Cambridge, MA

Corradini A, Montanari U, Rossi F (1996) Graph Processes. *Fundamenta Informaticae* 26:241-265

Costagliola G, Deufemia V, Risi M (2006) A Multi-layer Parsing Strategy for On-line Recognition of Hand-drawn Diagrams. *Proc. 2006 IEEE Symposium on Visual Languages and Human-Centric Coomputing.* Brighton, UK, pp 103-110

Costaglioga G, Orefice S, Polese G, Tortora G, Tucci M (1993) Automatic parser generation for pictorial languages. *Proc. 1993 IEEE Symposium on Visual Languages*. Bergen, Norway

Costagliola G, Tortora G, Orefice S, Lucia AD (1995) Automatic Generation of Visual Programming Environments. *IEEE Computer* 28: 56-66

Cox P, Pietrzykowski T (1985) Advanced Programming Aids in PROGRAPH. *Proc. 1985 ACM SIGSMALL Symposium on Small Systems.* Danvers, USA, pp 27-33

Cruz IF, Averbuch M, Lucas WT, Radzyminski M, Zhang K (1997) Delaunay: A Database Visualization System. *Proc. ACM SIGMOD Int. Conf. on Management of Data.* pp 510-513

Cruz IF, Lucas WT (1997) A Visual Approach to Multimedia Querying and Presentation. *Proc. ACM Multimedia'97.* Seattle, USA, 8-14 Nov. pp 8-14

Davidson A, Fuchs M, Hedin M, Jain M, Koistinen J, Lloyd C, Maloney M, Schwarzhof K (1999) Schema for Object-Oriented XML 2.0. *W3C Document*

Dean TR, Cordy JR (1995) A Syntactic Theory of Software Architecture. *IEEE Transactions on Software Engineering* 21:302-313

Di Battista G, Eades P, Tamassia R, Tollis IG (1999) *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice Hall, Englewood Cliffs, NJ

Doemel P (1994) WebMap – A Graphical Hypertext Navigation Tool. *Proc. 2nd Int. World Wide Web Conf.*. USA, pp 785-789

Dong J, Zhang K (2003) Design Pattern Compositions in UML. In: Zhang K (Ed) *Software Visualization – from Theory to Practice.* Kluwer Academic Publishers, pp 287- 208

Drewes F, Hoffmann B, Janssens D, Minas M, Van Eetvenlde N (2006) Adaptive Star Grammars. *Proc. Int. Conf. on Graph Transformation (ICGT'06).* Natal, Brazil

Eades P (1984) A Heuristic for Graph Drawing, *Congressus Numerantium* 42:149-160

Eades P, Zhang K (1996) *Software Visualization.* World Scientific Publishing Co., Singapore

Ehrig H, Heckel R, Korff M, Löwe M, Ribeiro L, Wagner A, Corradini A (1997) Algebraic Approach to Graph Transformation II: Single Pushout Approach and Comparison with Double Pushout Approach. In:

Rozenberg G (1997) (Ed) *Handbook on Graph Grammars and Computing by Graph Transformation: Foundations*. Vol.1, World Scientific, pp 247-312

Engelfriet J, Rozenberg G (1997) Node Replacement Graph Grammars. In: Rozenberg G (Ed.) *Handbook on Graph Grammars and Computing by Graph Transformation: Foundations*. Vol.1, World Scientific, 1-94

Engels G, Heckel R, Küster JM, Groenewegen L (2002) Consistency-Preserving Model Evolution Through Transformations. *Proc. UML'02*. LNCS 2460, Springer, pp 212-227

Ermel C, Rudolf M, Taentzer G (1999) The AGG Approach: Language and Environment. In: Ehrig H, Engels G, Kreowski HJ, Rozenburg G (Eds) *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools*. Vol.2, pp 551-604

Erwig M (2000) A Visual Language for XML. *Proc. 16th IEEE Symposium on Visual Languages*. Seattle, USA, 10-13 September, pp 47-54

Feiner S, McKeown K (1993) Automating the Generation of Coordinated Multimedia Explanations. In: MayBury M (Ed) *Intelligent Multimedia Interface*. AAAI Press/MIT Press, Cambridge, MA, pp 117-138

Ferrucci F, Pacini G, Satta G, Sessa M, Tortora G, Tucci M, Vitellio G (1996) Symbol-Relation Grammars: A Formalism for Graphical Languages. *Information and Computation* 131:1-46

Ferrucci F, Tortora G, Tucci M, Vitellio G (1994) A predictive parser for visual languages specified by relational grammars. *Proc. 10th IEEE Symposium on Visual Languages*. St. Louis, USA, pp 245-252

Ferrucci F, Tortora G, Tucci M, Vitellio G (2001) A System for Rapid Prototyping of Viaual Language Environments. *Proc. of IEEE Symposia on Human-Centric Computing*. Stresa, Italy, Sep. 5-7, pp 382-389

Fischer T, Niere J, Torunski L, Zündorf A (1998) Story Diagrams: A New Graph Rewrite Language Based on the Unifed Modeling Language and Java. *Proc. Theory and Application to Graph Transformations*. LNCS 1764, pp 296-309

Frank AU (1996) Qualitative Spatial Reasoning: Cardinal Directions as an Example. *Int. J. Geographical Information Science* 10:269-290

Frecon E, Smith G (1998) WebPATH – A Three Dimensional Web History. *IEEE Symp. Information Visualization*. N. Carolina

Freire J, Kumar B, Lieuwen D (2001) WebViews: Accessing Personalized Web Content and Services. *Proc. 10th Int. World Wide Web Conf.*. Hong Kong, China

Fujaba (2005) http://www.fujaba.de

Furnas G (1986) Generalized Fisheye Views. *Proc. CHI'86*, Boston

Gamma E, Helm R, Johnson R, Vlissides J (1995) *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley

Garlan D, Allen R, Ockerbloom J (1994) Exploiting Styles in Architectural Design Environments. *Proc. 2nd ACM SIGSOFT Symposium on Foundations of software engineering*. pp 175-188

Garlan D, Cheng SW, Kompanek AJ (2002) Reconciling the Needs of Architectural Description With Object-modeling Notations. *Science of Computer Programming* 44:23-49

Glinert EP (1990a) *Visual Programming Environments: Paradigms and Systems*. IEEE CS Press

Glinert EP (1990b) *Visual Programming Environments: Applications and Issues*. IEEE CS Press

Goldberg A, Burnett MM, Lewis T (1994) What Is Visual Object-Oriented Programming? In: Burnett MM, Goldberg A, Lewis T (Eds) *Visual Object-Oriented Programming*. Manning Publications Co.

Golin EJ (1991) A Method for the Specification and Parsing of Visual Languages. *Ph.D. Thesis*, Brown University

Golin EJ, Magliery T (1993) A Compiler Generator for Visual Languages. *Proc. 9th IEEE Symposium on Visual Languages*. Bergen, Norway, August 1993, IEEE CS Press, pp.314-323.

Gómez J, Cachero C, Pastor O (2001) Conceptual Modeling of Device-Independent Web Applications. *IEEE Multimedia*, pp 26-39

Graham S (1987) Programming Languages and Systems – Introduction to Part I. *ACM Turing Award Lectures – The First Twenty Years 1966-1985*. ACM Press, pp 1

Grimshaw A, Weissman J, West E, Loyot E (1994) Metasystems: An Approach Combining Parallel Processing and Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed Computing* 21:257-270

Guan SU, Yu H-Y, Yang J-S (1998) A Prioritized Petri Net Model and its Application in Distributed Multimedia Systems. *IEEE Transactions on Computers* 47:477-481

Habel A, Heckel R, Taentzer G (1996) Graph Grammars with Negative Application Conditions. *Fundamenta Informaticsae* 26:287-313

Halevy A, Ives Z, Suciu D, Tatarinov I (2003a) Schema Mediation in Peer Data Management Systems. *Proc. 2003 Int. Conf. Data Engineering*. pp 505-518

Halevy A, Ives Z, Tatarinov I, Mork P (2003b) Piazza: Data Management Infrastructure for Semantic-Web Applications, *Proc. 2003 Int. World Wide Web Conf.*. Budapest, Hungary, pp 556-567

Herman I, Melancon G, Marshall MS (2000) Graph Visualization and Navigation in Information Visualization. *IEEE Transactions on Visualization and Computer Graphics* 6:24-43

Hernández D, Clementini E, Felice PD (1995) Qualitative Distance. *Proc. Spatial Information Theory: a Theoretical Basis for GIS*. pp 45-58

Hinton S (1998) From Home Page to Home Site: Effective Web Resource Discovery at the ANU. *Proc. 7th Int'l World Wide Web Conf.*. Brisbane, Australia

Hirakawa M, Nishimura Y, Kado M, and Ichikawa T (1991) Interpretation of icon overlapping in iconic programming, *Proc. 7th IEEE Workshop on Visual Languages*. Kobe, Japan, pp 254-259

Huang ML, Eades P, Cohen RF (1998a) WebOFDAV - Navigating and Visualizing the Web On-line with Animated Context Swapping. *Proc. 7th Int. World Wide Web Conf.*. Brisbane, Australia

Huang ML, Eades P, Wang J (1998b) On-line Animated Visualization of Huge Graphs Using a Modified Spring Algorithm. *Journal of Visual Languages and Computing* 9:623-645

Inverardi P, Wolf AL (1995) Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering* 21:373-386

Ishizaki S (2003) *Improvisational Design – Continuous Responsive Design Communication*. MIT Press

Jahnke JH, Zudorf A (1998) Using Graph Grammars for Building the Varlet Database Reverse Engineering Environment. Technical Report tr-ri-98-201, University of Paderborn

Jonyer I (2003) Context-Free Graph Grammar Induction Based on the Minimum Description Length Principle. *Ph.D. Thesis*. The University of Texas at Arlington

Karsai G, Sztipanovits J, Ledeczi A, Bapty T (2003) Model-Integrated Development of Embedded Software. *Proceedings of the IEEE* 91:145-164

Kaul M (1982) Parsing of Graphs in Linear Time. *Proc. 2nd Int. Workshop on Graph Grammars and Their Application to Computer Science*. LNCS 153, pp 206-218

Keller RK, Schauer R (1998) Design Components: Towards Software Composition at the Design Level. *Proc. 20th Int. Conf. on Software Engineering*. pp 302-311

Klapsing R, Neumann G, Conen W (2001) Semantics in Web Engineering: Applying the Resource Description Framework. *IEEE Multimedia*, pp 62-68

Kleyn M (1995) A High Level Language for Specifying Graph-Based Languages and Their Programming Environments. *Ph.D. Thesis*, The University of Texas at Austin

Kong J (2005) Visual Programming Languages and Applications. *Ph.D.*

Thesis. Department of Computer Science, The University of Texas at Dallas

Kong J, Zhang K (2004a) Parsing Spatial Graph Grammars. *Proc. 2004 IEEE Symposium on Visual Languages and Human-Centric Computing*. Rome, Italy, pp 99-101

Kong J, Zhang K (2004b) On a Spatial Graph Grammar Formalism. *Proc. 2004 IEEE Symposium on Visual Languages and Human-Centric Computing*. Rome, Italy, pp 102-104

Kong J, Zhang K, Dong J, Song GL (2003) A Graph Grammar Approach to Software Architecture Verification and Transformation. *Proc. 27th Annual Int. Computer Software and Applications Conf. (COMPSAC'03)*. Dallas, USA, pp 492-499

Kong J, Zhang K, Dong J, Song GL (2005) A Generative Style-driven Framework for Software Architecture Design. *Proc. 29th Annual IEEE/NASA Software Engineering Workshop (SEW-29)*. Greenbelt, MD, USA, pp 173-182

Kong J, Zhang K, Zeng X (2006) Spatial Graph Grammars for Graphical User Interfaces. *ACM Transactions on Computer-Human Interaction* 13:268-307

Krasner GE, Pope ST (1988) A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming* 1:26-49

Kuikka E, Leinonen P, Penttonen M (2002) Towards Automating of Document Structure Transformations. *Proc. 2002 ACM Symposium on Document Engineering*. McLean, USA, pp 103-110

Kukluk J, Holder L, Cook D (2006) Inference of Node Replacement Recursive Graph Grammars. *Proc. 6th SIAM International Conference on Data Mining*. Washington DC, USA, pp 544-548

Kuske S (2001) A Formal Semantics of UML State Machines Based on Structured Graph Transformation. *Proc. UML 2001*. pp 241-256

Lassila O, R. R. Swick RR (1999) Resource Description Framework (RDF) Model and Syntax Specification. *W3C Recommendation*. February 1999

Lee HS, Schor MI (1992) Match Glgorithms for Generalized Rete Networks. *Artificial Intelligence* 54:249-274

Leinonen P (2003) Automating XML Document Structure Transformations. *Proc. 2003 ACM Symposium on Document Engineering*. Grenoble, France, pp 26-28

Liu B, Zhao K, Yi L (2002) Visualizing Web Site Comparison. *Proc. 6th Int. World Wide Web Conf.*. Honolulu, pp 693-703

Luckham D, Kenney J, Augustin L, Vena J, Bryan D, Mann W (1995) Specification and Analysis of System Architecture Using Rapide.

IEEE Transactions on Software Engineering 21:336-355

McWhirter JD, Nutt GJ (1994) Escalante: An Environment for the Rapid Construction of Visual Language Applications. *Proc. 10th IEEE Symposium on Visual Languages*. St. Louis, Missouri, pp 15-22

Maarek YS, Shaul IZB (1997) WebCutter: A System for Dynamic and Tailorable Site Mapping, *Proc. 6th Int. World Wide Web Conf.*. pp 713-722

Madhavan J, Bernstein PA, Rahm E (2001) Generic Schema Matching Using Cupid. *Proc. 27th VLDB Conf.*. Roma, Italy, pp 49-58

Madhavan J, Halevy AY (2003) Composing Mappings Among Data Sources. *Proc. 29th VLDB Conf.*. Berlin, German, pp 572-583

Maier, Zündorf A (2003) The Fijaba Statechart Synthesis Approach. *Proc. 2nd Int. Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*

Makoto M (2000) RELAX (REgular LAnguage Description for XML). http://www.xml.gr.jp/relax/

Mankoff J, Abowd GD, Hudson SE (2000) OOPS: A Toolkit Supporting Mediation Techniques for Resolving Ambiguity in Recognition-Based Interfaces. *Computers and Graphics* 24:819-834

Marriott K (1994) Constraint Multiset Grammars. *Proc. 10th IEEE Symposium on Visual Languages*. St. Louis, USA, pp 118-125

Marriott K, Meyer B (1997) On the Classification of Visual Languages by Grammar Hierarchies. *Journal of Visual Languages and Computing* 8:375-402

Marriott K, Meyer B (1998) (Eds) *Theory of Visual Languages*. Springer-Verlag

Marriott K, Meyer B, Tardif L (2002) Fast and Efficient Client-Side Adaptability for SVG. *Proc. 11th World Wide Web Conf.*. Hawaii, USA, pp 496-507

Martinez A, Patiño-Martinez A, Jiménez-Peris R, Pérez-Sorrosal F (2005) ZenFlow: A Visual Web Service Composition Tool for BPEL4WS. *Proc. 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*. Dallas, USA, Sep. 20-24, pp 181-188

Medvidovic N, Taylor RN (2000) A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering* 26:70-93

Medvidovic N, Rosenblum DS, Redmiles DF, Robbins JE (2002) Modeling Software Architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology* 11:2-57

Mehta NR, Medvidovic N (2003) Composing Architectural Styles From Architectural Primitives. *Proc. ESEC/FSE'03*. pp 347-350

Melnik S, Rahm E, Bernstein PA (2003) Rondo: A Programming Platform for Generic Model Management. *Proc. SIGMOD 2003 Conf.*. San Dieago, CA, pp 193-204

Métayer DL (1998) Describing Software Architecture Styles Using Graph Grammars. *IEEE Transactions on Software Engineering* 24:521-533

Miller RJ, Hernandez MA, Haas LM, Yan L, Ho C, Fagin R, Popa L (2001) The Clio Project: Managing Heterogeneity. *SIGMOD Record* 30:78-83

Milo L, Zohar S (1998) Using Schema Matching to Simplify Heterogeneous Data Translation. *Proc. VLDB'98*. New York City, USA, pp 122-133

Minas M (1998) Hypergraphs as a Uniform Diagram Representation Model. *Proc. Theory and Applications of Graph Transformations*. Paderborn, Germany

Minas M (2002) Concepts and Realization of a Diagram Editor Generator Based on Hypergraph Transformation. *Science of Computer Programming* 44:157-180

Minas M (2006) Parsing Star Grammars. *Proc. VL/HCC'06 Workshop on Graph and Model Transformation*. Brighton, UK

Minas M, Shklar L (1996) Visual Definition of Virtual Documents for the World-Wide Web. *Proc. 3rd Int. Workshop on Document Processing*. LNCS 1293, pp 183-195

Minas M, Viehstaedt G (1993) Specification of Diagram Editors Providing Layout Adjustment with Minimal Change. *Proc. 1993 IEEE Symposium on Visual Languages*. pp 324-329

Minas M, Viehstaedt G (1995) DiaGen: A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams. *Proc. 11th IEEE Symposium on Visual Languages*. Darmstadt, Germany, pp 203-210

Misue K, Eades P, Lai W, Sugiyama K (1995) Layout Adjustment and the Mental Map. *Journal of Visual Languages and Computing* 6:183-210

Moriconi M, Qian XL, Riemenschneider RA (1995) Correct Architecture Refinement. *IEEE Transactions on Software Engineering* 21:356-372

Muchaluat DC, Rodrigues RF, Soares LFG (1998) WWW Fisheye-View Graphical Browser. *Proc. IEEE Multimedia Modeling*

Myers BA (1990) Taxonomies of Visual Programming and Program Visualisation. *Journal of Visual Languages and Computing* 1:97-123

Myers BA (1998) Authoring Interactive Behaviors for Multimedia. *Proc. 9th NEC Research Symposium*. Nara, Japan

Mylopoulos J, Borgida A, Jarke M, Koubarakis M (1990) Telos: Representing Knowledge About Information Systems. *ACM Transactions on Information Systems* 8:325-362

Nakayama T, Kato H, Yamano Y (2000) Discovering the Gap Between Web Site Designers' Expectations and Users' Behavior. *Proc. 9th Int.*

World Wide Web Conf.. Amsterdam, Netherlands

Pietriga E, Vion-Dury J-Y, Quint Y (2001) VXT: A Visual Approach to XML Transformations. *ACM Symposium on Document Engineering*. Atlanta, USA, pp 1-10

Pietriga E (2005) A Toolkit for Addressing HIC Issues in Visual Language Environments. *Proc. 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*. Dallas, USA, Sep. 20-24, pp 145-152

Pimental M, Abowd G, Ishiguro Y (2000) Linking by Interacting: A Paradigm for Authoring Hypertext and Hypermedia. *Proc. Hypertext 2000*. Austin, USA, pp 39-48

Plimmer B, Grundy J, Hosking J, Priest R (2006) Inking in the IDE: Experuences with Pen-based Design and Annotation. *Proc. 2006 IEEE Symposium on Visual Languages and Human-Centric Coomputing*. Brighton, UK, pp 111-115

Plump D (1993) Hypergraph Rewriting: Critical Pairs and Undecidability of Confluence. In: Sleep MR, Plasmeijer MJ, van Eekelen M (Eds) *Term Graph Rewriting*. pp 201-214

Pong MC, Ng N (1983) PICS – A System for Programming with Interactive Graphical Support. *Software – Practice and Experience* 13:847-855

Pottinger RA, Bernstein PA (2003) Merging Models Based on Given Correspondences. *Proc. 29th VLDB Conf.*, Berlin, Germany, pp 826-873

Prabhakaran B (2000) Multimedia Authoring and Presentation Techniques, Guest-Editor's Introduction. *Multimedia Systems* 8:157

Radermacher A (1999) Support for Design Patterns through Graph Transformation Tools. *Proc. Application of Graph Transformations with Industrial Relevance*. LNCS 1779, pp 111-126

Rahm E, Bernstein PA (2001) A Survey of Approaches to Automatic Schema Matching. *The VLDB Journal* 10:334-350

Rekers J, Schürr A (1995) A Graph Grammar approach to Graph Parsing. *Proc. 1995 IEEE Symposium on Visual Languages*. pp 195-202

Rekers J, Schürr A (1996) A Graph Based Framework for the Implementation of Visual Environments. *Proc. 12th IEEE Symposium on Visual Languages*. Boulder, Colorado

Rekers J, Schürr A (1997) Defining and Parsing Visual Languages with Layered Graph Grammars. *Journal of Visual Languages and Computing* 8:27-55

Revesz GE (1983) *Introduction to Formal Languages*. McGraw-Hill Computer Science Series

Rosenfeld G (1976) Array and Web Grammars: An Overview. In: Lindenmayer A, Rozenberg G (Eds) *Automata, Languages, and*

Development. North-Holland

Rozenberg G (1997) (Ed) *Handbook on Graph Grammars and Computing by Graph Transformation: Foundations*. Vol.1, World Scientific

Rozenberg G, Welzl E (1986) Boundary NLC Graph Grammars – Basic Definitions, Normal Forms, and Complexity. *Information and Control* 69:136-167

Salis P, Tate G, MacDonell S (1995) *Software Engineering*. Addison-Wesley Publish Company

Sarkar M, Brown MH (1994) Graphical Fisheye Views. *Communications of the ACM* 37:73-84

Schefstrom D, van den Broek G (1993) *Tool Integration-Environments and Frameworks*, John Wiley, Chichester, England

Schürr A (1991) PROGRES, A VHL-Language Based on Graph Grammars. *Proc. 4th Int. Workshop on Graph Grammars and Their Application to Computer Science*. LNCS 532, pp 641-659

Schürr A (1994) Specification of Graph Translators with Triple Graph Grammars. *Proc. Int. Workshop on Graph-Theoretic Concepts in Computer Science*. Herrsching, Germany

Schürr A, Winter A, Zündorf A (1999) The PROGRES Approach: Language and Environment. In: Rozenberg G (Ed) *Handbook on Graph Grammars and Computing by Graph Transformation: Applications*. Vol.2, pp 487-550

Schürr A, Zundorf A, Winter A (1995) Visual Programming with Graph Rewriting Systems. *Proc. 11th IEEE Symposium on Visual Languages*. Darmstadt, Germany, pp 326-333

Selonen P, Xu J (2003) Validating UML Models Against Architectural Profiles. *Proc. ESEC/FSE'03*. pp 58-67

Shatz S (1993) *Development of Distributed Software*. Macmillan Publishing Company, New York

Shaw M, Garlan D (1995) Software Architecture: Perspectives on an Emerging Discpline. Prentice Hall

Shaw M, DeLine R, Klein DV, Ross TL, Young DM, Zelesnik G (1995) Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering* 21:314-335

Sheth A, J. Larson J (1990) Federated Database Systems. *ACM Computing Surveys* 22: 183 – 236

Shu NC (1986) A Perspective and A Dimensional Analysis. In: Chang SK, Ichikawa T, Ligomenides PA (Eds) *Visual Languages*. Plenum, New York

Shui WM, Wong RK (2003) Application of XML Schema and Active Rules System in Management and Integration of Heterogeneous Biological Data. *Proc. 3rd IEEE Symposium on Bioinformatics and*

Bioengineering, pp 367-374

Siegel D (1997) *Creating Killer Web Sites: The Art of Third Generation Site Design*. 2nd Ed., Hayden Books

Six JM, Kakoulis KG, Tollis IG (2000) Techniques for the Refinement of Orthogonal Graph Drawings. *Journal of Graph Algorithms and Applications* 4:75-103

Smith DC (1975) Pygmalion: A Computer Program to Model and Simulate Creative Thought. *Ph.D. Thesis*, Stanford University

Smith DC, Cypher A, Spohrer J (1994) KidSim: Programming Agents without a Programming Langauge. *Communications of the ACM* 37:54-67

Soares LFG, Rodrigues RF, Saade DCM (2000) Modeling, Authoring and Formatting Hypermedia Documents in the HyperProp System. *Multimedia Systems* 8:118-134

Song GL (2006) A Graphical Framework for Model Management. *Ph.D. Thesis*. Department of Computer Science, The University of Texas at Dallas

Song GL, Zhang K, Wong RK, Kong J (2004a) Management of Web Data Models Based on Graph Transformation. *Proc. 2004 IEEE/WIC/ACM Int. Conf. on Web Intelligence*, Beijing, China, pp 398-404

Song GL, Zhang K, Kong J (2004b) Model Management Through Graph Transformations. *Proc. 2004 IEEE Symposium on Visual Languages and Human-Centric Computing*, Rome, Italy, pp 75-82

Stasko JT, Domingue JB, Brown MH, Price BA (1998) *Software Visualization – Programming as a Multimedia Experience*. MIT Press, Boston

Su H, Kuno H, Rundensteiner EA (2001) Automating the Transformation of XML Documents. *Proc. 3rd Int. Workshop on Web Information and Data Management*, Atlanta, USA, pp 68-75

Sutherland IE (1963) Sketchpad: A Man-machine Graphical Communications System. *Ph.D. Thesis*. MIT

Taentzer G, Geodicke M, Meyer T (1998) Dynamic Change Management by Distributed Graph Transformation: Towards Configurable Distributed Systems. *Proc. 6th Int. Workshop Theory and Application of Graph Transformations*. LNCS 1764, pp 179-193

Thompson HS, Beech D, Maloney M, Mendel-Sohn N (2000) (Eds) XML Schema Part 1: Structures. *W3C Document*

Torlone R, Atzeni P (2001) A Unified Framework for Data Translation Over the Web. *Proc. WISE'01*, Kyoto, Japan

Vadaparty K, Aslandogan YA, Ozsoyoglu G (1993) Towards a unified visual database access. *Proc. ACM SIGMOD Conf. on Management of Data*, pp 357-366 van Wijk JJ (2006) Views on Visualization. *IEEE Transactions on Visualization and Computer Graphics* 12:421-432

Vazirgiannis M, Kostalas I, Sellis T (1999) Specifying and Authoring Multimedia Scenarios. *IEEE Multimedia*. July-September:24-37

Vazirgiannis M, Theodoridis Y, Sellis T (1998) Spatio-Temporal Composition and Indexing for Large Multimedia Applications. *Multimedia Systems* 6:284-298

Vion-Dury J-Y, Lux V, Pietriga E (2002) Experimenting with the Circus Language for XML Modeling and Transformation. *Proc. 2002 ACM Symposium on Document Engineering*, McLean, USA, pp 82-87

Wadge WW, Yildirim T (1997) Intensional HTML. *Proc. 10th Int. Symposium on Languages for Intensional Programming*, Victoria, Canada, pp 34-40

Weitzman L, Wittenburg K (1994) Automatic Presentation of Multimedia Documents Using Relational Grammars. *Proc. 1994 ACM Int. Conf. on Multimedia*. San Francisco, USA, pp 443-451

Weitzman L, Wittenburg K (1998) Grammar-Based Articulation for Multimedia Document Design. Maybury MT, Wahlster W (Eds) *Readings in Intelligent User Interfaces*. Morgan Kauffmann, San Francisco, pp 310-327

Wermelinger M (1998) Towards a Chemical Model for Software Architecture Reconfiguration. *Proc. 4th Int. Conf. on Configurable Distributed Systems*. pp 111-118

Wermelinger M, Fiadeiro JL (2002) A Graph Transformation Approach to Software Architecture Reconfiguration. *Science of Computer Programming* 44:133-155

Wilde N, Lewis C (1990) Spreadsheet-based Interactive Graphics: From Prototype to Tool. *ACM SIGGHI Special Issue, Proc. CHI'90*, pp 153-159

Wills LM (1992) Automated Program Recognition by Graph Parsing. *Ph.D. Thesis*, MIT AI Lab

Wittenburg K (1992) Earley-Style Parsing for Relational Grammars. *Proc. 8th IEEE Workshop on Visual Languages*. Seattle, Washington, pp 192-199

Wittenburg K, Weitzman L (1996) Relational Grammars: Theory and Practice in a Visual Language Interface for Process Modeling. *Proc. AVI'96*, Gubbio, Italy

Wittenburg K, Weitzman L, Talley J (1991) Unification-based grammars and tabular parsing for graphical languages. *Journal of Visual Languages and Computing* 2:347-370

W3C (2001) Synchronized Multimedia Integration Language (SMIL 2.0). http://www.w3.org/TR/2001/REC-smil20-20010807/

W3C (2004a) Extensible Markup Language (XML) 1.0 (3rd Edition). http://www.w3.org/TR/REC-xml.html

W3C (2004b) Cascading Style Sheets (CSS). http://www.w3.org/Style/CSS/

W3C (1999) XSL Transformation (XSLT). http://www.w3.org/TR/xslt

Yan L, Miller RJ, Haas LM, Fagin R (2001) Data-Driven Understanding and Refinement of Schema Mappings. *Proc. ACM SIGMOD'2001*, pp 485-496

Zhang DQ (1997) Generation of Visual Languages. *Ph.D. Thesis*, Department of Computing, Macquarie University, Sydney, Australia

Zhang DQ, Zhang K (1997) Reserved Graph Grammar: A Specification Tool for Diagrammatic VPLs. *Proc. 13th IEEE Symposium on Visual Languages*. Capri, Italy, pp 284-291

Zhang DQ, Zhang K (1998a) On a Visual Distributed Programming Environment and Its Construction by a Meta Toolset. *Proc. 10th Int. Conf. on Software Engineering and Knowledge Engineering*. San Francisco, USA, pp 347-350

Zhang DQ, Zhang K (1998b) VisPro: A Visual Language Generation Toolset. *Proc. 14th IEEE Symposium on Visual Languages*. Halifax, Canada, pp 195-202

Zhang DQ, Zhang K, Cao J (2001a) A Context-sensitive Graph Grammar Formalism for the Specification of Visual Languages. *The Computer J* 44:186-200

Zhang K (2003) (Ed), *Software Visualization – From Theory to Practice*. Kluwer Academic Publishers, Boston

Zhang K, Kong J, Qiu MK, Song GL (2005a) Multimedia Layout Adaptation Through Grammatical Specifications. *ACM/Springer Multimedia Systems* 10:245-260

Zhang K, Song GL, Kong J (2005b) Interoperating XML-Style of Digital Artifacts for Information Reuse. *Proc. 2005 IEEE Int. Conf. on Information Reuse and Integration (IRI'05)*. Las Vegas, USA, pp 126-131

Zhang K, Huang ML, Li KC (2002) An Integrated Visual Framework for Human-Web Interface. *Proc. 4th IEEE Int. Workshop on Advanced Issues of E-Commerce and Web-based Information Systems (WECWIS'02)*. Newport Beach, California, USA, pp 195-202

Zhang K, Ma X, Hintz T (1999) The Role of Graphics in Parallel Program Development. *Journal of Visual Languages and Computing* 10:215-243

Zhang K, Zhang DQ, Cao J (2001b) Design, Construction and Application of a Generic Visual Language Generation Environment. *IEEE Transactions on Software Engineering* 27:289-307

Zhang K, Zhang DQ, Deng Y (2001c) Graphical Transformation of Multimedia XML Documents. *Annals of Software Engineering* 12:119-137

Zhang KB, Zhang K (1999) An Incremental Approach to Graph Layout Based on Grid Drawing. *Proc. 3rd Workshop on Software Visualization,* University of Technology, Sydney

Zhang KB, Zhang K, Orgun MA (2002) Grammar-Based Layout for A Visual Programming Language Generation System. *Proc. 2nd Int. Conf. on the Theory and Application of Diagrams.* Georgia, USA, LNCS 2317, Springer, pp 106-108

Zhang Y, Zhang K (2000) Associative Query for Multi-version Web Documents. In: Gergatsoulis M, Rondogiannis P (Eds) *Intensional Programming II.* World Scientific, pp 55-64

Appendix RGG+: An Generalization to the RGG

A.1 Introduction

Like textual programming languages that are usually equipped with proper formal syntax definitions and parsers, graphical or visual languages need the support of such mechanisms. Formal definitions of graph grammars offer the following advantages (Rekers and Schürr 1995):

- Without a proper syntax definition, new users can only guess the syntax of a graphical language by generalizing from the provided examples,
- A definition could serve as an unambiguous specification for syntax-directed editors over the language,
- A graphical parser could be generated out of a proper definition, and
- A syntax definition is a necessary precondition for a definition of the semantics of the language.

Among a large variety of visual languages only a few are equipped with proper formal syntax definitions. This is mostly due to the fact that the extension from one-dimensional textual languages to two-dimensional graphical languages raises new issues. The existing graph grammar formalisms, though proven to be useful in many practical applications, need to be improved in many aspects, such as their expressive power and easiness to be developed and implemented.

As reviewed in Chapter 2, a number of grammar formalisms and their parsing algorithms have been proposed, most of which are used to define context-free grammars with limited expressive power. The best known context-sensitive graph grammar formalism is the layered graph grammar (Rekers and Schürr 1997), which is relatively expressive but requires exponential parsing time. The RGG improves the LGG in its expressiveness as well as the parsing efficiency. With a spatial extension to the RGG, the spatial graph grammar (SGG) discussed in Chapter 3 improves the parsing efficiency further due to the additional spatial constraints. Although both RGGs and SGGs had successfully solved many graph related problems in applications described in Chapters 4 through 7, the confluence condition restricts their applications.

This appendix discusses an attempt to generalize the RGG formalism in two directions: ease the development of graph grammars, and enhance the expressiveness of the grammar formalism by removing the confluence restriction in the RGG. As expected, the parsing efficiency will have to be sacrificed. We will call this generalized version *RGG+*.

The contribution of the RGG+ is twofold. One is to replace the multiple layers of labels with the usual two layers, i.e., terminal and non-terminal labels, and introduce a size-increasing condition to graph grammar productions to solve the membership problem. The size-increasing condition only imposes some weak restrictions on the structure of productions, and is more intuitive and easier to handle than that of the layer decomposition mechanism. The other is to give a more general parsing algorithm that has no requirement that graph grammars must be confluent. This greatly enhances the parsing capability.

The rest of this appendix is organized as follows. Section 2 defines the notations used in the subsequent sections. Section 3 formally defines the RGG+ and its languages, and proves the decidability of RGG+s. A parsing algorithm for RGG+s is described in Section 4 along with an analysis of its time and space complexities. Possible approaches to improving the efficiency of the parsing algorithm are discussed in Section 5. Finally, Section 6 summarizes this appendix.

A.2 Notations

The basic concepts and notations in this appendix are consistent with the RGG definition in Chapter 2, and are summarized below for expression clarity and simplicity.

Φ : An empty set.

Ω : A finite set of labels, consisting of two disjoint subsets, terminal label set Ω^T and non-terminal label set Ω^{NT}, i.e., $\Omega = \Omega^T \cup \Omega^{NT}$ and $\Omega^T \cap \Omega^{NT} = \Phi$.

$\lvert\,\rvert$:	Cardinality of a set.
N:	A node set consisting of a terminal node set N^T and a non-terminal node set N^{NT}, i.e., $N = N^T \cup N^{NT}$ with $N^T \cap N^{NT} = \Phi$.
f:	A labeling function establishing a mapping $N \to \Omega$.
$p := (L, R)$:	A production consisting of a left L and a right R graphs over the same label set Ω.
$R(H, G)$:	A set of redexes of graph G, which are sub-graphs of graph H.
$T(H, G', G, \widetilde{H})$:	Transforming graph H by replacing its sub-graph $\widetilde{H} \in R(H, G)$ with G' to yield a new graph.
$H \mapsto^R H'$:	R-application or reduction of a production $p := (L, R)$ to graph H, namely $H' = T(H, L, R, \widetilde{H})$.
$H \to^L H'$:	L-application or derivation of a production $p := (L, R)$ to graph H, namely $H' = T(H, R, L, \widetilde{H})$.
$H \mapsto^* H_n$:	A series of R-applications: $H \mapsto^{R1} H_1$, $H_1 \mapsto^{R2} H_2$, …, $H_{n-1} \mapsto^{Rn} H_n$ including the case $n = 0$ when $H = H_n$ and $H \mapsto H$.
$H \to^* H_n$:	A series of L-applications: $H \to^{L1} H_1$, $H_1 \to^{L2} H_2$, …, $H_{n-1} \to^{Ln} H_n$ including the case $n = 0$ when $H = H_n$ and $H \to H$.

A.3 The Generalization

The RGG+ inherits from the RGG the most basic concepts, such as graph element, graph, marking, isomorphism, redex, and graph transformations including L-application and R-application. In the RGG+'s grammar definition, a size-increasing condition is proposed to replace the layer decomposition mechanism used in the RGG. The condition makes RGG+'s easier to design while ensuring the membership problem of RGG+'s to be decidable.

A.3.1 Definition of a RGG+ and its language

It is well known that graph grammars with arbitrary graphs on the left and right sides of productions are able to generate type 0 languages, and the membership problem for type 0 languages is in general undecidable (Revesz 1983). Similar to LGGs, the requirement that the left graph of every production must be lexicographical smaller than its right graph ensures the decidability of RGGs. In order to relieve grammar designers from assigning labels to layers and ensuring a lexicographically smaller left graph than the right graph in each production, we introduce a size-increasing condition into the grammar definition, similar to the length-increasing condition in textual grammars. This is a weak constraint on the size between the right graph and the left graph of a production. Impacting little on the flexibility of context-sensitive grammars, the constraint is easy and intuitive to treat by grammar designers.

Definition A.1 $gg := (A, P, \Omega)$ is a graph grammar called *RGG+*, where A is an initial graph, P a set of graph grammar productions, Ω is a finite label set and can be further divided into two disjoint subsets, Ω^T and Ω^{NT}, for terminals and non-terminals respectively. For each production, $p = (L, R) \in P$, the following conditions must be satisfied:

- R is nonempty,
- L and R are over the same label set Ω, and
- the size of R must be no less than that of L, i.e., $|p.L.N| \leq |p.R.N|$; if they are equal, the number of non-terminal nodes in R must be more than that in L, i.e., $|p.L.N| = |p.R.N| \to |p.L.N^T| < |p.R.N^T|$.

The last condition ensures size-increasing, guaranteeing that a graph can be parsed in finite steps with the grammar and a definite answer as to whether or not the graph is in the grammar's language will be found. The language of the grammar is defined as follows.

Definition A.2 Let $gg := (A, P, \Omega)$ be a RGG+, its *language* $\Gamma(gg)$ is a set of graphs that can be derived from the initial graph A and each graph node has one terminal label, i.e.,

$$\Gamma(gg) = \{G \mid A \mapsto^* G \wedge f(G.N) \subseteq \Omega^T\}.$$

A.3.2 Decidability

When a RGG+ is given, its language is determined. Whether an arbitrarily given graph is in the language or not is decidable because of the following theorem.

Theorem A.1 For every RGG+ $gg := (A, P, \Omega)$ and for an arbitrary non-empty graph H ($H.N \neq \Phi$), it is decidable whether or not H is in $\Gamma(gg)$.

Proof: For a given graph H with a finite number of terminal nodes, namely $H.N = H.N^T$ being a finite set, the total number of graphs $\hat{H}$ with $f(\hat{H}.N) \subseteq \Omega$ and $|\hat{H}.N| \leq |H.N|$ must be finite under finite Ω. Considering a sequence of graphs

$$A = \hat{H}_0, \hat{H}_1, \hat{H}_2, ..., \hat{H}_{n-1}, \hat{H}_n = H$$

such that $\hat{H}_i.N^{NT} \neq \Phi$ and $|\hat{H}_i| \leq |\hat{H}_{i+1}|$ for $i = 0,1,...,n-1$ and $\hat{H}_i \neq \hat{H}_j$ if $i \neq j$, the number of such sequences without repetition is also finite. Thus we can enumerate all such sequences and check whether $\hat{H}_i \rightarrow^* \hat{H}_{i+1}$ ($i = 0,1,...,n-1$) holds for at least one of them. If so, then clearly $H \in \Gamma(gg)$, otherwise, $H \notin \Gamma(gg)$. □

In the proof, only inequality $|p.L.N| \leq |p.R.N|$ is employed, implying that the inequality itself can guarantee the decidability of the RGG+. However, introducing an implication $|p.L.N| = |p.R.N| \rightarrow |p.L.N^T| < |p.R.N^T|$ can help to speedup parsing. The implication and the inequality together decide that each R-application will at least either reduce a node or change a terminal node to a non-terminal node in the reduced intermediate graph. Therefore, R-applications can only be applied finite times to any host graph of a given size.

A.4 Graph Parsing

Having proven the membership problem for the languages defined by RGG+s to be decidable in the last section, this section presents a parsing algorithm for the languages of RGG+s.

The SFPA, the parsing algorithm of the RGG, is very efficient in parsing any graph in polynomial time under the assumption that if one parsing path fails, any other parsing paths will also fail. So, the SFPA is only suitable for those grammars called *selection-free grammars*. In order to support the specification of those context-sensitive grammars that are not selection-free, we need to develop a more general parsing algorithm that attempts to search all possible parsing paths instead of just one as in the SFPA. Such a parsing algorithm is no doubt more powerful but its performance may be seriously penalized because of the extremely large search space.

Generally, parsing is a process that applies a given graph grammar to perform a series of R-applications to reduce a given host graph to the initial graph. It usually needs to incorporate the following three interrelated actions:

1. Search in the host graph for the redexes of a production's right graph;
2. Accomplish an R-application with a found redex to produce a new intermediate graph from the host graph or from the current intermediate graph; and
3. Trace all the reduction paths by applying in turn the above two actions until a path leading to the initial graph is found or all possible paths have been exhausted.

In the following, we will discuss first how to trace all the reduction paths, and then about the search for all redexes of a production's right graph.

A.4.1 A Parsing Algorithm

A straightforward approach to parsing is to enumerate all the sequences and then check them one by one to see whether each derivation holds for at least one of the sequences, as shown in the proof of Theorem A.1. Unfortunately, this approach may have to handle many derivation-irrelevant sequences and thus is inefficient. In order to avoid irrelevant sequences, we try to trace all possible reduction paths starting from a given host graph to see if there exists one path that leads to the initial graph.

The following function realizes the tracing task, assuming that search for a redex and R-application has already been implemented (as in the SFPA). In the function two stacks are employed to separately store the redexes found and the intermediate host graph yielded. The tracing needs to maintain a correspondence between a redex and its host graph in order to perform the corresponding reduction. Since such a correspondence is usually many to one, the function uses a delimiter in the redex stack to delimit a group of redexes that correspond to the same host graph. The delimiter makes the correspondence manageable by synchronizing the contents in the two stacks. The function takes a graph and a set of productions as input and returns a definite answer indicating whether the graph is valid or not.

```
Parsing (Graph $H$ , ProductionSet $P$ )
{
loop-1: while ( $H \neq A$ )
        {
          DELIMITER → redexStack;                  // push
Loop-2:   for all $p \in P$
          {
            redexSet = FindRedexForRight( $H$ , $p.R$ );
Loop-3:     for all redex ∈ redexSet;
                redex → redexStack;                // push
          }
          redex ← redexStack;                      // pop
Loop-4:   while (redex = DELIMITER)
          {
            $H$ ← hostStack;                       // pop
            redex ← redexStack;                    // pop
            if (redex = NULL)
                return("Invalid");
          }
          hostStack ← $H$ ;                        // push
          $H$ = RightApplication( $H$ , $p$ , redex);
      }
      return("valid");
}
```

A4.2 Search for Redexes

The SFPA has partially solved the search problem, in which a function is constructed to search for only one redex once. Fortunately, it is easy to extend the function from searching for one redex to searching for all redexes. The extended function is given as follows, which takes a host graph and a right graph as input and returns a set of redexes found.

```
FindRedexForRight(Graph $H$ , Graph $R$ );
{
  redexSet = $\Phi$ ;
  nodeSequence = orderNodeSequence( $R$ );
  candidateSet = findNodeSequenceSet( $H$ , nodeSequence);
```

```
    for all $candidate \in candidateSet$
        $redexSet = redexSet$+match($candidate$, $H$, $R$);
    return($redexSet$);
}
```

The function orderNodeSequence(R) sequences the nodes in the right graph according to their labels' alphabetic order. The function findNodeSequenceSet(H, *nodeSequence*) finds all possible node sequences from the host graph, each of which is isomorphic to the nodeSequence. Finally, the function match(*candidate*, H, R) checks whether a candidate in the host graph is a redex of the right graph, if so, the candidate is returned as a redex, otherwise, a null is returned.

A.5 Parsing Complexities

Complexity analysis of an algorithm is helpful in evaluating the algorithm's performance. In this section we analyze the time complexity and the space complexity of the above parsing algorithm.

A.5.1. Time Complexity

Theorem 5.1 The time complexity of the parsing algorithm is $O((\frac{n}{r!})^h (h^h h!)^r)$, where h is the number of nodes in the host graph to be parsed, r is the maximal number of nodes in the right graphs of all productions, and n is the number of productions in the given RGG+.

Proof: According to the structure of the parsing algorithm, its maximal time complexity can be expressed as:

$$t = O(l_1(l_2(t_1 + l_3) + l_4 + t_2)),$$

where l_1 is the maximal number of iterations in the outmost loop-1, l_2 is the number of iterations in the first inner loop-2, l_3 is the number of iterations in the innermost loop-3, l_4 is the number of iterations in the second inner loop-4, and t_1 and t_2 are the time complexities of FindRedexForRight() and RightApplication() respectively.

We first consider l_2, which is in fact the number of productions, i.e., $l_2 = n = |P|$. Since $l_2 \cdot l_3$ is the total number of actions in pushing redexes into the redex stack and l_4 is the partial number of actions in popping redexes from the redex stack, l_4 should be no more than $l_2 \cdot l_3$, and thus can be ignored. Since l_3 is the number of redexes found in the host graph with respect to the right graph of a given production, the maximal number of redexes is all possible node combinations C_h^r, thus $l_3 \le C_h^r = O(h^r)$. When considering l_1, the worst case is when the algorithm's result is 'invalid', and all redexes found during parsing will enter the stack. Each of the redexes, when popped out of the stack, leads to an iteration of the outmost loop. Therefore, l_1 equals to the number of the redexes found.

An iteration of the outmost loop produces no more than nC_h^r redexes for n productions and performs one R-application. According to the size-increasing condition, each R-application would reduce the size of the derived host graph. Since there are at most h R-applications that may not reduce the host graph size and an R-application will reduce the host graph size by at least 1, the following derivations hold for l_1:

$$l_1 \le (nC_h^r)^h nC_{h-1}^r nC_{h-2}^r \ldots nC_{h-(h-r-1)}^r nC_{h-(h-r)}^r \qquad (1)$$

$$= n^{2h-r}(C_h^r)^h C_{h-1}^r C_{h-2}^r \ldots C_{r+1}^r C_r^r$$

$$= n^{2h-r}\left(\frac{h!}{(h-r)!r!}\right)^h \frac{(h-1)!}{(h-1-r)!r!} \ldots \frac{(r+1)!}{1!r!}\frac{r!}{0!r!}$$

$$= n^{2h-r}\left(\frac{h!}{(h-r)!r!}\right)^h \prod_{u=1}^{h-r-1}\frac{(u+r)!}{r!u!}$$

$$= \frac{n^{2h-r}}{(r!)^{2h-r-1}}(h(h-1)\ldots(h-r+1))^h \prod_{u=1}^{h-r-1}(u+r)(u+r-1)\ldots(u+2)(u+1)$$

$$= O\left(\left(\frac{n}{r!}\right)^{2h} h^{rh} \prod_{u=1}^{h-r-1} u^r\right)$$

$$= O\left(\left(\frac{n}{r!}\right)^{2h} h^{rh}((h-r-1)!)^r\right)$$

$$= O((\frac{n}{r!})^h (h^h h!)^r)$$

As for t_1 and t_2, since the maximal possible number of selections of r nodes from h nodes is $A_h^r = h(h-1)...(h-r+1)$, the worst cases of searching for all redexes of a right graph in a given host graph must be $t_1 = O(h^r)$. Since t_2 is independent of h, it can be considered $t_2 = O(1)$ that is bounded by a constant time.

Combining all the above discussions, we can finally obtain:

$$t = O((\frac{n}{r!})^h (h^h h!)^r). \qquad \square$$

A.5.2 Space Complexity

Theorem A.2 The space complexity of the parsing algorithm is $O(h^{r+1})$, where h is the number of nodes in the host graph to be parsed, r is the maximal possible number of nodes in all the right graphs of productions.

Proof: Obviously the main space-consuming components are the redex stack and the host graph stack used in the parsing algorithm. We can therefore express the maximal space complexity as:

$$s = s_1 + s_2,$$

where s_1 is the space used by the redex stack and s_2 is that by the host graph stack. Without losing generality, we can assume that the space taken by a redex is r and that by a host graph is h. Different from time complexity, the use of the stack space is not always increasing because pop operations would release space for reuse. Hence, the worst case is the maximal occupied space along the longest reduction path, and the following derivations hold for the redex stack and the host graph stack respectively.

$$s_1 \le r(hnC_h^r + nC_{h-1}^r + ... + nC_{h-(h-r)}^r) \qquad (2)$$

$$= rn(hC_h^r + C_{h-1}^r + ... + C_r^r)$$

$$= rn\left(\frac{hh!}{(h-r)!r!} + \frac{(h-1)!}{(h-1-r)!r!} + ... + \frac{r!}{0!r!}\right)$$

$$= \frac{n}{(r-1)!}\left(\frac{hh!}{(h-r)!} + \sum_{u=0}^{h-r-1}\frac{(u+r)!}{u!}\right)$$

$$= \frac{n}{(r-1)!}(hh(h-1)...(h-r+1) \quad + \sum_{u=0}^{h-r-1}(u+r)(u+r-1)...(u+1))$$

$$= O(h^{r+1} + \sum_{u=0}^{h-r-1} u^r)$$

$$= O(h^{r+1});$$

$$s_2 = hh + (h-1) + ... + r$$

$$= O(h^2).$$

Since $r \geq 1$, we can obtain:

$$s = O(h^{r+1}). \qquad \Box$$

From the above analysis, we observe that the time complexity is extremely high while the space complexity is bounded by a polynomial factor. We also observe that the structure of productions plays an important role in determining the algorithm's complexity. For example, if a stronger constraint such as $| p.L.N | \lessdot | p.R.N |$ is enforced on productions, then the first h R-applications that do not reduce the host graph size can be removed from (1). In addition, the algorithm itself may be further improved to increase its efficiency, especially its average time cost.

A.5.3 Optimization Considerations

Although the worst time complexity of the algorithm is extremely high, the parsing algorithm provides a starting point for further improvement. In practice, the worst case rarely occurs during parsing. When applying the RGG+, we could concentrate on the reduction of the average time cost of parsing under appropriate assumptions applicable to the most common applications. The high parsing costs are due to the search for redexes and handling of backtracking, which should be the primary targets for performance improvement.

A straightforward approach to reducing the redex searching time is to narrow down the search space within the host graph. In most applications, the size of a host graph is usually much larger than that of right graphs in productions. So, when searching for redexes in a given host graph, we only need to search for redexes among those host graph nodes whose labels appear in the right graph under consideration rather than searching for all the nodes in the host graph. This can be done by first removing the irrelevant nodes from the host graph, and then searching for redexes in the remaining nodes. Having obtained the node sequences of the host graph and right graph, irrelevant nodes can be removed from the host graph in linear time, i.e. rh. Let h' be the number of the remaining nodes, the time for searching for a redex can be expressed as $rh + C_{h'}^{r}$ instead of C_{h}^{r}. In the applications where the host graph is significantly larger than right graphs, the labels appearing in a right graph merely make a small portion of all the labels appearing in the host graph, that is, $h >> r$ and $h' \approx r$. The complexity of searching for a redex in such a case can be redcued from $O(h^{r})$ to $O(h)$. The assumption that the host graph is significantly larger than right graphs is realistic in many applications.

Another optimization approach is to perform as early as possible the R-applications that reduce the size of the host graph, so that the subsequent search space would be reduced significantly. This can be done by ordering the productions according to their values of $|p.R.N| - |p.L.N|$. Based on the order, the production with a larger right graph would be applied later, which inversely, due to stack operations, gives its corresponding redexes higher priority for R-applications. Of course, if some productions with the large right graphs are not the ones needed for graph reduction, the above approach improves little in performance. However, the ordering of productions can be preprocessed without adding extra cost to the parsing algorithm. Therefore, on average, the approach may be able to reduce the parsing time.

As mentioned before, another major cost in the parsing algorithm is in its backtracking, which attempts all the productions and their redexes. An ideal approach to optimizing backtracking is to avoid as many unnecessary productions and redexes as possible in each step of reduction, i.e. to push less redexes into the stack during parsing. To perform such an optimization, more information is required to limit the scope of productions as well as their redexes. Because graphs are inherently 2-dimensional and spatial, one general solution is to use the spatial information, such as direction, topology, alignment, etc., to introduce extra constraints into the search space. The Spatial Graph Grammar formalism (SGG) presented in Chapter 3 employs the spatial information specified in the grammar to improve its parsing performance. In the SGG, a mechanism for specifying spatial information is introduced into productions so that the nodes in right graphs may be specified to have some types of spatial relationships. Since a given host graph can be preprocessed to establish relevant spatial relationship among its nodes according to their positions in the graph, the matching of spatial information between the host graph and the right graphs makes a useful constraint in restricting the selection scope for productions and redexes.

Some existing approaches may be adapted and extended to improve the parsing performance. For example, the delta-based approach used in PROGRES (Schürr 1991), incremental pattern matching based on RETE networks (Lee and Schor 1992), and conflict detection with critical pair analysis (Bottoni et al. 2000). Furthermore, additional information, especially domain knowledge in specific applications, may also be helpful in reducing aimless selection of productions and redexes during parsing.

A.6 Summary

This appendix has presented the RGG+ formalism, more general version the RGG formalism. Compared with the original RGG, two general extensions are made in the RGG+. One is to replace the layer decomposition mechanism by the size-increasing condition to ensure the decidability of RGG+s. The other is a general parsing algorithm that works on any RGG+ regardless whether it is selection-free or not. These generalizations make the grammar easier to develop and the parsing algorithm more widely applicable than the original RGG. As noted above, the penalty for these generalizations is the more complex and time-consuming parsing, especially in the worst cases, i.e. the upper bound of the parsing complexity. Hence, improving the parsing efficiency is necessary.

Index