Issues in the 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

Abstract. Graphs are a popular data structure, and graph-manipulation programs are common. Graph manipulations can be cleanly, compactly, and explicitly described using graph-rewriting notation. However, when a software developer is persuaded to try graph rewriting, several problems commonly arise. Primarily, it is difficult for a newcomer to develop a feel for how computations are expressed via graph rewriting. Also, graph-rewriting is not convenient for solving all aspects of a problem: better mechanisms are needed for interfacing graph rewriting with other styles of computation. Efficiency considerations and the limited availability of development tools further limit practical use of graph rewriting. The inaccessible appearance of the graph-rewriting literature is an additional hindrance. These problems can be addressed through a combination of "public relations" work, and further research and development, thereby promoting the widespread use of graph rewriting.

1. Introduction

Graph rewriting has the potential to be useful in a large variety of applications. Graphs provide an expressive and versatile data representation. Typically, nodes represent objects or concepts, and edges represent relationships among them. In addition, 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, high-level representation of a solution to a computational problem. Also, the theoretical foundations of graph rewriting assist in proving correctness and convergence properties.

Despite this potential, graph rewriting has not attained widespread practical use. To discover the reasons for this, it is helpful to consider an outside viewpoint:

> Mr. and Mrs. Maggraphen manage a small software house in Bavaria. Most of their important data structures are graphs. Currently, all of their programs are written in C, with much of the code devoted to graph manipulations.
>
> The Maggraphens are planning for the future, and want to switch from C to a graph-rewriting language.

The Maggraphens are enthusiastic about graph rewriting, but have many questions. To begin with, important practical considerations arise. Will the graph-rewriting language be fast enough? Are there tools for developing, displaying, and debugging graph-rewrite rules? Suppose, optimistically, that the answer to both questions is "Yes".

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

Even so, there is another major hurdle: the Maggraphens can't imagine how to recast their C programs in terms of graph rewriting. They desperately need small-scale advice (how to formulate individual rewrite rules) and large-scale advice (how to organize a collection of rules). Let us consider a sampling of their questions. (Figure 1 shows our terminology.)

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:

- **$g_l \rightarrow g_r$** g_l and g_r are unattributed graphs. During rule application, an attributed subgraph g_l^{host} (isomorphic to g_l) is replaced by g_r^{host} (a subgraph created to be isomorphic to g_r).
- **Embedding Information**
  Calculates post-embedding edges from pre-embedding edges (defined below). Embedding information can be textual or graphical. Gluing models specify embedding with a gluing isomorphism.
- **Application Condition** (Optional)
  Defines conditions on attribute values or host-graph structure. These conditions must hold for rule application to proceed.
- **Attribute Transfer Function** (Optional)
  Assigns attribute values to g_r^{host}, using attribute values in g_l^{host}.

Host Graph g — The graph to which a rule is being applied.

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}. (The "-" operator denotes removal of all nodes and edges of g_l^{host} and all edges with one or both endpoints in g_l^{host}.)

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

Post-embedding Edges — the set of edges joining g_r^{host} to RestGraph

Figure 1. Our terminology for graph rewriting. These definitions assume the use of subgraph isomorphism, where some models actually allow for a general graph morphism.

---

2. Mrs. Maggraphen: We are new to graph rewriting. Where do we start?

The Maggraphens are looking to us, the graph-rewriting community, as a source of information about how to express computations in graph rewriting. Consider an analogous 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 should make an effort to promote such a culture, to allow newcomers to quickly develop a proper mindset for performing practical, effective computations using graph rewriting. Relevant materials include the following:

- Accessible written expositions about the practical use of graph rewriting: systems organizations, styles of computation, etc.
- Easily-available tools for creating, editing, executing, debugging graph rewriting systems (Section 4).
- 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. (Relevant references, discussed in [BlFG95], include: software engineering [EnLS87] [ELNSS92] [LoKa92] [Pfei90], syntactic pattern recognition [Fu82], document image analysis [Bunk82a] [FaBl93] [GrBl95] [CoTV93], 3D object recognition [LiFu86], visual programming environments [EgPM92], diagram editors [Gött92] [DoTo88], databases [EhKr80], and semantic networks [EhHK92]. Further discussion is given by [Panel91].)

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

3. Mr. Maggraphen: In C, we use standard algorithms (searching, sorting, hashing) and algorithm-design methods (divide-and-conquer, dynamic programming, greedy algorithms). What is the equivalent to this in graph rewriting?

Currently, we have little to offer the Maggraphens, in terms of graph-rewrite-oriented techniques for algorithm design or analysis. We have few libraries of standard graph-rewriting code. (An inspiring example is given by the parameterized graph-rewrite rules for abstract-syntax-tree manipulation reported in [ELNSS92]).

We need to develop specialized algorithm design techniques, geared toward graph rewriting as the primitive operation. Precedents for such specialized algorithm design techniques include VLSI design (with area*time used as a cost function) and optical computing (where primitive operations include Fourier transform, convolution, union and intersection of figures, coordinate transforms).

4. Mrs. Maggraphen: What development tools are available?

As everyone is well aware, practical use of graph rewriting depends heavily on the availability of development and debugging tools. Unfortunately, construction of these tools is a time-consuming, complex task, due to the need to combine textual and diagrammatic elements, the need to provide readable displays of large graphs, and the need to visualize the interactions among graph rewriting rules. Development of graph-rewrite debugging techniques is an interesting and challenging research topic. Currently it is difficult even to define what kind of tools are needed to support widespread practical use of graph rewriting. This will become clearer over time, as the improving set of available tools allow us to gather more extensive experience with executable graph-rewriting systems.

For the reader interested in experimenting with graph rewriting, here is a brief list of graph-rewriting environments. The first two environments are mature enough to be in widespread use, and are under active further development. The remaining environments may become available for general use. Our apologies if this list is incomplete.

- PROGRES provides extensive facilities for ordered graph rewriting [NaSc91] [ELNSS92]. Contact andy@i3.informatik.rwth-aachen.de to obtain this software.

- GraphEd [Hims91] provides extensive graph-display capabilities, and supports a limited form of graph-rewriting (direct-derivation steps of context-free rewrite rules). Contact himsolt@fmi.uni-passau.de to obtain this software.
- 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.
- A prototype implementation of algebraic graph transformation is described in [LöBe93]. At that time, the tool performed direct derivation steps in the single-pushout approach.
- Göttler [Gött92] mentions a succession of implementations for executing ordered graph rewriting (Y and X notation); a new C implementation is under development, including a graphical editor for X notation rules.

5. Mr. Maggraphen: Can graph rewriting be efficient? Isn't subgraph-isomorphism testing intractable?

This question readily comes to mind, but we can give some reassurance. It is true that subgraph-isomorphism testing is an NP-complete problem in general, but various factors make it tractable in a graph-rewriting system. Firstly, it is often possible to express a computation using small subgraphs on the left-hand-side of rewrite rules. Secondly, node labels, edge labels, and directed edges drastically reduce the search space for isomorphic subgraphs. Finally, some graph-rewriting systems have certain phrases that frequently appear in application conditions; these can be exploited to greatly reduce the search space for isomorphic subgraphs that meet the application condition. The optimization of subgraph-isomorphism testing is discussed in [BuGT91] [Zünd94].

Of course, graph rewriting should not be marketed as a fast style of computation: 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 could motivate the development of a new computer architecture with graph-rewriting as a fundamental operation. First we would need to develop suitable graph-rewriting architectures in software, and thus popularize graph rewriting as a style of computation. Special-purpose graph-rewriting hardware may sound far-fetched, but consider neural-network computations as an analogy: years of research with software-implemented neural-net architectures have now resulted in commercially-available neural-net architectures implemented as VLSI circuits.

6. Mrs. Maggraphen: How can we organize rewrite rules?

The graph-rewriting literature reports on various methods of organizing a collection of graph-rewrite rules: unordered, ordered and event-driven graph-rewriting systems, as well as graph grammars (Table 1). This taxonomy arose from our efforts to organize our reading of the graph rewriting literature. (This literature is confusing because many systems are called "grammars", whether they define a graph-language or not.) An understanding of these systems-organizations provide a helpful starting point in the process of deciding how a computation could be expressed as graph rewrite rules.

The choice of system organization greatly affects the number of rewrite-rule applications that must be tried during execution. Parsing with a grammar normally requires backtracking, and frequent testing of inapplicable rules. In contrast, an ordered graph rewriting system can directly transform an input graph into an output graph, with a limited number of production rules under consideration at any given time [Bunk82a]. Event-driven graph-rewriting systems can be highly time-efficient, applying rules only in direct response to external actions. Thus, if an application is such that it can be

System Components	System Execution
Unordered Graph-rewriting System	
A set of graph-rewrite rules.	Rewrite the given host graph (choosing nondeterministically among applicable rules) until no further rules apply.
Graph Grammar	
A set of graph-rewrite rules (*productions*). A start graph. A designation of labels as terminal or nonterminal.	In *generative* use, rewrite the start graph to obtain a terminal graph (no non-terminal labels.) The set of generatable terminal graphs is the *language* of the grammar. For *recognition*, parse the given graph: find a sequence of rewrite-rules that derive the given graph from the start graph.
Ordered Graph-rewriting System	
A set of graph-rewrite rules. A control specification (provides complete or partial ordering of rule-application).	Rewrite the given host graph (choosing nondeterministically among applicable rules consistent with the control specification) until a final state in the control specification is reached.
Event-driven Graph-rewriting System	
A set of graph-rewrite rules. An externally-arising sequence of events.	Rewrite the given initial host graph: rewrite rules are executed in response to events.

Table 1. Four organizations for graph-rewriting systems.

implemented using event-driven graph-rewriting, then likely it can run with acceptable time-efficiency. If the application calls for ordered (or partially ordered) graph rewriting without backtracking, then it may well 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. In any case, graph rewriting can be useful even if it does not provide an acceptably efficient implementation: a practical software development cycle can include the use of graph rewriting to form an executable specification (e.g. [ZüSc92]).

We now briefly review the practical use of these four system organizations.

Unordered graph rewriting

An excellent example of unordered graph rewriting is provided by Δ-rewriting [KaLG91] [LoKa92]. 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-rewrite rules, either infinitely (as in the dining philosophers example of [LoKa92, p. 112]), or with termination (as in the quicksort example). The *platform* concept used to modularize Δ-

rewriting is discussed in Section 8. Unfortunately, no Δ-rewriting environment is available; current experience is limited to paper-based descriptions of Δ-rewriting systems.

Graph grammars

In a pure graph grammar, productions can be listed in any order, but 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" 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.

In addition to order-dependence, there is the issue of reversibility. Can a given grammar be used both for recognition and generation? While a pure grammar is reversible, in practice non-reversible constructs like application conditions and attribute computations are common. Reversibility is desired in various domains, but difficult to achieve. For example, there is on-going research into reversible string-grammars for natural language processing [Strz90]. On a related note, a graph grammar with non-reversible rules is limited to either bottom-up or top-down parsers.

Practical use of graph grammars is seriously hampered by the high complexity of parsing. Sub-exponential parsers have been developed for certain restricted classes of graph grammars. A selection of parsing references are as follows. Kaul presents a linear-time precedence parser for a special class of context free graph-grammars [Kaul83]. Bunke and Haller describe an extension of Early's parser for context-free plex languages [BuHa92]; this parser permits left-recursion and is capable of recognizing partial structures. Recently, a parsing algorithm applicable to context-sensitive graph grammars has been developed [ReSc94]. Egar et al. use a graph-grammar parser 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 [LiFu86]. 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]. More recently, Klauck reports on a heuristically-driven chart parser and it's application to CAD/CAM [Klau94]. On a related note, Henderson and Samal discuss efficient parsing of stratified shape grammars, building on the table-driven methods used for LR(k) string grammars [HeSa86]; these techniques might be relevant to graph-grammar parsing.

Ordered graph rewriting

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. Similarly, a recognition approach for music notation [FaBl93] uses ordered recognition stages, each of which consists of three ordered phases (*Build* creates edges, *Weed* removes inconsistent edges, and *Incorporate* prunes the graph

while adding semantic information to attributes). Graph applications in software engineering have made extensive use of ordered graph rewriting (e.g. [ELNSS92]).

Various forms of ordered graph rewriting are possible, depending on the use of non-determinism and backtracking:

- A completely deterministic system results from pairing a deterministic control specification with the use of cursor-nodes (also called demon nodes) to indicate the desired host graph location for rule application. Determinism is desirable in editing applications, where end-users expect a deterministic response to an editing command (e.g. [Gött92]).
- Partially ordered rewrite systems, without backtracking, have been used for software engineering (e.g. [ELNSS92]) and diagram recognition (circuit-diagrams [Bunk82a], music-notation [FaBl93] [Fahm95], math-notation [GrBl95]). In the diagram recognition work, the control specification orders the phases that make up the recognition process; rules within a phase are unordered or partially ordered, and all non-deterministic alternatives lead to a desired result.
- Partially ordered rewrite systems, with backtracking, can be expressed in the PROGRES language [ZüSc92]. The PROGRES interpreter automatically backtracks in the search for a successful path through the control specification: alternate matches for g_l^{host}, and alternate control paths, are tried as needed. This allows straightforward coding of classical AI search problems as a partially-ordered collection of rewrite rules.

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 (*control diagrams*) are used by [Bunk82a], with extensions by [DoTo88] [FaBl93] and others. For example, a *block condition* allows the control diagram to test attribute values of any nodes involved in the most recent production [DoTo88]. To permit more flexible control constructs, the control specification can take a textual form, similar to an imperative programming language. For example, PROGRES provides deterministic and non-deterministic versions of And, Or, Loop [ZüSc92][ELNSS92], in addition to encapsulation tools such as transactions and subdiagrams.

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. The authors mention an anticipated need for control structures within a single transaction.

Ordered graph rewriting can be used to regulate event-driven graph rewriting. In the Forrester-diagram editor of [DoTo88], the control specification defines which editing events are legal at any given point. Events not foreseen by the control specification are disallowed, resulting in an error message to the user. A similar structure is used by the diagram editors described in [Gött92].

7. Mr. Maggraphen: How do we choose a graph-rewriting mechanism?

A large variety of graph-rewriting mechanisms have been investigated. No one rewriting mechanism is universally suitable. Practical choice of a rewriting mechanism depends on the application, on the availability of tools, and on personal taste. Relevant factors include the power of the embedding, formal properties of rewrite rules, readability and intellectual manageability, and efficiency of rule application.

Power of the Embedding

Complex embedding mechanisms permit significant graph inspection and graph manipulation during the embedding step. Conversely, highly-restricted embedding mechanisms, such as the invariant embedding of the gluing models, are inconvenient for expressing certain common graph operations such as node deletion (Figure 2).

The choice of an embedding mechanism involves a tradeoff between using fewer, but complex, rewrite rules versus using a larger number of simpler rules. Up to now, we have few practical examples of graph-rewriting systems that make heavy use of complex embeddings. It appears that many software designers find it is easier or more natural to express a computation using more rules of a restricted embedding type.

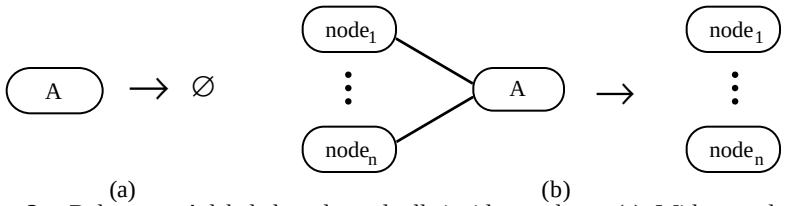


Figure 2 Delete an A-labeled node and all incident edges. (a) With an elementary embedding mechanism. (b) With a gluing model. The invariant embedding necessitates that g_l be expanded to include all edges incident on the A-labeled node. A set of rewrite rules is used to enumerate each possible configuration of incident edges. (The "..." notation, denoting variable repetition of nodes and edges, is adapted from [EhHK92]. Similarly, Δ–notation uses *-groups, which denote zero or more occurrences of starred graph elements, to implement node-deletion [KaLG91, p. 478]. A Δ-rule that deletes a node is syntactic shorthand for an infinite collection of Δ-rules that meet the gluing condition.)

Formal Properties of Rewrite Rules

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 simplifies construction of proofs about the integrity of a database system, as illustrated by the library-transaction system of [EhKr80].

Using rewrite rules with formally-characterized properties, graph rewriting can provide a formal definition of graph classes; examples include the class of well-formed Forrester diagrams [DoTo88] and the class of well-formed semantic networks [EhHK92].

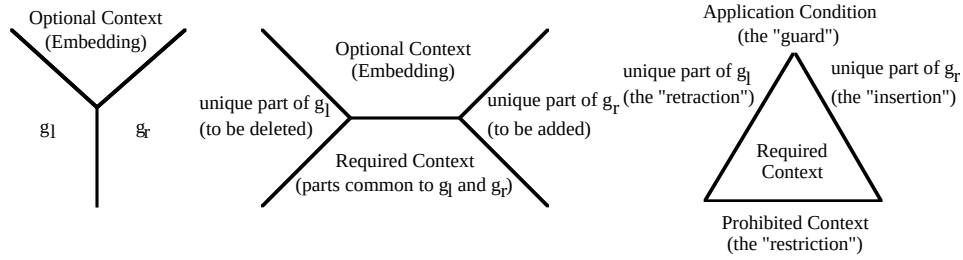
Readability and Intellectual Manageability

Readability of rewrite rules affects intellectual manageability, system development time, ease of maintenance, and ease of debugging. It can be particularly difficult to present complex embeddings in a readable way. Since textual embedding specifications can be difficult to read, various diagrammatic notations have been proposed (Figure 3). Visual presentation can be simplified by avoiding the duplication of graph-parts

common to g_l and g_r (Figure 4). In our opinion, these diagrammatic depictions are advantageous for embeddings of intermediate complexity:

- Elementary embeddings can be specified textually, and are easily perceived from visually-corresponding nodes in g_l and g_r (Figure 5). Similarly, gluing isomorphisms are effectively conveyed by the visual correspondence of g_l and g_r nodes, as in [EhHK92].
- Embeddings that are more complex than the elementary type (e.g., they involve testing of node-labels in RestGraph, or following of edges in RestGraph) 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 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 [ELNSS92].

Some applications require complex embeddings, others don't. In our experience, 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 3 Three diagrammatic notations for graph-rewrite rules. In Y and X notations, the embedding is shown as optional context: these diagrammatic depictions of embedding are used *if* they match in the host graph. The required context must match in order for the rewrite rule to be applied. In Δ notation, the center of the Δ is used both for required and optional context, with a * placed next to the optional parts. (Elements of a * group may occur zero, one or more times.) The prohibited context depicts host-graph structure that must not be present; restrictions on labels and attributes are expressed textually in the guard.

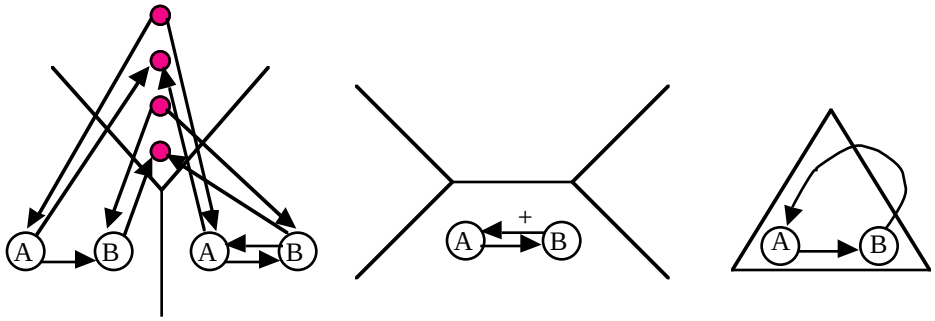


Figure 4 Graph-rewrite rules to add a second edge between an A-labeled node and a B-labeled node. Avoiding duplication of graph-parts common to g_l and g_r shrinks the drawing of g_l and g_r, and greatly reduces the graphical depiction of the embedding. (The Y-notation rule appears in [Gött92, Fig. 14].)

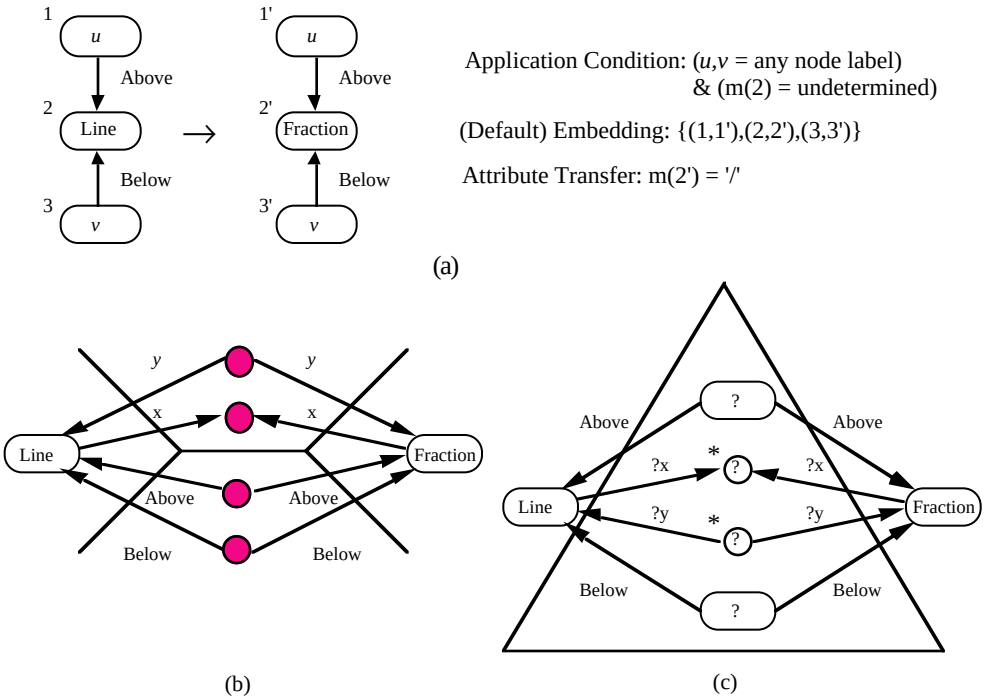


Figure 5 Textual (a) versus graphical (b, c) depiction of a simple embedding. These are three notations for a graph-rewrite rule to replace a *Line*-labeled node by a *Fraction*-labeled node, in the context of incoming *Above* and *Below* edges (as used in [GrBl95]). (a) The analogous embedding is conveyed by similarly-denotated nodes in visually-corresponding places; this is reinforced by the textual description "{(1,1'), (2,2'), (3,3')}". (b) In X-notation, the embedding is conveyed as optional context. One filled-in node (indicating arbitrary node label) and two edges depict a node-correspondence. Since directed edges are used, this must be repeated for incoming and outgoing edges. (c) In Δ-notation, the embedding is conveyed similarly, using *-groups to indicate 0 or more occurrences of the starred structures.

Isomorphisms versus General Graph Morphisms

Selection of a rewrite mechanism involves choosing isomorphisms or general morphisms for finding a subgraph g_l^{host} matching g_l. The utility of general graph morphisms is illustrated by small examples in the literature ([EhHK92, p. 560], [KrRo90, p. 200]). However, general morphisms could easily result in unexpected matches. We would be interested to hear of the use of general graph morphisms in large-scale system; debugging of such rewrite systems could be difficult.

A useful compromise is to allow the rule-author to selectively and explicitly indicate where general morphisms may be used. For example, Δ-rewriting uses subgraph isomorphism, but with a label-subscript notation (called a *fold*) to explicitly indicate groups of nodes which can optionally be matched to a single host-graph node [KaLG91] [LoKa92]. The utility of this construct is demonstrated by a rule to insert an element into a circular list: one rule works for circular lists of any length ≥ 1.

Extensions to the Rewrite Mechanism

Many extensions to rewrite mechanisms are useful in practice [BlFG95]. These include hierarchical label organization; calculation of attribute values; application conditions; parameters to graph-rewrite rules; variable node and edge labels within rewrite rules; variable graph structure within rewrite rules (e.g. optional or repeated nodes and/or edges). While all of these extensions are useful in certain applications, care must be used to select only the features necessary to cleanly express the graph transformations needed in a given application.

8. Mrs. Maggraphen: How do we modularize a graph-rewriting system?

A graph-rewriting system that is constructed in a modular way is easier to design, implement, debug, and maintain. Various aspects of a graph-rewriting system can be modularized -- the host-graph structure, the rewrite rules, the control specification. This is an active research area. Selected approaches to modularization are listed below. Several of these approaches can be used in combination.

Modular specification of host-graph structure

A description of allowable host-graph structure provides a foundation for the design of a graph-rewriting system. For example, the graph scheme in PROGRES defines statically-declarable graph properties [ELNSS92]. The graph scheme defines a class hierarchy for node labels and edge labels (multiple inheritance is allowed). Based on this, edge typing information is declared: for each edge-label, define what node-types are admissible at the endpoints of the edge. This static type information allows useful compile-time and run-time checks on graph-rewrite rules and on host-graph structure.

Host-graph triggers

This method of modularization is proposed for an unordered graph-rewriting system (wherein a host-graph is nondeterministically transformed by a set of graph-rewrite rules, with no control specification). To allow the designer to divide a large problem into more manageable subproblems, Δ-rewrite systems use *platforms* of related rules [LoKa92] [ToKa94]. These platforms are defined via specially labeled nodes called *trigger* nodes. To define a platform, choose a new trigger label. Every rewrite rule in the platform contains this trigger node in g_l (i.e., in the required context or retraction). If some rewrite rule wishes to invoke rules in a particular platform P, the rewrite rule adds the P trigger to the host graph. This satisfies one of the preconditions of rule-application from platform P, and thus may result in execution of a P-platform rule. The label of a trigger node is a tuple of arbitrary structure, and can include parameters to

influence the resultant application of a P-platform rule. This style of computation has been used to solve (on paper) a variety of specification and concurrency problems.

Modular control specification

In an ordered graph-rewriting system, the control specification can be structured in a modular way. For example, PROGRES provides transactions and subdiagrams as encapsulation tools [ZüSc92]. Ordering can be used to structure the computation into phases; for example, Build-Constrain-(Rank)-Incorporate recognition stages are used in [FaBl93] [GrBl95].

Two-level rewrite rules

Generic graph-rewrite rules (expressed as graphs) can be transformed via meta-rules, to produce executable rewrite rules. This has been used in a system to describe legal database transactions [GöHi94]: complex transactions are conveniently expressed as a hyperproduction, which is transformed by a metaproduction to produce the final production. This construct allows general operations to be expressed generically, as a hyperproduction, and then used in a variety of ways. For example, a hyperproduction for the manipulation of geometric objects can be specialized (via metaproductions) to treat polylines or rectangles.

Modules of rewrite rules arising from host-graph locality

In many applications, a host graph can be represented hierarchically, with an abstract level, as well as a refined level (consisting of local graphs and interfaces). In this case, graph productions can be modularized, with some modules transforming local graphs, others changing interfaces or the global graph, and yet others changing the graph hierarchy (split or join local graphs) [EhEn94] [Taen94].

Inheritance

Inheritance is a powerful tool for layering in object-oriented system design. Several forms of inheritance can be used within a graph-rewriting system; some examples are mentioned earlier in this list, as well as in [EhEn94].

Import-Export-Interface

As described in [EhEn94], graph transformations can be organized into modules, where each module has an import interface, local operations, and an export interface. This is challenging to implement, because imported graph-rewrite rules are known by name only.

9. Mr. Maggraphen: How can we design a graph-rewriting system to accommodate evolving host-graph structure?

The Maggraphens are producing software for clients with changing needs. Thus they need to plan for evolution of their graph-rewriting system. Adding a new feature may require extensions to the host-graph representation; for example, new node labels and edge labels may be introduced. When this happens, the Maggraphens expect most of their old rewrite rules to continue to work properly, and they want it to be clear which of the old rules must be updated in response to the expanded host-graph representation. Many aspects of a rewrite system bear on this problem, such as the use of graph schemes to statically declare permissible host-graph structure [ELNSS92]. Here we consider only the effect of choosing induced versus non-induced subgraph matching. (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 of g that connect two g_l^{host} nodes. A non-induced subgraph may omit some or all of these edges. This is illustrated in Figure 6.)

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: g_l cannot match unless the rule-author has anticipated all the edges present in that part of the host graph. Various edge-configurations must be enumerated in separate graph-rewrite rules (where a single non-induced rewrite rule could suffice).
- Non-induced subgraphs require extra application conditions, necessary to ensure the absence of certain host-graph edges.
- Implicit 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 6).

These points become particularly significant in case of host-graph evolution. Consider the addition of a new type of edge, with the new edge-label "Grow". Ideally, the old graph-rewrite rules should continue functioning as before, so that we merely need to create a few new rules that directly process the Grow edges. Both induced and non-induced subgraphs disappoint us.

- Using induced subgraphs, the presence of a Grow edge prevents application of any of the old rules. The old rules must be replicated, to enumerate all possible permutations of Grow edges that might occur in the g_l^{host} area.
- Using non-induced subgraphs, the old graph-rewrite rules continue to apply, but they perform implicit 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 all gluing and embedding models that use removal of g_l^{host} during the rewriting step. Improved semantics can be defined by using non-induced subgraph matching and avoiding node deletion where possible. (If g_l^{host} and g_r^{host} contain corresponding nodes, then these nodes are identified, rather than removing the g_l^{host} node and replacing it with the g_r^{host} node.) Such incomplete removal of non-induced subgraphs is provided in the definition of structured graph rewriting [KrRo90], and in the current PROGRES language [Schü91, p. 652]. (These semantics evolved over time: an earlier PROGRES reference describes the removal from host-graph of the complete subgraph corresponding to the non-induced g_l^{host} [EnLS87, p. 192]). 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.

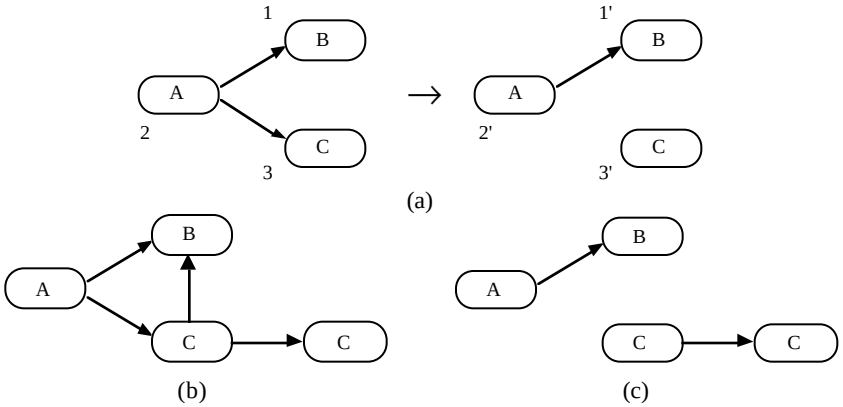


Figure 6 Induced versus non-induced subgraphs. Rewrite rule (a) is applied to the host graph (b). If an induced g_l^{host} is required, the isomorphism test fails and the rewrite rule cannot be applied. If non-induced subgraph matching is used, a suitable g_l^{host} is found and replaced, resulting in the new host graph (c). (We use the standard removal and replacement of g_l^{host}, as in the LEARRE steps: Locate, establish Embedding Area, Remove, Replace, Embed [Roze87].) Note the implicit edge-deletion in (c): 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).

10. Mrs. Maggraphen: Can hierarchical graphs be rewritten?

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

It is possible to consider hierarchical graphs as merely a notational device pertaining to graph display: 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, a full implementation of hierarchical-graph rewriting must give many special considerations to these edges. There is significant interest in the topic of hierarchical graph rewriting. Relevant references include a chart-based parser for hierarchical-graphs [MaKl92]; abstract graphs in a prototype algebraic-rewrite environment [LöBe93]; graphs where node labels can be graphs themselves [Schn93]; flat host-graph structure with hierarchy-expressing rewriting rules used to zoom in and out [EhHK92] and to manage and display a derivation [Hims94]; use of hierarchical graphs in a formal approach to plan generation [ArJa94]; use of hierarchically distributed graph transformations [Taen94].

11. Mr. Maggraphen: A lot of our C code performs graph inspections. How can we translate this into graph-rewrite rules?

The Maggraphens' current software freely mixes graph-inspection operations with graph-manipulation operations. Their graph inspection operations test local or global host-graph properties; examples include searching for a short path between nodes, or testing whether a graph is bipartite. The Maggraphens are concerned about the feasibility of translating to a pure graph-rewriting language. It is true that some host-graph inspection is performed during a graph-rewriting step (find g_l^{host}, find embedding edges, test the application condition). But these host-graph inspections accompany or follow subgraph-isomorphism testing, making it clumsy and expensive to express graph inspections that should be undertaken before the subgraph-isomorphism test.

More direct methods for expressing host-graph inspections are desirable. The designers of PROGRES recognize this, providing a variety of graph-inspection language constructs [ELNSS92]. Statically-declarable graph properties are defined in the graph scheme; these include the class hierarchy for node labels and edge labels, as well as restrictions on the source- and target-node-labels for edges with a particular edge label. In addition to this static construct, a variety of dynamic graph-inspection constructs are provided. General control structures direct the application of graph tests and graph productions [ZüSc92]. A rule's 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}. Independent of rewrite-rule application, path descriptions can be used to compute values for derived attributes. The applicability of a rewrite rule (or a subprogram of rules) can be tested without executing it. Global on-going graph inspection is proposed in [NaSc91]: global runtime conditions are used to state host-graph conditions that should always (or never) hold.

In summary, practically-usable graph rewriting languages must provide general facilities for graph inspection. Different language constructs may be suitable for unordered, grammar-based, or ordered graph-rewriting environments.

12. Mrs. Maggraphen: What about our user-interface and image-processing code? We want to leave that coded in C.

Graph rewriting is a suitable formalism for expressing only part of the Maggraphens' computation. To encourage widespread use of graph rewriting, we need convenient methods to combine graph rewriting with other styles of computation. This is an interesting research topic. A few possible approaches include combining graph rewriting with a blackboard architecture (with the host graph stored as part of the blackboard); combining 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); using 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]).

13. The Maggraphens: Thanks for the information. We'll probably continue to use C...

Currently we cannot advise the Maggraphens to stake their financial future on graph rewriting as their tool for product development. We hope that this situation will

change, so that in perhaps ten years time we could give different advice. Here's what we have to do to achieve this.

- Make it less difficult for an outsider to learn how to use graph rewriting in a practical application. The Maggraphens' experience mirrors our own: as we set out to apply graph rewriting to diagram recognition [FaBl93] [GrBl95] [Fahm95], we found it hard to figure out how to organize our computation. A careful reading of the literature was only of limited help: we found extensive discussion of graph-rewriting mechanisms, but little discussion of systems issues, and few examples of significantly-large graph-rewriting systems. Currently, the graph-rewriting literature appears confusing and uninviting to an outsider.
- Disseminate the graph-rewriting research/experience that is currently available. Graph rewriting is an intuitive, widely appealing concept, and outsiders are continually reinventing it. (Several attendees at Williamsburg invented graph rewriting during the course of their research, only later to discover that there already existed research on this subject, and thus found their way to the workshop. Other reinventors of graph-rewriting never find us. This should not be happening for a research community that has a decades-long history.) The profile of graph-rewriting must be raised. One important goal is to have graph-rewriting included in the standard undergraduate computing science curriculum. A few lectures' worth of material can be included in a data-structures or algorithms course, where graph-representation techniques and graph-inspection algorithms are already taught. Alternatively, graph grammars can be introduced in a formal languages class.
- Develop a better sense for which applications (or parts of applications) are suitable for implementation via graph rewriting. (We found an enthusiastic atmosphere at the Williamsburg conference: all sorts of computer-science applications were eagerly characterized as "yes, yes, graph grammars would be a great way to solve that problem".) We need to develop guidelines for identifying when graph rewriting use is advisable, and we need to develop methods for integrating graph rewriting into systems that use other styles of computation as well.
- Continue to develop and refine environments for graph rewriting. We are delighted that the PROGRES environment (and other environments to follow) are sufficiently mature to be generally usable. (When we began our diagram-recognition work, we found that the [Bunk82a] software was not in a state to be reused. Thus we had to create our own modest graph-rewriting environment; this took time, and the poor quality of the executing environment hampered our debugging and testing. We are happy that now, if we interest other colleagues in graph rewriting, we can direct them to existing graph-rewriting environments!)

In summary, our current situation is this. We are very enthusiastic about graph rewriting as a style of computation, and we are eager to convince other researchers to use graph rewriting. However, when we do succeed in convincing someone to try graph rewriting, we are left in the awkward position of being flooded with Maggraphen-type questions, few of which we can answer satisfactorily. Let us continue to work toward giving graph rewriting the widespread use it deserves.

References

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

[ArJa94] O. Arnold and K. Jantke, "Therapy Plans as Hierarchically Structured Graphs," in [IWGG94], pp. 338-343.[BlFG95] D. Blostein, H. Fahmy, A. Grbavec, "Practical Use of

Graph Rewriting," Technical Report No. 95-373, Computing and Information Science, Queen's University, Jan 1995.

[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.

[BuGT91] H. Bunke, T. Glauser, T. Tran, "An Efficient Implementation of Graph Grammars Based on the RETE Matching Algorithm," in [IWGG91], pp. 174-189.

[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.

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

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

[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.

[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.

[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.

[EhEn94] H. Ehrig and G. Engels, "Pragmatic and Semantic Aspects of a Module Concept for Graph Transformation Systems," in [IWGG94], pp. 157-168.

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

[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.

[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.

[Fahm95] H. Fahmy, "Reasoning in the Presence of Uncertainty via Graph Rewriting," Ph.D. Thesis, Computing and Information Science, Queen's University, March1995.

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

[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.[GöHi94] H. Göttler and B. Himmelreich, "Modeling of Transactions in Object-Oriented Databases by Two-level Graph Productions," in [IWGG94], pp. 151-156.[GrBl95] A. Grbavec and D. Blostein, "Mathematics Recognition Using Graph Rewriting," *Third International Conference on Document Analysis and Recognition*, Montreal, Canada, August 1995, pp. 417-421.

[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.

[Hims91] M. Himsolt, "GraphEd: An Interactive Tool for Developing Graph Grammars," in [IWGG91], pp. 61-65.

[Hims94] M. Himsolt, "Hierarchical Graphs for Graph Grammars," in [IWGG94], pp. 67-70.

[IWGG79] *Intl. Workshop on Graph Grammars and Their Application to Computer Science and Biology*, LNCS Vol . 73, V. Claus, H. Ehrig, G. Rozenberg Eds, Springer, 1979.

[IWGG83] *Second Intl. Workshop on Graph Grammars and Their Application to Computer Science*, LNCS Vol. 153, H. Ehrig, M. Nagl, G. Rozenberg Eds, Springer, 1983.

[IWGG87] *Third Intl. Workshop on Graph Grammars and Their Application to Computer Science*, LNCS Vol. 291, Ehrig, Nagl, Rozenberg, Rosenfeld Eds, Springer, 1987.

[IWGG91] *Fourth Intl. Workshop on Graph Grammars and Their Application to Computer Science*, LNCS Vol. 532, H. Ehrig, H. Kreowski, G. Rozenberg Eds, Springer, 1991.

[IWGG94] Pre-proceedings of the *Fifth Intl. Workshop on Graph Grammars and Their Application to Computer Science*, Williamsburg, VA, Nov. 1994. Full versions of selected papers appear in this volume.

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

[Kaul83] M. Kaul, "Parsing of Graphs in Linear Time," in [IWGG83], pp. 206-218.

[Klau94] C. Klauck, "Heuristic Driven Chart-Parsing," in [IWGG94], pp. 107-113.

[KrRo90] H.-J. Kreowski, G. Rozenberg, "On Structured Graph Grammars, I, II" *Information Sciences*, Vol. 52, 1990, pp. 185-210, 210-246.

[LiFu86] W. Lin and K.S. Fu, "A Syntactic Approach to Three-Dimensional Object Recognition," *IEEE Trans. Systems Man and Cybernetics*, **16**(3), May 1986, pp. 405-422.

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

[MaKl92] J. Mauss and C. Klauck, "A Heuristic Driven Parser Based on Graph Grammars for Feature Recognition in CIM," *Advances in Structural and Syntactic Pattern Recognition*, Ed. H. Bunke, World Scientific, 1992, pp. 611-620.

[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.

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

[ReSc94] J. Rekers and A. Schürr, "Parsing for Context-Sensitive Graph Grammars," in [IWGG94], pp. 89-94.

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

[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," *Theoretical Computer Science*, Vol. 109, 1993, pp. 257-275.

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

[Strz90] T. Strzalkowski, "Reversible Logic Grammars for Natural Language Parsing and Generation," *Canadian Computational Intelligence Journal*, **6**(3), pp. 145-171, 1990.

[Taen94] G. Taentzer, "Hierarchically Distributed Graph Transformations," in [IWGG94], pp. 430-435.[ToKa94] W. Tolone and S. Kaplan, "A Semantic Definition for Introspect using Δ-Grammars," in [IWGG94], pp. 418-423.[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*, LNCS Vol 570, Springer Verlag, 1992.

[Zünd94] A. Zündorf, "Graph Pattern Matching in PROGRES," in [IWGG94], pp. 174-178.