Visual Modeling and Programming with Graph Transformations

VL'98 Tutorial

14th IEEE Symposium on Visual Languages

Halifax, Nova Scotia, Canada, Sept. 1-4, 1998

Dorothea Blostein
Dept. Computing and Information Science
Queen's University, Kingston,
Ontario, Canada K7L 3N6
blostein@qucis.queensu.ca
phone: (613) 545-6537

Andy Schürr
Institute for Software Technology
Department of Computer Science
University of the German Federal Armed Forces
D-85577 Neubiberg, Germany
schuerr@informatik.unibw-muenchen.de

Table of Contents

Glossary iv

1. Introduction 1

1.1 Visual Languages and Graph Transformation 1

1.1.1 History of Graph Transformations 2

1.2 Software Treatment of Graphs 3

1.3 Introductory Example of A Graph Production 4

1.4 Application Areas Which Use Graph Transformation 4

1.5 Overview of These Notes 6

2. Notations and Mechanisms for Graph Productions 7

2.1 Terminology and an Example 7

2.2 Steps in the application of a graph production 10

2.3 Overview of Embedding Mechanisms 11

2.3.1 Hierarchy of Graph Language Classes 12

2.3.2 Unrestricted Embeddings: Expression Notation 13

2.3.3 Diagrammatic Notations for Unrestricted Embedding: Y, X and Δ Notation 14

2.3.4 Depth1 Embeddings: NCE 17

2.3.5 Elementary Embeddings: Schneider's Notation 18

2.3.6 Invariant Embeddings 18

2.4 Other Aspects of Graph Productions 19

2.4.1 Induced versus Non-induced Subgraph Isomorphism 19

2.4.2 A Class Hierarchy for Node Labels 22

2.4.3 Homomorphism versus Isomorphism 23

2.4.4 Hierarchical Graphs and Distributed Graph Rewriting 23

2.4.5 Efficiency of Production Application 23

2.4.6 Tools and Languages for Graph Transformation 24

2.5. Methods of Controlling the Application of a Set of Graph Productions 24

2.5.1 Unordered Graph Transformation Systems 24

2.5.2 Graph Grammars 25

2.5.3 Ordered Graph Transformation Systems 25

2.5.4 Event-driven Graph Transformation Systems 26

3 . The Graph Transformation Language PROGRES 27

3.1 Object Oriented Modeling and Graph Transformation 27

3.2 Related Specification and Programming Languages 29

3.2.1 Semiformal Modeling and Formal Specification Languages 29

3.2.2 Visual Rule-Based Programming Languages 29

3.2.3 Graph Transformations and Meta Programming Tools 30

3.3 Graph Schemata and Derived Graph Properties 31

3.3.1 The Running Example and DIANE Graphs 31

3.3.2 Node Classes, Node Types, and Edge Types 32

3.3.3 Standard Attribute Types and Functions 34

3.3.4 External Attribute Types and Functions 34

3.3.5 Intrinsic Node Attributes 35

3.3.6 Derived Node Attributes and Constraints 36

3.4 Graph Queries and Graph Transformations 37

3.4.1 Restrictions and Path Declarations 37

3.4.2 Subgraph Tests and Attribute Conditions 39

3.4.3 Productions and Attribute Assignments 40

3.4.4 Advanced Pattern Matching Concepts 42

3.4.5 Control Structures and Transactions 44

3.4.6 Consistency Checking with Constraints 46

3.5 Modules and Updatable Graph Views 49

3.5.1 PROGRES Packages 50

3.5.2 Specification-in-the-Large with Packages 52

3.5.3 Graphical Modeling with Updatable Graph Views 53

3.6 The Programming and Prototyping Environment 54

3.6.1 Basic Components and their Interdependencies 54

3.6.2 Editing and Analyzing Specifications 56

3.6.3 Executing and Debugging Specifications 58

3.6.4 Prototyping 59

3.7 Summary 61

4 . Graph Transformation Applied to Document Image Analysis 63

4.1 Overview of Document Image Analysis 63

4.2 Systems Using Graph Rewriting for Document Image Analysis 64

4.2.1 Parsing Images of Neural Networks [Pfal72] 65

4.2.2 Analysis of Circuit Diagrams and Flowcharts [Bun82] 65

4.2.3 Analysis of Dimension Sets in Engineering Drawings [DoPn88] 65

4.2.4 Analysis of Music Notation [FaBl93] 66

4.2.5 Analysis of Music Notation Using Simplified Graph Productions [Pies94] [Baum95] 66
4.2.6 Analysis of Mathematical Notation [GrBl95] 67
4.2.7 Analysis of Mathematical Notation [LaPo97] 67
4.2.8 Analysis of Music Notation with Symbol Uncertainty [FaBl98] 67
4.3 PROGRES Program for Interpreting Images of Mathematical Notation 68
4.3.1 Build, Constrain, Parse 68

5. Re-Design of Legacy Applications with Graph Transformations 77
5.1 Motivation 77
5.2 Methodology 77
5.3 Design Recovery 79
5.4 Visualization of the Design Recovery Information 79
5.5 Re-Engineering 81
5.5.1 Structural Changes 81
5.5.2 Source Code Alterations 84
5.6 Summary 84

6. Modelling a Visual Language with PROGRES 85
6.1 Concepts of Modelling a Visual Language 85
6.1.1 The HotVla Language 86
6.1.2 The HotVla Syntax 86
6.2 Generating a HotVla Editor 89
6.3 An Analysing Mechanism for HotVla 89
6.4 Executing HotVla in the Generated Prototype 91
6.5 Summary 92

Conclusion 93

Bibliography 94

Glossary

The following terms and symbols are used in these notes. A more detailed explanation of the relevant concepts is provided in the notes.

application condition	Defines conditions on LHS^{host}, typically testing attribute values or host graph structure. These conditions must hold for rule application to proceed.
attribute transfer clause	Assigns attribute values to RHS^{host}, calculated from attribute values in LHS^{host}.
attributes	The nodes and edges in a graph may be attributed, i.e. have attribute information associated with them. Attributes may be of any data type. They are used to record auxiliary information, which is not captured by the structure of the graph.
context free production	A production in which LHS consists of exactly one node. (This node has a nonterminal label.).
context sensitive production	A produciton in which a portion of LHS exists as a subgraph of RHS.
DIANE graph	DIrected Attributed Node and Edge labeled graph. PROGRES operates on DIANE graphs.
edge type	In PROGRES, the edge type determines the label of the edge, as well as the permissible types of source and target nodes
embedding rule	Synonym of *embedding specification*
embedding specification	Calculates post-embedding edges from pre-embedding edges. Embedding information can be provided using textual or graphical notation.
folding	In PROGRES, when two nodes of a rule's LHS are folded, this means that they are allowed to match the same host graph node.
graph grammar	A set of productions and an initial graph (the axiom). Node and edge labels are designated as terminal or non-terminal. A terminal graph is a graph in which all labels are terminal. The graph language defined by a grammar consists of the set of terminal graphs that are derivable from the initial graph.
graph production	A construct which replaces one subgraph by another subgraph, analogous to the way that a string-grammar production replaces one substring by another. A graph production consists of LHS, RHS, an embedding specification, and possibly application conditions and attribute transfer clauses.
graph rewriting system	Synonym for *graph transformation system*.
graph schema	In PROGRES, the graph schema provides a type definition of a class of graphs. This includes a declaration of the node and edge types which are used, a declaration of node attributes, a declaration of which node types can act as source and destination for each edge type, and additional integrity constraints.
graph transformation system	A set of rules which implement the graph inspection and modification operations needed by an application.
host graph	The graph to which graph productions are being applied. The host graph may be directed or undirected, depending on the needs of the application. In most applications, the host graph is attributed. The phrase *host graph* is often shortened to *graph*., where the meaning is clear from context.

induced subgraph	Let M be a subgraph of graph N. M is an induced subgraph if it satisfies the following conditions: if an edge of graph N connects two nodes of M, then that edge must be part of M.
labels	The nodes and edges in a graph may be labeled, i.e. marked with a specific label drawn from a finite set of labels.
LHS	The Left Hand Side of a graph production. This is an unattributed graph.
LHS^{host}	A subgraph of the host graph, isomorphic to LHS. This is an attributed (sub)graph. Some graph transformation mechanisms require that LHS^{host} is an induced subgraph: if a host-graph edge connects two nodes of LHS^{host}, then that edge must be part of LHS^{host}.
LHS match	Synonym for LHS^{host}
LHS occurrence	Synonym for LHS^{host}
linear production	A context free production where RHS contains any at most one node with a nonterminal label. (Any number of terminal-labeled nodes are allowed in RHS.)
monotone production	A production in which the number of nodes in LHS ≤ number of nodes in RHS.
node class	The node class is always an abstract class in PROGRES; its extension is a set of node types with common properties.
node denotations	Unique names for the nodes in LHS and RHS. These names are only used within the production itself (e.g. within the application condition and attribute transfer clause) to refer to nodes of LHS and RHS. Often, numbers are used for node denotations. PROGRES node denotations take the form \`1,\`2, \`3 ... in LHS, and 1´, 2´, 3´ ... in RHS.
node type	A PROGRES node type determines the properties of a node. Node types are declared within the graph schema. This declaration includes the name used to label nodes of this type, the attributes that nodes of this type have, and the permissible types of incoming and outgoing edges which may connect to nodes of this type. The same name is used for the node label as for the node type as a whole.
package	A PROGRES module which contains a number of related declarations, typically including production rules and type definitions.
path expression	A PROGRES procedure which describes a graph traversal.
pre-embedding edges	The set of edges joining LHS^{host} to RestGraph
production	See “graph production”
post-embedding edges	The set of edges joining RHS^{host} to RestGraph
regular production	A context-free produciton in which RHS has a unique maximum node. All other RHS nodes are predecessors of the maximum node. The maximum node may have a terminal or non-terminal label, all other RHS nodes have terminal labels.
RestGraph	The host graph minus LHS^{host}. (The “minus” operator removes all nodes and edges of LHS^{host}, and all edges with one or both endpoints in LHS^{host}.)
restriction	A PROGRES condition on a single node, restricting its attribute values and/or its context in the graph.
RHS	The Right Hand Side of a graph production. This is an unattributed graph.
RHS^{host}	A newly-created subgraph isomorphic to RHS; used to replace LHS^{host}. This is an attributed graph. The graph structure is copied from RHS, and attribute values are assigned by the attribute transfer clause.
rule	A synonym for “graph production”.

subgraph isomorphism	Formal definitions of subgraph isomorphism can be found in many textbooks, such as [CoLR90]. Roughly, the structure of the graphs has to match: the nodes are put into correspondence in such a way that edge connectivity matches. Node and edge labels must match for isomorphism, but attribute values are often ignored.
subgraph test	A PROGRES construct which checks for the existence of a certain graph pattern without changing the host graph.
transaction	A atomic and consistency-preserving PROGRES construct which defines a programmed graph transformation.
unrestricted production	A produciton with no restriction on LHS or RHS.

1. Introduction

1.1 Visual Languages and Graph Transformation

Andy wrote this

1.1.1 History of Graph Transformations

Andy wrote this

1.2 Software Treatment of Graphs

Graphs are widespread. Society uses graph-based notations such as organizational charts, circuit diagrams, and flow charts. Computer programs use graph data structures such as traveling-salesman networks and semantic networks. Typically, nodes represent objects or concepts and edges represent relationships. Auxiliary information, of any data type, is stored in node attributes or edge attributes[1].

Software treatment of graphs involves the representation, inspection, modification, display, and recognition of graphs, as illustrated in Figure 1.1. In current undergraduate computer science curricula, graph modification receives far less attention than graph representation and graph inspection. In existing software, it is common to modify graphs through pointer-manipulation code for adding and deleting nodes and edges. This is a low-level, error-prone coding style. *Graph productions* (also called *graph rewrite rules* and *graph transformation rules*) provide a higher-level interface for graph modifications. The graph productions are clearly delineated from the rest of the code, and can be conveniently expressed using visual language constructs. Graph productions replace one subgraph by another subgraph, analogous to the way that string-grammar productions replace one substring by another. Graph transformation has been intensively studied for several decades [IWGG] [Roz97] [Roz99], but is not widely known in the general computer science community. As a result, the concept gets reinvented, for example in reference [ArHW90]. Graph transformation is an intuitive and useful tool for specification and design in many software applications.

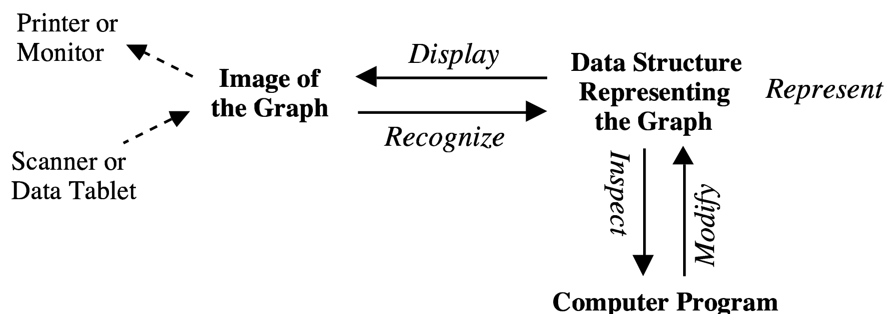


Figure 1.1 Five aspects of graph processing. *Represent* the graph; standard methods include an adjacency list, adjacency matrix, or incidence matrix. *Inspect* the graph; standard algorithms include breadth-first or depth-first search. *Modify* the graph; graph productions can be used, as illustrated in these notes. *Display* the graph, using a graph-layout algorithm. *Recognize* drawings of graphs, using document-image analysis techniques.

[1]Chapters 1 and 2 include excerpts from references [BlFG95], [BlFG96] and [BlSc98].

1.3 Introductory Example of A Graph Production

Figure 1.2 illustrates the definition and use of a graph production. The graph production performs a local update on a graph, replacing a subgraph that matches LHS (Left Hand Side) by a copy of RHS (Right Hand Side). We use sequential graph transformations. In contrast, parallel graph transformation simultaneously replaces all matches of LHS by copies of RHS. Graph transformation is analogous to the string transformation performed by production rules in string grammars. However, graph transformation has more degrees of freedom. There are various ways of attaching the copy of RHS to the graph. This is called the *embedding* of RHS in the graph. Figure 1.2 (c) and (d) show two possible embeddings when rule (a) is applied to graph (b). On its own, the execution environment cannot choose the embedding that is most appropriate for the application. The source code for the graph production must specify the desired embedding.

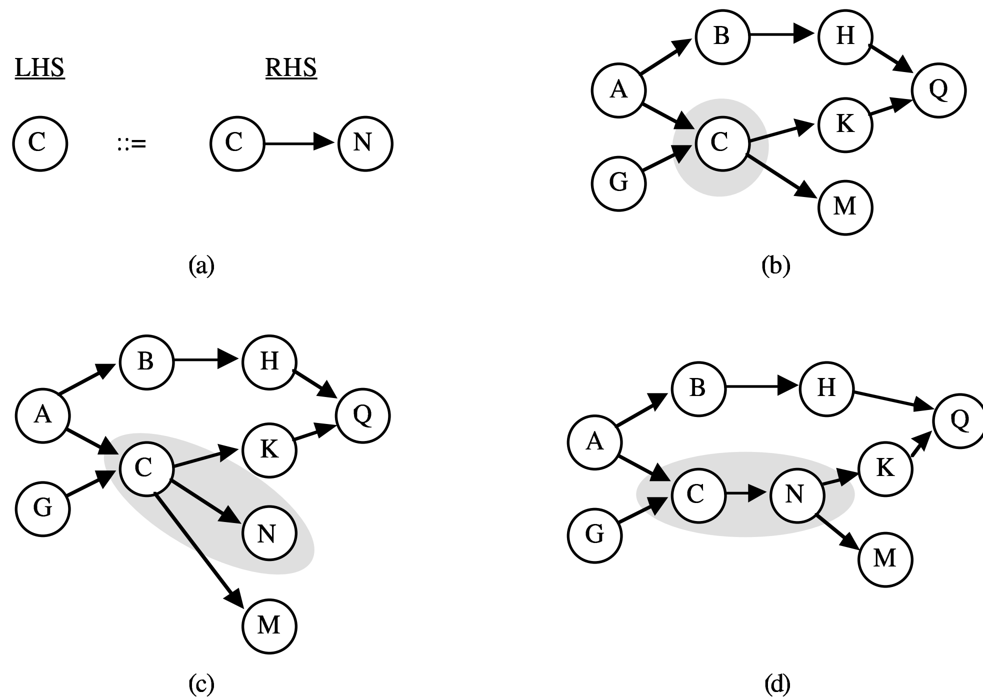


Figure 1.2 The graph production (a) adds a new node N following node C. This transforms the graph in (b) into a result such as (c) or (d), depending on how the copy of RHS is embedded in the graph.

1.4 Application Areas Which Use Graph Transformation

Graphs and graph transformations have been used in many application areas. These applications involve the manipulation of structured data; this data is stored as a graph, and the desired manipulations are carried out by graph productions. Execution of the productions by an interpreter provides a prototype implementation of the system. Once the system design is finalized, faster execution can be provided by translating the specifications to an implementation language. Applications are discussed more extensively in [Roz99].

- *Define Visual Languages and their Tools* [DoTo88] [ArHW90] [Göt92] [ReSc97] [Min97]

  Presently, many visual languages are defined informally, through examples; these include class diagrams, state transition diagrams, object interaction diagrams, and message sequence charts used in object-oriented software development. The syntax and static semantics of a visual language can be unambiguously defined using graph transformation. The *validity* of an abstract syntax graph is defined through static type declarations and run-time constraints. Graph productions define the syntax of the language and generate a parser. It can be guaranteed that all editing operations leave the syntax graph in a valid state.

  An editor for the visual language can be defined using a similar approach. The editor shows the visual-language program as a diagram on the screen, but represents it internally by an abstract syntax graph. Graph productions specify the semantics of each editing operation. Execution of these rules provides a prototype editor for the visual language.

- *Create a Software Development Environment* [Nag96] [ELNSS92]

  A graph represents all of the files associated with a software development effort, including the requirements specification, design, source code, and documentation. Each file gives rise to nodes and edges that represent that file and its internal relationships. Edges between files support cross-referencing and version management. Graph productions implement coarse-grained operations such as configuration management as well as fine-grained operations used within editors, interpreters, and integration tools.

- *Define Constraint Programming Algorithms* [MoRo93] [KKS97]

  A graph represents a constraint network, using a node for each variable and an edge for each binary constraint. The possible values for each variable are stored as node attributes. Constraint-solving algorithms are defined via graph productions. Rules can modify the structure of the constraint network or update the node attributes storing the possible values for variables.

- *Model Distributed Systems of Processes* [Bart96] [Hri98]

  A graph models the state of a distributed system, using a node for each process or message. Edges define communication channels as well as sender and recipient for individual messages. Graph productions model message passing among processes. Various types of execution models can be treated, including performance models, interaction diagrams, and distributed tracing features.

- *Reconfigure an Array of Processors* [DeDe96]

  The goal is to design and analyze reconfiguration algorithms. A graph represents a processor array, with one node per processor and an edge for each active communication line between processors. Graph productions activate and deactivate communication lines to replace faulty processors with spares.

- *Implement Functional Languages* [PeJo87]

  The abstract syntax tree of a functional language program becomes a Directed Acyclic Graph (DAG) due to sharing of subexpressions. Graph productions reduce the DAG to a single node. This has the effect of evaluating the program represented by the DAG.

- *Build a Neural Network* [Grua95]

  Design of neural networks is a difficult problem, attacked here by searching the space of possible neural networks. The system starts with an initial network consisting of a single neuron. This network is stored as a graph. Graph productions cause growth of the neural network. A set of trees directs which graph productions to apply next. Genetic algorithms are applied to the set of trees, to evolve to a situation in which a desirable neural network is produced.

- *Model Plant Growth* [PrLi90]

  Biologists and mathematicians have long been interested in modeling the growth of plants. Fascinating developmental processes determine the shape of leaves, pine cones, flowers, and so on. Parallel graph transformations are one of the tools that have been used. A graph represents the arrangement of cells in a plant. Graph productions model the various types of cell division that occur in plant growth. These graph productions are applied in parallel to all parts of the host graph.

1.5 Overview of These Notes

Chapter 2 provides a more detailed discussion of graph transformation. This includes notations and mechanisms for describing graph productions, as well as methods of controlling the application of a set of graph productions.

Next, Chapter 3 provides an introduction to the language *PROGRES* and its tools. PROGRES uses a mixture of textual and visual constructs to support computation via graph transformation. PROGRES tools include a syntax directed editor, an interpreter, a debugger, and a facility for translating PROGRES code to C and Tcl/Tk. PROGRES constructs are introduced using the development of a library database as a running example. A different running example is used in the oral tutorial presentation: the development of a process management tool based on precedence diagrams.

The final chapters discuss the use of graph transformation to solve problems in particular application areas. Chapter 4 discusses pattern recognition applications, particularly the recognition of mathematical notation in document images. Chapter 5 describes the redesign of legacy applications with graph transformations. Chapter 6 discusses the modeling of a visual dataflow language. PROGRES source code is available for these applications and running examples.

Chapter 2

Notations and Mechanisms for Graph Productions

This chapter provides an overview of notations and mechanisms used for graph transformation. We begin by defining terminology and describing the steps typically involved in application of a graph production. this is followed by a presentation of embedding mechanisms of varying power and generality. Both textual and diagrammatic notations can be used for these. Various classes of graph languages can be generated by graph grammars that use these embedding mechanisms.

2.1 Terminology and an Example

The following terminology is used to describe graph productions.

host graph	the graph to which graph productions are being applied
LHS	the Left Hand Side of a graph production
RHS	the Right Hand Side of a graph production
LHS^{host}	the part of the host graph which matches LHS (i.e. which is isomorphic to LHS)
RHS^{host}	the copy of RHS which is inserted into the host graph, in place of LHS^{host}
Graph Production	A rule specified by:
• LHS and RHS	LHS and RHS are unattributed graphs. A subgraph isomorphic to LHS is to be replaced by one isomorphic to RHS
• Embedding information	A textual or graphical description of the embedding. This describes how pre-embedding edges are converted to post-embedding edges. Algebraic graph rewriting (Section 1.1.1) uses a gluing isomorphism in place of embedding information.
• Application condition	Restrictions on rule application. Optional. These can include restrictions on the existence of host-graph nodes and edges, as well as restrictions on attribute values.
• Attribute Transfer Function	Assigns attribute values. Optional. Attribute values for RHS^{host} are computed from attribute values of LHS^{host} .
LHS^{host}	A subgraph of the host graph g, isomorphic to LHS. In some models, LHS^{host} must be an *induced* subgraph: if an edge of g connects two nodes of LHS^{host}, then that edge must be part of LHS^{host}.
RestGraph	The host graph minus LHS^{host}. (The “minus” operator removes all nodes and edges of LHS^{host}, and all edges with one or both endpoints in LHS^{host}.)
RHS^{host}	A subgraph isomorphic to RHS; used to replace LHS^{host}.

Pre-embedding edges	the set of edges joining LHS^{host} to RestGraph. (These edges "embed" LHS^{host} in g.)
Post-embedding edges	the set of edges joining RHS^{host} to RestGraph.

This terminology is illustrated by the graph production in Figure 2.1. This production is part of a graph transformation system for understanding music notation [FaBl93]. Starting with a scanned image of music notation, a symbol recognizer provides a list of music symbols and their location on the page. (The symbol recognizer is assumed to operate perfectly, to make the subsequent recognition easier.) The list of music symbols is converted to an initial graph. The production shown in Figure 2.1 handles music notation where one notehead symbol needs to be interpreted as two separate notes.

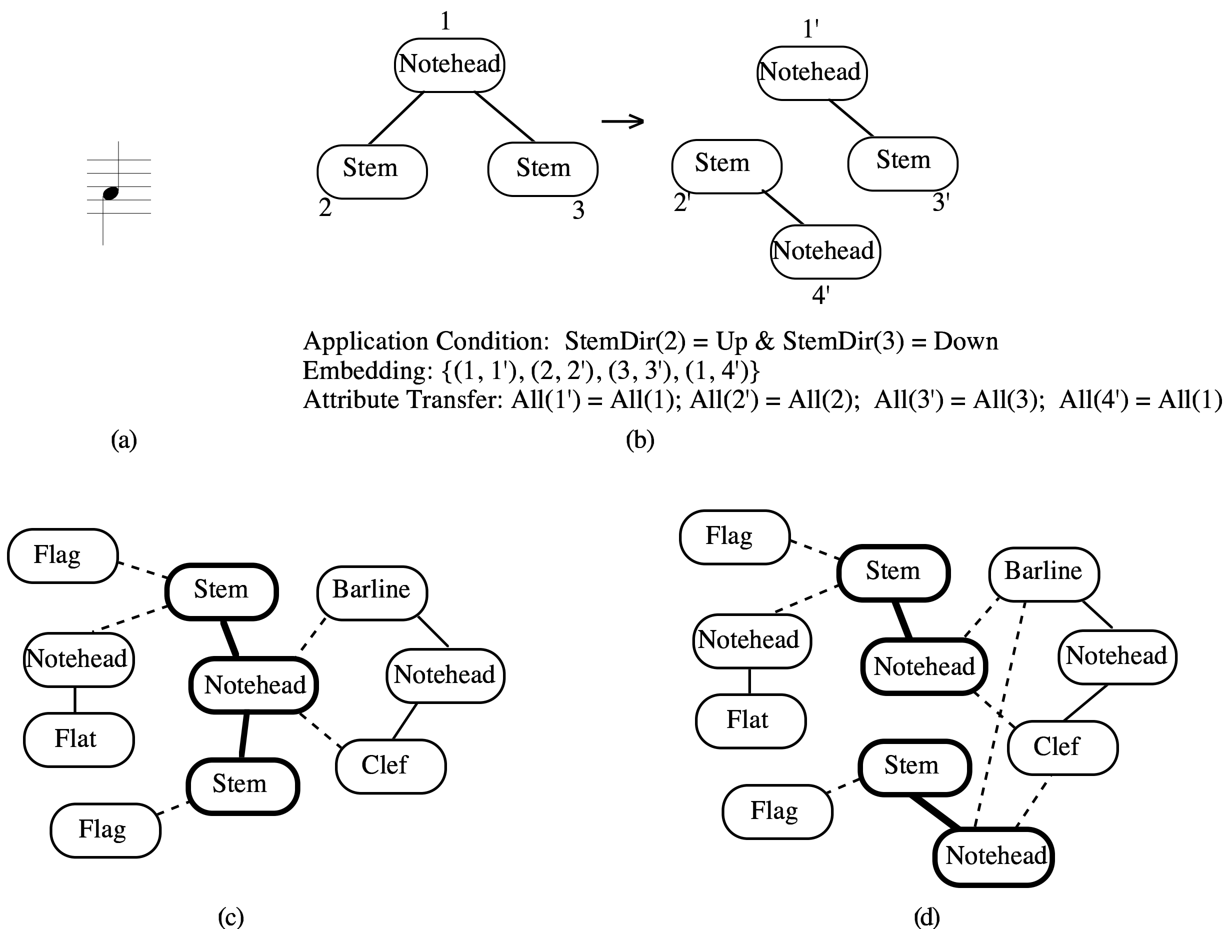


Figure 2.1 Notation (a) uses one notehead symbol to indicate two notes, one belonging to the voice printing with downward stems, the other belonging to the voice printing with upward stems. Production (b) handles this by splitting one Notehead node into two. When production (b) is applied to host graph (c), the result is graph (d). The production matches a Notehead node that has two associated stems. The application condition states that node 2 must have a StemDir attribute with value Up, and node 3 a StemDir attribute with value Down. The embedding clause (1, 1') specifies that any pre-embedding edges connected to LHS^{host} node 1 give rise to post-embedding edges connected to RHS^{host} node 1'. The other three clauses are similar. In (c), LHS^{host} is drawn bold, and the five pre-embedding edges are drawn dashed. RestGraph has 7 nodes. In (d), RHS^{host} is bold, and the seven post-embedding edges are drawn dashed.

2.2 Steps in the application of a graph production

The following steps are used to apply a graph production. Details vary depending on which graph transformation mechanism is used.

1. Locate LHS^{host} , a subgraph of the host graph that is isomorphic to LHS.
   If no isomorphic subgraph can be found, report failure of rule application.
   If there are several isomorphic subgraphs, choose any one of them.

2. Test the application condition. If it fails, this LHS^{host} cannot be used for production application. Try a different LHS^{host} (choosing among the isomorphic subgraphs in step 1). If all LHS^{host} locations fail the application condition, then report failure of rule application.

3. Remove LHS^{host} from the host graph. Keep track of the set of pre-embedding edges (i.e. edges that connected RHS^{host} to RestGraph).
   Sometimes only part of LHS^{host} is removed. Removal does not occur for nodes and edges which are common to both LHS and RHS. Examples of this include algebraic graph rewriting (the gluing points are not removed) and PROGRES (where preserved nodes, notated as n´ = `n in RHS, are not removed). This is explained in detail later.

4. Create RHS^{host} by making a copy of RHS.
   If step 3 involved leaving part of LHS^{host} in the host graph, then that part of RHS does not need to be copied in step 4.

5. Embed RHS^{host} into RestGraph, using the embedding information supplied with the production. Typical embedding information might take the form: if a pre-embedding edge connected LHS^{host} node X to some RestGraph node, then construct a post-embedding edge connecting RHS^{host} node Z to that same RestGraph node.

6. Compute attribute values for RHS^{host}, using the attribute transfer function supplied with the production.

2.3 Overview of Embedding Mechanisms

A graph production implicitly or explicitly provides embedding information. Embedding mechanisms vary in complexity and power, allowing a varying amount of computation to be performed in converting the pre-embedding edges to post-embedding edges.

A pre-embedding edge is converted into zero, one or more post-embedding edges. Thus, the embedding mechanism specifies, for each possible pre-embedding edge,

- the direction of the post-embedding edges (if directed edges are used)
- the label of the post-embedding edges (if edge labels are used)
- the endpoints of the post-embedding edges. One endpoint is a node in RHS^{host}. The other endpoint is a RestGraph node, designated by a path that starts with a pre-embedding edge, and sometimes continues with a sequence of RestGraph edges.

Some embedding mechanisms permit unrestricted specification of post-embedding edges in terms of pre-embedding edges and RestGraph. Other mechanisms impose some restrictions, as discussed below. The choice of an embedding mechanism can involve a tradeoff between using fewer, but complex, productions, versus using a larger number of simpler rules.

Embedding mechanisms can be classified as follows [Nag79b] [Nag87]. This classification is central to the study of the generative power of graph grammars. For a more extensive and rigorous presentation, see [Roz97]. Examples of these embedding mechanisms follow this overview. From most to least powerful, an embedding mechanism can be:

Unrestricted Each LHS^{host} node has embedding paths specified for it. Each path starts at the LHS^{host} and follows a sequence of edges (starting with a pre-embedding edge), where each edge has a prescribed orientation and edge label. Following every such path that is present in this graph produces a set of RestGraph nodes. A subset of these nodes is chosen (based on node label) to act as endpoints for post-embedding edges. The directions and edge-labels of the post-embedding edges can be chosen freely. Textual and diagrammatic notations for unrestricted embeddings are presented below.

Orientation and Label Preserving (olp) As in the unrestricted case, but when we begin by following a pre-embedding edge with a certain orientation and edge-label, then all post-embedding edges we construct must have this same orientation and label. In the expression notation of Figure 2.2 below, the set In_i has to begin by following an I_i edge; the set Out_j has to begin by following an O_j edge.

Depth1 As in the unrestricted case, except that the RestGraph endpoints for post-embedding edges are restricted to the direct neighbors of LHS^{host}.

Simple Depth1, and Orientation and Label Preserving.

Elementary Simple, with the additional restriction that the embedding cannot depend on the labels of nodes in RestGraph. A pre-embedding edge can be identified only by its orientation, edge label, and the LHS^{host} node it connects to; if there are several such pre-embedding edges, they must all be transformed the same way, independent of the node label of the RestGraph nodes they connect to.

Analogous Elementary, and the embedding transformation is independent of the orientations and labels of the pre-embedding edges.

Invariant	There is a mapping between nodes of LHS and RHS such that RHS^{host} directly takes over the pre-embedding edges of LHS^{host}. This is the only type of embedding that does not allow the splitting and contracting of edges: the number of post-embedding edges equals the number of pre-embedding edges. Algebraic graph rewriting provides an important use of invariant embedding. It has a strong mathematical basis, with useful theorems concerning order-independence and parallelism in rule application.

2.3.1 Hierarchy of Graph Language Classes

The Chomsky hierarchy for string grammars is well-known [HoUl79]. String productions can be unrestricted, context sensitive, context free, or regular. Each successive restriction reduces the set of string languages that can be generated. An analogous situation exists for graph grammars. The set of generatable graph languages depends both on the allowed complexity of LHS in production rules, as well as on the power of the embedding specification [Roz97]. The most powerful embedding mechanisms allow search of the whole graph, attaching the RHS copy to any nodes in the graph. In this case, descriptive power is the same whether production rules are context sensitive or context free. This is described further below.

The power of a graph grammar depends on two factors:

(1) the type of embedding. The embedding can be unrestricted, olp, depth1, simple, elementary, analogous or invariant.

(2) the type of production.

unrestricted production:	no restriction on LHS or RHS.
monotone production:	number of nodes in LHS ≤ number of nodes in RHS.
context sensitive production:	a portion of LHS exists as a subgraph of RHS.
context free production:	LHS consists of exactly one node. (This node has a nonterminal label.).
linear production:	context free, and RHS contains at most one node with a nonterminal label. (Any number of terminal-labeled nodes are allowed in RHS.)
regular production:	context-free, and RHS has a unique maximum node. All other RHS nodes are predecessors of the maximum node. The maximum node may have a terminal or non-terminal label, all other RHS nodes have terminal labels.

Only the second factor has a counterpart in the Chomsky hierarchy.

Let T be a particular production type (unrestricted, monotone, context-sensitive, context-free, linear, regular). Once T is fixed, a hierarchy of graph language classes arise based on the embedding mechanism [Nag87]:

$$\text{unrestricted-T} \supseteq \text{olp-T}, \text{depth1-T} \supseteq \text{simple-T} \supseteq \text{elementary-T} \supseteq \text{analogous-T} \supseteq \text{invariant-T}.$$

The equalities hold when the productions are of type unrestricted. An interesting hierarchy of graph language classes arises when the embedding is fixed to be unrestricted, and the production type is allowed to vary. This is discussed in the next section.

Other types of graph grammars have been invented for parsing purposes. Precedence graph grammars are a subclass of context free graph grammars with rather restricted embedding specifications. They are closely related to precedence grammars and may be parsed in linear time [Kau83]. Layered graph grammars and reserved graph grammars on the other hand are subclasses of unrestricted graph grammars, which replace the requirement |LHS| <= |RHS| of monotonic graph grammars by a more elaborated alphabetic ordering criterion on LHS/RHS graphs. The language of layered graph grammars is decidable but requires an exponential parsing algorithm [ReSc97]. Reserved graph grammars (layered graph grammars with an additional local confluence criterion) have a parsing algorithm with polynomial time and space requirements [ZZ97].

2.3.2 Unrestricted Embeddings: Expression Notation

Nagl formalizes and generalizes embedding specifications using an expression notation [Nag79b] [Nag87]. Expression graph transformation operates on an (un)directed, edge-(un)labeled, node-labeled graph, with LHS^{host} required to be an induced subgraph. An embedding specification consists of 2n expressions, where each post-embedding edge has one of n edge labels, and one of two edge directions. The symbol In_i represents the incoming edges of label i, and Out_i represents the outgoing edges of label i. The notation is illustrated in Figure 2.2. This notation is quite difficult to read. Visual notations can be used instead, as discussed in Section 2.3.3.

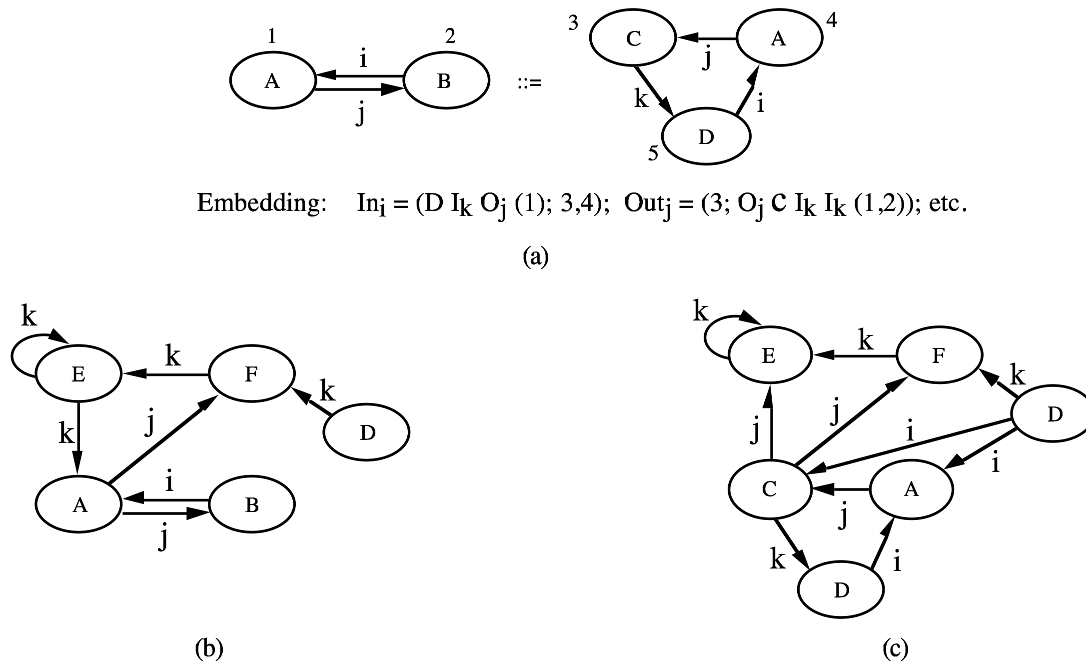


Figure 2.2 Production (a) uses the expression embedding notation defined by [Nag79b]. When this production is applied to host graph (b), the result is (c). Edge labels are written next to the edge. Node labels are written inside the node, and node denotations are outside. Clause In_i describes the construction of incoming post-embedding edges that have label i. The set of nodes "D I_k O_j (1)" are source nodes of i-labeled edges ending in nodes 3 and 4 of RHS^{host}. The expression "D I_k O_j (1)" describes the following set of nodes: starting from node 1 of LHS^{host}, follow an outgoing edge with label j (O_j) and then an incoming edge with label k (I_k); include those nodes that have the label D.

Similarly, clause Out_j describes outgoing post-embedding edges that have label j. The set of nodes "O_j C I_k I_k (1,2)" are target nodes of j-labeled edges originating in node 3 of RHS^{host}. The expression "O_j C I_k I_k (1,2)" describes the following set of nodes: starting from nodes 1 or 2 of LHS^{host}, follow a chain of two incoming edges with label k or one outgoing edge with label j.

Expression graph grammars have the following language hierarchy. (The language of a graph grammar is defined to be the set of all terminally labeled graphs that can be derived from the initial graph S.)

(A) The set of recursively enumerable graphs is generated by *unrestricted* productions with expression embedding.

(B) All of the following generate the same set of graphs (a strict subset of the graphs generated by (A), and a strict superset of the graphs generated by (C)):

- *monotone* productions with expression embedding

- *context sensitive* productions with expression embedding
- *context free* productions with expression embedding

(C) Both of the following generated the same set of graphs

- *linear* productions with expression embedding
- *regular* productions with expression embedding

Interestingly, whether LHS is restricted to be monotone, context sensitive, or context free does not affect the set of generatable graph languages. Certainly the Chomsky hierarchy for string languages leads us to expect a difference between context sensitive and context free productions. This does not occur here, because the expression mechanism provide a means of using context through the embedding, not just through complex forms for LHS.

It is difficult to understand the effect of an embedding expression. This motivated the development of visual notations for embeddings. Examples are the X, Y and Δ notations shown next.

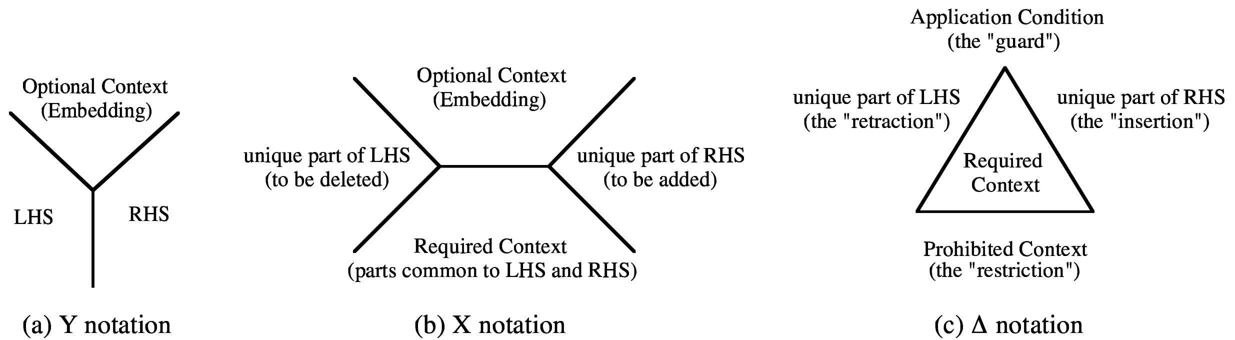
2.3.3 Diagrammatic Notations for Unrestricted Embedding: Y, X and Δ Notation

Several diagrammatic notations for embedding have been developed. Göttler's Y and X notation [Göt92] have been used in specifying and implementing diagram editors. Kaplan's Δ notation [KaLG91] [LoKa92] has been used for concurrent system specification. These diagrammatic notations can describe unrestricted embeddings, since they allow the following of paths within RestGraph to determine the source and target nodes of post-embedding edges. (However, some of the set operators that are available in expression notation are not available in the diagrammatic notations.)

An overview of Y, X and Δ notations is given in Figures 2.3, 2.4, 2.5 and 2.6. The latter example shows that diagrammatic notations for embedding are not always preferable to textual ones.

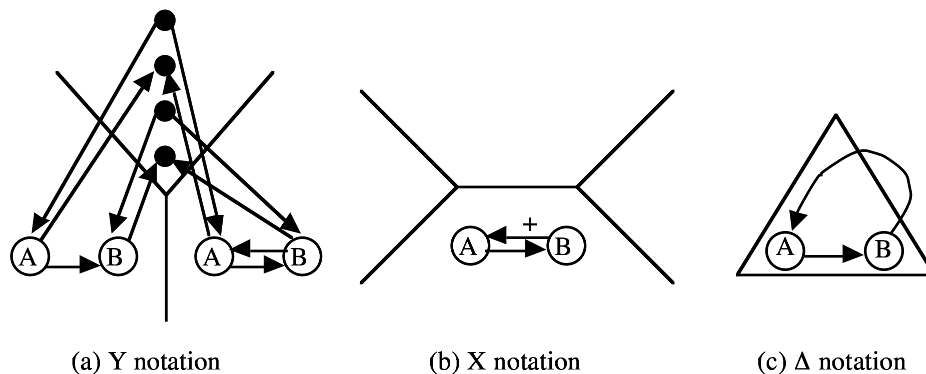
Briefly, Y notation contains a complete, separate drawing of LHS and RHS. This is found to be wasteful when graph productions have parts that are common to LHS and RHS. (These graph parts act as context for the production, but do not get altered by application of the production.) In contrast, X and Δ notations depict the common parts just once, in a separate area called the *required context*. The required context must exist for production application to occur, but it is left unchanged by the production.

In Δ notation the required and optional contexts are in the middle of the Δ (with a * next to each node of the optional context) The area below the Δ contains a new field, the *prohibited context*. This is a subgraph connected to LHS. If it can be matched to the host-graph area surrounding LHS^{host}, then rule application is forbidden. Thus Δ notation gives the application condition in two portions: graph-related restrictions are shown diagrammatically as the required and prohibited context, whereas attribute-related restrictions are given textually.



(a) Y notation (b) X notation (c) Δ notation

Figure 2.3 Three diagrammatic notations for graph productions. In Y and X notations, the embedding is shown as optional context: these diagrammatic depictions of embedding are used *if* they match in the host graph. The required context must match in order for the production to be applied. In Δ notation, the center of the Δ is used both for required and optional context, with a * placed next to the optional parts. (Elements of a * group may occur zero, one or more times.) The prohibited context depicts host-graph structure that must not be present; restrictions on labels and attributes are expressed textually in the guard.



(a) Y notation (b) X notation (c) Δ notation

Figure 2.4 Three diagrammatic notations for a graph production which adds a second edge between an A-labeled node and a B-labeled node. Avoiding duplication of graph-parts common to LHS and RHS shrinks the drawing of LHS and RHS, and greatly reduces the graphical depiction of the embedding. Using Y notation, LHS and RHS are represented separately, with eight edges and four nodes used to show the embedding. Using X notation, the common parts of LHS and RHS are shown as required context, the additional edge is indicated with a + sign, and no embedding depiction is required. The Δ notation is similar, but depicts the added edge as looping to the right of the triangle, into the insertion region. (This Y notation rule appears in [Göt92, Fig. 14].)

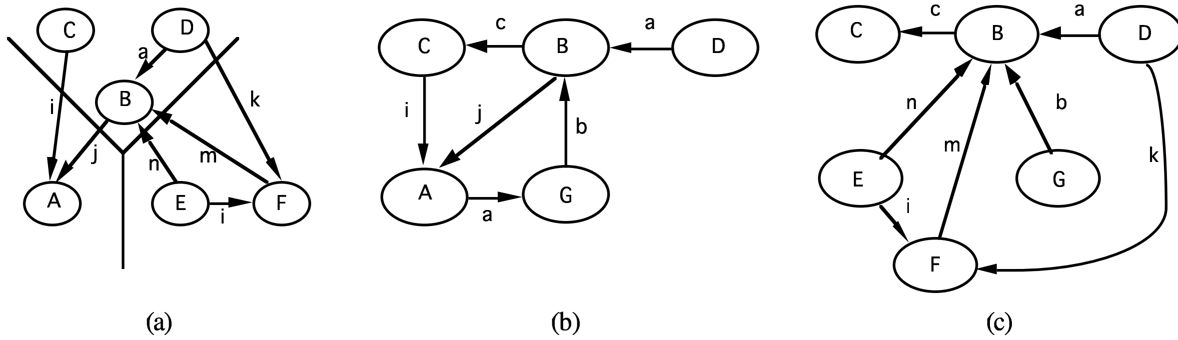


Figure 2.5 Diagrammatic depiction of a complicated embedding, using Y notation. Production (a) is applied to graph (b), resulting in graph (c). In (a), the embedding specification is shown in the top of the Y. Edges between LHS and the embedding specification match a subset of pre-embedding edges. Edges between RHS and the embedding specification show the post-embedding edges which are created as a result. (Edges directly between LHS and RHS are not permitted in this notation.) The embedding specification is an *optional context* -- after LHS^{host} has been located, the embedding specification is matched to the surrounding RestGraph. If node n' of the embedding specification finds a match in RestGraph, then any edge between n' and RHS causes the formation of a post-embedding edge. An optional-context node must have a connection to RHS in order to have an effect on the graph production. For example, node C and its i-labeled edge (in (a)) emphasize that the i-labeled edge is deleted (if present), but the production's effect would be the same without their inclusion in the embedding specification. The readability of this diagrammatic depiction of embedding can be compared to the textual expression notation of Figure 2.2a.

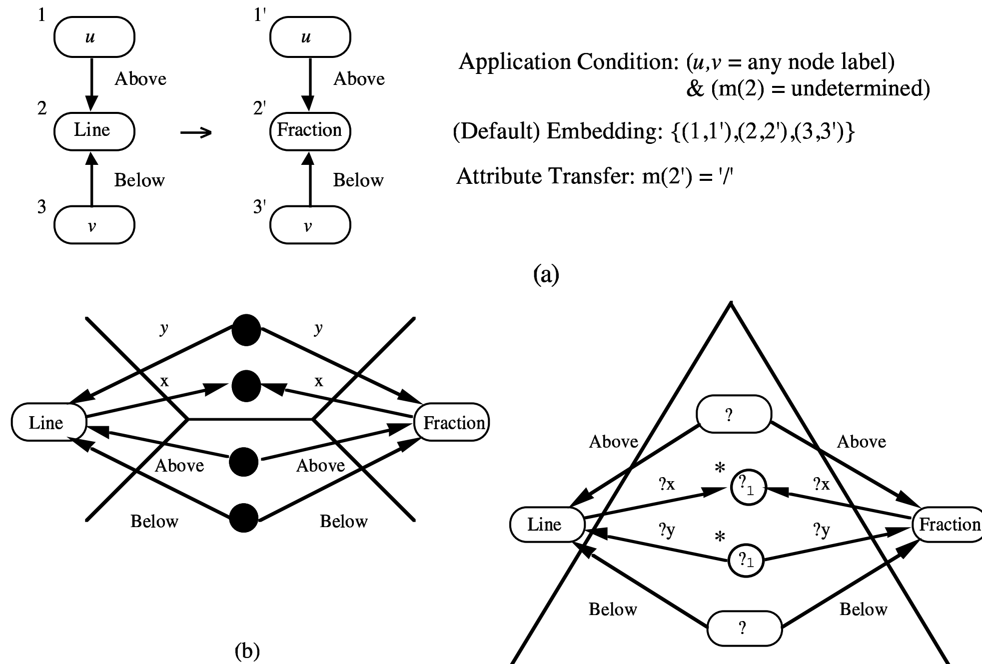


Figure 2.6 Specifying a straightforward embedding. In this case, a textual description of the embedding (a) can be simpler than a diagrammatic description (b) or (c). All three notations describe a graph production which replaces a *Line*-labeled node by a *Fraction*-labeled node, in the context of incoming *Above* and *Below* edges. (This production is used in [GrBl95], for recognition of mathematical notation.) In (a), the analogous embedding is easy to perceive, due to the correspondence in the placement of LHS nodes and RHS nodes. The embedding is formally specified by the textual description "{(1,1'), (2,2'), (3,3')}". The X notation in (b) conveys the embedding using the optional context region (top part of the X). A node correspondence is indicated via two edges and a filled-in node (which matches a node with any label). Since directed edges are used, this notation is repeated for incoming and outgoing edges. (c) In Δ notation, the embedding is conveyed similarly, using *-groups to indicate 0 or more occurrences of the starred structures. The subscripts on the ? node labels indicate that these two nodes can optionally match the same host graph node, handling the situation where a context node is related to the Line node via two edges, one incoming and one outgoing.

2.3.4 Depth1 Embeddings: NCE

In [JaRo82], Janssens and Rozenberg define graph transformation with neighborhood controlled embedding (NCE). This is a Depth1 embedding mechanism -- a post-embedding edge can only connect to RestGraph nodes that used to be connected to a pre-embedding edge. Edge labels may change, but edge-orientation is preserved in the transformation from pre-embedding edges to post-embedding edges. An embedding consists of two sets, In and Out. The set In applies to incoming edges (i.e., pre-embedding edges which terminate in LHS^{host}), while the set Out applies to outgoing edges. Each set contains quintuplets, as in this example:

In = {(1,3,a,b,E), (1,5,b,b,F)}

Here (1,3,a,b,E) applies to any pre-embedding edge that has label "a", and connects a RestGraph node with label E to LHS^{host} node 1. Such an edge causes formation of a post-embedding edge with label "b", connecting that same RestGraph node to RHS^{host} node 3.

2.3.5 Elementary Embeddings: Schneider's Notation

Early work with graph grammars [PR69] [Mont70] sparked interest in the embedding problem. In 1970, Schneider provided the first formal definition of the embedding problem; a summary can be found in [Nag79b]. Schneider's embedding specification consists of two sets In_i and Out_i, which specify the transformation of edges with edge label i. Each set contains pairs of the form (n,n'), where n denotes a node in LHS and n' denotes a node in RHS. This pair causes a pre-embedding edge connected to node n in LHS^{host} to be transformed into a post-embedding edge connected to node n' in RHS^{host}. The orientations, labels, and RestGraph-endpoints of the embedding edges remain unchanged. The notation simplifies if there are no edge labels (only two sets, In and Out, are needed) or if edges are undirected (each pair In_i and Out_i combine into one set).

2.3.6 Invariant Embeddings

Invariant embeddings establish a one-to-one correspondence between a subset of LHS and RHS nodes. These nodes, called gluing points, are not altered during application of the production. All embedding edges must connect to gluing points; rule application is disallowed at LHS^{host} if there is a pre-embedding edge connecting to a non-gluing node of LHS^{host}. The effect is that the set of embedding edges is unchanged by production application.

The restricted nature of the invariant embedding limits the expressiveness of a gluing production. A production expressed with a more complex embedding can be translated to an invariant embedding by suitable expansion of LHS and RHS. In the original rule, graph changes are accomplished by the embedding process as well as by subgraph replacement; in the new rule, all graph changes must be explicit in the expanded LHS and RHS. To enable such translation to invariant embedding, additional notation for LHS and RHS may be needed. Figure 2.7 shows an example: a production to "delete a node with label A". Using an analogous embedding, LHS is a single node, RHS is a null graph, and the embedding specification is null. To translate this to an invariant embedding, LHS must explicitly include all of the neighbors of node A. (These neighbors become gluing nodes.) Since we don't know the number of neighbors, we have to express LHS using some notation for replicated nodes and edges. This idea is incorporated into Δ notation: *-groups denote zero or more occurrences of starred graph elements. A Δ production that deletes a node is syntactic shorthand for an infinite collection of Δ production that meet the gluing condition [KaLG91, p478].

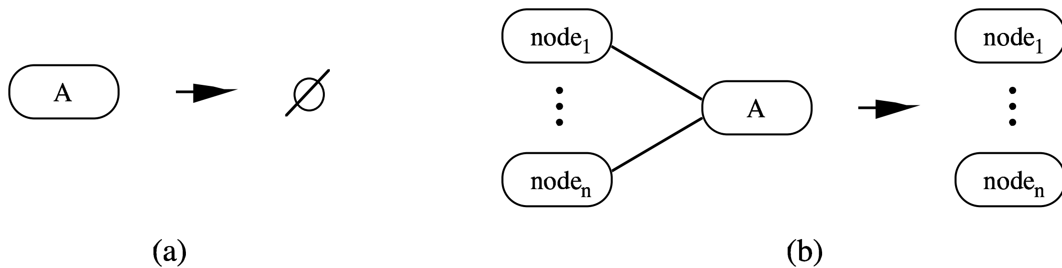


Figure 2.7 Production rules to delete an A-labeled node and all incident edges. (a) Elementary embedding mechanism. During rule-application, LHS is matched to an A-labeled node. When this LHS^{host} is replaced by RHS^{host} (an empty graph), all embedding edges are discarded. (b) A collection of rules in a gluing model, drawn using a shorthand notation. The invariant embedding of the gluing model necessitates that LHS be expanded to include all nodes neighboring the A-labeled node. Many productions are needed, to enumerate each possible configuration of incident edges. Here, the "..." notation indicates a match to any number of nodes and edges.

Because of the simplicity of invariant embedding, a graph production does not need to remove pre-embedding edges and replace them by post-embedding edges. Instead, it removes LHS^{host} except for the gluing nodes, and then

adds RHS^{host} except for its gluing nodes (connecting, instead, to the gluing nodes left by LHS^{host}). This carries over the embedding edges unchanged. As mentioned in Section 1.1.1, Algebraic graph rewriting uses category theory to model graph transformations, effectively using invariant embeddings [EhKL91] [Roz97]. Because of its strong mathematical basis, this approach has received much attention by those interested in the theory of graph grammars. The algebraic approach has been used in applications where concurrency and synchronization must be proven, for example in developing a small database system for managing library transactions [EhKr80] [EhHa86], and for high-level data-structure manipulation [Pfe90].

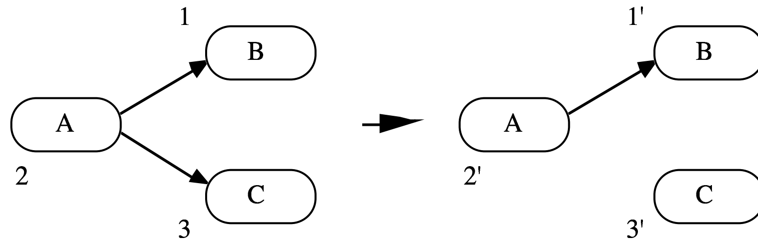
2.4 Other Aspects of Graph Productions

A variety of notations and mechanisms can be used for graph productions. The following sections discuss the use of induced versus non-induced subgraph isomorphism, the use of class hierarchies for node labels, homomorphic versus isomorphic graph matching, hierarchical host graphs, efficiency of production application, and tools and languages available for graph transformation.

2.4.1 Induced versus Non-induced Subgraph Isomorphism

A graph transformation system can use either induced or non-induced subgraph isomorphism. This choice has important consequences on the structure of graph productions.

The difference between induced and non-induced subgraph matching is illustrated in Figure 2.8. If LHS^{host} is an induced subgraph of the host graph, then LHS^{host} must include all local edges of the host graph (i.e. all edges that connect two LHS^{host} nodes). A non-induced subgraph may omit some or all of these edges.

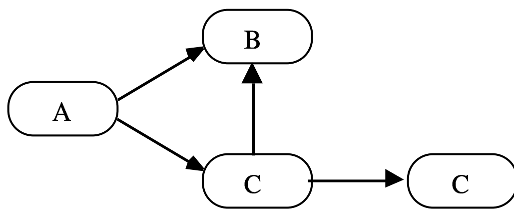


Application Condition: ((x(1)-x(2)) < (x(3)-x(1)))

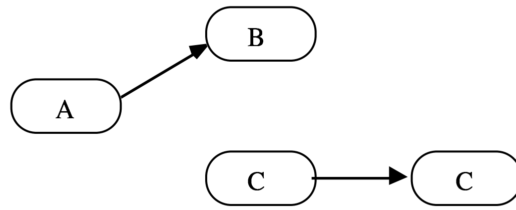
Embedding: {(1,1'),(2,2'),(3,3')}

Attribute Transfer: {ALL(1')=ALL(1); ALL(2')=ALL(2); ALL(3')=ALL(3)}

(a)

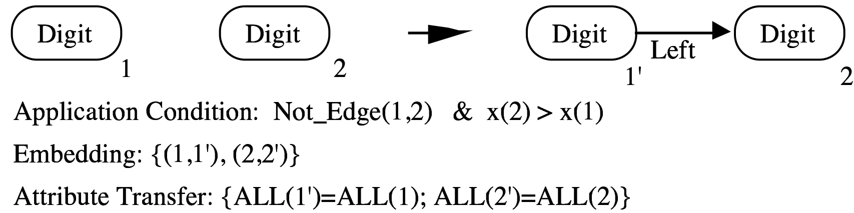


(b)



(c)

Figure 2.8 Induced versus non-induced subgraphs. Production (a) is applied to host graph (b). If an induced LHS^{host} is required, the isomorphism test fails and the production cannot be applied. On the other hand, if non-induced subgraph matching is used, a suitable LHS^{host} is found. Production application results in the new host graph (c). During production application, LHS^{host} is removed from the graph, and is replaced by RHS^{host}, so the edge from the C-labeled node to the B-labeled node is removed in from the host graph. This effect may not have been anticipated by the author of production (a).



Application Condition: Not_Edge(1,2) & x(2) > x(1)

Embedding: {(1,1'), (2,2')}

Attribute Transfer: {ALL(1')=ALL(1); ALL(2')=ALL(2)}

Figure 2.9 When LHS^{host} is a non-induced subgraph, an extra application condition may be required. This production, used in the [GrBl95] system, adds a Left edge to the host graph. A Not_Edge (1, 2) test is used in the application condition, to avoid the attempt to insert a Left edge when one is already present. Without this clause in the application condition, the production could be applied repeatedly. This would waste time (perhaps even cause an infinite loop), but would not result in parallel Left edges. Any existing edges between nodes 1 and 2 are discarded when LHS^{host} is removed. If LHS^{host} were required to be an induced subgraph, then the Not_Edge(1,2) condition is no longer needed.

Compared to non-induced subgraphs, induced subgraphs meet more stringent matching criteria, and provide more information about local host graph structure. The following consequences result.

- Using induced subgraphs increases the number of productions: LHS cannot match unless the rule-author has anticipated all the edges present in that part of the host graph. If a variety of edge-configurations are possible, these must be enumerated in separate graph productions (where a single non-induced production could suffice).

- Non-induced subgraphs require extra application conditions, necessary to ensure the absence of certain host-graph edges (Figure 2.9).

- Unanticipated edge-deletion is a major pitfall of non-induced subgraphs. Edges present in host-graph but not mentioned in LHS are deleted by rule application (as in Figure 2.8).

These points become particularly significant in case of host-graph evolution. Suppose a graph transformation system is being extended, by adding a new type of edge with the edge label “Grow”. Ideally, the old graph productions should continue functioning as before, so that we merely need to create a few new rules that directly process the Grow edges. Both induced and non-induced subgraphs disappoint us.

- Using induced subgraphs, the presence of a Grow edge prevents application of any of the old rules. The old rules must be replicated, to enumerate all possible permutations of Grow edges that might occur in the LHS^{host} area.

- Using non-induced subgraphs, the old graph productions continue to apply, but they perform implicit deletion of Grow edges. Productions apply whether or not a Grow edge is present, but if a Grow edge was present before rule application, it is no longer present after rule application.

These problems are independent of the embedding mechanism, arising similarly in all models that use removal of LHS^{host} during the transformation step. Improved semantics can be defined by using non-induced subgraph matching and avoiding node deletion where possible. This means that whenever LHS^{host} and RHS^{host} contain corresponding nodes, these nodes are preserved during production application. Preserved nodes are not removed as part of LHS^{host} node and reinserted as part of RHS^{host}; instead they are left unchanged by production application. Such semantics (incomplete removal of non-induced subgraphs) are provided in the definition of structured graph transformation [KrRo90], and in the PROGRES language. Many graph-transformation papers give scant mention of their choice to use induced or non-induced subgraph matching. This issue is important both theoretically and practically.

2.4.2 A Class Hierarchy for Node Labels

Both node and edge labels can be used to represent information in a graph. For subgraphs to be considered isomorphic, they must match not only in structure, but also in labeling. The set of possible labels can be organized as a flat set or as a hierarchy of label classes. Using flat labels, an X-labeled node in LHS must match an X-labeled node in the host graph. Using hierarchically structured labels, an X-labeled node in LHS matches any host graph node whose label occurs at or below X in the label tree.

Hierarchical node labels have been used in several graph transformation systems [ELNSS92] [Göt92]. The PROGRES language allows multiple-inheritance among node-classes, resulting in lattice-structured rather than tree-structured node labels [ELNSS92]. The distinction between labels and attributes is blurred in Δ notation [KaLG91]. Here labels are tuples of arbitrary structure, and hence combine the functions otherwise divided among labels and attributes. Unification is used when matching variable labels in a Δ production to labels in the host-graph. Advantages of a hierarchical label structure are as follows.

Inheritance for the definition of attributes. Every node label has an associated set of attributes. The definition of available attributes is simplified by the use of inheritance among classes of node labels. A label class inherits all of the attributes of its parent label, and may have additional attributes defined specifically for it.

LHS nodes match with a variable degree of specificity. With hierarchical node labels, each production rule can be designed to match node labels with a greater or lesser specificity. An LHS node that has a node-class label high in the inheritance tree will match many host graph nodes. An LHS node with a more specific node-class label will match a much narrower set of host graph nodes. In contrast, difficult design choices must be made when a flat label set is used.

- Option 1: Use many labels to make fine case distinctions.
- Option 2: Use a small number of general labels, augmented by attributes for fine case distinctions.

Efficiency problems arise under both options. These arose in our design of a graph transformation system for recognition of math notation [GrBl95]. This system was first designed using flat label sets; a later translation to PROGRES gave us access to hierarchical label sets (Chapter 4). With flat label sets, our options were as follows.

- Use highly-specific node labels, such as *a*, *b*, *c*, *1*, *2*, *3* for selected letters and digits.
- Use general node labels, such as *letter*, *number*, *digit*, with node attributes recording which character gave rise to the node.

Graph transformation can involve operations on individual letters, or on all letters. The above options are suitable for one of these operations but not the other.

- Using highly-specific labels, it is expensive to perform an operation on all letters. A separate production rule can be written for each letter, or wildcard node-labels can be used in LHS.
- Using general labels, it is expensive to search for a particular letter. This has to be done, for example, by a production rule that looks for the sequence *cos* to interpret it as the name of a trigonometric function, rather than as the implied multiplication *c*o*s*. Here LHS contains a node labeled *letter*, with an application condition to specify the desired attribute value “c”. The search for LHS^{host} is quite inefficient under standard implementations, since matches to all *letter* nodes in host-graph are undertaken, with most matches rejected by the application condition. It is possible, but cumbersome, to make this search efficient by customizing the subgraph-isomorphism code so that application conditions limit the search space for suitable LHS^{host}.

The use of hierarchical node labels solves these problems, allowing LHS nodes to match with a variable degree of specificity.

2.4.3 Homomorphism versus Isomorphism

Most commonly, a subgraph isomorphism test is used when finding a subgraph that matches LHS. Alternatively, a general graph morphism can be used. The utility of general graph morphisms is illustrated by small examples in the literature ([EhHK92, p. 560], [KrRo90a, p. 200]). However, the use of general morphisms might result in productions that match the host graph in unintended ways. A useful compromise is to selectively indicate where general morphisms may be used. For example, Δ graph rewriting uses subgraph isomorphism, but with a label-subscript notation (called a *fold*) to explicitly indicate groups of nodes and edges which can be matched to the same host-graph entity [KaLG91] [LoKa92]. A *fold* construct is also available in PROGRES. The utility of this construct is demonstrated by a production to insert an element into a circular list. (In the example, this is a circular list of nodes representing philosophers and forks, used to simulate the dining philosophers problem.) The production works on non-empty lists of any length. When matching long lists, each LHS node maps to a unique host-graph node, but a short list causes several LHS nodes to map to one host graph node. This construct provides controlled deviation from strict isomorphism: a rule-author selectively and explicitly indicates where LHS node identifications are permissible.

2.4.4 Hierarchical Graphs and Distributed Graph Rewriting

Hierarchical host-graph structures arise naturally in many applications. In a strict definition of hierarchical graphs, all edges must connect siblings, or connect a parent and a child node. However, many practical problems cannot be modeled without additional edges that cross the hierarchy, for example to connect “cousin” nodes. The presence of such hierarchy-crossing edges greatly complicates the construction of tools for hierarchical graph transformation. Various notations for hierarchical graph structures are described in [Hare88] [SiGJ93]. Hierarchical structure assists in the display of a large graph. Zoom-in and zoom-out operations reduce the graph to manageable proportions for viewing, or delimit selected portions of the graph for processing.

There is significant interest in the topic of hierarchical graph transformation. Relevant references include a chart-based parser for hierarchical-graphs [MaKl92]; abstract graphs in a prototype algebraic-transformation environment [LöBe93]; graphs where node labels can be graphs themselves [Sch93]; and flat host-graph structure with hierarchy-expressing productions used to zoom in and out [EhHK92].

Distributed graph rewriting is an interested research topic that is related to the use of hierarchical graphs [Taen96] [Taen96b]. A distributed graph is a two-level hierarchical graph, where the fine-grained level contains the graph *per se* and the coarse-grained level describes the partitioning of the graph and the relationships between the partitions. Distributed graph rewriting systems are an interesting approach for modeling dynamic, distributed data structures. There are not yet any implementations of distributed graph transformations.

2.4.5 Efficiency of Production Application

Graph transformation can be computationally expensive. Nevertheless, it is useful in the specification and design stages of software development, as well as in the construction of a prototype. Final implementation may have to resort to another programming language, in the interest of faster execution. The execution speed of graph transformations depends heavily on the characteristics of the graph productions, as well as on the control structures which invoke them.

A subgraph-isomorphism test is involved in the execution of every graph production. This test locates a part of the host graph which has the same structure as LHS; node labels and edge labels must match, but attribute values can differ. Subgraph-isomorphism testing is an NP-complete problem in general, but various factors make it tractable in graph transformation. Usually the LHS graph in a graph production is small. This greatly reduces the search space for an isomorphic subgraph. There are polynomial algorithms for subgraph isomorphism given a fixed upper bound on the size of the subgraphs to be matched. Also, node labels, edge labels, and directed edges drastically

reduce the search space for isomorphic subgraphs. In some applications, subgraph isomorphism search is eliminated by keeping a unique *cursor node* (or *demon node*) in the host-graph, which indicates where the graph production should be applied. For example, this is a way to keep track of the insertion point in an editing application [Göt92] [ELNSS92]. Production rules used for tool specifications usually match in constant time [Zün96b]. For the mathematics-recognition example, Build rules match in $O(n^2)$ time and the remaining rules generally take $O(n)$ time. Many techniques can be used to optimize subgraph-isomorphism testing [Zün96b] [BuGT91].

2.4.6 Tools and Languages for Graph Transformation

Many notations have been devised for graph transformation [IWGG] [Roz97] [BlFG95], but only some of them are supported by generally-available tools. Links to available tools, including PROGRES, may be found at the URL in reference [URL98]. These include tools for generating pictures and films using collage graph grammars (BIZARR2 and The Collage System); a general purpose, pure, lazy, functional programming language implemented using transformations of directed acyclic graphs (Concurrent CLEAN); a generator for diagram editors using hypergraph grammars (DiaGen); a database of biochemical compounds which uses layered graph grammars as its modeling language (Klotho); a generator for compiler optimizers which combines DATALOG and graph transformation (OPTIMIX); a pattern matching language which includes term and graph transformation (PROP); and an extension of ANSI C for matching and transformation trees and graphs (SMART).

All graph transformations in these notes are node oriented. Edge and hyperedge graph transformations have been investigated as well [Hab92].

2.5. Methods of Controlling the Application of a Set of Graph Productions

Issues concerning individual graph productions are discussed above. A computation is carried out by a collection of graph productions. These productions can be organized in a variety of ways, which we call unordered graph transformation, graph grammars, ordered graph transformation (with and without backtracking), and event-driven graph transformation.

2.5.1 Unordered Graph Transformation Systems

An *unordered graph transformation system* consists of a collection of graph productions. A given graph is transformed by nondeterministically choosing the next production to apply, until no further production application is possible.

Unordered graph transformation is used in the Δ notation introduced in Section 2.3.3. The transformation system is given an initial host graph to which productions are applied. For the quicksort example of [LoKa92, p. 177], the initial host graph is a list of numbers to be sorted. For the specification of the Actor language [KaLG91, p. 484], the initial host graph is compiled from an Actor program. For the dining philosophers example (a dynamic version in which philosophers can join or leave the table) [LoKa92, p. 112], the initial host graph represents five philosophers and five forks. The initial host graph may be transformed by a finite number of production applications (as in the quicksort example), or with indefinite application of productions (as in the dining philosopher example).

To divide a large problem into more manageable subproblems, Δ productions are organized into *platforms* of related rules. The interface between platforms is provided by *triggers*, special host-graph nodes that are used to

trigger the application of rules in a platform. Every rule in a given platform contains this trigger node as part of its LHS. To invoke rules belonging to platform G, the G trigger is placed somewhere into the host graph. This satisfies one of the preconditions of rule-application from platform G; of course, successful rule-application from platform G also depends also on proper matching of LHS^{host}. The label of the trigger node is a tuple of arbitrary structure, and can include parameters to influence the resultant application of a G-platform rule. This style of computation has been used to solve a variety of specification and concurrency problems. The Δ notation hides many synchronization details from the programmer. A variety of techniques for proving properties such as deadlock freedom and termination are presented in [LoKa92]. The structuring of Δ productions into platforms is helpful for proof construction; for example, a different proof method can be used to demonstrate termination of each platform.

Unfortunately, there is no mention of plans to implement an environment for Δ notation; current experience is limited to paper-based descriptions of Δ transformation systems. Implementation of Δ program execution involves three steps [LoKa92, p. 100]: (1) *matching* (find a set of applicable production, each with a suitable LHS^{host} and a unification of variable labels); (2) *conflict resolution* (find a non-conflicting subset of the applicable productions); and (3) *application* (apply the non-conflicting rules in parallel). Conflict resolution seems difficult to implement, since it must guarantee fairness, it must permit maximal parallelism, and it must prevent parallel application of rules A and B if application of rule A modifies the host graph so that rule B is no longer applicable. (A "prohibited context" greatly complicates this latter test: rule A might add host-graph structure that matches rule B's prohibited context.)

2.5.2 Graph Grammars

A *graph grammar* consists of a set of productions, a start graph, and a designation of labels as "terminal" or "nonterminal" [ReSc97]. In generative use of a graph grammar, productions are applied to the start graph until a terminal graph results. For recognition, a parser determines whether a given graph can be derived from the start graph. Unless the form of graph productions is severely restricted, backtracking is necessary during parsing. The parser can operate either top-down, beginning with the start graph and attempting to transform it into the target graph, or bottom-up, beginning with the target graph and applying the production rules (in reverse direction) in order to transform it into the start graph. In practice, the use of attribute computations makes productions applicable only in the forward (top-down) or reverse (bottom-up) direction. Thus the style of the productions limits the applicable parsing algorithms.

In a pure graph grammar, productions can be listed in any order. However, order-dependence often arises in practice. Once a developer has chosen a particular parser, the developer is usually aware of the order in which the parser tries alternatives. The developer may make use of this to design a smaller or faster graph grammar. For example, Anderson [Ande77] uses a set-based "coordinate grammar" (not a graph grammar) to recognize mathematical notation. He describes his reliance on production-rule ordering to distinguish an input "cos" as a word denoting a trigonometric function, rather than as an implied multiplication denoting "c*o*s". It would be possible to rewrite the grammar to avoid this order dependence, but the grammar would increase in size and complexity. The drawback of such order dependence is that the language is no longer defined by the grammar alone, but arises through the interaction of the grammar with a particular parser.

2.5.3 Ordered Graph Transformation Systems

Ordered graph transformation systems (elsewhere called *programmed graph grammars*) consist of productions and a control specification [Bun82]. There is no need for a distinction between terminal and nonterminal labels, since the control specification indicates when production application should terminate. The PROGRES language supports ordered graph transformation, using procedural control constructs to invoke productions. The success or failure of production application influences future control flow [ZüSc91]. For example, a PROGRES loop can be written to apply one or more productions as often as possible, iterating until all productions fail. Similarly, a conditional

statement can be written to take one action upon successful production application, and another action upon failure. Productions can be grouped into *transactions*, which succeed or fail as a whole.

Nondeterminism arise from two sources in an ordered graph transformation system. Firstly, there is nondeterminism in the selection of a particular LHS^{host} when there are several possible matches for LHS. Secondly, nondeterministic constructs can be used in the control specification. If there is non-determinism, it may be sufficient merely to follow any of the possible execution paths, or it may be necessary to backtrack in case a path fails. PROGRES programs can be executed with or without backtracking. For example, backtracking is used to search the space of possible moves in the Ferryman's Problem, in which a goat, wolf and cabbage are transported across a river [ZüSc91]. Backtracking involves significant overhead. In applications such as diagram recognition, ordered graph transformation (without backtracking) is found to be very efficient compared to parsing with a graph grammar [Bun82]. The ordered graph transformation can directly transform an input graph into an output graph. The control structure limits the number of productions that are under consideration, reducing the number of subgraph isomorphism tests that are needed to find the next applicable production. The system is written such that all non-deterministic alternatives lead to a desired result.

Completely deterministic graph transformation results from using a deterministic control specification, paired with the use of cursor-nodes (also called demon nodes) to indicate the desired host graph location for rule application. Such determinism is desirable in editing applications, since end users expect a deterministic response to an editing command. For example, in [Göt92] each graph production LHS includes a cursor node, which is matched to the unique host-graph cursor node; RHS leaves a new cursor node in the host graph. (Overall control in this editing system is event-driven; short sequences of ordered, deterministic graph transformations are used to respond to an individual editing command. When the end user selects an insertion point in the diagram, this moves the cursor node in the underlying graph representation.)

2.5.4 Event-driven Graph Transformation Systems

Event-driven graph transformation systems have an ordering imposed by an external sequence of events. For example, events related to a library database are the loaning, returning, or ordering a library book. Each event results in the invocation of a corresponding production rule [EhKr80]. Editors are event-driven as well. A more general control specification can be used, to define which set of editing events are legal at any given point [DoTo88] [Göt92]. Event-driven systems can be quite efficient since they can be almost fully deterministic. External events are used to select the next production to apply, and little search of the host graph is needed since there is a known insertion point.

It is possible to combine ordered graph transformation with event-driven graph transformation, as in the Forrester-diagram editor of [DoTo88]. Here the control specification (which uses host-graph inspection) is used to describe which editing events are legal at any given point. Events not foreseen by the control specification are disallowed, resulting in an error message to the user. A similar structure is used by the diagram editors described in [Göt92].