Technical Report No. 95-373

Practical Use of Graph Rewriting

Dorothea Blostein, Hoda Fahmy, Ann Grbavec

January, 1995

Department of Computing and Information Science
Queen’s University, Kingston, Ontario, Canada   K7L 3N6

Contents

Practical Use of Graph Rewriting

Dorothea Blostein, Hoda Fahmy, Ann Grbavec
Department of Computing and Information Science
Queen's University, Kingston, Ontario, Canada K7L 3N6
blostein@qucis.queensu.ca

Graphs provide an expressive and versatile data representation. Typically, nodes represent objects or concepts, and edges represent relationships among them. Hierarchical relationships can be depicted by node-nesting [Hare88] [SiGJ93]. Auxiliary information is expressed by adding attributes to nodes or edges. Given the widespread use of graphs as a data representation, it is natural that graph manipulations form the basis of many useful computations. Graph manipulations can be represented implicitly, embedded in a program that, among other things, constructs or modifies a graph. Alternatively, graph manipulations can be represented explicitly, using clearly-delineated graph rewriting rules that modify a host graph. The explicit use of graph-rewriting rules offers several advantages. Graph rewriting provides an abstract and high-level representation of a solution to a computational problem. Also, the theoretical foundations of graph rewriting ease the task of proving correctness and convergence properties.

Graph rewriting has the potential to be useful in a large variety of applications. However, it is difficult for software designers to evaluate the merits of using graph-rewriting solutions to their software problems. The literature for graph rewriting is extensive and highly technical. Here we present an intuitive overview of graph rewriting, with emphasis on application to practical problems. Detailed definitions and proofs may be found in the references. Section 1 reviews the mechanisms and notations that are available for expressing a single graph-rewriting rule, and discusses practical considerations related to the selection of a particular rewriting mechanism. Graph-rewriting rules are extended in Section 2, to include attribute processing, application conditions, parameters, and variables. Unordered rewriting, graph grammars, ordered rewriting, and event-driven rewriting (Section 3) provide alternative methods for organizing a computation that centers on graph-rewriting. Section 4 discusses considerations related to the practical application of graph rewriting; these include methods of graph inspection, efficiency considerations, tool availability, and the inclusion of graph rewriting sections within a larger software system. Existing graph-rewriting systems (Section 5) illustrate the wide range of methods used in practical application of graph-rewriting.

If graph rewriting is to be used in an application, one has to choose a particular graph-rewriting mechanism, and a particular notation in which to express the rewrite rules. The choice is large, as seen in Section 1. In addition, it is of great practical importance is to decide *how* the desired computation can be expressed using graph rewriting. What role do attributes play? Should the rewrite be organized as a graph grammar, an unordered rewriting system, or an ordered rewriting system? (This overview concentrates on sequential graph rewriting, but parallel rewriting is more appropriate for some problems.) What methods can be used to organize a large collection of rewrite rules, so that they remain intellectually manageable, and can be debugged and updated? How can information be structured in a graph? For example, how can alternatives or uncertainty be represented and manipulated? As of yet, there are no general answers to these questions.

Existing graph-rewriting systems provide a useful source of information and inspiration. We are motivated by our personal experience in applying graph rewriting to diagram recognition. (The goal of diagram recognition is to extract the meaning conveyed by an image of music notation, math notation, circuit-diagram notation, and so on [ICDAR93]). Graph rewriting is a powerful tool with a strong theoretical basis. Further research is needed to encourage practical use of graph rewriting, to give this computational tool the accessibility it deserves.

1. Graph Rewriting Mechanisms

A graph-rewriting rule is applied to a host graph to replace one subgraph by another. The sequential graph-rewriting mechanisms we consider here involve the replacement of one subgraph at a time; parallel rule application is considered only briefly. Sequential and parallel rewriting are reviewed by [Nagl79]. Other reviews and overviews include [Nagl87] [Ehri87] [KrRo90ab] [EhKL91] [FaBl92].

This research is supported by Canada's Natural Sciences and Engineering Research Council.

Graph-rewriting requires a decision on how to attach the new subgraph to the host graph. This can be done via an explicit *embedding specification* or implicitly by a gluing isomorphism. Kreowski and Rozenberg combine these two approaches in their definition of a structured graph rewriting rule; their presentation [KrRo90a] [KrRo90b] is recommended as a formal introduction to the embedding and gluing approaches to graph rewriting. Before reviewing these approaches, we must establish terminology, since varied terms are used in the literature.

1.1 Terminology

Terminology for graph rewriting is not standardized; we use the following terms. Note that these definitions incorporate some assumptions (for example, the use of subgraph isomorphism, where some models actually allow for a general graph morphism).

Graph g	A directed or undirected graph. Nodes and/or edges may be labeled, and may have associated attributes.
Graph Rewrite Rule	A rule specified by:
• $\mathbf{g_l \rightarrow g_r}$	g_l and g_r are unattributed graphs. A subgraph isomorphic to g_l is to be replaced by one isomorphic to g_r
• **Embedding information**	This can be textual or graphical (Appendix⊠.1 to A.4) Gluing models specify embedding with a gluing isomorphism (Appendix⊠.5)
• **Application condition**	Restrictions on rule application (Section☐3). 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 (Section☐2). Optional. Attribute values for g_r^{host} are computed from attribute values of g_l^{host}
Host graph g	The graph to which a rule is being applied.
$\mathbf{g_l^{host}}$	A subgraph of the host graph g, isomorphic to g_l. In some models, g_l^{host} must be an *induced* subgraph: if an edge of g connects two nodes of g_l^{host}, then that edge must be part of g_l^{host}.
RestGraph	The graph g - g_l^{host}. (This "-" operator denotes removal of all nodes and edges of g_l^{host}, as well as edges with one or both endpoints in g_l^{host}.)
$\mathbf{g_r^{host}}$	A subgraph isomorphic to g_r; used to replace g_l^{host}
Pre-embedding edges	the set of edges joining g_l^{host} to RestGraph (These edges "embed" g_l^{host} in g)
Post-embedding edges	the set of edges joining g_r^{host} to RestGraph

The basic primitive being manipulated by graph rewriting varies depending on the approach. It may be a node (Appendix⊠.1 to A.5.1), an edge (Appendix⊠.5.2), or a hyperedge (Appendix⊠.5.3). Most approaches assume that the basic primitive being acted upon is a node.

1.2 Classification of Embedding Mechanisms

A graph rewriting rule implicitly or explicitly provides embedding information. Gluing models use the gluing isomorphism to implicitly specify an invariant embedding. With explicit embedding models, graph rewriting is accomplished by locating g_l^{host}, removing it along with all pre-embedding edges, and then constructing post-embedding edges according to the 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 needs to specify, 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 (zero, one or more sets of endpoints). One endpoint is a node in g_r^{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, rewrite rules, versus using a larger number of simpler rules. Nagl developed the following classification of embedding mechanisms [Nagl79] [Nagl87], informally summarized here from most complex to least complex. This classification imposes a useful organization on the many rewrite mechanisms reviewed in Appendix⊠.

Unrestricted – From any g_l^{host} node, we can follow a sequence of edges (starting with a pre-embedding edge), conditional on edge orientations and edge labels, ending at a set of RestGraph nodes. A subset of these nodes can be 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.

Unrestricted embeddings can be expressed by the textual expressions discussed in Appendix⊠.1, or by the diagrammatic notations presented in Appendix⊠.2.

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⊠.1, 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 g_l^{host}. A well-known example of a Depth1 embedding mechanism is Neighborhood Controlled Embedding (NCE), defined by [JaRo82] (Appendix⊠.3). NCE applies the additional restriction that the embedding is orientation-preserving (but not necessarily label-preserving).

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 g_l^{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. Schneider's embedding mechanism (Appendix⊠.4) is of the Elementary type.

Analogous – Elementary, and the embedding transformation is independent of the orientations and labels of the pre-embedding edges. For example, consider the application of Schneider's embedding to an undirected, unlabeled graph.

Invariant - There is a mapping between nodes of g_l and g_r such that g_r^{host} directly takes over the pre-embedding edges of g_l^{host} (e.g. [Mont70]). 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.
The gluing models of Appendix⊠.5 provide an important use of invariant embedding. These models have a strong mathematical basis, with useful theorems concerning order-independence and parallelism in rule application.

This classification characterizes the class of graph languages generated by a graph grammar. Given a particular production type T (such as context-free or context-sensitive, Figure⊠.2), a hierarchy of graph language classes arise based on the embedding mechanism [Nagl87]:

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

The equalities hold when the productions are of type unrestricted [Uesu78].

At this point it is appropriate to give more detailed consideration to the various graph rewriting mechanisms. Please refer to Appendix⊠, which reviews the following topics.

- Expression notation provides unrestricted embeddings. The graph-language hierarchy results (Figure A.2).
- Diagrammatic notations provide nearly unrestricted embeddings. The Y, X, and Δ notations use *required context* to notate graph parts common to g_l and g_r, *optional context* to notate the embedding, and *prohibited context* to notate disallowed host-graph structure (which other rewrite mechanisms include as part of the application condition).
- NCE (Neighbourhood Controlled Embedding) and NLC (Node Label Control) provide Depth1 embedding, and have important theoretical properties.
- Schneider's mechanism, defined in 1970, provides an elementary embedding.
- Gluing models use an invariant embedding, specified via the gluing isomorphism. Gluing models include the algebraic approach, edge-replacement systems, and hyperedge replacement systems.
- Structured graph rewriting is a comprehensive model that combines embedding and gluing into one framework. Review of this model is highly informative.

Given this bewildering diversity of rewriting mechanisms, it is important to consider the practical issues involved in choosing one particular mechanism.

1.3 Practical Considerations for Selecting a Rewrite Mechanism

No single rewriting mechanism is suitable for all applications. The choice of rewriting mechanism depends greatly on the application, on the availability of tools, and on the tastes of the system designer. Analogously, a variety of programming languages continue to be in widespread use, despite the existence of general-purpose programming languages.

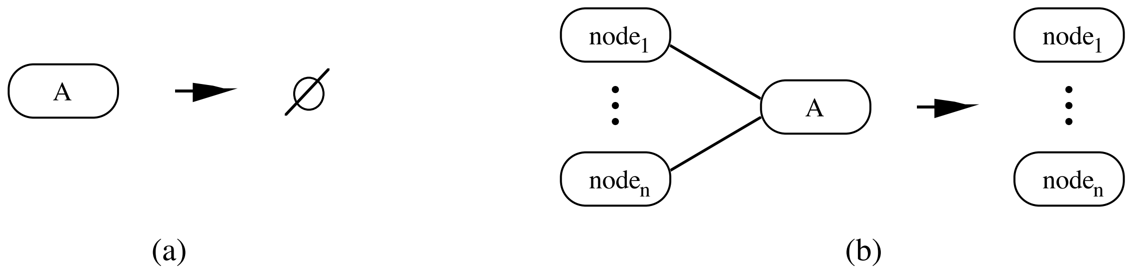
Many factors are relevant in choosing a rewriting mechanism, including the power of the embedding, formal properties of rewrite rules, readability and intellectual manageability, efficiency of rule application, and tools for rule development and debugging. We include personal observations, influenced by our experience in applying graph rewriting to diagram recognition. While no definite recommendations can be made, practical use of graph rewriting can be furthered by clarifying the issues involved in selecting a rewrite mechanism.

1.3.1 Power of Embedding

The complex embedding mechanisms permit significant graph inspection and graph manipulation during the embedding step. The most restricted embedding mechanisms, such as the invariant embedding of the gluing models, are inconvenient for expressing certain common graph operations. For example, consider the operation of deleting an A-labeled node from the host graph. All edges incident on the A-labeled node are to be deleted also. This operation is easily accomplished using an elementary embedding, but is difficult to accomplish with an invariant embedding (Figure□).

The choice of an embedding mechanism involves a tradeoff between using fewer, but complex, rewrite rules, versus using a larger number of simpler rules. Some graph-rewrite applications may call for a natural level of embedding complexity. For example, in reading math notation, we use an elementary embedding mechanism [Grba94], but use of a complex embedding mechanism would not reduce the number of rules in our particular rule set. Change to a gluing model does greatly increase the number of rules, as illustrated by Figure□b.

We have not come across practical examples of graph-rewrite systems that make heavy use of complex embeddings. It appears that many software designers find it is easier to express a computation using more rules of a restricted embedding type. We cannot judge whether this is inherent to the problem domains, or whether this arises because we have insufficient knowledge for taking practical advantage of complex embeddings.



Figure□ Rewrite rules to delete an A-labeled node and all incident edges. (a) Elementary embedding mechanism. During rule-application g_l is matched to an A-labeled node. When this is replaced by g_r^{host} (an empty graph), all embedding edges are discarded. (b) Collection of rules in a gluing model, using a shorthand notation. The invariant embedding of the gluing model necessitates that g_l be expanded to include all nodes neighboring the A-labeled node. Many rewrite rules are needed, to enumerate each possible configuration of incident edges. (The "..." notation denotes a variable number of nodes and edges, and is adapted from [EhHK92, p. 568]; Ehrig et al. do not mention an interpreter for this notation. The *-groups of Δ-rewriting, which denote zero or more occurrences of starred graph elements, can be used to implement node-deletion in a gluing approach [KaLG91, p478]. A Δ-rule that deletes a node is syntactic shorthand for an infinite collection of Δ-rules that meet the gluing condition.)

1.3.2 Formal Properties

Formal properties of graph rewriting are practically important. The strong theoretical foundations of the gluing models can offer significant advantages. For example, algebraic graph rewriting allows easy construction of proofs about the integrity of a database system; the library-transaction system of [EhKr80] provides an example. However, the gluing models are appropriate only if invariant embedding suffices for conveniently expressing the needed graph rewriting.

Graph rewriting can be used to perform local transformations that preserve well-formedness properties of the host graph. For example, graph rewriting has been used to formally define the class of well-formed Forrester diagrams [DoTo88], and the class of well-formed semantic networks [EhHK92].

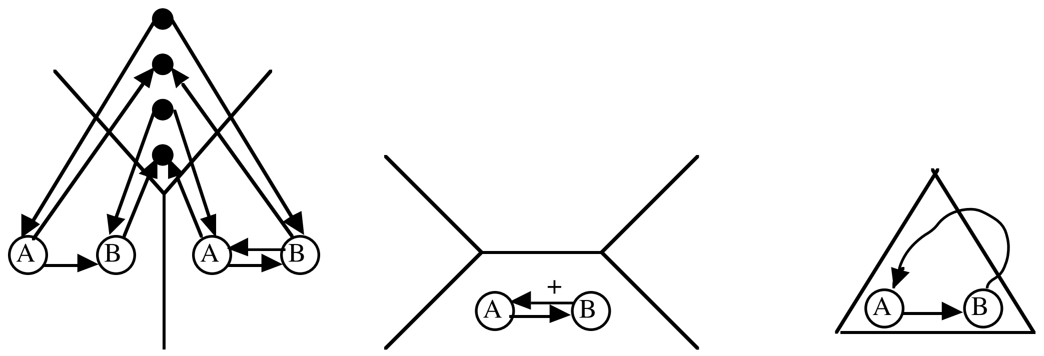
1.3.3 Readability and Intellectual Manageability

Readability is an important consideration, with effects on intellectual manageability, system development time, ease of maintenance, ease of debugging. Complex embeddings can be difficult to specify, understand, and debug. Diagrammatic notations (such as Y notation [Gött79] [Gött83], X notation [Gött87] [GöGN91] [Gött92], Δ-notation [KaLG91] [LoKa92], and ⊢ notation [Reke94]) can help (Section A.2). Diagrammatic notations show embedding edges graphically, as edges between the context (which is part of RestGraph) and g_r. If a complicated embedding is used (e.g. expression graph rewriting), then diagrammatic notations provide an easily-perceived depiction of the following of edges in RestGraph. Visual presentation can be simplified by avoiding the duplication of graph-parts common to g_l and g_r (Figure 2). In summary, diagrammatic depiction of embeddings is advantageous for fairly complex embeddings:

- In case a simple embedding suffices, a textual embedding specification can be used, with the embedding easily perceived from visually-corresponding nodes in g_l and g_r. The drawings of g_l and g_r become unnecessarily cluttered if a graphical depiction of a simple embedding is used (Figure 3). Similarly, gluing isomorphisms are effectively conveyed by the visual correspondence of g_l and g_r nodes, as in [EhHK92].
- Generally, embeddings that are more complex than the elementary type (Section A.4) are easier to perceive if a diagrammatic notation is used instead of a textual one.
- Selected embedding paths that are very long and highly complex may benefit from textual rather than diagrammatic depiction. An example is the use of the PROGRES "path" construct, which permits extensive searching and testing of the host-graph, as part of the embedding process (Section 5.1).

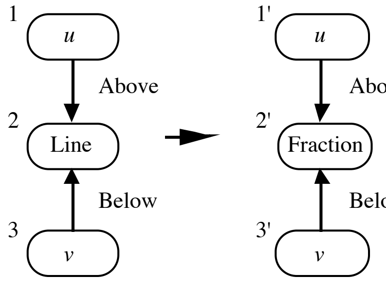
The use of color could be helpful in diagrammatic notations, for example to distinguish optional, prohibited, and required contexts. Color distinctions can be more readable than textual annotations. For example, the *-groups in the Δ-notation of Figure 3b might be more easily perceived through the use of color. Of course, an equivalent black-and-white notation is needed, since color isn't always available.

Intellectual manageability is an important consideration. Some applications require complex embeddings, others don't. In practical uses of graph rewriting that we have seen, rewrite rules tend to be fairly simple, with uncomplicated embeddings. Generally, the major difficulties arise not in the formulation of individual rewrite rules, but in the structuring of a large collection of rules that interact in a desired way.



(a) Y notation [Gött83] (b) X notation [Gött92] (c) Δ notation [LoKa92]

Figure 2 Three notations for a graph-rewrite rule to add a second edge between an A-labeled node and a B-labeled node. Avoiding duplication of graph-parts common to g_l and g_r simplifies both the drawing of g_l and g_r, and especially the graphical depiction of the embedding. Using Y notation, g_l and g_r are represented separately, with eight edges and four nodes used to show the embedding. Using X notation, the common parts of g_l and g_r 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. (Part (a) reproduces the Y-notation rule of [Gött92, Fig. 14].)

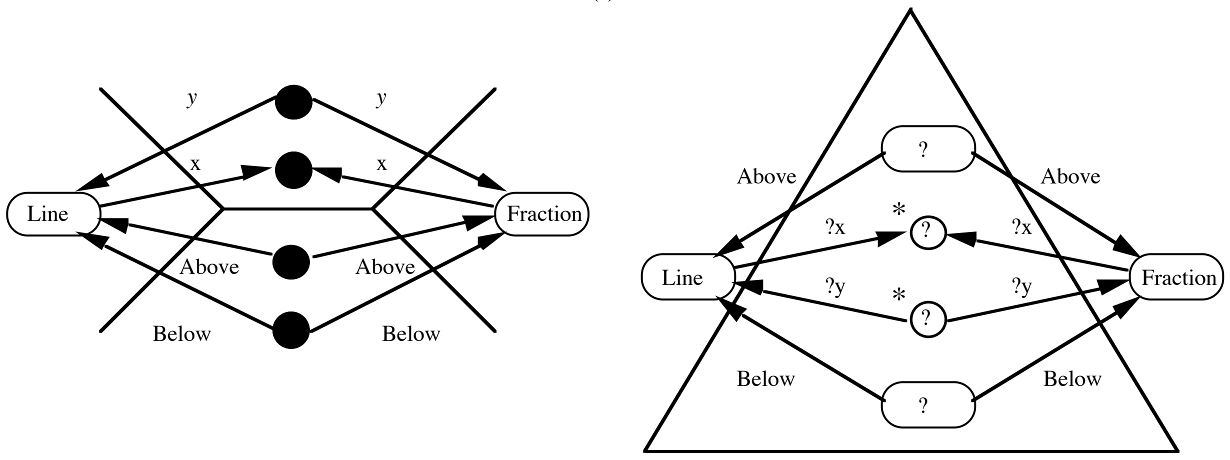


Application Condition: (u,v = any node label) & (m(2) = undetermined)

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

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

(a)



(b) (c)

Figure 3 Three equivalent notations of a graph-rewriting rule to replace a *Line*-labeled node by a *Fraction*-labeled node, in the context of incoming *Above* and *Below* edges. (This rule occurs as part of a system for recognition of math-notation [Grba94].) (a) The analogous embedding is conveyed by similarly-denotated nodes in visually-corresponding places; this is reinforced by the textual description "{(1,1'), (2,2'), (3,3')}", which states that all pre-embedding edges connected to node 1 become post-embedding edges connected to node 1', and so on. Here u and v are variable node labels, matching a variety of host-graph labels. (b) In X-notation, the embedding is conveyed as optional context, in the upper part of the X. Here one node (filled-in, indicating arbitrary node label) and two edges are used to depict a node-correspondence; since directed edges are used, this must be repeated for incoming and outgoing edges. Execution of X-notation performs repeated matching of the structures in the optional context. (c) In Δ-notation, the embedding is conveyed similarly, using *-groups to indicate 0 or more occurrences of the starred structures. Here ?x and ?y are variable edge labels; all occurrences of ?x are unified to the same value, which may be distinct from the value the ?y variable unifies to. (We adopted similar use of x and y in (b), although this may not be accepted X-notation syntax.)

1.3.4 Isomorphisms versus General Graph Morphisms

For clarity, our exposition assumes that a subgraph isomorphism test is used when finding a subgraph g_l^{host} matching g_l. However, the use of general graph morphisms is certainly possible. The decision to use a general graph morphism has great practical impact on the resulting rewrite rules. The utility of general graph morphisms is illustrated by small examples in the literature ([EhHK92, p. 560], [RoKr90a, p. 200]). On the downside, the use of general morphisms means that rewrite rules easily give rise to unexpected matches. There may be practical examples where general graph morphisms are advantageous; we would be interested to hear about debugging of such rewrite systems.

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 / edges which can optionally be matched to single host-graph entities [KaLG91] [LoKa92]. The utility of this construct is demonstrated by, for example, a rule to insert an element into a circular list; when the host-graph list is long, each g_l node maps to a unique host-graph node, but a short list causes several g_l nodes map to one host-graph node. In summary, the fold construct provides controlled deviation from strict isomorphism: a rule-author selectively and explicitly indicates where g_l node identifications are permissible.

1.3.5 Efficiency of Rewrite-rule Application

Graph-rewriting has well-recognized efficiency problems. Application of a graph-rewrite rule requires that g_l^{host} be located in the host graph g, which involves subgraph-isomorphism testing. Since the general form of the subgraph-isomorphism problem is NP-complete, graph rewriting can be computationally expensive. However, it is often possible to express a computation using small subgraphs g_l. Node labels, edge labels, and directed edges drastically reduce the search space for isomorphic subgraphs. (Efficiency issues related to node labels are discussed in Section 2.1.) In addition, some graph rewriting systems have certain phrases that frequently appear in application conditions; these can be exploited to further reduce the search space for isomorphic subgraphs that meet the application condition. For example, picture-processing applications commonly use host-graph node attributes to record the (x, y) location of the image feature represented by a node. In this case, it is possible to preprocess the host graph so that computational geometry algorithms can quickly answer nearest-neighbor queries. This would permit very fast identification of a g_l^{host} subgraph in which two nodes are not connected by an edge, but are restricted by an image-proximity clause in the applicability predicate. Such customized subgraph-isomorphism code could be worthwhile for certain applications.

The von Neumann architecture (geared toward instruction fetch and execution, with a bottleneck between processor and memory), is not well-suited to the interpretation of graph rewriting. Strong demand might motivate the development of a new computer architecture with graph-rewriting as a fundamental operation. Certainly this has happened for other styles of computation: there are special architectures (using custom VLSI chips) for fast implementation of neural network computations and for AI applications. Standard parallel-processing hardware may also prove useful for the implementation of graph rewriting.

Simpler rewrite formalisms can provide more efficient, and less expressive, alternatives to graph rewriting. These include tree grammars [Fu82] and coordinate grammars [Ande77]. However, these restricted formalisms are not suitable for many applications.

Rewriting efficiency depends not only on the subgraph-isomorphism test, but also on the algorithm for applying a rewrite rule. In some cases this latter time is dominant. (For example, subgraph-isomorphism search can be eliminated by a unique cursor-node in the host-graph, which indicates where the graph rewrite should occur. Use of cursor nodes is natural in editing applications, where the end-user specifies an insertion point, e.g. [Gött92] [ELNSS92.) Efficient rule-application requires an implementation that minimizes the updating of host-graph structure. This is achieved by identifying graph portions that are common to g_l and g_r, thus avoiding the overhead of first removing and then replacing these nodes and edges. Some rewrite notations explicitly identify the common portions of g_l and g_r (e.g. X and Δ notation) whereas others provide separate descriptions, repeating the common portions (e.g. Y and Schneider's notation). The implementation can easily identify the common portions of g_l and g_r if the notation fails to do so. Exploitation of these common portions results in orders of magnitude speedup in the implementation of X-notation versus Y-notation [Gött92].

Here we've looked at the cost of applying a single rewrite rule. The overall structuring of a graph-rewriting system greatly influences the number of attempted rewrite-rule applications (Section 4.4).

2 . Extensions to Graph Rewriting Mechanisms

In practical applications of graph rewriting, extensions are made to the basic rewriting formulations of Section □ and Appendix A. Extensions to an individual graph-rewriting rule include attributes, application conditions, and parameterization. The use of such extensions can greatly increase the expressiveness of a graph rewriting rule. However, the extensions also limit how well the theoretical foundations of pure graph rewriting carry over to practical use. For example, in a pure graph grammar, a graph rewriting rule can be applied either in the forward direction (as for generating examples of graphs in the language) or in the reverse direction (as for bottom-up parsing to recognize whether a given graph is in the language). However, if the graph grammar must be applied to attributed graphs, then attribute computation rules generally dictate a unidirectional application of rewrite rules; it becomes necessary to create separate graph grammars for generation and for recognition.

We begin with a discussion of labels and attributes in graph representations. This introduces the need for rewrite-rule features such as attribute-transfer functions, application conditions, variable labels, and parameterized rules. Finally we consider the evolution of graph-rewriting systems; the ease of evolution is affected by the use of structuring, and by the decision to use induced or non-induced subgraph isomorphism.

2.1 Labels and Attributes; Hierarchical Label Organization

In a graph, both node and edge labels can be used to represent information. The organization of possible labels can be flat (a set of labels) or hierarchically structured (a tree of labels). For subgraphs to be considered isomorphic, they must match not only in structure, but also in labeling. Using flat labels, an X-labeled node in g_l matches an X-labeled node in g_l^{host}. Using hierarchically structured labels, an X-labeled node in g_l matches a g_l^{host} node with a label that occurs at or below X in the label tree. (Thus subgraph isomorphism becomes non-commutative.) Hierarchical node labels have been used in several graph rewriting systems [ELNSS92] [Gött92]. (The PROGRES language allows multiple-inheritance among node-classes, resulting in lattice-structured rather than tree-structured node labels [ELNSS92].) For some applications, hierarchical edge labels might be useful as well. Advantages of a hierarchical label structure are discussed below.

Both with flat and hierarchical labels, the implementation can make use of label statistics to reduce subgraph-isomorphism times. The implementation records how many labels of each kind there are in the host graph. Matching begins with the rarest node-label found in g_l.

Graphs are often augmented with attributes, to express non-structural aspects needed to model an application. Attributes can be associated with nodes or edges, and can have arbitrary data types, such as integer, string, or table. The attributes available at a node depend on the label of that node. When node-labels are hierarchically organized, inheritance is used: a sublabel inherits all of the attributes of it's parent label, and may have additional attributes defined specifically for the sublabel [Gött92].

A label can be thought of as a special kind of attribute. A label is distinguished from other attributes in that graph isomorphism requires labels to match, but permits other attribute values to differ. In practice node labels often indicate the fundamental characteristics of the object represented by the node, whereas attributes provide detailed characterization.

The distinction between labels and attributes is blurred in some systems, such as Δ-rewriting [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-rule to labels in the host-graph.

The use of attributes gives rise to a tradeoff between representing information via graph structure or via the attributes, and of embodying computations via graph rewriting or via attribute-value computations. Such a tradeoff arises whenever we choose the set of node-labels for a graph representation -- should we use many labels, to make fine case distinctions, or should we use a small number of general labels, using attributes to make the fine case distinctions. (For now we consider only flat label sets rather than hierarchical ones.)

- Option 1: Use highly-specific node labels. For example, the simple library-transaction system of [EhKr80] [EhHa86] uses unattributed graphs -- node labels encode the names of books, publishers, and library patrons.
- Option 2: Use general node labels. An alternative library-representation uses a small number of node labels (Book, Publisher, LibraryPatron), with node attributes to record details such as names and titles.

Both readability and efficiency are affected by this tradeoff. Graph readability suffers under Option 1, since a graph node labeled with a person's name can be either an author or a library patron. The node's role cannot be determined locally, but must be deduced from the topology of the graph, from the nodes and edges along the path to the root node.

Efficiency problems arise under both Options 1 and 2. The problems can be avoided by adopting hierarchically organized node labels. The efficiency problems are illustrated by our design of a graph-rewriting system for recognition of math notation [Grba94]

- Option 1: Use highly-specific node labels, such as *a*, *b*, *c*, *1*, *2*, *3*.
- Option 2: Use general node labels, such as *letter*, *number*, *digit*, with node attributes recording the actual character that gave rise to the node.

Graph rewriting can involve operations on individual letters, or on all letters. The above options are efficient for one of these operations, and inefficient for the other.

- Using Option 1, it is expensive to perform an operation on all letters. Either separate production rules have to be written for each letter, or a variable node-label notation needs to be introduced (Section 5).
- Using Option 2, it is expensive to search for a particular letter (as done 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 g_l contains a node labeled *letter*, with an application condition to specify the desired attribute value "c". The search for g_l^{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 g_l^{host} (Section 3.5).

The use of hierarchical node labels solves these efficiency problems. The implementation maintains a tree to define the label hierarchy. The label of a node is represented by a pointer into this tree. To match a "letter", we search for a node with label-pointer that goes anywhere into the subtree rooted at "letter". With a suitable tree-storage scheme, this is a constant time test, involving comparison of two tree-addresses to determine if one denotes an offspring of the other. Thus, implementation speed is similar for flat and hierarchical label sets. (This discussion does applies only to single-inheritance hierarchies, not when node-labels have a lattice structure with multiple inheritance, as in the PROGRES language [ELNSS92].)

Hierarchies play an important role in many potential applications for graph-rewriting. For example, reading of sketch maps benefits from hierarchical node labels and hierarchical graph structure (as demonstrated by the schema-based implementation of [MuMH88]; graph-rewriting is a promising alternative). As another example, hierarchies (and inheritance) play a central role in semantic nets. While hierarchies can be represented in a flat graph [EhHK92], a hierarchical graph representation (Section 7.1) combined with hierarchical node labels could prove highly useful.

2.2 Attribute Transfer Functions

In standard graph rewriting, attribute values are calculated by attribute transfer functions: whenever a rule is applied, values for attributes in g_r^{host} are calculated based on values of attributes in g_l^{host}.

In some applications, host-graph attribute values depend on each other, and an automatic attribute-update mechanism is desirable. Attribute expressions can be used to encode dependencies among host-graph attributes in [Gött92]. Application of a rewrite rule updates the attribute values or attribute expressions stored with g_r^{host} nodes. Eventually (after several rules have been applied) an attribute-evaluation algorithm iterates over the host graph to find an appropriate evaluation ordering for evaluating the attribute expressions. To ensure unambiguous results, there must not be a cyclic dependency among host-graph attributes.

Derived attributes in the PROGRES language are completely separated from graph-rewrite rules [ELNSS92]. (*External attributes* are set in the standard way, from user input or in conjunction with rewrite-rule application.) The evaluation mechanism for a derived attribute is defined statically, independent of rewrite rules, via a directed attribute equation in the graph scheme. (The graph scheme is a type-definition for host-graph, defining node labels, associated attributes, and permissible edge types). Directed attribute equations can follow complex host-graph paths to locate nodes required for the attribute-value computation. Alternative paths (or default values) can be specified, for use when the first path does not exist in the host-graph. Thus, significant attribute computations occur in a data-flow manner, decoupled from the application of graph-rewriting rules.

2.3 Application conditions

During application of a graph-rewrite rule, we find a g_l^{host} that is isomorphic to g_l. Often a simple isomorphism test is insufficient for ensuring that rewrite-rule application is appropriate at this location in the host graph. Thus application *conditions* are commonly used to express additional constraints on a prospective g_l^{host}. These application conditions can be implicit or explicit. Implicit application conditions are used by certain graph-rewriting models, and apply to every graph rewriting rule. An example is the gluing condition imposed by the gluing models (Appendix .5). Explicit application conditions are under the control of the rewrite-rule author, and are expressed by boolean predicates associated with individual rules. Ehrig and Habel formally introduce application conditions to algebraic graph rewriting, and provide illustrative examples drawn from a library system [EhHa86].

The clauses of an explicit application condition can test both graph structure and attribute values. Commonly-tested graph-structure features include the non-existence of edges in g_l^{host} (necessary for a non-induced g_l^{host}, see Section 7.2), and the (non)existence of edges between g_l^{host} and RestGraph. For rewrite systems with an attributed host graph, application conditions are used to express constraints on the attribute values.

Application conditions are often expressed textually as boolean predicates. In Δ-notation however, application conditions which pertain to the structural features of the g_l^{host} are given diagrammatically, whereas the application conditions which pertain to non-structural features, such as attribute values, are given textually by boolean predicates. The former category of conditions is given by the *restriction* part of the Δ-rule (also termed the *prohibited context*), and the latter category of conditions is given by the *guard* portion of the Δ-rule [KaLG91] [LoKa92]. A prohibited context indicates that the rule can be applied only when the subgraph PC cannot be located in the host graph. (Under typical use, PC is connected to g_l^{host}; in other words, the rewrite rule specifies PC as a partial graph, with all incomplete edges connected to g_l. During rule application, after g_l^{host} has been located, an attempt is made to extend the match to include PC. If this succeeds, rule application is aborted.) For some applications (such as music-notation recognition [FaBl94]) it is useful to have a *conditional prohibited context*: here rule application is prohibited only if a subgraph PC can be found which satisfies a given condition. The condition imposes restrictions on PC, in a fashion analogous to the use of application conditions to impose restrictions on g_l^{host}. The notation chosen by [FaBl94] superimposes a shaded box on the subgraph and condition that constitute a prohibited context region.

Path expressions in the PROGRES language, connecting two g_l nodes, are application conditions that test graph structure [ZüSc92]. Here path evaluation is integrated with the subgraph-isomorphism test. A path can be used to express complex relationships between two nodes of g_l^{host}, and can contain subpaths, conditional iteration, guarded alternatives and transitive closures. For example, for a graph representing the syntax of a program, a path can be used to connect an identifier-use node with the corresponding, far-removed identifier-declaration node [ELNSS92, p. 156]. In this example, subgraph-isomorphism begins without search: the host-graph contains a unique *cursor* node; this cursor node is matched to g_l, identifying a particular identifier-use node, and then the path evaluation is undertaken, to locate the identifier-declaration node that makes up the rest of g_l^{host}.

2.4 Parameters to Graph Rewrite Rules

Parameters are often used to specify attribute values. These parameter values can be used to test attribute values in the application condition or to assign attribute values in the attribute transfer function. Parameters can also be used to bind variable node and edge labels, prior to subgraph isomorphism testing.

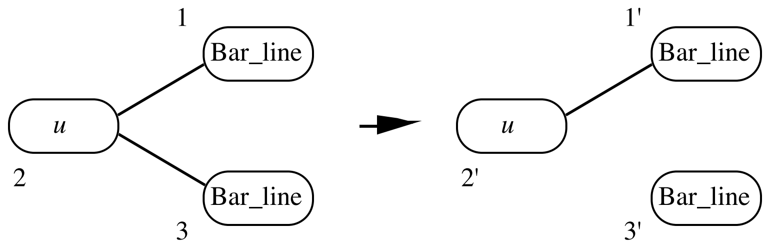
The use of parameters is possible when rewrite rules are invoked, as when graph rewriting is ordered or event-driven (Section 3). For example, in an event-driven library system [EhKr80] the event of borrowing a book results in invocation of a rewrite rule with parameters indicating the catalogue number of the book, and the ID number of the library patron. In this case, rule parameters are used to bind variable node labels before subgraph isomorphism testing. (Variable node labels that are not bound by parameters are bound during the subgraph isomorphism operation, thus allowing a generalized g_l to adapt itself to the node labels found in a particular host-graph environment.) Parameters can denote attribute values as well as node labels. Attribute parameters can be used to test attribute values (in the application condition), or to assign attribute values (in the attribute transfer function). The PROGRES language has parameterized production rules (both labels and attribute values can be parameterized); production rules can also return values [ELNSS92] [ZüSc92].

2.5 Rewrite Rules with Variable Node and Edge Labels

Various graph-rewriting notations allow for variable node labels or variable edge labels (Figure 3). The use of variables allows one graph-rewrite rule to apply in a variety of situations. This permits a reduction in the number of rules needed (Figure 4), and is an important structuring tool in the design of large rewrite systems. Variable node labels can be quite general (matching all possible node labels), or quite specific (matching only one or two node labels).

Variables are bound before or during subgraph isomorphism. In the former case, a parameter to the graph rewrite rule (Section 2.4) binds the value of the variable. It is the latter case which we consider for the rest of this section. Here, the subgraph isomorphism code proceeds as usual, except that certain node or edge labels aren't uniquely constrained in g_l. Once a matching g_l^{host} is found, the particular labels that occur in g_l^{host} serve to bind the variable node and edge labels for the remainder of the application of this graph rewriting rule. (Thus, for example, the application condition can test which value was bound to a variable node label, as in Figure 4.)

Variable labels can give rise to a significant performance cost, potentially causing large increases in the search space for subgraph isomorphism. Hierarchical node labels (Section 2.1) reduce this problem significantly: the most frequent label-generalizations are stated efficiently via the label hierarchy (avoiding the use of variable node labels), leaving variable use only to the less common case when rule applicability cuts across the label hierarchy.



Application Condition: (x1(2) > x1(1) > x(3)) & ($u \in$ {Nodehead, Accidental})

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

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

Figure 4 The use of variable node-labels permits a reduction in the number of rewrite rules. This rule removes a conflicting bar-line association. The application condition states that this rule applies when node *u* matches either a Notehead or Accidental node [FaBl93].

2.6 Rewrite Rules with Variable Graph Structure

The successful use of variable node labels leads us to consider more general variability in the *structure* of g_l. Possibilities include

- If g_l is a directed graph, provide a notation for specifying variable edge direction.
  For example, variable-edge direction notation would reduce the number of rewrite rules in our math-recognition system [Grba94].
- Provide a notation for optional or repeated nodes or edges in g_l.
  Structures for repetition and optional items are provided by Δ-notation (Appendix A.2) and others.
  Notations for a variable number of nodes and edges are particularly useful for invariant embeddings, such as the gluing models (Figure 1).

As indicated, availability of such structural variability can greatly reduce the number of graph-rewrite rules needed by a given application. One general rule with variable structure is used in place of many specific rules. This combination of rewrite rules eliminates duplication of information, which is a crucial aid to debugging, maintenance, and further system development. Careful implementation is required, so that variable-structure rewrite rules can execute with acceptable efficiency.

2.7 Planning for Evolving Graph-Rewriting Systems

Design of a graph-rewriting system includes the design of a host-graph encoding suitable for the application, the design of the rewrite rules, and the design of other components such as a user interface. All of these must be maintained as the graph-rewriting system evolves. Consider the addition of a new feature to a graph-rewriting system. This may require extensions to the host-graph representation; for example, new node labels and edge labels may be introduced. It is desirable that many of the old rewrite rules continue to work properly, and particularly desirable that it be clear which of the old rules must be updated in response to the expanded host-graph representation. What characteristics of a graph-rewriting system help to achieve this? Of the many relevant considerations, we discuss two: use of hierarchical host-graph structure, and the effect of induced or non-induced subgraph matching.

2.7.1 Hierarchical Host-Graph Structure

Hierarchical structures provide encapsulation, thus helping to localize the changes needed during system evolution. Various aspects of a graph rewriting system can be structured. Elsewhere we discuss hierarchical node labels (Section 2.1), and methods for organizing a collection of rewrite rules (Section 3). Here we turn to the structure of the host-graph itself.

Processing or displaying a large graph is difficult. Giving the graph a hierarchical structure permits meaningful portions of the graph to be treated as units. Zoom-in and zoom-out operations are used to reduce the graph to manageable proportions for viewing, or to delimit selected portions of the graph for processing. Various notations for hierarchical graph structures are described in [Hare88] [SiGJ93].

Hierarchical graphs can be defined in various ways, depending on whether edges are allowed to cross the hierarchy. In a strict definition, all edges must go either between siblings or between parent and child nodes. 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 rewriting.

It is possible to consider hierarchical graphs as merely a notational device pertaining to graph display. This is because a hierarchically structured graph can easily be translated into a flat graph, with the addition of special edges to indicate parent/child relationships in the hierarchy. However, in a full implementation of hierarchical-graph rewriting, many special considerations must be given to these edges. There is significant interest in the topic of hierarchical graph rewriting; the following is an incomplete list of relevant research efforts.

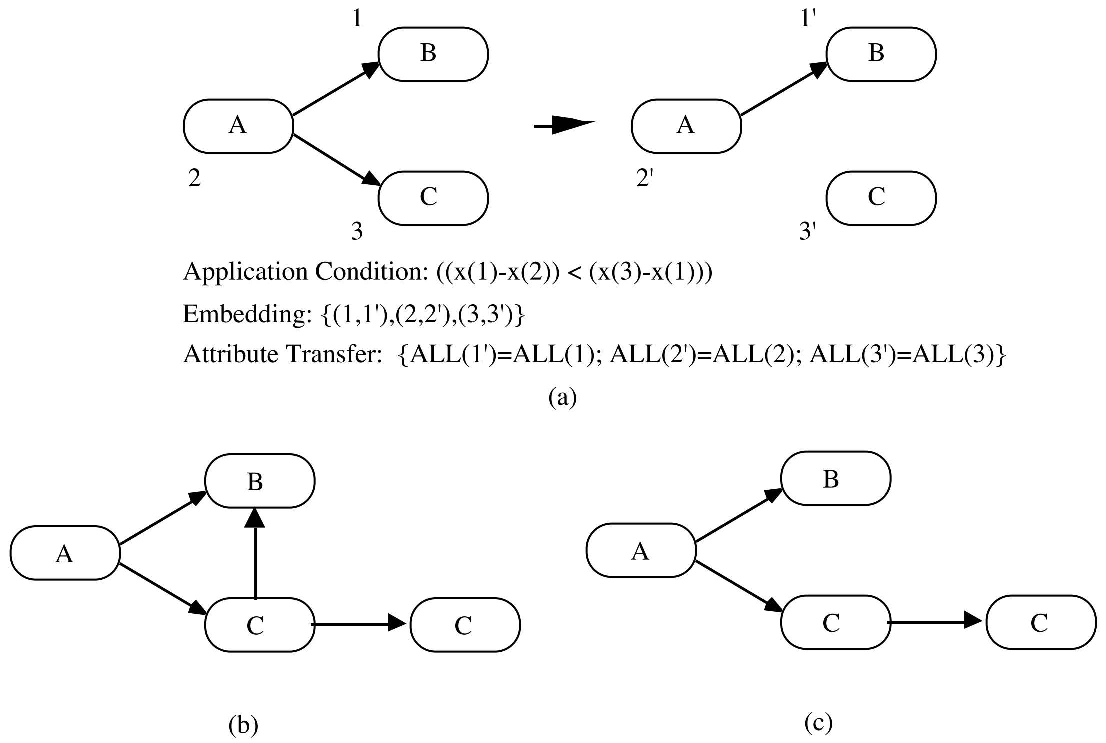
A chart-based parser for hierarchical-graphs is discussed in [MaKl92]; here each production collects a set of nodes into a supernode. Abstract graphs (in which subgraphs are represented by a single node, and groups of edges are bundled into a single edge) are supported in the prototype algebraic-rewrite environment of [LöBe93]. This graph structuring is motivated by the need for convenient selection of subgraphs and morphisms for rule application, not by a desire to rewrite hierarchical host-graphs.

Hierarchical graph structure can be described by allowing node-labels to be graphs themselves. In the algebraic rewriting formalism of [Schn93], labels can be complex categorical objects (including graphs), instead of being restricted to atomic objects drawn from an alphabet. These complex labels can be rewritten during rewrite-rule application (unlike the label-preserving graph morphisms of more restrictive algebraic rewrite systems). It would be interesting to gain practical experience in the use of this rewriting formalism.

Hierarchical graph structure can be captured by a flat host-graph combined with a set of graph-rewriting rules. In [EhHK92], a hierarchical semantic-network is stored as a host graph (which contains the semantic net at some level of zooming), combined with a set of rewrite rules that implement the zoom-in and zoom-out operations. This is an interesting proposal, although it complicates semantic-net information-retrieval, since rewrite rules may need to be applied before the desired information (if present) appears in the host graph.

2.7.2 Effect of (Non)Induced Subgraphs on Graph Evolution

Graph-rewriting systems can use induced or non-induced subgraphs; each option has important consequences for the structure of rewrite rules, and the extensibility of the system. Recall that If g_l^{host} is an induced subgraph of g, then g_l^{host} must include all local edges of g (i.e. all edges that connect two g_l^{host} nodes). A non-induced subgraph may omit some or all of these edges. This is illustrated in Figure5.



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)

Figure 5 Induced versus non-induced subgraphs. A rewrite rule (a) is being applied to the host graph shown in (b). If an induced g_l^{host} is required, the isomorphism test fails and the rewrite rule cannot be applied. On the other hand, if non-induced subgraph matching is used, a suitable g_l^{host} is found; rewrite-rule application results in the new host graph (c). Note the hidden edge-deletion: the edge from the C-labeled node to the B-labeled node is removed in host-graph, an effect that may or may not have been anticipated by the author of rewrite-rule (a).

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.

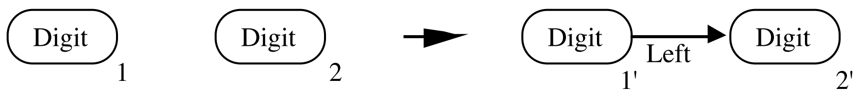
- Using induced subgraphs increases the number of rewrite rules. A g_l cannot match unless the rule-author has anticipated all the edges present in that part of the host graph. If a variety of edge-configurations might be present, these must be enumerated in separate graph-rewriting rules (where a single non-induced rewrite rule could suffice). The rule-author must be able to anticipate all possible edge-configurations.
- Non-induced subgraphs result in extra application conditions, to test the presence of host-graph edges (Figure 5) (Note that the library system in [EhKr80] manages without application conditions both by using induced subgraphs, and by using unattributed graphs. An extension, still unattributed, but with application conditions is presented in [EhHa86].)
- Hidden edge-deletion is a major pitfall of non-induced subgraphs. Edges present in host-graph but not mentioned in g_l are deleted by rule application (Figure 5).

These points become particularly significant in case of host-graph evolution, e.g. the addition of a new type of edge, with the new edge label “Grow”. Ideally, we hope that most of the old graph-rewriting rules 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, keeping the original Grow-less form, as well as all possible permutations of Grow edges that might occur in the g_l area.

- Using non-induced subgraphs, the old graph-rewrite rules continue to apply, but they perform hidden Grow-edge deletion. Rewrite rules 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 gluing models as in unrestricted-embedding models. A general solution to this graph-evolution problem is needed. Perhaps the greatest need is for improved treatment of rules which do not delete host-graph nodes. (Rules in this class update attribute values, insert and delete edges, and insert nodes.) For this restricted rule class, a graph-rewriting system can provide a hybrid of induced and non-induced subgraph semantics, so that (a) rules apply generally, as in the non-induced case, and (b) no hidden edge-deletion occurs. This issue is addressed by incomplete removal of non-induced subgraphs in the PROGRES language [Schü91, p. 652], although earlier references speak of removal of the corresponding complete subgraph [EnLS87, p. 191]. Similarly, structured graph rewriting [KrRo90a,b] provides non-induced subgraph matching, with no edge-deletion in cases when there is no node deletion. Gluing models often use induced subgraphs (e.g. [Nagl87]). A compromise in [FaBl94] uses induced g_l^{host} matching (to avoid hidden edge-deletion) combined with non-induced matching for prohibited contexts (which represent host-graph structure, that, if present, prevents rule application). Many graph-rewriting papers give scant mention of their choice to use induced or non-induced subgraph matching. This issue is important both theoretically and practically.



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 5 Structural application condition arising from a non-induced g_l^{host}. This rewrite rule adds an edge to a graph [Grba94]. To avoid repeated rule application (when an edge is already present), the application condition must test for non-existence of an edge. (Note that repeated rule application would waste time, but would not result in parallel edges, since any existing edges between nodes 1 and 2 are discarded even though they are not mentioned in g_l.)

3. Computation via Graph Rewriting

Graph rewriting is an expressive mechanism; many formulations of graph rewriting are theoretically sufficiently powerful to express any computation. Nevertheless, when faced with a particular application, it can be difficult to determine how to conveniently and effectively express the computation via graph rewriting. A collection of graph-rewrite rules can be organized in a variety of ways:

- An unordered rewriting system simply consists of a collection of graph rewriting rules. The system operates by being presented with an initial host graph. Rewrite rules are applied in any order, until no further rules apply.
- The heart of a graph grammar consists of a set of graph rewrite rules (also called *productions*). This set of productions is augmented by (1) a start graph, which provides the initial host graph for rule application, and (2) a designation of labels as “terminal” or “nonterminal”. In generative use of a graph grammar, rewrite rules are applied to the start graph until a terminal graph results. (A terminal graph contains no non-terminal labels.) Alternatively, graph grammars can be applied for recognition: the outside world provides a graph, and a parser is used to determine whether this target graph can be derived from the start graph by the application of production rules. 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 often makes the set of production rules directional, so that they can only be applied in the forward (top-down) or reverse (bottom-up) direction. Thus the style of the rewrite rules limits the applicable parsing algorithms.

- Ordered graph rewriting gives the system designer control over the order in which rewrite rules are applied. The ordering may be complete, or may be partially non-deterministic. The application determines whether execution of non-deterministic systems requires backtracking. An ordered rewrite system is also referred to as a "programmed graph grammar" [Bunk82a], since the ordering of rewrite rules is reminiscent of the ordering of statements in an imperative programming language. An ordered rewrite system does not require a distinction among terminal and nonterminal labels, since the control specification indicates when rule-application should terminate. Control specifications can be given as graphical control diagrams, or as textual specifications similar to an imperative programming language (Section□3).
- Event-driven graph rewriting systems have an ordering imposed by an external sequence of events (Section□4).

The practical graph rewriting systems of Section□illustrate further use of these styles of computation.

3.1 Unordered Graph Rewriting

An excellent example of unordered graph rewriting is provided by Δ-rewriting [KaLG91] [LoKa92]. (Section□.2 introduces basic constructs of Δ.) The rewriting system is given an initial host-graph (e.g. the quicksort example of [LoKa92, p. 177] uses a list of numbers to be sorted, the specification of the Actor language of [KaLG91, p. 484] uses a graph compiled from an Actor program). This initial host-graph is transformed via graph-rewriting rules, either infinitely (as in the dining philosophers example of [LoKa92, p. 112]), or with termination (as in the aforementioned quicksort example).

To divide a large problem into more manageable subproblems, Δ graph-rewriting rules 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 g_l. To invoke rules in a given platform G, the G trigger is placed somewhere into the host graph. (This satisfies one of the preconditions of rule-application from platform G; successful rule-application from platform G also depends also on proper matching of g_l^{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. Unfortunately, there is no mention of plans to implement an environment for Δ-rewriting; current experience is limited to paper-based descriptions of Δ rewriting systems.

3.2 Graph Grammars and Parsers

Graph grammars have been intensively studied, and have met with some practical success. As described above, graph grammars can be applied either in a forward direction, to generate sample graphs from the language, or in the reverse direction, to recognize and parse graphs in the language. In practice it can be difficult to produce one grammar that is suitable for both generation and recognition. For example, this problem arises in the use of string grammars for natural language processing. Natural language systems tend to have different grammars for analysis and generation. Analysis grammars tend to overaccept, being able to parse all legal sentences and some extra ones. Synthesis grammars tend to undergenerate. A reversible grammar matches analysis and synthesis capabilities, and is easier to maintain; construction of such grammars is a topic for active research (e.g. [Strz90]). Reversible graph grammars are similarly difficult to construct, due to the need for attribute processing and variable binding. Rekers mentions the difficulty of predicting the unintended sentences that are recognized by a graph grammar for a visual language, and suggests that grammar debugging could be aided if this recognition grammar could be used for generation [Reke94]. In our own work, we are considering the possibility of reversible notational conventions, which could be used both to generate and to recognize a diagrammatic notation such as music. This is a challenging research topic, which may be impractical due to differences in the knowledge needed for generation and recognition [HaBl94].

Graph grammars have numerous interesting and useful formal properties. Appendix□ reviews formulations of context-free graph grammars, the relations between formal properties of graph and string grammars, and the hierarchies of graph languages that arise from embedding complexity and production complexity.

Practical use of graph grammars is seriously hampered by efficiency problems. Sub-exponential parsers have been developed for certain restricted classes of graph grammars. A selection of parsing references are as follows. Bunke and Haller describe an extension of Early's parser for context-free plex languages [BuHa89] [BuHa92]; this parser permits left-recursion and is capable of recognizing partial structures. Egar et al. develop a parser for attributed graph grammars, used in the design of a visual programming environment for clinical protocols [EgPM92]. Lin and Fu recognize three-dimensional objects (in two-dimensional images) using a semantic-directed top-down backtrack parser for plex grammars [LiFu89]. Collin et al. interpret dimensions in engineering drawings using a plex-grammar parser that mixes top-down and bottom-up processing [CoTV93]. A chart-based parser for hierarchical graphs is discussed in [MaKl92]. The following work does not deal with graph grammars, but contains relevant discussion of parsing complexity. Henderson and Samal discuss efficient parsing of stratified shape grammars [HeSa86], building on the table-driven methods used for LR(k) string grammars.

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.

3.3 Ordered Graph Rewriting: Organization and Modularity for Rewrite Rules

For many computations it is convenient to order, or partially order, a collection of rewrite rules. For example, Bunke recognizes circuit diagrams by first applying a collection of noise-reduction rules [Bunk82a]. It is critical that these noise-reduction rules be applied first, and exhaustively, before application of rules for recognition of transistors, capacitors, and so on. Another example is provided by Fahmy's recognition approach for music notation [FaBl93]. Recognition is ordered into subgrammars, each of which consists of three phases -- Build creates edges, Weed removes inconsistent edges, and Incorporate prunes the graph while adding semantic information to attributes. The ordering among these phases is critical to the design and functioning of this graph rewriting system. In addition, ordered graph rewriting can be relatively efficient compared to parsing with a graph grammar (Section 4). On the other hand, the theoretical underpinnings of ordered graph rewriting are not as strong as for graph grammars and unordered graph rewriting, making convergence and correctness proofs more difficult. This effect can be mitigated by only partially ordering the rewrite system. For example, a rewrite system can consist of an ordered sequence of phases, where production rules within each phase are unordered [FaBl94].

Ordered graph rewriting can be partially or completely deterministic. If there is non-determinism, the implementation may be required merely to follow some path through the control specification, or may be required to provide backtracking in case a path fails. The following examples illustrate circumstances in which these various options can be used. Note that non-determinism can arise from the selection of a particular g_l^{host} (when there are several possible matches for g_l) as well as from the selection of a particular path through the control diagram.

- A completely deterministic system 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 [Gött92] each graph-rewrite rule g_l includes a cursor node, which is matched to the unique host-graph cursor node; g_r leaves a new cursor node in the host graph. (Overall control in this editing system is event-driven; short sequences of ordered, deterministic graph rewrites 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.)
- Partially ordered rewrite systems, without backtracking, have been used for diagram recognition: circuit-diagram recognition [Bunk82a] and music-notation recognition [FaBl93] [FaBl94]. Here ordering is used to control the sequence of phases that make up the recognition process; rules within a phase are unordered. Rewrite rules are written such that all non-deterministic alternatives lead to a desired result. Such a partially-ordered system has attractive properties: the unordered portions retain a strong theoretical basis (useful for convergence proofs), and the partial ordering provides control and encapsulation. The encapsulation into phases simplifies extensions to the rewrite system, and enhances intellectual manageability, since only a subset of the rules needs to be considered at any one time.

- Partially ordered rewrite systems, with backtracking, can be expressed in the PROGRES language [ZüSc92]. The PROGRES interpreter automatically provides backtracking in the search for some successful path through the control specification -- other matches for g_l^{host}, and other control paths are tried as needed. This allows straightforward expression of search problems, such as the ferryman's problem [ZüSc92], as a partially-ordered collection of rewrite rules.

These approaches differ greatly in execution cost, and in the types applications they are suited to implement. This list of examples may provide a system designer with some guidance.

Control specifications can be expressed in a variety of forms, including lists, diagrams, or text. The simplest control specification associates two sets with each production rule. The Success set lists the possible production(s) to try after successful application of the current production. The failure set lists productions to try after unsuccessful application of the production. This can be specified in tabular form [Fu82], which quickly becomes difficult to read. Diagrammatic control specifications (called control diagrams) are used by [Bunk82a], with extensions by [DoTo88], [FaBl93], and others. Sometimes an if-condition is needed, to allow host-graph inspection before determining the next set of allowable productions. For example, the *block condition* of [DoTo88] allows the control diagram to test attribute values of any nodes involved in the most recent production. Notations for non-determinism (Figure 7) and subgrammars are introduced in [FaBl93]. To permit more flexible control constructs, the control specification can take a textual form, similar to an imperative programming language. For example, the PROGRES language provides both deterministic and non-deterministic versions of And, Or, and Loop statements [ZüSc92][ELNSS92], in addition to encapsulation tools such as transactions and subdiagrams. An earlier version of the language adopted Modula-2 control structures [EnLS87, p. 194].

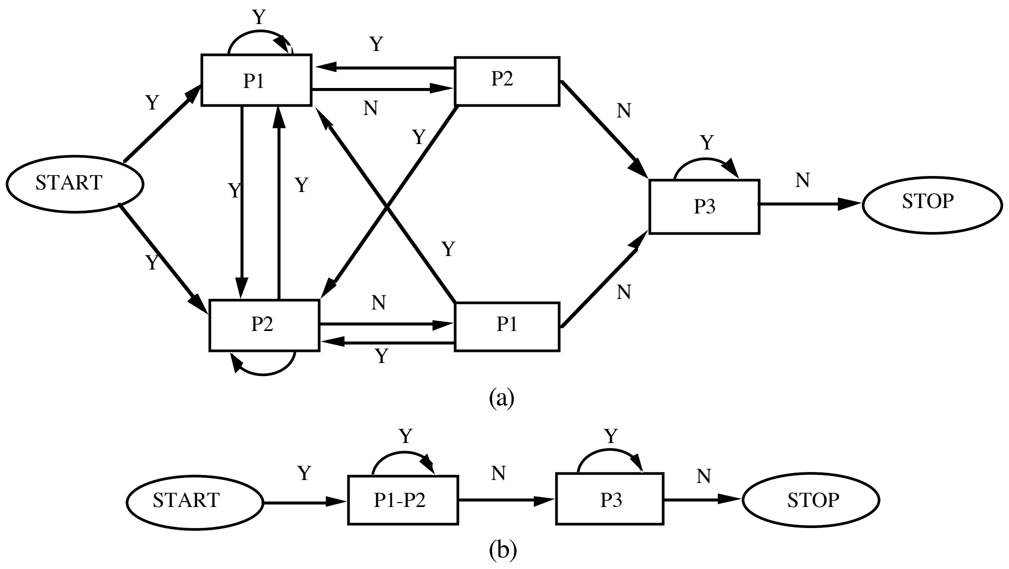


Figure 7 A graph-rewriting system may be partially ordered, necessitating a control specification that can express non-determinism conveniently. Consider the following verbal control description: *apply productions 1 and 2 as often as possible; when neither can apply, apply production 3 as often as possible.* (a) The traditional control diagram notation of [Bunk82a] results in a complex diagram. (b) A modified control-diagram notation for expressing non-determinism [FaBl93].

3.4 Event-driven Graph Rewriting

Whereas ordered graph rewriting systems provide an internally-imposed ordering of the rewrite rules, event-driven systems have an externally imposed ordering, arising from the ordering of external events.

This is illustrated by the library system of Ehrig and Kreowski [EhKr80]. An external event, such as loaning, returning, or ordering a library book, results in the invocation of a corresponding rewrite rule. Parameters provide the rewrite rule with information describing the details of the event. To extend the system, [EhKr80] see the need for control structures within a single transaction. Thus external events direct the overall sequence of actions. The system response to a given external event is controlled by ordered graph rewriting, which invokes the appropriate parameterized production rules. (A follow-on to this system, with application conditions, is presented in [EhHa86].)

It is possible to combine ordered graph rewriting with event-driven graph rewriting, 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 describe in [Gött92].

4. Popularization of Graph Rewriting

Graphs are a popular data structure, and graph-manipulation programs are widespread. Graph-manipulations can be cleanly, compactly, and explicitly described using graph-rewriting notation. This may convince software developers to consider using graph rewriting. Commonly, the first task is to convert existing graph-manipulation code (written in a high-level programming language) to a graph rewriting formulation. Three problems that may be encountered are:

- It can be difficult for a newcomer to develop a feel for how computations are expressed via graph-rewriting (Section 4.1).
- It may be difficult or clumsy to express the necessary graph-inspection operations via graph-rewriting (Section 4.2).
- Graph rewriting can express only part of the computation. Mechanisms are needed to interface graph rewriting with the remaining programming-language code (Section 4.3).

These are three reasons why graph rewriting has not attained more widespread practical use. Other reasons include efficiency considerations (Section 4.4) and the limited availability of development tools (Section 4.5). Further related discussion may be found in [Panel91]. The European COMPUGRAPH working group is carrying out extensive graph-rewriting research to demonstrate the potential of graph transformation to serve as a unifying paradigm for computing [Comp92]; their recent theoretical results appear in [TCS93].

4.1 A Graph-Rewriting Culture

While we can convince software designers that graph rewriting is a powerful and useful tool, we need further research and additional experience with *how* computations can be expressed in graph rewriting. A newcomer to graph-rewriting must be given access to these collective experiences. Consider, by analogy, the change from C to Lisp programming. Avid C programmers who cannot use Lisp effectively (due to a C mindset that dominates their approach to programming), can absorb "Lisp culture" by immersing themselves in an environment of experienced Lisp programmers. These same C programmers, in attempting to learn graph rewriting, may have trouble locating sources of "graph-rewrite culture". The graph-rewriting community can make an effort to promote such a rewrite culture, to allow newcomers to quickly develop a proper mindset for performing practical, effective computations using graph rewriting. Relevant materials could include the following:

- Accessible written introductions to the practical use of graph rewriting.
- Easily-available tools for creating, editing, executing, debugging graph rewriting systems. The design and implementation of such tools is a major undertaking, and yet tool availability is an absolute necessity for widespread use of graph rewriting. Tools could include libraries of graph-rewriting rules for common operations; an example is given by parameterized graph-rewrite rules for abstract-syntax-tree manipulation [ELNSS92, p161].
- Easily-available examples of non-trivial, practical uses of graph rewriting. Complete, executable systems are most helpful. These illustrate various computational styles in which graph rewriting may be used.
- Algorithm-analysis oriented toward graph-rewriting. A user of an imperative programming language can draw on a large body of algorithms (e.g. searching, sorting, hashing) and algorithm-design techniques (e.g. divide and conquer, dynamic programming, greedy algorithms). Specialized forms of algorithm analysis are needed for computing styles that are not based on the operations of a von Neumann computer. For example, chip-layout considerations in VLSI design give rise to algorithm analysis using area*time as a cost function.

The fostering of a graph-rewriting culture could go far toward the popularization of graph rewriting.

4.2 Graph Rewriting as Part of a Larger System

A large variety of computations can be conveniently expressed using the various approaches toward organizing graph-rewriting rules. However, it can happen that graph rewriting is a suitable formalism for expressing only *part* of a computation. If graph rewriting is to be of practical use in such a situation, we need convenient methods to combine graph rewriting with other styles of computation. This is an interesting research topic. A few possible approaches are as follows: (1) combine graph rewriting with a blackboard architecture (with the host graph stored as part of the blackboard), (2) combine graph rewriting with methods for performing major computations on attributes (where attributes can be complex entities such as tables or lists or even other graphs), and (3) use graph rewriting with or on top of a standard programming language (as is already being done with some ordered graph-rewriting systems such as PROGRES [ZüSc92]).

4.3 Graph Inspection

One can make a distinction between graph algorithms (mainly concerned with the inspection of graphs) and graph rewriting (which provides a means for generating, manipulating and reducing graphs) [EhHK92]. However, most existing graph-manipulation software freely mixes graph-inspection operations with graph-manipulation operations. Thus, if we want to convince software designers of the benefits of recasting their code in terms of graph rewriting, we must provide convenient methods for graph inspection. Current graph-rewriting mechanisms offer various graph-inspection capabilities, but are particularly lacking in graph-inspection capabilities that precede subgraph-isomorphism testing. Here are the current ways to inspect the host graph during graph rewriting:

- The subgraph isomorphism test inspects the host graph for a subgraph that is isomorphic to g_l, with matching node and edge labels.
- The embedding inspects the graph to identify pre-embedding edges. For non-invariant embeddings, this allows an operation such as "delete a node and all incident edges", without requiring g_l to state the number or labels of the incident edges).
- The application condition can arbitrarily inspect the host graph, including graph structure and attribute values. However, this method is clumsy and expensive for graph inspections that should be undertaken before, or as part of, the subgraph-isomorphism test.
- The attribute computation rules are used to compute attribute values for g_r^{host}, and may include arbitrary computations, including graph inspection.
- The control specification can inspect the graph to decide on rule order, and can use rule order to ensure that certain graph properties hold whenever a particular rule is invoked. Thus a rewrite rule can assume that certain graph properties hold whenever it is invoked. The control specification provides a powerful and convenient graph-inspection method, but it is also distances a system from the theoretical underpinnings of "pure" graph grammars.

In our work we feel a need for more natural means of host-graph inspection. During music-notation recognition [FaBl94], we use the application condition to perform general graph inspection, for example to test whether two nodes in g_l^{host} have the same neighbors. However, we do not find these application conditions to be a general and natural way to inspect the host graph. Math-notation recognition requires complex graph-inspection [Grba94], which we would like to express in a more direct way.

The PROGRES language provides strong integration of graph inspection and graph manipulation, through general control structures for directing the application of graph tests and graph productions [ZüSc92]. In addition, g_l can be augmented with *path* constructs, permitting complex, far-reaching examination of graph structure as part of the localization of g_l^{host}. The path acts like an application condition that tests graph structure; however, the path evaluation is integrated with the subgraph-isomorphism test. In a separate mechanism, independent of rewrite-rule application, complex path descriptions can be used for the computation of values for derived attributes (Section□2).

Global on-going graph inspection is proposed for the PROGRES environment in [NaSc91]. Global runtime conditions could be used to state host-graph conditions that should always (or never) hold. The graph-rewriting environment is responsible for checking these conditions between rule applications.

In summary, graph rewriting mechanisms could benefit from improved facilities for graph inspection. Current graph-rewriting mechanisms force the first graph-inspection operation to be a subgraph isomorphism test. This is not always appropriate or convenient.

4.4 Efficiency of a Graph Rewriting System

The cost of applying a single graph-rewrite rule is discussed in Section 3.5. The number of rule applications depends greatly on the organization of a rewrite system. Ordered graph-rewriting systems (versions without backtracking) tend to be more efficient than graph grammars [Bunk82a]. The non-deterministic nature of grammars necessitates backtracking during parsing, and may involve frequent testing of inapplicable rules. In contrast, an ordered graph rewriting system can directly transform an input graph into an output graph. Such transformational graph rewriting greatly reduces the number of subgraph-isomorphism tests (compared to parsing), because the control specification limits the number of production rules under consideration at any given time. Event-driven graph-rewriting systems can be highly-efficient in rule-application, only applying rules directly in response to an action by an external human operator.

These general statements do not imply that event-driven systems or ordered systems are preferable to graph grammars. Rather, the situation should be viewed the other way around. If an application is such that it can be implemented using event-driven graph-rewriting, then we have hope that it can run with acceptable efficiency. If the application calls for ordered (or partially ordered) graph rewriting without backtracking, then we have some hope that the system might run with acceptable efficiency. If the application calls for graph grammar use, then careful grammar and parser construction (context free, if possible) are necessary if there is to be hope of parsing speeds allowing large-scale practical use.

Event-driven graph rewriting seems close to near-term practical use as an implementation language, for example in implementing diagram editors [DoTo88] [Gött92] [EhHK92] [Reke94]. Here an external human user issues an interactive, fairly slow stream of editing operations. The primary time constraint is adequate response; Card et al. suggest less than 0.1 seconds for perceptual processing, 1 second for immediate response, and 10 seconds for a unit task [CaRM91]. Since only very few rewrite rules need be applied (or considered) in response to an editing operation, such a system may be practical using current graph-rewriting technology. Graph grammars, on the other hand, are further from acceptable performance in an end-user setting. Advances in parsing will help, but may not be sufficient to make graph grammars practical as an implementation tool. Thus it is important to expand research into the use of graph grammars at the specification and design level. A practical software development cycle could include the use of graph rewriting to form an executable specification (e.g. [ZüSc92]), even if it does not provide an acceptably efficient implementation.

4.5 Graph-rewrite Tools

Practical use of graph-rewriting depends heavily on the availability of development and debugging tools. Development of these tools is a complex and involved task. Here we briefly consider some of the issues involved.

Graph-rewriting notations have both textual and diagrammatic elements. At minimum, g_l and g_r need to be created and viewed diagrammatically (since textual graph descriptions are unwieldy and error-prone). Other features, such as the embedding, can be given either textually or diagrammatically. Some features, such as attribute computations and attribute-testing (in the applicability predicate) must be given textually. Development tools must achieve a smooth interaction between textual and diagrammatic elements. Rule displays must portray the relationships between textual and diagrammatic elements. Editors must integrate the editing both text and diagrams, ensuring that a complete and self-consistent rule is produced. Debugging may involve trace operations on textual as well as diagrammatic features of a rule.

Complex readability issues arise in development environments for graph-rewriting. Graph layout is an important and well-studied problem (e.g. [PaTi90]); note that different problems arise in the readable display of small rewrite-rule graphs versus the display of a large host graph. Debugging tools face the problem of displaying a large, time-varying host graph in a readable way.

Tools are needed to visualize the interactions among graph rewriting rules. A zoom operation may be needed to allow a user to see several rules simultaneously, for comparison purposes. Search and substitute operations are central to a text editor; similar operations are needed for a graph-rewrite-rule editor. The run-time interaction among graph-rewrite rules is difficult to display, since distributed changes are taking place in a large host graph, and since rewrite rules are often chosen non-deterministically. The user may need to trace how the application of one rule enables or disables future application of another rule. Development of graph-rewrite debugging techniques is an interesting research topic.

An extensive collection of tools are implemented or planned for the PROGRES language and the IPSEN environment. The implementation is organized into four layers: the common graph storage provides data abstraction, supporting the internal graph layer, the transformation layer, and tool control [NaSc91].

Göttler [Gött92] mentions a succession of implementations for executing ordered graph rewriting (Y and X notation). The unacceptably slow 1983 Fortran implementation was followed by a 1986 Lisp implementation (successfully used for a variety of structure editors). A new C implementation is under development, including a graphical editor for X notation graph-rewriting rules.

Pfeiffer describes development plans for a graphical editing environment for algebraic graph rewriting [Pfei90]. In the meantime, a textual representation of a graph grammar is compiled into C.

The GraphEd system [Hims93] provides extensive facilities for graph layout, with support for context-free graph rewriting. A derivation-step is accomplished by selecting a host-graph location and a graph-rewrite rule. A graph-rewrite rule is specified diagrammatically, with the single g_l node drawn large enough to enclose the g_r subgraph. Diagrammatic depiction of eNCE embeddings is supported. More extensive support of graph-rewriting is planned for the future.

A prototype implementation of algebraic graph transformation is described in [LöBe93]. The tool currently performs direct derivation steps in the single-pushout approach. Graph morphisms are illustrated graphically as lightly-drawn edges connecting the identified objects, either a pair of nodes or a pair of edges. Thus morphisms between edges are displayed as "higher-order edges" (edges between edges). To trigger rule application, the user selects a rule and a morphism for applying this rule. This selection process is facilitated by the use of abstract graphs (in which subgraphs are abstracted to single nodes, and bundles of morphism edges are abstracted to a single edge).

5. Selected Application Areas for Graph Rewriting

Graph rewriting has been applied to a variety of problem areas. Selected references are given here. See [Panel91] for excellent overviews of successes and problems in many graph-grammar application areas; these include developmental biology (cell division patterns), pattern recognition, parallel programming, visual languages, hypertext systems, software engineering.

5.1 Software Engineering

Graph rewriting has been successfully applied to various problems in software engineering, including as a specification tool, for the study of concurrent and distributed systems, for the implementation of functional languages, and for data-structure manipulation.

PROGRES (formerly called "PROGRESS") is a comprehensive language for ordered graph rewriting, designed to specify tools within the software engineering project IPSEN (Integrated Project Support ENvironment), but useful for other application areas as well. An excellent overview of both PROGRES and IPSEN, illustrated by a realistic example, is given in [ELNSS92]. Further information may be found in [EnSc85] [Nagl86] [EnLS87] [NaSc91] [Schü91] [ZüSc92]. Within IPSEN, the PROGRES language is used to specify the operational behaviour of document processing tools like syntax-directed editors, static analyzers, or incremental compilers and interpreters. For many of these applications, the host-graph has a dominant tree part, enriched by various context sensitive relations, that are derived via graph-rewriting rules [NaSc91]. Graph-rewrite rules constitute only a fraction of the text in a PROGRES specification. As discussed in [ELNSS92, Section 5], a major part of the specification is structural (node class hierarchy, attributes, attribute evaluation, edge types, paths); the operational part consists of rules and transactions. Language structures such as paths and derived attributes are used in place of graph-rewrite rules that only change attribute values or perform graph traversals.

The PROGRES language is designed to combine the advantages of object-oriented languages, attribute grammars, and ordered graph rewriting within a strongly-typed language. Important features of PROGRES include the following [Schü91] [ZüSc92] [ELNSS92].

- Node labels are defined hierarchically, with inheritance. Multiple inheritance is permitted [ELNSS92, p148].
- Graph components (nodes, edges and attributes) are strongly typed, within a *graph scheme*. For example, a PROGRES edge-type declaration states that an edge with label X can go from a node labeled Y to a node labeled Z. The resultant type checking is valuable in the construction of large graph-rewriting systems, helping to detect errors in graph productions, and controlling errors introduced during system extensions (such as the addition of new node labels). Explicit declaration of all possible edge types is facilitated by the hierarchical (multiple-inheritance) organization of node labels. For example, "edge type X: Thing -> Word" declares that an X-labeled edge can go from a node labeled Thing (or any label below "Thing" in the node-label hierarchy) to a node labeled Word (or any label below "Word" in the node-label hierarchy).

Graph schemata prove highly useful in practice; the type-checking of graph rewrite rules detects many specification errors before runtime (Andy Schürr, personal communication). This is particularly important for a language such as PROGRES, where programming errors often result in backtracking, rather than in runtime exceptions. Such errors can be very difficult to find at runtime.

- Evaluation of derived attributes is described within the graph scheme, decoupled from rewrite-rule application (Section□2).
- Graph rewriting productions are parameterized, and have return values. A transaction concept allows a series of rewriting steps to be treated as atomic, restoring the host-graph to an unchanged state in case of failure.
- Graph inspection is accomplished using tests and paths. A test is like the left half of a production; test execution involves locating a g_l^{host} that meets the stated conditions. Conditions within a g_l can include named paths, where a "path" is a separately-declared structure that describes a possible relationship between two nodes. Path expressions form the query part of PROGRES, analogous to database query languages [ELNSS92, p 159] -- the functionality of some tools (browsers, static analyzers) can be almost completely specified by path expressions alone.
- A comprehensive set of control structures direct the nondeterministic selection and application of rewriting rules. Control structures include nondeterministic and deterministic versions of And, Or, and Loop. Productions, transactions (composed out of tests and productions), and subdiagrams all have the following three characteristics: they are boolean and atomic (either succeeding and changing the host graph, or failing and making no change), and nondeterministic (selecting a particular matching g_l^{host}, or a particular "yes" or "no" edge in a subdiagram). In case there is no path through a subdiagram, the PROGRES interpreter restores the host-graph state and backtracks in the control specification, trying other g_l^{host} matches, or other paths through subdiagrams.
- The language notation integrates textual and graphical components of a graph-rewriting system. Productions use graphical notation for g_l and g_r. Elementary embeddings are indicated via node-correspondences within g_r nodes. More complex embeddings can be described textually. Graph-structure application conditions (called restrictions) are described textually, and are invoked by including the application-condition name within g_l. (A graphical user interface for the PROGRES environment is under development.)

[ZüSc92] reports that a prototype PROGRES-interpreter is under development. In any case, PROGRES specifications can be transformed (partly by hand) into corresponding implementations in a standard programming language [ELNSS92]. This long-standing research effort provides an impressive demonstration of the practical utility of graph rewriting.

The Δ-notation for graph rewriting was proposed by [KaLG91] [LoKa92], as a specification tool for concurrent languages and systems. Whereas PROGRES provides many powerful mechanisms in addition to graph rewriting, Δ-notation offers the attraction of being almost purely a graph-rewriting language. As illustrated in Figure□.3, the diagrammatic notation uses a Δ shape to notate the unique part of g_l (to be deleted), the unique part of g_r (to be inserted), the parts common to g_l and g_r (the required context), and undesired host-graph structure (the prohibited context). Elements in a Δ-rule can form a *-group, a syntactic shorthand for an infinite sequence of Δ-rules which contain, respectively, 0, 1, 2, 3... repetitions of the elements in the *-group. (Perhaps the term "syntactic shorthand" understates the significance of a construct that converts a finite set of graph-rewriting rules into an infinite set.) The Δ−notation is based on double-pushout algebraic graph rewriting, yet it permits deletion of a node without enumeration of incident edges. (Refer to Figure□ A Δ-rule containing node-deletion can be translated into an equivalent rule containing a *-group instead of node deletion; this rule in turn corresponds to an infinite set of rewrite rules that meet the requirements of the double-pushout construction). A useful style of computation is to structure Δ-rules into platforms, which communicate via triggers (Section□1).

A major strength of Δ-notation lies in the specification of concurrent systems, and its ability to hide synchronization details from the programmer. This is illustrated by solutions to a remote client-server problem, the dynamic dining philosophers problem, and definition of the semantics of the concurrent language Actors [KaLG91]. A variety of techniques for proving properties such as deadlock freedom and termination are presented in [LoKa92]. The structuring of Δ-rules into platforms is helpful for proof construction; for example, a different proof method can be used to demonstrate termination of each platform.

So far Δ-notation provides a paper-based specification method; no plans are mentioned for developing an implementation of Δ-notation. Implementation of Δ-program execution involves three steps [LoKa92, p. 100]: (1) *matching* (find a set of applicable rewrite rule, each with a suitable g_l^{host} and a unification of variable labels); (2) *conflict* *resolution* (find a non-conflicting subset of the applicable rewrite rules); and (3) *application* (apply the non-conflicting rules in parallel). Conflict resolution is 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. (The “prohibited context” construct greatly complicates this latter test: rule A might add host-graph structure that matches rule B’s prohibited context.) This complex conflict-resolution step provides the Δ environment with many desirable concurrency properties, allowing elegant solutions to traditional problems such as the dining philosophers and the bounded buffer.

Algebraic graph rewriting is being used for high-level data structure manipulation [Pfei90]. Graph rewriting is used in place of pointers, allowing the compiler to prevent common errors such as access to unallocated or deallocated structures, and failure to deallocate structures. Graph rewriting defines an abstract data type, which can be included as a separately compiled module in a source program. A flat label set is used, with enumeration of all legal host-graph edge-types (restricting the legal source and target node-labels, based on the edge label). *Expression-labeled nodes* are used as applicability predicates, and for recursive invocation of graph-rewriting rules. A graphical editing environment for graph-rewriting is under development. In the meantime, a textual representation of a graph grammar is compiled into C.

Graph rewriting is commonly used to implement functional programming languages [PeJo87]. A program is converted to an abstract syntax tree, which becomes a DAG (directed acyclic graph) due to repeated expressions. Graph rewriting is used to reduce this DAG to a single node. Only a few rewrite rules are needed, due to the limited number of constructs in a functional programming language. Carefully-tuned methods are used to achieve efficient implementation of this select set of rewrite rules.

5.2 Syntactic Pattern Recognition

Classification problems can be solved syntactically by constructing a separate grammar for the recognition of each pattern class. For example, [Fu82] uses string grammars to define the shapes of submedian, telocentric, and other chromosomes. Similarly, string grammars have been used to define normal and abnormal electrocardiograms. These applications involve a difficult grammatical inference problem, constructing appropriate grammars based on a set of samples from each pattern class. “Transformation systems” provide an alternative [Gold92]. While a grammar must generate all the patterns in a pattern class from an artificial origin (the start symbol), Goldfarb’s transformation system uses weighted substitution operations to capture the differences between positive and negative patterns. The transformation system is only required to discriminate between classes, not to reconstruct the classes as a whole. This discussion is phrase in terms of string grammars, but carries over to graph grammars as well.

Graph grammars are difficult to use in pattern recognition, due to the grammatical inference problem, the high cost of parsing, and the problem of noisy and distorted input (Bunke, in [Panel91, p45]). However, often these problems can be reduced by using ordered graph-rewriting instead of a graph grammar. This applies particularly to problems where a structural description, rather than a classification, is desired. The diagram-recognition systems of Section□3 illustrate th is point.

5.3 Picture Processing and Document Image Analysis

Graph grammars were originally proposed in [PfRo69] [Pfal72] as a means of solving picture-processing problems. String grammars had proven highly successful in the manipulation of one-dimensional problems. Since strings are not well-suited to the description of two-dimensional data, as in a picture or image, the extension to graph grammars was proposed. In the graph-representation of an image, image primitives are represented by labeled attributed nodes. Node attributes typically include the (x, y) image location of the primitive that gives rise to the node. Graph edges represent relations among the primitives; these can be spatial relations (such as “above”, “below”, “near”) as well as other relations derived during the course of graph-rewriting. The success of a graph-rewriting application depends heavily on the proper choice of image primitives, and on the reliability of the primitive-detection algorithms.

Some applications have a straightforward mapping from image to graph. These include graph-based notations such as circuit diagrams [Bunk82a], or software engineering diagrams such as SDL and SADT [Gött92]. In other cases, such as for music notation [FaBl93], difficult decisions are involved in mapping an image to a suitable graph representation. Further research may result in general guidelines to assist in devising a graph-representation of an image.

Although graph-rewriting research was instigated by picture processing problems, graph rewriting has not gained extensive use in this field. This is due, in part, to the following problems: (1) large graph-rewriting systems are difficult to write and debug; (2) parsing with graph grammars is computationally expensive; and (3) graph rewriting is difficult to use when uncertainty (associated with the identity of a picture primitive) is present. Research on these problems continues.

Ordered graph-rewriting has been used for various diagram-recognition applications, including circuit diagrams, music notation and math notation. In [Bunk82a], ordered graph rewriting is proposed for the recognition of circuit diagrams. The high cost of parsing is avoided; instead rule ordering is used to transform an input graph (representing line-segment primitives in a circuit-diagram image) into an output graph (representing a circuit, with recognized components such as transistors and capacitors). Rule ordering permits exhaustive application of error-correcting rules (which close small gaps in lines and eliminate small overhanging lines) prior to the application of the component-recognition rules. This approach is extended to music-notation recognition in [FaBl93]. The semantics of music notation depend both on nearby symbols (such as a sharp preceding a note) and on distant symbols (such as a distant key signature and a note). Thus, the necessary node-interactions cannot be predicted during construction of the input graph. Instead, a discrete input graph is used, and the necessary edges are built using Build, Weed, Incorporate phases of rewrite-rule application. (The Build phase creates edges between potentially-interacting nodes. The Weed phase deletes excess or conflicting associations. The Incorporate phase collapses the information from nodes and edges into the attributes of the remaining nodes.) This work is extended to address the uncertainty that may be associated with the identity of the notational primitives [FaBl94], and to perform recognition of mathematical notation [Grba94].

A plex-grammars recognition system for three-dimensional objects is discussed in [LiFu89]. A 3D surface may be partially or totally occluded in the image plane. This makes it difficult to construct a three-dimensional plex as input for the recognizer. A semantic-directed top-down backtrack recognizer is used. Results are shown for several images of machine parts.

Plex-grammars have been used to interpret dimensions in engineering drawings [CoTV93]. The parser mixes top-down and bottom-up processing. The processing of errors and uncertainty is postponed until after contextual information is obtained from higher-level analysis.

Graph-grammars are used in [EgPM92] to develop a visual programming environment for medical protocols. To achieve good response time and free-form editing, a syntax-neutral graph editor is used in conjunction with compilers for various diagram types. This arrangement avoids the effort of producing custom-tailored diagram editors for each visual language. It also avoids the rigid top-down approach imposed on users by syntax-directed editing. The authors have built a graph-grammar parser and tested it in three visual-language compilers. They conclude that attributed graph grammars are an appropriate tool for the specification and generation of visual-language parsers.

5.4 Diagram Editors

Graph rewriting has been applied to the editing of a variety of diagrams. Göttler et al. use X and Y notation (Section⍍.2) to express executable requirements-specification for diagram-editing, used to created editors for SADT, EADT, Petri-nets, and HIPO [Gött83] [Gött87] [GöGN91] [Gött92]. Rekers [Reke94] applies the $\vdash$ notation (Section⍍.2) to the definition of syntax for graphical editors. A syntax for diagrams of finite state machines is provided as an illustration.

Dolado and Torrealdea use graph-rewriting to formally describe Forrester diagrams [DoTo88]. Nodes in a Forrester diagram denote rates, materials sources, material delays, information delays, and so on. Edges denote information or flow coupling. Graph-rewrite rules define primitive operations for modifying a Forrester diagram -- productions insert and delete various types of nodes and edges. The control specification uses host-graph inspection to limit the currently-allowable set of productions, so that a legal Forrester diagram is guaranteed to result. This ordered graph rewriting system is capable of generating all legal Forrester diagrams. In practical use as an editor, a human operator issues commands that direct the application of production rules, with the system issuing error messages in the event the operator selects an operation that is currently disallowed. As of 1988, the system was formally defined but not implemented.

5.5 Databases and Semantic Nets

Ehrig and Kreowski [EhKr80] discuss the application of graph rewriting in the construction of data base systems. A small library system is constructed as an example. The library database is represented as an attributed graph, with nodes for books, authors, library patrons, publishers, and catalog numbers. Each rewrite rule represents a transaction, such as loaning out a book, accepting a returned book, acquiring a new book, or adding a new library patron. Parallel rule application is permitted in this distributed database. It is critical to ensure that proper synchronization is observed and consistency is maintained. Thus the authors choose the algebraic approach to graph rewriting (Section⍍.5.1), which has strong theoretical results (such as the Church Rosser theorem) that are used to analyze interactions among transactions, achieve synchronization, and prove consistency conditions. A follow-on to this system is presented in [EhHa86], illustrating the use of application conditions to enforce constraints including (1) reader-numbers must be distinct, and (2) a maximum of ten books can be checked out by a single reader.

Ehrig et al. propose the use of algebraic graph grammars to generate well-formed semantic networks [EhHK92], and mention this as a possible basis for a syntax-directed editor used to build up a semantic net. The authors discuss the need for structuring principles, to organize the information in a large semantic net. Graph-rewrite rules are proposed as a method for defining the hierarchical structure of a graph, applying a rewrite rule in the forward direction to expand a parent node into the components it encompasses, and applying the rewrite rule in the reverse direction to collapse the components back into a representative parent node. This is an interesting proposal, but one that hampers the accessibility of the information in a semantic net. The information in the semantic net is no longer stored purely as a graph; much of the information is encoded in the graph rewrite rules themselves. In order to find information in such a semantic net, we must search all graphs derivable from the current host graph. Further research is needed to establish whether this can be done practically. A more straightforward solution is to encode hierarchical graphs in a static data structure [Hare88] [SiGJ93], with zoomed graph views generated on request for the user. Graph rewriting could be applied to hierarchical graphs, probably with minimal modifications to the graph rewriting mechanism.

5.6 Biology

Parallel graph rewriting is extensively used in biology and in computer graphics (visualization of biological structures). Parallel rewriting is needed to model cell divisions; the graph rewriting captures both the developmental process and the final structure of the organism [Panel91, pp. 41-43, 47-48]. Details are omitted here, since our focus is on sequential graph rewriting.

6. Summary and Conclusions

Graphs are high-level, versatile constructs with widespread practical use. Graph rewriting is a natural way to specify the manipulation of these high-level constructs. We have reviewed a variety of notations and mechanisms that have been proposed for graph rewriting. We have discussed various approaches that have been used to express computations via graph rewriting. Graph rewriting is a promising formalism, well-understood theoretically, with the potential to be practically useful in a tremendous variety of application areas.

A variety of important tradeoffs face a software developer choosing to use graph rewriting. Proper resolution of these tradeoffs is often application dependent, or a matter of taste.

- Selection of a rewrite mechanism has important consequences for the power of the embedding, formal properties of the rewrite system, readability and intellectual manageability, and efficiency (Section 1.3).
- Details of the rewrite mechanism have important practical consequences: isomorphisms versus general graph morphisms (Section 1.3.4), variable node and edge labels (Section 2.5), induced subgraph matching (Section 2.7.2).
- Design of the host-graph representation is critical and involves a tradeoff between using graph structure or graph attributes to represent certain information. Hierarchically-organized labels can be very helpful (Section 2.1).
- Given a particular rewrite mechanism, graph rewriting rules can be organized to use unordered rewriting, graph grammars, ordered rewriting (partially or totally ordered, with or without backtracking), or event-driven rewriting (Section 3). The choice of rule organization has great impact on the nature, theoretical properties, and efficiency of the graph-rewriting system.

Faced with this bewildering array of choices, a software designer can feel ill-equipped to address a fundamental question: how should he or she express, develop, and debug computations using graph rewriting? For practical use of graph rewriting, a "computational style" is at least as important as the exact choice of graph rewriting formalism (Section 4).

Abstraction facilities are critical, to allow structuring of large host graphs and of large graph-rewriting systems. Possible structuring facilities include hierarchical labels (Section 2.1), hierarchical graphs (Section 2.7.1), and control abstractions such as subgrammars and rule-parameters. In addition to these abstraction capabilities, rewrite-rule authors need technology for incorporating graph rewriting into a larger software system (Section 4.2). Finally, many graph-rewriting mechanisms could benefit from more flexible host-graph inspection operations (Section 4.3). Further difficulties facing practical use of graph-rewriting are discussed in [Panel91].

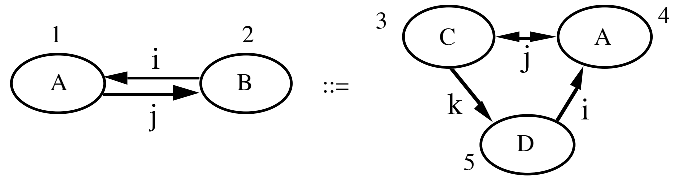
Ideally, anyone faced with implementing a graph-oriented computation should consider whether graph rewriting is an appropriate mechanism. However, it can be enormously difficult to envision *how* to perform the graph rewriting. The graph-rewriting community can respond by providing better tools, better notations, and a “graph-rewriting culture” -- a readily-accessible body of knowledge, experience, and algorithms relating to the practical application of graph-rewriting.

Appendix A. Graph Rewriting Mechanisms

Graph rewriting mechanisms are reviewed in the order from most-powerful to least-powerful embedding mechanisms.

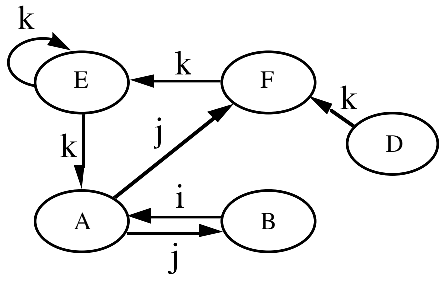
A.1 Unrestricted Embeddings: Expression Notation

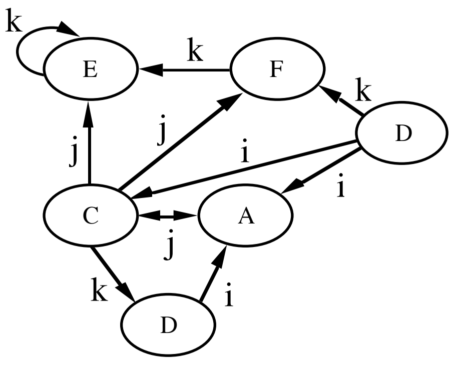
Nagl formalizes and generalizes embedding specifications using an expression notation [Nagl79] [Nagl87]. Expression graph rewriting operates on an (un)directed, edge-(un)labeled, node-labeled graph, with g_l^{host} required to be an induced subgraph. An embedding specification consists of 2n expressions, where each pre-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 j. The notation is illustrated in Figure A.1.



Embedding: $In_i = (D\ I_k\ O_j\ (1);\ 3,4)$; $Out_j = (3;\ O_j \cup I_k\ I_k\ (1,2))$; etc.

(a) a rewrite rule using the expression notation





(b) initial host graph

(c) resultant graph

Figure A.1 Illustration of the expression embedding mechanism defined by [Nagl79]: the production of part (a), applied to the host graph (b), produces the result (c). Node labels are inside the node while node denotations are outside. In_i embeds incoming edges of label i, and Out_j embeds outgoing edges of label j. In_i specifies that the set of nodes "$D\ I_k\ O_j$ (1)" are source nodes of i-labeled edges ending in nodes 3 and 4 of g_r^{host}. The expression "$D\ I_k\ O_j$ (1)" describes the following set of nodes: starting from node 1 of g_l^{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, Out_j specifies that the set of nodes "$O_j \cup I_k\ I_k$ (1,2)" are target nodes of j-labeled edges originating in node 3 of g_r^{host}. The expression "$O_j \cup I_k\ I_k$ (1,2)" describes the following set of nodes: starting from nodes 1 or 2 of g_l^{host}, follow a chain of two incoming edges with label k or one outgoing edge with label j.

The theoretical significance of expression graph grammars is apparent in the hierarchy for graph languages shown in Figure A.2 [Nagl79]. Monotone, context-sensitive, and context-free graph grammars produce the same graph languages, despite the differences in restrictions on g_l. This is because the expression mechanism provide a means of inducing context by embedding rather than by complex forms for g_l.

Expression graph rewriting is the most general form of graph rewriting. Formulation of embedding expressions is undoubtedly difficult, yet such expressions permit a compact and time-efficient graph rewriting system. It may take several rewrite rules of a non-expression type to simulate the effect of a single expression rewrite rule. Time-efficiency results from subgraph-isomorphism testing involving a smaller number of nodes: g_l can be relatively small since expressions provide contextual information. It is difficult to visualize the effect of an embedding expression; this problem is reduced by the diagrammatic X, Y and Δ notations for rewrite rules (Section A.2). Alternatively, paths in the PROGRES language provides a more readable textual notation for complex embeddings [EnLS87, p. 193].

[Exp-**unrestricted**] **=** **[Recursively enumerable graphs]**

$\cup$

[Exp-**monotone**] = [Exp-**context-sensitive**] = [Exp-**context-free**]

$\cup$

[Exp-**linear**] = [Exp-**regular**]

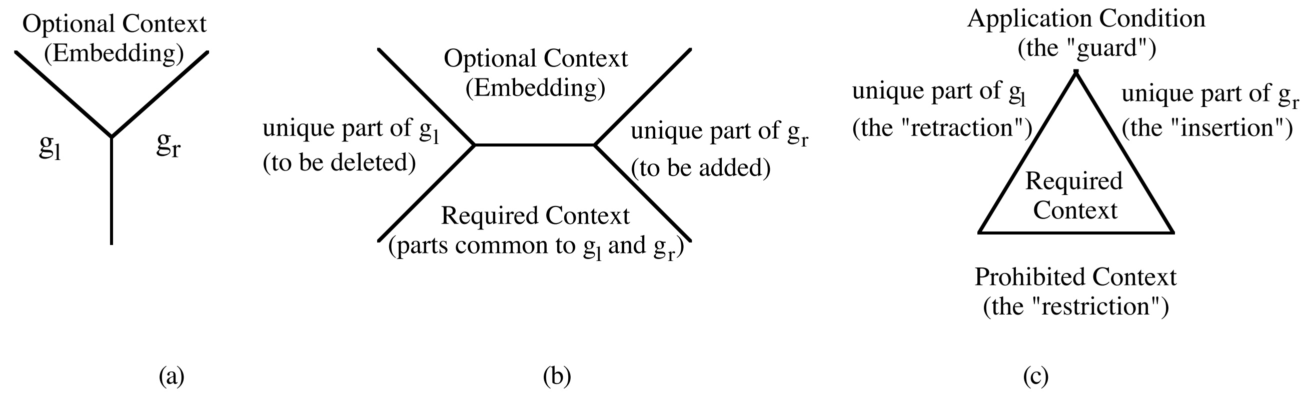
Figure A.2 The hierarchy for graph languages generated by graph grammars using the unrestricted expression embedding mechanism [Nagl79]. (The language of a graph grammar is defined to be the set of all terminally labeled graphs that can be derived from the initial graph S.) Three language classes emerge, where production types are defined as follows (assuming that the basic primitive being manipulated is a node):

- *unrestricted* production – no restriction on g_l or g_r.
- *monotone* production – the number of nodes in g_l is less than or equal to the number of nodes in g_r.
- *context-sensitive* production – a portion of g_l exists as a subgraph of g_r.
- *context-free* production – g_l consists of exactly one node. (This node has a nonterminal label.)
- *linear* production – a restricted context-free production, where g_r contains at most one nonterminally-labeled node (plus any number of terminally-labeled nodes).
- *regular* production – a restricted context-free production, where g_r has a unique maximum node; all other g_r nodes are predecessors of the maximum node. The maximum node may have a terminal or non-terminal label, all other g_r nodes have terminal labels.

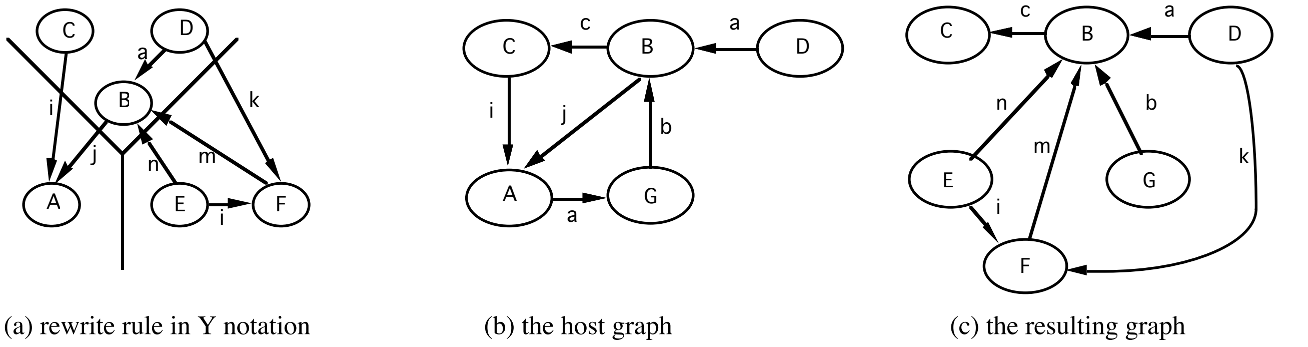
Recently, [AiNa91] propose node-replacement graph rewriting with path-controlled embedding (n-PCE). Although the authors do not remark on this, their approach strongly resembles expression graph rewriting: RestGraph nodes which take part in the embedding are connected to g_l^{host} by a path, where a path is characterized by a sequence of edge labels. Differences between n-PCE and the expression approach are: (1) n-PCE restricts g_l to be one labeled node (as in NLC, Section A.3); and (2) n-PCE does not use node labels in the selection of source and target nodes of post-embedding edges.

A.2 Unrestricted Embeddings: Diagrammatic Notations

Textual specifications of embedding mechanisms can be difficult to read. To address this problem, diagrammatic notations for embedding have been developed. Göttler's Y notation [Gött79] [Gött83] and X notation [Gött87] [GöGN91] [Gött92] have been used extensively 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. An overview of these notations is given in Figure A.3. The depiction of embedding is illustrated by the Y notation of Figure A.4. Further illustrations are provided by Figure 2 and Figure 3.



Figure▲.3 Three diagrammatic notations for graph-rewriting operations; diagrammatic depiction is used not only for g_l and g_r, but also for the embedding. In Y and X notations, the embedding is shown in the so-called optional context: these diagrammatic depictions of embedding are used *if* they match in the host graph. The required context, on the other hand, must match in order for the rewrite rule to be applied. In the 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 Δ notation includes application conditions: host-graph structure that must be present is placed in the required context, host-graph structure that must not be present is placed in the prohibited context, and restrictions on labels and attributes are expressed textually in the guard.



Figure▲.4 An example of Y notation. The rule shown in (a) is applied to the graph shown in (b) resulting in the graph shown in (c). Rule (a) displays g_l on the left, g_r on the right, and the embedding specification on top. Edges between g_l and the embedding specification depict a subset of pre-embedding edges. Edges between g_r and the embedding specification depict the corresponding post-embedding edges. (Edges between g_l and g_r are not permitted.) The embedding specification is an *optional context* -- after g_l^{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 g_r causes the formation of a post-embedding edge. An optional-context node must have a connection to g_r in order to have an effect on the graph rewrite. For example, node C and its i-labeled edge (in (a)) emphasize that the i-labeled edge is deleted (if present), but the rewriting effect would be the same without their inclusion in the embedding specification.

In many graph-rewrite rules, g_l and g_r have parts in common. In most notations, including the Y notation of Figure▲.4, these common parts are notated twice, once in the definition of g_l and again in the definition of g_r. The X and Δ notations instead notate the common parts just once, in a separate area called the *required context*. The required context must exist for the rewrite to occur, but it is left unchanged by the rewrite.

X-notation rules are analogous to Y-notation rules except that the bottom part of the X contains the required context: the subgraph which is common between g_l and g_r. The unique portion of g_l is given in the left-hand side of the X, and the unique portion of g_r is given in the right-hand side of the X. The embedding (the optional context) is given in the upper portion of the X. In implementing graph rewriting using X-rules, only the portion of g_l given in the left-hand side of the X needs to be removed. Göttler reports that this improves the efficiency of the implementation by orders of magnitude [Gött92]. This increase in efficiency may seem astounding, since the costly subgraph-isomorphism test is not affected by the change from Y to X notation. However, in Göttler's rewriting systems subgraph isomorphism is extremely fast due to the presence of unique cursor nodes (demon nodes) that indicate the location of the desired g_l^{host}. Thus most of the execution time is spent directly on graph manipulations during rule application, which for simple rules (add an edge, update an attribute value) are far more involved for Y notation than X notation.

As a practical matter, the choice between Y and X notations should be made on the basis of user preference, not implementation efficiency. (The implementation can easily make an internal conversion of rules from the Y to the X form.) In our limited trials, we find Y notation easier to use: during rewrite-rule development we frequently update either g_l or g_r, and can do this without concern about the exact portions that are common to g_l or g_r. On the other hand, for applications where rewrite rules have large required contexts, developers may prefer the X notation for its compactness. This compactness results not only from savings in depicting g_l or g_r, but also from savings in depicting the embedding (Figure 2).

The Δ-notation was proposed by [KaLG91] [LoKa92], as a specification tool for concurrent languages and systems. Δ-notation is a superset of X-notation. The portions of an X-notation rule are placed as follows: the unique part of g_l is to the left of the Δ, the required and optional contexts are in the middle of the Δ (with a * next to each node of the optional context), and the unique part of g_r is to the right of the Δ. In addition, the area below the Δ contains the *prohibited context*, a subgraph connected to g_l; if this subgraph can be matched to the host-graph area surrounding g_l^{host}, then rule application is forbidden. Thus, in Δ notation the application condition is given in two portions: graph-related restrictions are shown diagramattically as the required and prohibited context, whereas attribute-related restrictions are given textually. Δ-notation uses *-groups to denote repetition: elements of a *-group are repeated zero, one or many times. This is used both for optional context (the embedding) and to express complicated graph matches. The *fold* construct, denoted by subscripts on node labels, is used to permit several nodes to be matched to a single host-graph node. While isomorphism is used to locate g_l^{host}, the use of folds permits selected application of a more general morphism. This is convenient, for example, in describing a rewrite of a circular list of nodes: a long list can provide unique node-matches for g_l, where a short list requires several host-graph nodes to match a single g_l node. The dining philosopher problem illustrates this [LoKa92]. To structure Δ rewriting systems, rules can be organized into platforms, with special trigger nodes placed into the host-graph to communicate among platforms.

Compared to Δ notation, the X notation has a more readable separation between required and optional context. On the other hand, the prohibited context of the Δ notation is useful in practice. This same restriction would have to be expressed textually in Y or X notation. (An extension of this construct to a *conditional* prohibited context is discussed in Section 2.3.)

Rekers independently introduces another graphical notation for rewrite rules [Reke94]; this notation uses a $\vdash$ delimiter, with g_l above the line, and g_r and required context below. Dotted lines are used to distinguish required-context from g_r. A diagrammatic notation for a limited prohibited context is proposed; for example, the required absence of an edge is indicated by a crossed-out edge starting or ending in g_l.

Special diagrammatic notations can be developed for context-free graph-rewrite rules. For example, the GraphEd system [Hims93] limits g_l to a single node. This node is drawn quite large, defining two regions (inside and outside). The g_r subgraph is drawn inside, and optional-context nodes are drawn outside. An eNCE embedding is specified as a set of edges between g_r nodes and optional-context nodes. This provides a compact and readable notation for context-free rewrite rules.

A.3 Depth1 Embeddings: NCE and NLC

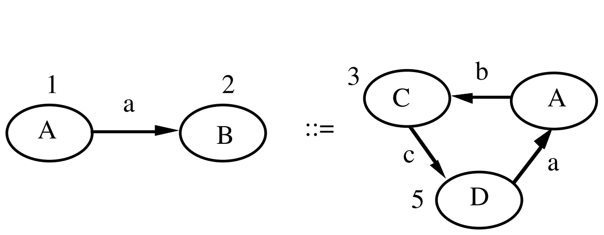
In [JaRo82], Janssens and Rozenberg formally define graph rewriting with neighbourhood controlled embedding (NCE). This is a Depth1 embedding mechanism -- a post-embedding edge can connect to a RestGraph node only if that node used to be connected to a pre-embedding edge. Edge labels may change, but edge-orientation is preserved in the transformation from pre-embedding edges to post-embedding edges. This is illustrated in Figure A.5.

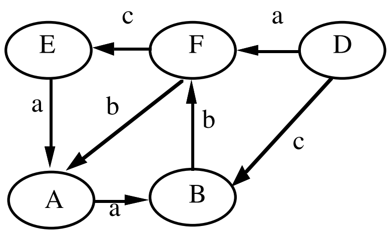
The NCE model is predated by the related and theoretically significant Node Label Control (NLC) model of graph-rewriting [JaRo80a] [JaRo80b]; see also [Roze87], [Welz87], [EnLR87], [EnRo91]. The NLC approach is restricted to cases where g_l consists of one non-terminally labeled node; a collection of NLC rewrite rules is used to define a context free graph grammar. Instead of specifying a separate embedding for each rewrite rule, one global embedding is defined for the grammar as a whole. This embedding consists of a set of pairs (x,y) where x is the label of a node in g_r^{host} and y is the label of a node in the direct neighbourhood of g_l^{host}. A pair (x,y) implies that a post-embedding edge is to be added between a node in g_r^{host} with label x and a node in the direct neighbourhood of g_l^{host} with label y. Hence, this approach is node-label controlled. To apply NLC to directed graphs, the embedding consists of two sets of pairs (x,y); one set embeds the incoming edges, while the other embeds the outgoing edges. Optionally, to alter the direction of embedding edges, four sets of pairs (x,y) can be used; two to embed edges that don't change direction and two to embed edges that do change direction. Finally, if the graph has edge labels, then the pairs (x, y) can be replaced by four-tuples (x, a, y, b), to convert a-labeled pre-embedding edges to b-labeled post-embedding edges. One can see that NCE (Figure▯.5) is a variant of this that is applicable when g_l consists of more than one node.

The NLC approach is theoretically significant in that it defines a (node-oriented) class of context-free graph grammars. This is useful in determining whether properties of string context-free grammars can be generalized to graph grammars. For example, string context free grammars have the property that the result of a derivation is independent of the order in which the productions are applied. This property guarantees the existence of derivation trees. However, NLC graph grammars, do not necessarily have this property. Graph grammars with this property are said to be *confluent* or *order-independent*. For NLC grammars to be confluent, no edge can connect two non-terminally labeled nodes in g_r or the initial graph. NLC grammars with this restriction are called *boundary* NLC or B-NLC [RoWe86] [Welz86] [Welz87] [EnLR87] [EnLW90]. Boundary NLC grammars do not have normal forms. However, Chomsky and Greibach normal forms do exist for the boundary form of eNCE grammars [EnLW90]. Here "eNCE" denotes an edge-labeled context-free NCE grammar.

The context-free nature of NLC graph grammars limits their expressive power. Extending the model to "handle NLC" grammars, where g_l consists of two nodes connected by an edge, greatly increases expressive power [MaRo87a]. Handle NLC grammars are capable of generating the recursively enumerable set of terminally labeled graphs.

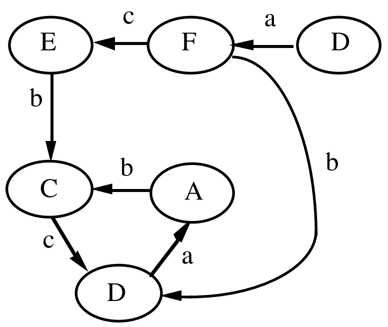
Edge-label controlled (ELC) grammars have been proposed by [MaRo87b] [MaRo90]. These grammars are analogous to the NLC approach, except that the basic primitive which is rewritten is a labeled edge rather than a labeled node. Hence, the embedding is defined in terms of edge labels.

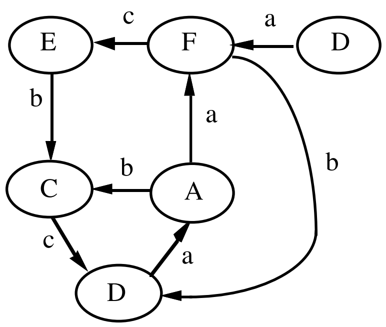




Embedding: In = {(1,3,a,b,E), (1,5,b,b,F)}
Out = {(2,4,b,a,F), (2,5,c,a,E)}

(a) the rewriting rule (b) the initial host graph





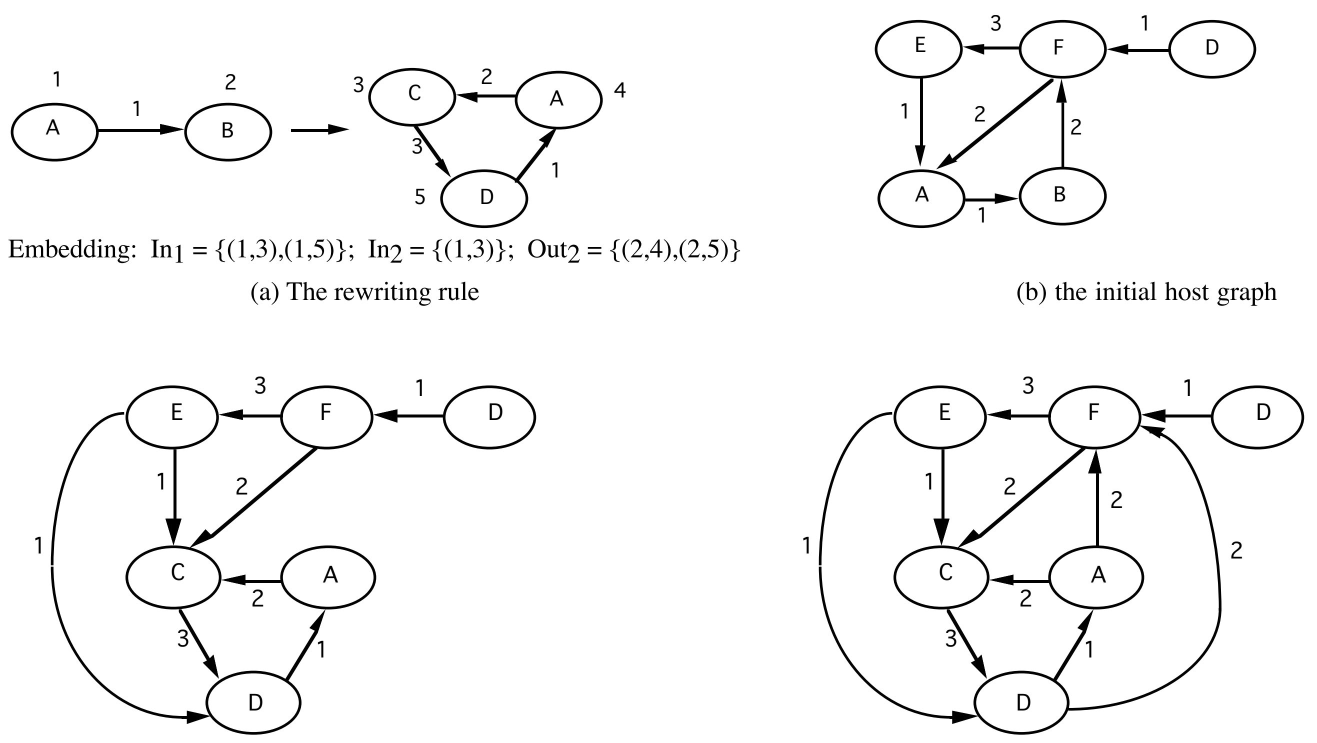
(c) after processing the In part of Embedding (d) final result

Figure A.5 An example of the NCE rewriting mechanism, applied to an edge-labeled directed graph (edNCE). In the rewrite rule (a), node labels are inside the node while node denotations are outside. The embedding specification consists of two sets In, Out. The set In handles pre-embedding edges which terminate in g_l^{host}, while the set Out handles those which originate from g_l^{host}. Each set consists of quintuples of the form (n,n',w_1,w_2,v), where n denotes a g_l node, n' denotes a g_r node, w_1 and w_2 are edge labels, and v is a node label in RestGraph. To apply a quintuple, look for any pre-embedding edge that has label w_1 and connects node n of g_l^{host} to a RestGraph node labeled v; if such an edge is found, it becomes a post-embedding edge of label w_2 connecting node n' of g_r^{host} to that same v-labeled RestGraph node. In this example, the pre-embedding edge with label c does not match any quintuple, and hence does not give rise to a post-embedding edge. Similarly, pre-embedding edges that match several quintuples give rise to several post embedding edges.

As noted in [JaRo82], this depth1 embedding mechanism can be restricted to a simple embedding mechanism (which preserves the labels of the pre-embedding edges), by using triplets (n,n',v) instead of the quintuplets (n,n',w_1,w_2,v). In this case, all pre-embedding edges between node n and a v-labeled host-graph node are transformed into post-embedding edges connecting to node n'. One could also use quadruplets (n,n',w,v), if the desired embedding of an edge depends on its edge label.

A.4 Elementary Embeddings: Schneider's Notation

Early work with graph grammars [PfRo69] [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 [Nagl79]. Schneider's mechanism is illustrated in Figure A.6. Comparing this to the NCE mechanism shown in Figure A.5, Schneider's mechanism cannot express (1) embedding conditional on the label of a RestGraph node, or (2) a change in the label of embedding edges. On the other hand, the simplicity of Schneider's mechanism makes it relatively readable and easy to use, as in [Bunk82a], [FaBl93].



Embedding: $In_1 = \{(1,3),(1,5)\}$; $In_2 = \{(1,3)\}$; $Out_2 = \{(2,4),(2,5)\}$

(a) The rewriting rule

(b) the initial host graph

(c) after processing the In_w parts of Embedding

(d) final result

Figure A.6 An example of Schneider's embedding mechanism. In the rewrite rule (a), node labels are inside the node while node denotations are outside. The embedding specification consists of sets In_i, Out_i, which specify the transformation of edges with edge label i. Each set In_i and Out_i contains pairs of the form (n,n'), where n denotes a node in g_l and n' denotes a node in g_r. Such a pair causes a pre-embedding edge connected to node n in g_l^{host} to be transformed into a post-embedding edge connected to node n' in g_r^{host}. The orientations, labels, and RestGraph-endpoints of the embedding edges remain unchanged. The notation simplifies if there are no edge labels (use only one set In and one set Out) or if edges are undirected (use one set instead of two In Out sets).

A.5 Gluing Models

In the gluing approach to graph rewriting, a rule's gluing isomorphism acts as the embedding specification. Both nodes and edges can be used as gluing points; here we consider the case of nodes. The rewrite rule designates a subset of g_l nodes as gluing nodes, the gluing isomorphism puts these nodes into correspondence with a subset of g_r nodes. To apply a rewrite rule, g_l^{host} is located in RestGraph. Next, the *gluing condition* is checked: every pre-embedding edge must connect to a gluing node in g_l^{host}. (If there is a pre-embedding edge that connects to a non-gluing node in g_l^{host}, then the rule application is disallowed.) The gluing isomorphism provides an invariant embedding, specifying which g_r^{host} node inherits the embedding edges of each gluing node in g_l^{host}. Because of the simplicity of invariant embedding, a graph rewrite step does not need to remove pre-embedding edges and replace them by post-embedding edges. Instead, it removes g_l^{host} except for the gluing nodes, and then adds g_r^{host} except for its gluing nodes (connecting, instead, to the gluing nodes left by g_l^{host}). This carries over the embedding edges unchanged.

The restricted nature of this model has important advantages for the analysis of rewrite-rule interactions. The effect of a rewrite rule is localized to g_l^{host}, and perhaps even localized to the non-gluing part of g_l^{host}. (Some models allow gluing nodes and/or edges to change label or direction, for example [Schn93].) This localization allows analysis of parallel or order-independent rule application. Theoretical results are derived from a formulation of gluing in terms of category theory.

Of course, the restricted nature of the invariant embedding also limits the expressiveness of a gluing rewrite rule. A rewrite rule expressed with a more complex embedding can be translated to an invariant embedding by suitable expansion of g_l and g_r. 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 g_l and g_r. To enable such translation to invariant embedding, additional notation for g_l and g_r may be needed. Referring back to Figure□, consider the example of a rewrite rule to "delete a node with label A". Using an analogous embedding, g_l is a single node labeled A, g_r is a null graph, and the embedding specification is null. To translate this to an invariant embedding, g_l 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 would have to express g_l using some notation for replicated nodes and edges.

We now briefly review three main gluing models: the algebraic model (Section A.5.1); edge replacement systems (Section A.5.2), and hyperedge replacement systems (Section A.5.3).

A.5.1 The Algebraic Approach

The algebraic approach to graph grammars ([EhPS73], [Ehri87], [EhKL91]) was first proposed in 1973 as a way of generalizing string concatenation to a graph operation. (It is also referred to in the graph-grammar literature as the Berlin approach. Ehrig et al. extend the approach from graphs to more general structures such as partial graphs [EKMRW81].) The *pushout* construct of category theory is adopted in this approach; well-known techniques and results from category theory are applied to graph derivations. Because of its strong mathematical basis, this approach has received much attention by those interested in the theory of graph grammars.

As outlined above, a graph rewrite rule consists of g_l (with designated gluing points), g_r (with designated gluing points), and an (iso)morphism between the two sets of gluing points. Thus, g_l consists of gluing points (which are not deleted), and other graph structure (which is deleted); g_r consists of a replication of the gluing points, along with new graph items. The gluing morphism can be modeled using a double (or single) pushout, applied in the category of graphs and total (partial) graph morphisms.

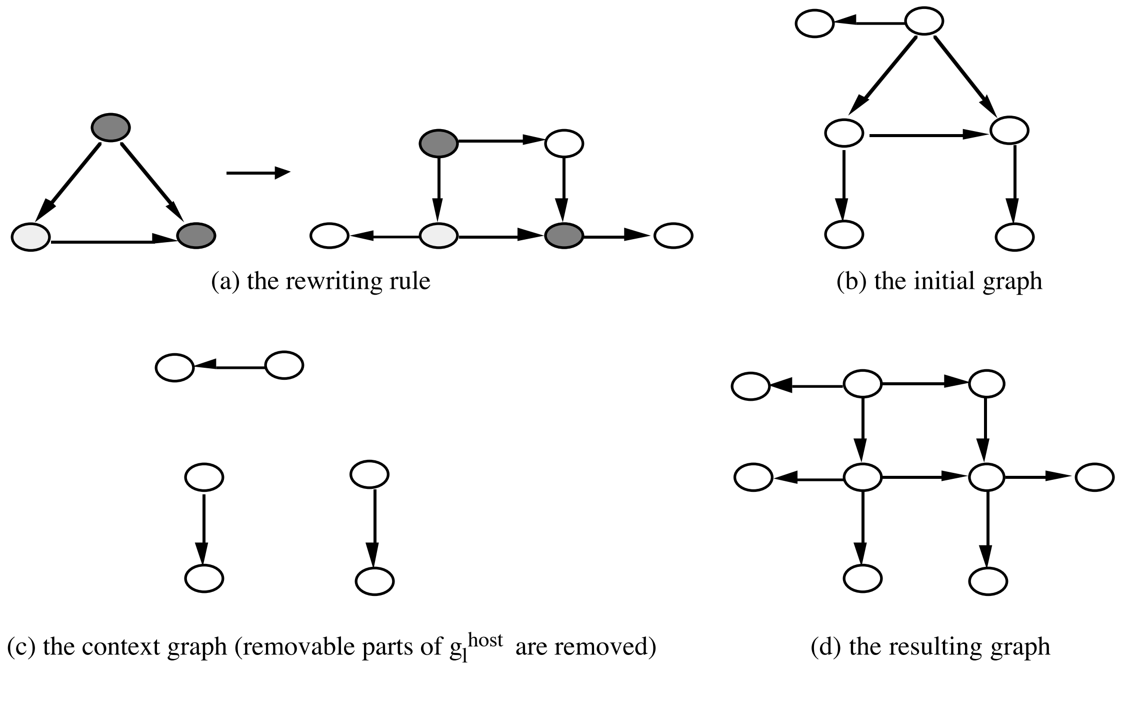

(a) the rewriting rule
(b) the initial graph
(c) the context graph (removable parts of g_l^{host} are removed)
(d) the resulting graph

Figure A.7 Application of a rewrite rule using the algebraic approach. Shaded nodes in (a) represent the injective mapping of the distinguished gluing points. Application of the rule shown in (a) to the graph shown in (b) results in the graph shown in (d). The context graph (c) is an intermediate step.

Figure A.7 illustrates a rewrite rule in the double-pushout approach, as well as the definition of the *context graph*. The application of a rewrite rule, transforming a graph g into a graph g', can be viewed as a double gluing: the gluing of g_l^{host} and the context graph along the gluing points (which produces g), and the gluing of g_r^{host} and the context graph along the gluing points (which produces g'). This can be represented by a pair of *gluing diagrams* [Ehri87]. Since gluing diagrams are pushouts in the category of graphs, techniques and results from category theory can be applied to graph derivations. This enables proofs of the Church-Rosser Property, parallelisms, amalgamation, and concurrency [EhKL91].

The algebraic approach is advantageous in applications where concurrency and synchronization must be proven. For example, the algebraic approach has been used to develop a toy database system for managing library transactions [EhKr80] [EhHa86], and is being used for high-level data-structure manipulation [Pfei90].

A.5.2 Edge Replacement Systems

The edge replacement grammar discussed in [HaKr83] and [HaKr87a] is a restricted case of both the algebraic approach (Section A.5.1) as well as the hyperedge replacement approach (Section A.5.3). Here, g_l is restricted to be a *handle:* a graph consisting of two unlabeled nodes connected by a non-terminally labeled edge. This restricted form of g_l makes the search for an isomorphic g_l^{host} inexpensive. The two endpoints of the handle are gluing nodes. The gluing condition is automatically satisfied, since all nodes in g_l are gluing nodes. Application of a rewrite rule results in replacement of an edge by the graph g_r^{host} (Figure A.8). Of all the gluing models, the rewriting mechanism of edge-replacement grammars is most similar to that of string grammars: the labeled edge is analogous to a character.

Edge-replacement grammars define context-free graph languages, since each production specifies the replacement of one labeled edge by a graph. (This edge-based definition contrasts with the node-based definition of context free graph grammars given by the NLC grammars of Section A.3.) Edge-replacement grammars are considered analogous to context-free string grammars. Carrying string-grammar properties over to edge-replacement graph grammars, Chomsky's pumping lemma has a graph-grammar analogue [Kreo79], but no analogue of Chomsky normal form has been defined.

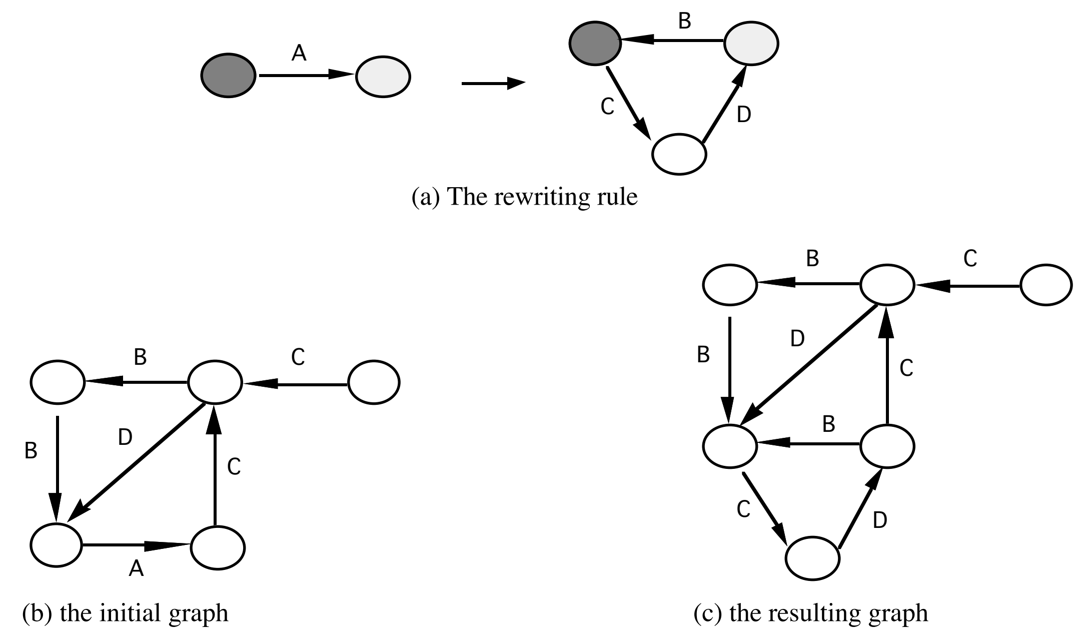


Figure A.8 An application of an edge-replacement rewriting rule. Shaded nodes in (a) represent the injective mapping of the distinguished gluing points. Application of the rule shown in (a) to the graph shown in (b) results in the graph shown in (c)

A.5.3 Hyperedge Replacement Systems

The hyperedge replacement systems described in [HaKr87b] [HaKV89] [DrKr91] manipulate *hypergraphs*, which consist of *hyperedges*. Each hyperedge has a label, m ordered incoming tentacles, and n ordered outgoing tentacles (Figure A.9a). A hypergraph consists of a set of hyperedges joined at sockets (Figure A.9b). Each rewrite rule specifies the replacement of a hyperedge by a hypergraph. Thus these systems define another class of context-free graph languages. The embedding is implicitly specified: the terminal nodes of the g_l hyperedge act as the set of gluing points k_l. The g_r hypergraph consists of a set of gluing points k_r, where there is a bijective mapping between k_l and k_r. This type of rewriting is highly similar to Feder's plex grammars [Fede71]. If all hyperedges have exactly one incoming tentacle and exactly one outgoing tentacle, then we have an edge replacement system, as above.

The significance of hyperedge systems is that they deal with a sophisticated primitive, and yet are rich in results corresponding to the properties of context-free string languages. Additionally, they are rich in decidability results concerning graph-theoretic properties of the members of the generated languages [DrKr91]. Hyperedge systems have furthered graph-language theory. We are not aware of their use in applications.

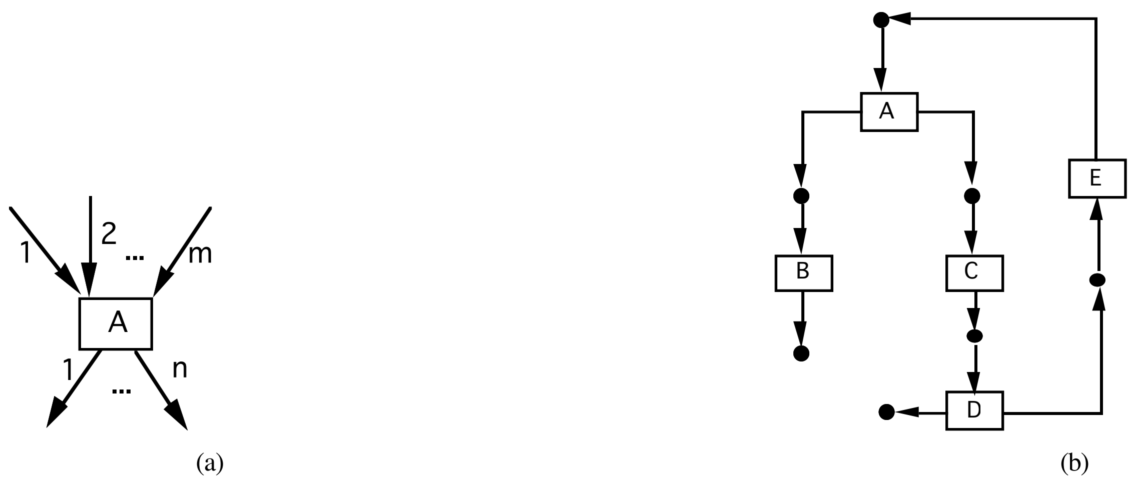


Figure A.9 (a) A hyperedge with label A. (b) A hypergraph; the solid nodes are sockets joining hyperedges.

A.6 Structured Graph Rewriting: Embedding combined with Gluing

The structured graph rewriting model of Kreowski and Rozenberg unites a wide variety of graph-rewriting approaches under one general framework [KrRo90a] [KrRo90b]. A complex model results from this effort to accommodate both the embedding and the gluing approaches to graph rewriting. While this model may be too complex for practical use, it provides an excellent overview of graph-rewriting mechanisms. Existing approaches to graph rewriting are modeled using selected subsets of the features present in the structured graph rewriting model. This clarifies the similarities and differences among various approaches to graph rewriting.

Structured graph rewriting is applied to directed graphs with labeled nodes and edges. A structured rewrite rule has the following components.

- The graph g_l. This is a structured graph, meaning that it contains three designated subgraphs.
  A subset of the nodes of g_l is designated as the <u>contact part of g_l</u>.
  A subgraph of g_l is designated as the <u>protected part of g_l.</u>
  A subgraph of the protected part is designated as the <u>gluing part of g_l</u>.
- The graph g_r. This is a weakly structured graph, meaning that it contains one designated subgraph.
  A subgraph of g_r is designated as the <u>gluing part of g_r</u>.
- The gluing isomorphism, which maps the gluing part of g_l to the gluing part of g_r.
- An NCE-like embedding specification (Section A.3): the relations C_{in} and C_{out} embed incoming and outgoing edges according to five-tuples that specify the g_l node, the g_r node, the pre-embedding edge label, the post-embedding edge label, and the node-label of the affected RestGraph node.

Rule application involves five steps.

(1) <u>Choose</u> an occurrence of g_l in g. The result is g_l^{host}.

(2) <u>Check</u> the application condition. This includes the contact condition, a generalization of the gluing condition of Section A.5: any node in g_l^{host} which is adjacent to nodes in RestGraph must be in the contact part of g_l^{host}. The

application condition may also include conditions on attribute values (Section□2). Rule application proceeds only if all conditions are met.

(3) Remove the unprotected parts of g_l^{host}. Many graph rewriting approaches require the complete removal of g_l^{host}. Exceptions are the diagrammatic X and Δ notations (Section⊠.2), and the gluing models of Section⊠.5, which require removal of only part of g_l^{host}. The remaining graph is called *ContextGraph.* (If g_l^{host} is completely removed, then ContextGraph = RestGraph.)

(4) Add g_r^{host} to ContextGraph, by gluing according to the gluing isomorphism. For pure embedding approaches, the Add is a disjoint union of ContextGraph and g_r^{host}; the embedding edges are formed next, in the Connect step.

(5) Connect g_r^{host} to ContextGraph. Post-embedding edges are formed according to the embedding specification.

Various graph-rewriting models fit into this framework. The gluing models of Section⊠.5 rely completely on gluing (the Add step) to embed g_r^{host} in the host graph. Here the gluing and protected parts of g_l are identical, and contain the contact part of g_l; the gluing isomorphism is used; and the embedding specification is null. Embedding approaches to graph rewriting are modeled using null gluing parts and null protected parts; the embedding specification can express embeddings that are orientation-preserving and depth1. Existing graph-rewriting mechanisms that are modeled in [KrRo90a] use either the Add or Connect steps, but not both. An example requiring both Add and Connect is provided by the diagrammatic X and Δ-notations of Section⊠.2. Recall that these models have a required context: a subgraph common to g_l and g_r, which is unaltered by the graph rewrite, but must be present for the rewrite to occur. To model this with structured graph rewriting, the required context becomes the gluing (and protected) part of g_l; the Add step models this gluing. The Connect step models a (depth1, orientation-preserving) subset of the embedding specification, which is represented via the "optional context" of X and Δ notations.

References

[AiNa91] K. Aizawa, A. Nakamura, "Graph Grammars with Path-Controlled Embedding," *Theoretical Computer Science*, Vol. 88, No. 1, Sept. 1991, pp. 151-170.

[Ande77] R. Anderson, "Two Dimensional Mathematical Notation," in *Syntactic Pattern Recognition, Applications*, K. S. Fu editor, Springer 1977, pp. 147-177.

[Bunk82a] H. Bunke, "Attributed Programmed Graph Grammars and Their Application to Schematic Diagram Interpretation," *IEEE Pattern Analysis and Machine Intelligence*, Vol. 4, No. 6, Nov. 1982, pp. 574-582.

[Bunk82b] H. Bunke, "On the Generative Power of Sequential and Parallel Programmed Graph Grammars," *Computing*, Vol. 29, 1982, pp. 89-112.

[BuHa89] H. Bunke and B. Haller, "A parser for context free plex grammars," *Proc. 15th Intl. Workshop on Graph-Theoretic Concepts in Computer Science*, June 1989, (in LNCS 411, Springer), June 1989.

[BuHa92] H. Bunke and B. Haller, "Syntactic Analysis of Context-Free Plex Languages for Pattern Recognition," in *Structured Document Image Analysis*, Eds. Baird, Bunke, Yamamoto, Springer 1992, pp. 500-519.

[CaRM91] S. Card, G. Robertson, and J. Mackinlay, "The Information Visualizer, An Information Workspace," *Proc. ACM SIGCHI 1991 Conference on Human Factors in Computing Systems*, pp. 181-188, New Orleans, Louisiana, April 1991.

[CoTV93] S. Collin, K. Tombre, and P. Vaxiviere, "Don't Tell Mom I'm Doing Document Analysis; She Believes I'm in the Computer Vision Field," *Proc. Second International Conference on Document Analysis and Recognition*, Tsukuba, Japan, Oct. 1993, pp. 619-622.

[Comp92] *COMPUGRAPH Computing by Graph Transformation: Final Report*, ESPRIT Basic Research Working Group No. 3299, edited by H. Ehrig and M. Löwe, March 1992.

[DoTo88] J. Dolado and F. Torrealdea, "Formal Manipulation of Forrester Diagrams by Graph Grammars," *IEEE Trans. Systems, Man and Cybernetics* 18(6), pp. 981-996, Nov. 1988.

[DrKr91] F. Drewes, H.-J. Kreowski, "A Note on Hyperedge Replacement," in [IWGG91], pp. 1-11.

[EgPM92] J. Egar, A. Puerta, M. Musen, "Automated Interpretation of Diagrams for Specification of Medical Protocols," *AAAI Symposium: Reasoning with Diagrammatic Representations,* Stanford University, March 1992, p 189-192.

[Ehri87] H. Ehrig, "Tutorial Introduction to the Algebraic Approach of Graph Grammars," in [IWGG87], pp. 3-14.

[EhHa86] H. Ehrig, A. Habel, "Graph Grammars with Application Conditions," in [RoSa86], pp. 87-100.

[EhHK92] H. Ehrig, A. Habel, H. Kreowski, “Introduction to Graph Grammars with Applications to Semantic Networks,” *International Journal of Computers and Mathematical Applications*, Vol. 23, No 6-9, pp. 557-572, 1992.

[EhKL91] H. Ehrig, M. Korff, M. Löwe, “Tutorial Introduction to the Algebraic Approach of Graph Grammars Based on Double and Single Pushouts,” in [IWGG91], pp. 24-37.

[EhKr80] H. Ehrig and H. Kreowski, “Applications of Graph Grammar Theory to Consistency, Synchronization, and Scheduling in Data Base Systems,” *Information Systems*, Vol 5, pp. 225-238, 1980.

[EKMRW81] H. Ehrig, H.-J. Kreowski, A. Maggiolo-Schettini, B. Rosen, J. Winkowski, “Transformation of Structures: An Algebraic Approach,” *Math. Syst. Theory*, Vol. 14, 1981, pp. 305-334.

[EhPS73] H. Ehrig, M. Pfender, H. J. Schneider, “Graph Grammars: An Algebraic Approach,” *Proc. IEEE Conf. SWAT ‘73*, Iowa City, 1973, pp. 167-180.

[EnLR87] J. Engelfriet, G. Leih, G. Rozenberg, “Apex Graph-Grammars,” in [IWGG87], pp. 167-185.

[EnLW90] J. Engelfriet, G. Leih, E. Welzl, “Boundary Graph Grammars with Dynamic Edge Relabeling,” *Journal of Computer and System Sciences*, Vol. 40, 1990, pp. 307-345.

[EnRo91] J. Engelfriet, G. Rozenberg, “Graph Grammars Based on Node Rewriting: An Introduction to NLC Graph Grammars,” in [IWGG91], pp. 12-23.

[ELNSS92] G. Engels, C. Lewerentz, M. Nagl, W. Schafer, A. Schürr, “Building Integrated Software Development Environments Part 1: Tool Specification,” *ACM Trans. Software Engineering and Methodology*, Vol. 1, No. 2, Apr. 1992, pp. 135-167.

[EnLS87] G. Engels, C. Lewerentz, W. Schafer, “Graph Grammar Engineering: A Software Specification Method,” in [IWGG87], pp. 186-201.

[EnSc85] G. Engels, W. Schafer, “Graph Grammar Engineering: A Method used for the Development of an Integrated Programming Support Environment,” *Proc. International Joint Conference on Theory and Practice of Software Development*, (in *Lecture Notes in Computer Science*, Springer Verlag, Vol. 186, 1985), pp. 179-193.

[FaBl92] H. Fahmy and D. Blostein, “A Survey of Graph Grammars: Theory and Applications,” *11th International Conference on Pattern Recognition*, Delft, Netherlands, September 1992, Vol 3, pp. 294-298.

[FaBl93] H. Fahmy and D. Blostein, “A Graph Grammar Programming Style for Recognition of Music Notation,” *Machine Vision and Applications*, Vol 6, No 2, pp. 83-99, 1993.

[FaBl94] H. Fahmy and D. Blostein, “Reasoning in the Presence of Uncertainty via Graph Rewriting,” pre-proceedings of *Fifth Intl. Workshop on Graph Grammars and Their Application to Computer Science*, Nov 1994, Williamsburg, Virginia, pp 95-100.

[Fede71] J. Feder, “Plex Languages,” *Information Sciences*, Vol. 3, 1971, pp. 225-241.

[Fu82] K. S. Fu, *Syntactic Pattern Recognition and Applications*, Prentice Hall 1982.

[Gold92] L. Goldfarb, “Transformation Systems are More Economical and Informative Class Descriptions than Formal Grammars,” *11th International Conference on Pattern Recognition*, Delft, Netherlands, September 1992, Vol 2 pp. 660-664.

[Gött79] H. Göttler, “Semantical Description by Two-Level Graph-Grammars for Quasihierarchical Graphs,” in *Workshop WG 78 on Graphtheoretical Concepts in Computer Science, Applied Computer Science*, Vol. 13, Hanser-Verlag, Munich, 1979.

[Gött83] H. Göttler, “Attribute Graph Grammars for Graphics,” in [IWGG83], pp. 130-142.

[Gött87] H. Göttler, “Graph Grammars and Diagram Editing,” in [IWGG87], pp. 216-231.

[GöGN91] H. Göttler, J. Gunther, G. Nieskens, “Use Graph Grammars to Design CAD-Systems!” in [IWGG91], pp. 396-410.

[Gött92] H. Göttler, “Diagram Editors = Graphs + Attributes + Graph Grammars,” *International Journal of Man-Machine Studies*, Vol 37, No 4, Oct. 1992, pp. 481-502.

[Grba94] A. Grbavec, “Recognition of Mathematical Notation using Ordered Graph Rewriting,” MSc Thesis, Department of Computing and Information Science, Queen’s University, February 1995.

[HaKr83] A. Habel, J.-J. Kreowski, “On Context-free Languages Generated by Edge Replacement,” in [IWGG83], pp. 143-158.

[HaKr87a] A. Habel, J.-J. Kreowski, “Characteristics of Graph Languages Generated by Edge Replacement,” *Theoretical Computer Science,* Vol. 51, 1987, pp. 81-115.

[HaKr87b] A. Habel, J.-J. Kreowski, “May We Introduce to You: Hyperedge Replacement,” in [IWGG87], pp. 15-26.

[HaKV89] A. Habel, J.-J. Kreowski, W. Vogler, “Decidable Boundedness Problems for Hyperedge-Replacement Graph Grammars,” in *Lecture Notes in Computer Science*, Vol. 351, 1989, pp. 275-289.

[HaBl94] L. Haken and D. Blostein, “Shared Processing in Automatic Diagram Recognition and Generation: A Case Study of Music Notation,” in preparation.

[Hare88] D. Harel, “On Visual Formalisms,” *Communications of the ACM*, Vol 31, No 5, pp. 514-530, May 1988.

[HeSa86] T. Henderson and A. Samal, “Shape Grammar Compilers,” *Pattern Recognition*, Vol 19, No 4, pp 279-288, 1986.

[Hims93] M. Himsolt, “Konzeption und Implementierung von Grapheneditoren,” Dissertation, Universität Passua, 1993. The GraphEd system is freely available for non-commerical use (himsolt@fmi.uni-passau.de).

[ICDAR93] *Proc. Second International Conference on Document Analysis and Recognition*, Tsukuba, Japan, Oct. 1993.

[IWGG79] *International Workshop on Graph Grammars and Their Application to Computer Science and Biology*. (*Lecture Notes in Computer Science,* Vol. 73, V. Claus, H. Ehrig, G. Rozenberg Eds, Springer Verlag, 1979)

[IWGG83] *Second International Workshop on Graph Grammars and Their Application to Computer Science*. (*Lecture Notes in Computer Science,* Vol. 153, H. Ehrig, M. Nagl, G. Rozenberg Eds, Springer Verlag, 1983)

[IWGG87] *Third International Workshop on Graph Grammars and Their Application to Computer Science*. (*Lecture Notes in Computer Science,* Vol. 291, H. Ehrig, M. Nagl, G. Rozenberg, A. Rosenfeld Eds, Springer Verlag, 1987)

[IWGG91] *Fourth International Workshop on Graph Grammars and Their Application to Computer Science*. (*Lecture Notes in Computer Science,* Vol. 532, H. Ehrig, H. Kreowski, G. Rozenberg Eds, Springer Verlag, 1991)

[JaRo80a] D. Janssens, G. Rozenberg, “On the Structure of Node-Label Controlled Graph Languages,” *Information Sciences,* Vol. 20, 1980, pp. 191-216.

[JaRo80b] D. Janssens, G. Rozenberg, “Restrictions, Extensions and Variations of NLC Grammars,” *Information Sciences,* Vol. 20, 1980, pp. 217-244.

[JaRo82] D. Janssens, G, Rozenberg, “Graph Grammars with Neighbourhood-Controlled Embedding,” *Theoretical Computer Science*, Vol. 21, 1982, pp. 55-74.

[KaLG91] S. Kaplan, J. Loyall, S. Goering, “Specifying Concurrent Languages and Systems with Δ-grammars,” in [IWGG91], pp. 475-489.

[Kreo79] H.-J. Kreowski, “A Pumping Lemma for Context-free Graph Languages,” in [IWGG79], pp. 270-283.

[KrRo90a] H.-J. Kreowski, G. Rozenberg, “On Structured Graph Grammars, I,” *Information Sciences*, Vol. 52, 1990, pp. 185-210.

[KrRo90b] H.-J. Kreowski, G. Rozenberg, “On Structured Graph Grammars, II,” *Information Sciences*, Vol. 52, 1990, pp. 221-246.

[LiFu89] W. Lin and K.S. Fu, “A Syntactic Approach to Three-Dimensional Object Recognition,” *IEEE Trans. Systems Man and Cybernetics*, Vol 16, No 3, May 1986, pp 405-422.

[LöBe93] M. Löwe, M. Beyer, “AGG -- An Implementation of Algebraic Graph Rewriting,” *Proci Fifth Intl .Conf. on Rewriting Techniques and Applications*, Montreal, Canada, June 1993, in LNCS 690, Springer Verlag, pp. 451-456.

[LoKa92] J. Loyall and S. Kaplan, “Visual Concurrent Programming with Delta-Grammars,” *Journal of Visual Languages and Computing*, Vol 3, 1992, pp. 107-133.

[MaRo87a] M. Main, G. Rozenberg, “Handle NLC Grammars and R.E. Languages,” *Journal of Computer and System Sciences*, Vol. 35, No. 2, Oct. 1987, pp. 192-205.

[MaRo87b] M. Main, G. Rozenberg, “Fundamentals of Edge-Label Controlled Graph Grammars,” in [IWGG87], pp. 411-426.

[MaRo90] M. Main, G. Rozenberg, “Edge-Label controlled Graph Grammars,” *Journal of Computer and System Sciences*, Vol. 40, 1990, pp. 188-228.

[MaKl92] J. Mauss and C. Klauck, “A Heuristic Driven Parser Based on Graph Grammars for Feature Recognition in CIM,” *Proc. International Workshop on Structural and Syntactic Pattern Recogntion*, Bern Switzerland, August 1992 (in *Advances in Sructural and Syntactic Pattern Recognition*, Ed. H. Bunke, World Scientific, 1992), pp. 611-620.

[Mont70] U. Montanari, “Separable Graphs, Planar Graphs and Web Grammars,” *Information and Control*, Vol. 16, pp. 243-267.

[MuMH88] I. Mulder, A. Mackworth, W. Havens, “Knowledge Structuring and Constraint Satisfaction: The Mapsee Approach,” *IEEE Pattern Analysis and Machine Intelligence*, Vol 10, No 6, November 1988, pp. 866-879.

[Nagl79] M. Nagl, “A Tutorial and Bibliographical Survey on Graph Grammars,” in [IWGG79], pp. 70-126.

[Nagl86] M. Nagl, “Graph Technology Applied to a Software Project,” in [RoSa86], pp. 303-321.

[Nagl87] M. Nagl, “Set Theoretic Approaches to Graph Grammars,” in [IWGG87], pp. 41-54.

[NaSc91] M. Nagl, A. Schürr, “A Specification Environment for Graph Grammars,” in [IWGG91], pp. 599-609.

[Panel91] "Panel Discussion: The Use of Graph Grammars in Applications," in [IWGG91], pp. 41-60.

[PaTi90] F. Paulisch and W. Tichy, "EDGE: An Extendible Graph Editor," *Software Practice and Experience*, Vol.20, pp. 63-88, 1990.

[PeJo87] S. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice Hall, 1987.

[PfRo69] J. Pfaltz, A. Rosenfeld, "Web Grammars," *Proc. 1st Int. Joint Conf. on Artificial Intelligence*, Washington, 1969, pp. 609-619.

[Pfal72] J. Pfaltz, "Web Grammars and Picture Description," *Computer Graphics and Image Processing*, Vol. 1, 1972, pp. 193-220.

[Pfei90] J. Pfeiffer, "Using Graph Grammars for Data Structure Manipulation," in *Proceedings of the 1990 Workshop on Visual Languages*, 1990, pp. 42-47.

[Reke94] J. Rekers, "Graphical definition of graphical syntax," in preparation.

[Roze87] G. Rozenberg, "An Introduction to the NLC Way of Rewriting Graphs," in [IWGG87], pp. 55-70.

[RoSa86] G. Rozenberg, A. Salomaa (Eds.), *The Book of L*, Springer-Verlag, Berlin, 1986.

[RoWe86] G. Rozenberg, E. Welzl, "Boundary NLC Graph Grammars - Basic Definitions, Normal Forms, and Complexity," *Information and Control,* Vol. 69, 1986, pp. 136-167.

[Schü91] A. Schürr, "PROGRESS: A VHL-Language Based on Graph Grammars," in [IWGG91], pp. 641-659.

[Schn93] H. Schneider, "On categorical graph grammars integrating structural transformations and operations on labels," in [TCS93], pp. 257-275.

[SiGJ93] G. Sindre, B. Gulla, H. Jokstad, "Onion Graphs: Aesthetics and Layout," *Proc. 1993 IEEE Symposium on Visual Languages*, Bergen, Norway, August, 1993, pp. 287-291.

[Strz90] T. Strzalkowski, "Reversible Logic Grammars for Natural Language Parsing and Generation," *Canadian Computational Intelligence Journal*, Vol 6, No 3, pp. 145-171, 1990.

[TCS93] *Theoretical Computer Science*, Vol 109, 1993, special issue on computing by graph transformation.

[Uesu78] T. Uesu, "A System of Graph Grammars which Generates all Recursively Enumerable Sets of Labelled Graphs," *Tsukuba Journal of Mathematics*, Vol. 2, 1978, pp. 11-26.

[Welz86] E. Welzl, "On the Set of All Subgraphs of the Graphs in a Boundary NLC Graph Language," in [RoSa86], pp. 445-459.

[Welz87] E. Welzl, "Boundary NLC and Partition Controlled Graph Grammars," in [IWGG87], pp. 593-609.

[ZüSc92] A. Zündorf and A. Schürr, "Nondeterministic Control Structures for Graph Rewriting Systems," *Proc 17th Intl. Workshop on Graph-Theoretic Concepts in Computer Science WG91*, (in *Lecture Notes in Computer Science,* Vol 570, Springer Verlag, 1992).