RWTH Aachen
Fachgruppe
Informatik

Computing with Graphs and Graph Rewriting

D. Blostein / A. Schürr

AIB 97-8*

Aachener Informatik-Berichte

Computing with Graphs and Graph Rewriting

Dorothea Blostein
Dept. Computing and Information Science
Queen's University, Kingston,
Ontario, Canada K7L 3N6
blostein@qucis.queensu.ca

Andy Schürr
Lehrstuhl für Informatik III
RWTH Aachen,
D-52074 Aachen, Germany
andy@i3.informatik.rwth-aachen.de

Abstract: Graphs are a popular data structure. Programmers are faced with the need to represent, inspect, modify, display and recognize graphs. In this paper we describe a systematic approach to graph modification, using graph rewrite rules. Graph rewrite rules replace one subgraph by another subgraph. In other words, a graph rewrite rule specifies how, and under what conditions, to replace one piece of a graph by another piece. This is an intuitive and useful generalization of string rewriting. We illustrate these ideas using a sampling of applications: precedence-network creation, visual language editing, and recognition of mathematics notation. Graph rewriting examples are written in PROGRES, a programming language with both visual and textual elements.

1. Introduction

Graphs are widely used, both in society and in computer programs, to depict networks of related items. Graph nodes represent objects or concepts, and graph edges represent relationships between them. For example, in a circuit diagram, graph nodes represent chips or gates or resistors, and graph edges are electrical connections. In a flow chart, nodes represent pieces of code, and edges represent control flow. In an organizational chart, nodes represent employees, and edges represent the reporting structure. Node nesting can be used in applications where hierarchical relationships are important. Auxiliary information is stored as node attributes or edge attributes. In a computer program, attributes can be of any type, including integer, string, record, and array. These three graph elements — nodes, edges, and attributes — allow graphs to represent complex, structured information at a convenient level of abstraction.

Programmers are often faced with the need to represent, inspect, modify, display and recognize graphs (Figure 1). In this paper we describe a systematic approach to graph modification, using graph rewrite rules.

Graph rewrite rules replace one subgraph by another subgraph [8][15]. In other words, a graph rewrite rule specifies how, and under what conditions, to replace one piece of a graph by another piece. This is an intuitive and useful generalization of string rewriting. Many existing programs modify graphs, but do so with pointer-manipulation code to add and delete graph nodes and edges. This is a low-level and error-prone coding style. Graph rewriting languages provide a higher-level interface for graph modifications. The explicit use of graph rewrite rules offers several advantages. The graph rewrite rules are clearly delineated from the rest of the

This research has been supported by Canada's Natural Science and Engineering Research Council, the Deutsche Forschungsgemeinschaft, the ESPRIT working groups COMPUGRAPH and APPLIGRAPH, and the RWTH Graduiertenkolleg Informatik und Technik.

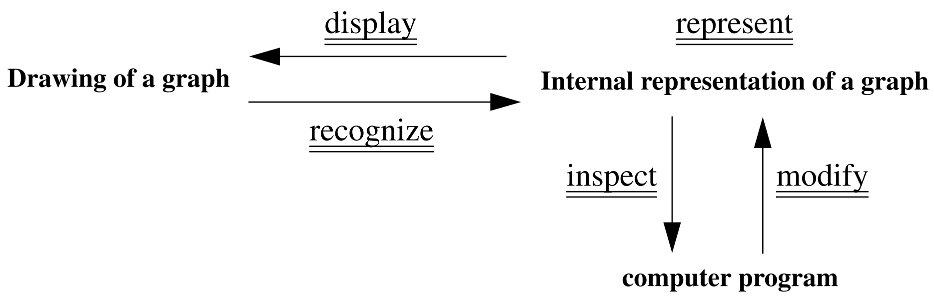


Figure 1 Treatment of graphs in a computer program. The program *represents* graphs, using an internal representation such as adjacency list, adjacency matrix, or incidence matrix [4]. The program *inspects* graphs, perhaps using standard algorithms such as breadth-first or depth-first search. The program *modifies* graphs; as discussed below, this can be done via graph rewrite rules. The program *displays* graphs to a user, using a graph-layout algorithm [16]. Finally, in applications with large quantities of paper-based drawings of graphs, the program *recognizes* graphs, using document-image analysis techniques [12].

code, making them easier to comprehend and maintain. Graph modifications are clearly and conveniently expressed using the visual language constructs provided by graph rewriting languages. The non-deterministic nature of graph rewriting is convenient for expressing specifications and implementations of software at a suitably abstract level. Also, the theoretical foundations of graph rewriting can assist in proving correctness and convergence properties of the software.

We illustrate these ideas using a sampling of applications: precedence-network creation, visual language editing, and recognition of mathematics notation. Graph rewriting examples are written in PROGRES, a programming language with both visual and textual elements. The PROGRES environment (available on the web [17]) executes under the Solaris operating system. For other formulations and applications of graph rewriting, please refer to [8] [15].

2. Applications of Graph Rewriting

Graph rewriting is used both in specifying and in implementing software. Table 1 summarizes a variety of applications of graph rewriting. The HostGraph represents the state of the computation; its structure is described in the first column of Table 1. The graph rewrite rules implement the dynamic behaviour of the system, as described in the second column.

For example, the first entry in Table 1 summarizes the use of graph rewriting in specifying the semantics of an editor for a visual language. Programs in the visual language appear as diagrams on the screen, but are internally represented by a HostGraph. The structure of a valid HostGraph is declared by a type declaration called a graph schema, use of which is illustrated later. Graph rewrite rules are defined for each editing operation. These rules specify the semantics of the editing operations. The rules can be executed, providing a fully-functional editor for the visual language. Eventually, the specifications can be automatically or manually translated to an implementation language such as C, for faster execution. The combination of graph schema and run-time constraints is used to ensure that all editing operations preserve the validity of the HostGraph. This enforces the syntax and semantics of the visual language.

Contents of the HostGraph	Rewriting the HostGraph
Generate Visual Language Editors [1] [2]	
The HostGraph represents the current state of editing, acting as an abstract syntax graph for the language under consideration. (See Section 4.)	Graph rewrite rules modify the HostGraph in response to each editing operation. The syntax (and static semantics) of the visual language are enforced.
Create a Software Development Environment [6]	
The HostGraph represents files, such as requirements specifications, designs, source code, and documentation. Graph edges between files support cross-referencing and version management.	Graph rewrite rules provide coarse-grained operations (configuration management) as well as fine-grained operations (in editors, interpreters, and integration tools).
Recognize the Information in a Document Image [7] [9]	
An initial HostGraph is constructed from character-recognition and symbol-recognition data. The graph edges express geometric relationships among symbols. (See Section 5.)	Graph rewrite rules use geometric relationships to deduce logical relationships, thus constructing a representation of the meaning of the document (including tables or mathematical expressions).
Define Constraint Programming Algorithms [11]	
The HostGraph represents a constraint network. Nodes represent variables and edges represent binary constraints. The possible values for each variable are stored as node attributes.	Graph rewrite rules provide constraint-solving algorithms which either modify the structure of a constraint network or manipulate the possible variable values associated with its nodes.
Specify Distributed Systems of Processes [3]	
The HostGraph models the state of the distributed system. Nodes are processes and messages, edges define communication channels as well as sender and recipient of messages.	Graph rewrite rules model message passing among processes. Properties of the distributed system are proven using properties of the graph rewrite rules.
Reconfigure an Array of Processors [5]	
The HostGraph represents a processor array, with one node per processor and an edge for each active communication line between processors.	Graph rewrite rules activate and deactivate communication lines to replace faulty processors with spares. The use of graph rewriting helps in the design and analysis of reconfiguration algorithms.
Implement Functional Languages [1] [13]	
The HostGraph represents the abstract syntax tree of a program; this tree becomes a Directed Acyclic Graph (DAG) due to sharing of subexpressions.	Graph rewrite rules are used to reduce the DAG to a single node. This has the effect of evaluating the program represented by the DAG.
Build a Neural Network [10]	
The HostGraph represents the interconnection structure of the current neural network. The initial neural network consists of a single neuron.	Graph rewrite rules cause growth of the neural network. A set of trees directs which graph rewrite rules to apply next. To find a good network, genetic algorithms are applied to the set of trees.
Model Plant Growth [14]	
The HostGraph represents the arrangement of cells in a plant.	Graph rewrite rules model the various types of cell division that occur in plant growth. These graph rewrite rules are applied in parallel to all parts of the HostGraph.

Table 1 Representative applications of graph rewriting. Additional references for these and other applications are available on the web [17].

3. A Notation for Graph Rewrite Rules

Different types of rewrite rules have been defined to operate on strings, trees, and graphs. String-rewrite rules replace one substring by another. They are commonly used in string grammars to define programming language syntax. Tree-rewrite rules replace one subtree by another. They are used, for example, to transform parse trees in order to translate between dialects of programming languages. Graph rewrite rules replace one subgraph by another.

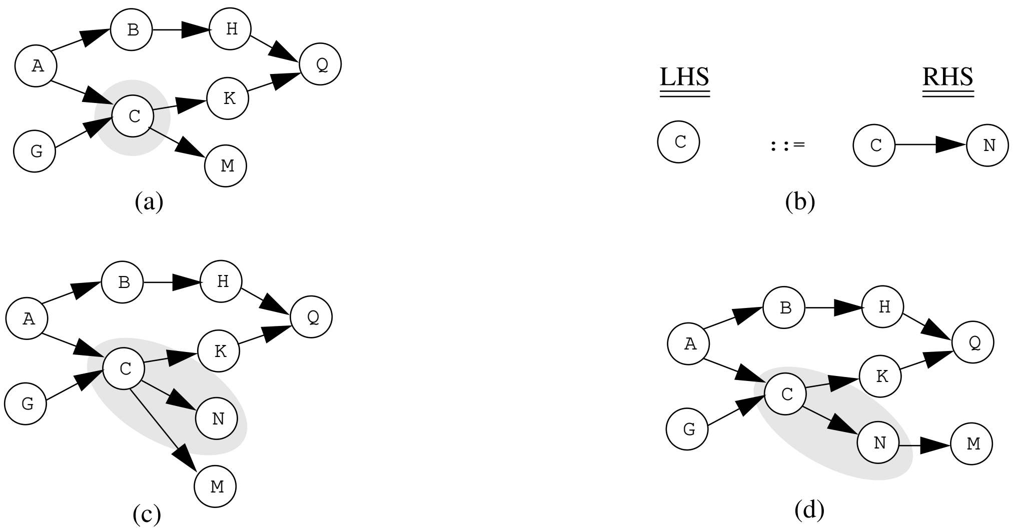


Figure 2 Example of a graph rewrite rule, using informal notation. The HostGraph in (a) is a precedence network. Edges indicate task-completion constraints. For example, task C cannot begin until tasks A and G complete. The graph rewrite rule in (b) adds a new task N, to follow task C. This rule transforms HostGraph (a) into one of several possible results, such as (c) or (d). These results use different *embeddings* of NewSubgraph in HostGraph. In other words, different sets of edges are used to connect NewSubgraph to HostGraph. On its own, the execution environment is unlikely to choose an embedding that is appropriate for the application. Instead, the author of graph rewrite rule (b) has to provide additional information to specify the desired embedding. Typically, the embedding edges for NewSubgraph are constructed based on the embedding edges that belonged to OldSubgraph.

As shown in Figure 2, a graph rewrite rule is defined by two graphs, *LHS (Left Hand Side)* and *RHS (Right Hand Side)*. LHS is used to find which part of HostGraph to change, and RHS specifies the changes to make there. The following steps are taken to apply a graph rewrite rule:

1. Find *OldSubgraph*, an instance of LHS within HostGraph. If HostGraph contains several instances of LHS, then the graph rewrite rule can be applied in any of these places. (If parallel graph rewriting is used, then the graph rewrite rule is applied in parallel in all of these places.)
2. Remove OldSubgraph from HostGraph. (Some parts of OldSubgraph may be unchanged by the rewrite rule, and hence do not need to be removed. This is indicated by a node-preserving notation in RHS, as introduced in Figure 5.)
3. Make a copy of RHS, named *NewSubgraph*, and insert it into HostGraph. Here we must decide what edges to create between NewSubgraph and HostGraph. These edges are known as the *embedding* of NewSubgraph in HostGraph. Graph rewriting languages include constructs for specifying the desired embedding. Thus, the author of a graph rewrite rule provides a description of LHS, RHS, and the embedding.

Sets of graph rewrite rules are used to carry out a computation. We illustrate this with a set of graph rewrite rules that build precedence networks. Figure 3 shows successive HostGraphs, starting with an empty graph, and ending with a precedence network containing seven nodes. The graph rewrite rules are given in Figure 4, Figure 5, and Figure 6. In this example, a fully-specified ordering of graph rewrite rules is used, beginning with CreateDiagram("A", "B"), InsertTask("A", "B", "C"). In some applications it is appropriate to allow nondeterminism in the ordering of rules. In case of nondeterminism, the execution environment arbitrarily chooses any of the applicable graph rewrite rules. Backtracking can be used to avoid dead ends in the graph transformation process.

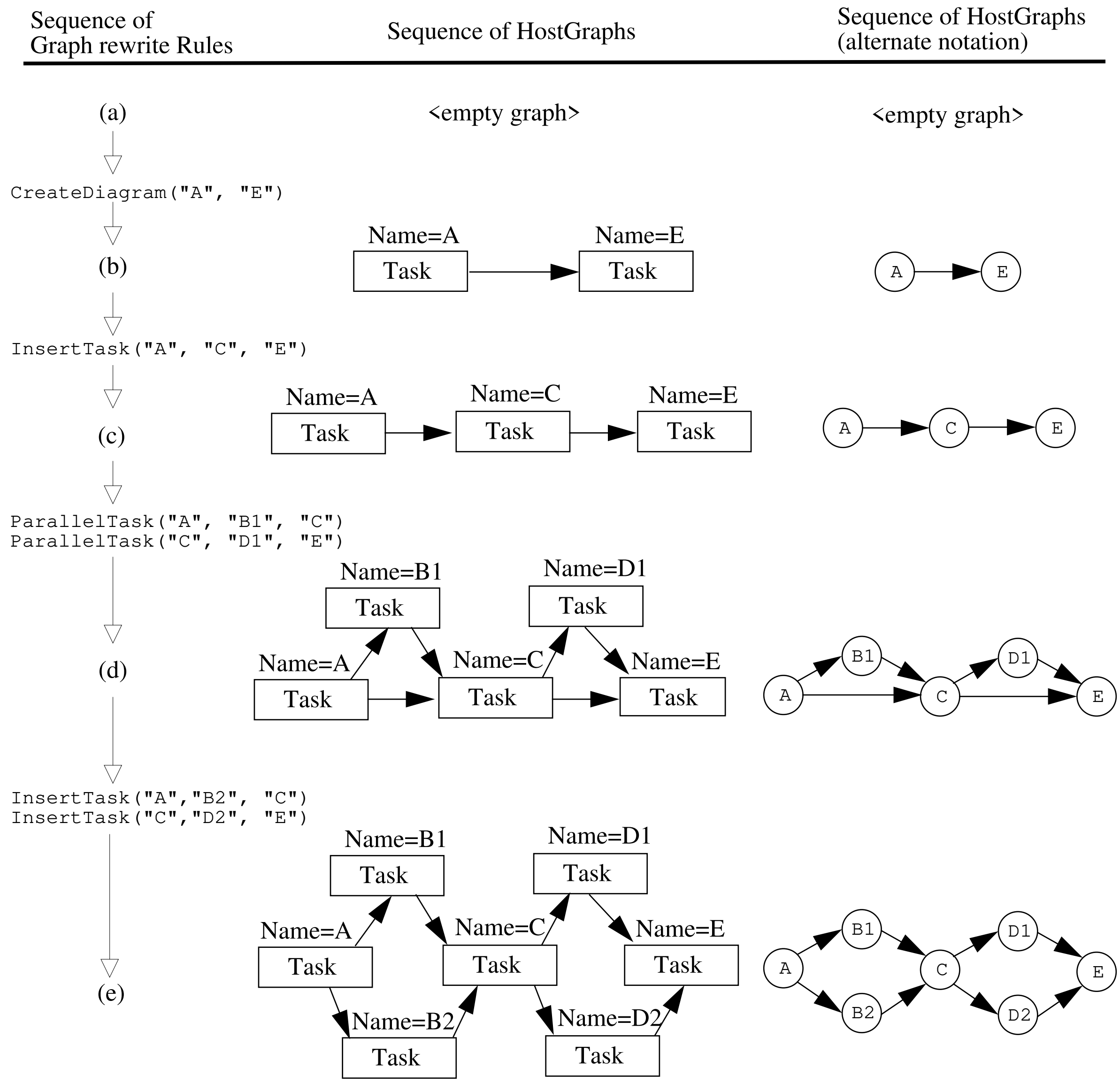
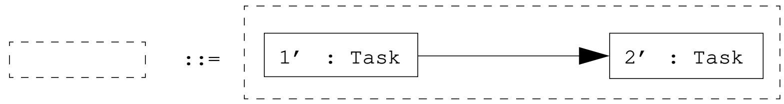


Figure 3 Generating precedence networks using graph rewrite rules. The leftmost column lists the order of application of the graph rewrite rules CreateDiagram **(Figure 4),** InsertTask **(Figure 5), and** ParallelTask **(Figure 6). Parameters to rewrite rules are used to specify the names of nodes that are being matched or being created. The second column shows the sequence of HostGraphs, starting with an empty graph in (a) and ending with a seven-node graph in (e). Every node in the graph has the node-label** Task**, and an attribute** Name **of type** string**. The third column shows a more compact display of the HostGraphs, as would be displayed in the user interface of this precedence network program.**

As illustrated by Figure 4 to Figure 6, a graph rewrite rule uses LHS and RHS to treat graph structure, and uses condition and transfer clauses to treat attributes. LHS and RHS describe networks of labeled nodes and labeled edges. There are no attributes in LHS and RHS. When a rule matches LHS to HostGraph, it matches the graph structure, finding corresponding nodes with the same edge connectivity and matching labels. More formally, this is called a *subgraph isomorphism test.* (Some graph rewriting mechanisms use a subgraph homomorphism test, in which several LHS nodes can map to a single HostGraph node [15].) Attribute values are ignored in the subgraph isomorphism test, but can be tested in a condition clause. Similarly, NewSubgraph obtains its structure from RHS, but obtains its attribute values from a transfer clause.

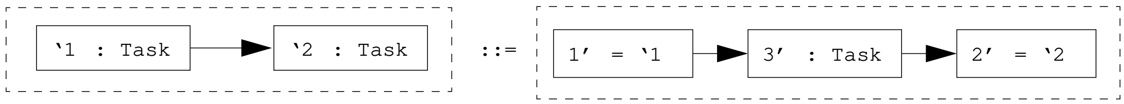
```
production CreateDiagram( StartName, StopName : string) =
```




```
  transfer 1'.Name := StartName;
           2'.Name := StopName;
end;
```

Figure 4 **PROGRES source code for** CreateDiagram**, a graph rewrite rule that creates a two-node precedence network. The code uses a mixture of textual and visual notation. Dashed rectangles, separated by ::=, enclose LHS and RHS. Since LHS is empty, the rule is applicable to any HostGraph, even an empty one. This rule creates two connected nodes, with** Name **attributes taken from the parameter list.** Task **is a node label; in this example all nodes have the same label. Node denotations, here 1' and 2', are local names for nodes. These denotations are used only within the graph rewrite rule. The** transfer **clause assigns attribute values; for example, the** Name **attribute of node 1' is assigned the value of the** StartName **parameter.**
This rule creates two new graph nodes, with no connections to other nodes in HostGraph. If CreateDiagram **is applied three times in succession, it adds three disconnected pairs of nodes to HostGraph.**

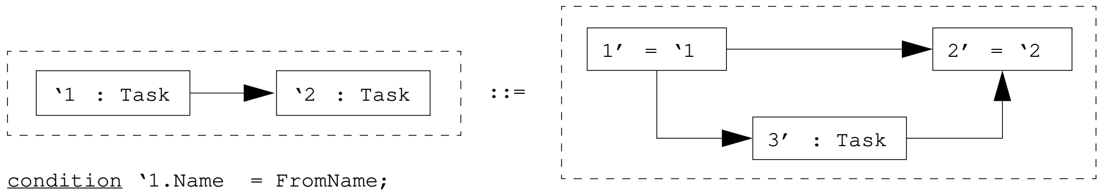
```
production InsertTask( FromName, NewName, ToName : string) =
```




```
  condition `1.Name  = FromName;
            `2.Name  = ToName;
  transfer  3'.Name := NewName;
end;
```

Figure 5 InsertTask **inserts a new** Task **node between two connected nodes. LHS nodes are denoted by '1 and '2 and RHS nodes by 1', 3' and 2'. To apply the rule, LHS is matched to a subgraph of HostGraph. The match is successful only if the** condition **clause is satisfied: the** Name **of the node matching '1 must equal the** FromName **parameter. Also, the** Name **of the node matching '2 must equal the** ToName **parameter.**
The effect of this rule is illustrated in Figure 3. The new 3' node has no embedding edges (no direct connection to other nodes in HostGraph); the only connections to 3' are the two edges shown in RHS. The embedding of nodes 1' and 2' is indicated by the *node preserving* notation 1' = '1 and 2' = '2. Here 1' and '1 refer to the same node, as do 2' and '2. The graph rewrite rule preserves the embedding edges and node attributes of these nodes.

```
production ParallelTask( FromName, NewName, ToName : string) =
```




```
  condition `1.Name  = FromName;
            `2.Name  = ToName;
  transfer  3'.Name := NewName;
end;
```

Figure 6 ParallelTask **creates a new task that is in parallel with an existing edge. This rule is similar to rule** InsertTask **(Figure 5); the only difference is in RHS.**

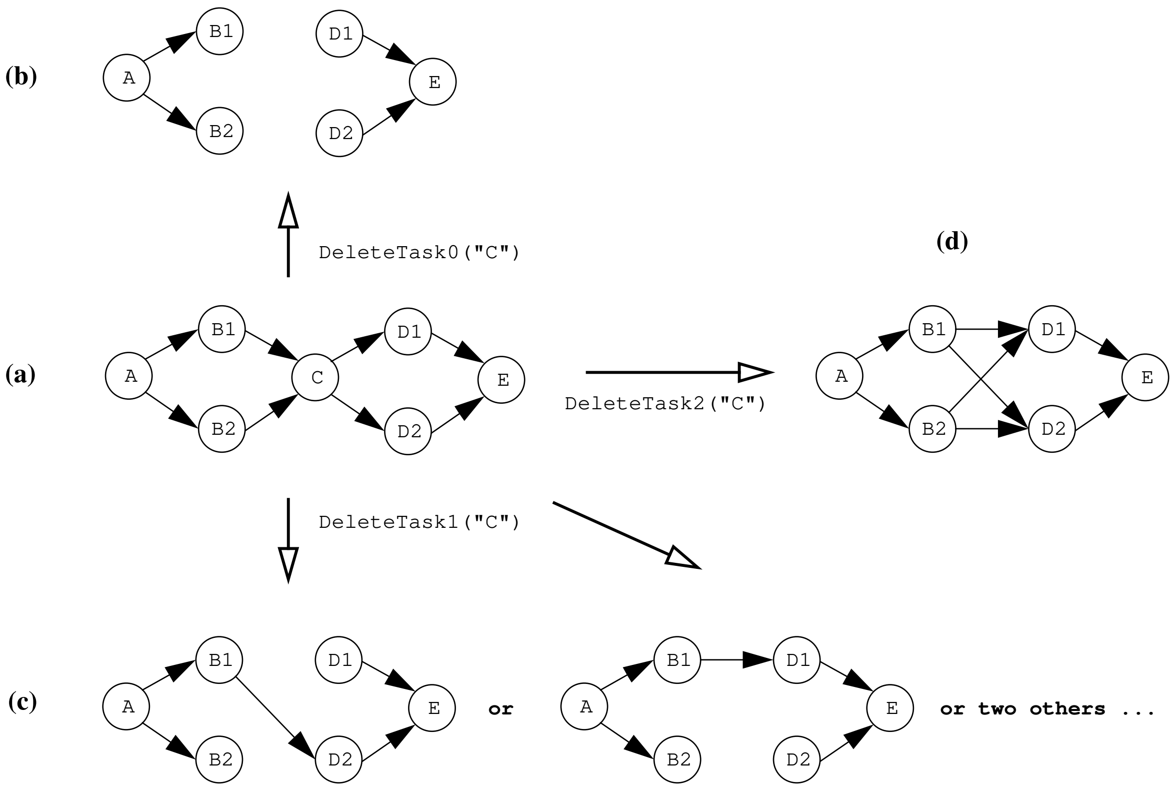


Figure 7 Deleting a node from HostGraph. Graph (a) at center is the initial HostGraph. We are deleting node C from this graph. Applying the rule from Figure 8 results in graph (b); the rule in Figure 9 results in one of the graphs (c), and the rule in Figure 10 gives result (d). In this application, result (d) is most appropriate: tasks that succeeded C cannot start until completion of all tasks that preceded C.

In some situations LHS needs to match an OldSubgraph of variable size. This is the case when we delete a node in a precedence network. Figure 7 illustrates the effect of three different node-deletion rules: the rules in Figure 8 and Figure 9 produce an undesired result and the rule in Figure 10 produces the desired result. Figure 10 uses *set nodes* to match a variable number of HostGraph nodes. Set nodes are used in a similar fashion when we split one task into two parallel tasks: each of the parallel tasks gets a copy of the incoming and outgoing edges of the node which was split.

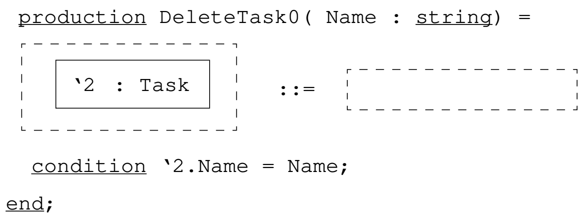
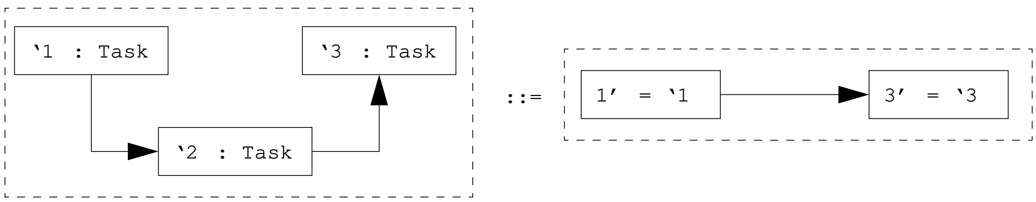


Figure 8 A simple rule to delete a node. Application of this rule is illustrated by Figure 7 (a) and (b). The node is deleted, as are all embedding edges that used to connect this node to other nodes in HostGraph.

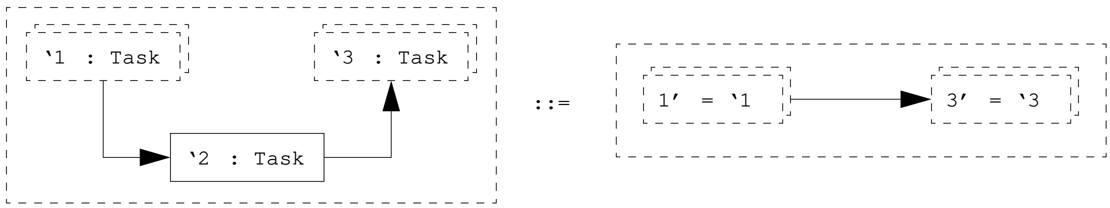
```
production DeleteTask1( Name : string) =
```




```
  condition `2.Name = Name;
end;
```

Figure 9 This rule deletes a node, while preserving one of the predecessor-successor links. Application of this rule is illustrated by Figure 7 (a) and (c). LHS matches the victim node ‘2, some predecessor ‘1, and some successor ‘3. As shown in Figure 7 (a) and (c), the resulting HostGraph contains an edge for one of the paths that used to run through the victim node.

```
production DeleteTask2( Name : string) =
```




```
  condition `2.Name = Name;
end;
```

Figure 10 This rule deletes a node, while preserving all of the predecessor-successor links. Application of this rule is illustrated by Figure 7 (a) and (d). The victim node ‘2 is deleted, and edges are created from all predecessors to all successors. This is accomplished by the *set nodes*, which have double-dashed borders. Set node ‘1 matches all predecessors and ‘3 matches all successors. Depending on the HostGraph, ‘1 and ‘3 match zero, one, two or more nodes. Thus the size of OldSubgraph is variable. The set nodes in RHS are joined by an edge; this specifies the creation of an edge from every node in set 1’ to every node in set 3’.
The HostGraph nodes matched by set nodes are unchanged. Set nodes always use the 1’ = ‘1 node preserving notation, thus preserving node labels and node attributes.

Now that we have introduced graph rewriting notation, we describe the use of graph rewriting in two larger applications.

4. Graph Rewriting for Visual Languages and Tools

Visual languages convey meaning using diagrams. A large variety of visual languages have been developed. Graph rewriting is a useful tool for precisely specifying the syntax and semantics of these languages, and for mechanizing the development of tools to edit and execute these languages. Currently, many visual languages are only informally defined, by example. This is true for languages used in object-oriented software development, such as class diagrams, state transition diagrams, object interaction diagrams, and message sequence charts. Development of visual-language editors and compilers is often based on user interface toolkits or generic graph editors. These do not provide support for parsing visual language sentences, checking static semantic rules, and building compilers or interpreters. Graph rewriting provides higher-level support for specifying the visual language, and for developing tools to support the language.

We illustrate this with a visual language for precedence networks. The PROGRES source code for this example is available on the web [17]. Figure 11 shows several screen dumps of a precedence-network tool which is automatically generated from the PROGRES code. This tool offers support for editing precedence networks, computing the earliest and latest start times for tasks, finding critical paths in a network, assigning needed resources to tasks, and semiautomatically eliminating resource scheduling conflicts.

a) Task View **b) Resource View**

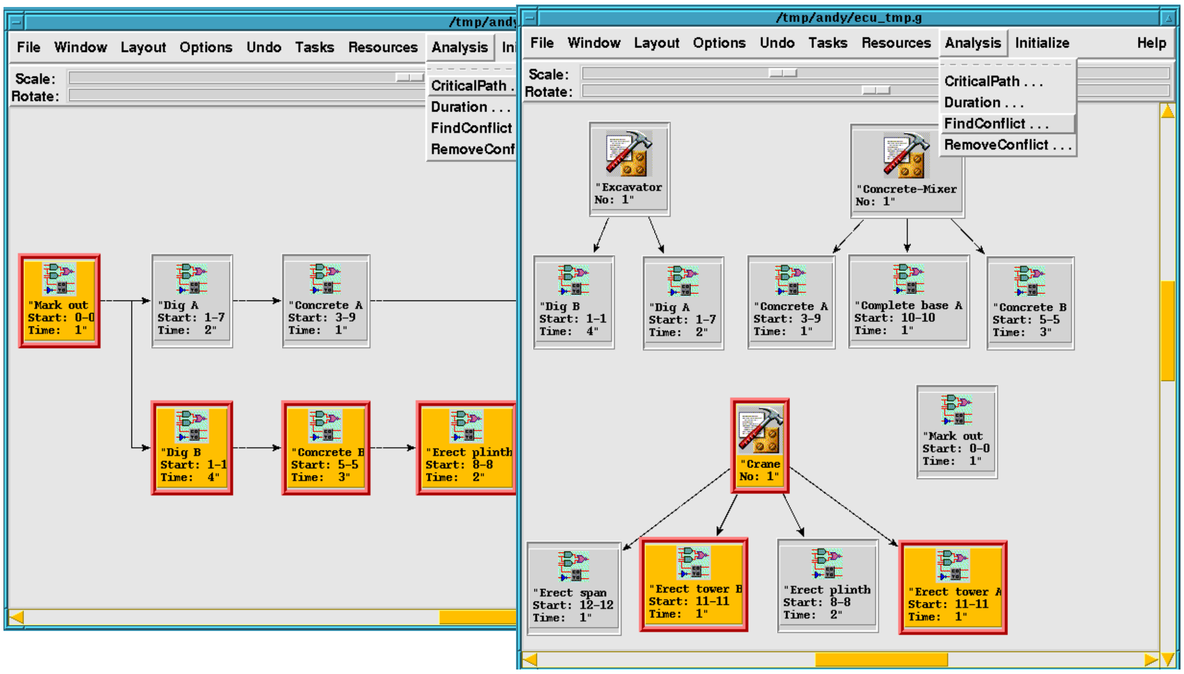


Figure 11 Two screen-dumps from a precedence-network tool. The C++ and Tcl/Tk source code for this tool is automatically generated from PROGRES source code. The task view in a) displays a precedence network for building a bridge. The highlighted critical path (shown in red on a color screen) includes all tasks except Dig A **and** Concrete A**. The resource view in b) supports the assignment of resources to tasks. The highlighted resource** Crane **and its two associated tasks** Erect Tower A **and** Erect Tower B **constitute a resource-scheduling conflict.**

The following steps are used to define the precedence-network tool of Figure 11.

- Write a *graph schema*, which acts as a type declaration for the HostGraph structure that underlies the visual language. This is illustrated in Figure 12. A graph schema can include static integrity constraints; these enforce the static semantics of the visual language.
- Write graph rewrite rules for each editing operation that is to be provided for the visual language. The execution of the visual language (its dynamic semantics) can be defined by additional graph rewrite rules. Complex graph operations can be defined as a combination of simpler graph rewrite rules.
- Specify the user interface as follows. Define menu hierarchies for those graph rewrite rules which implement user-level commands. Bind rule-parameters to mouse buttons or other input devices. Choose which aspects of the HostGraph are to be displayed in the user interface. Define the necessary icons.

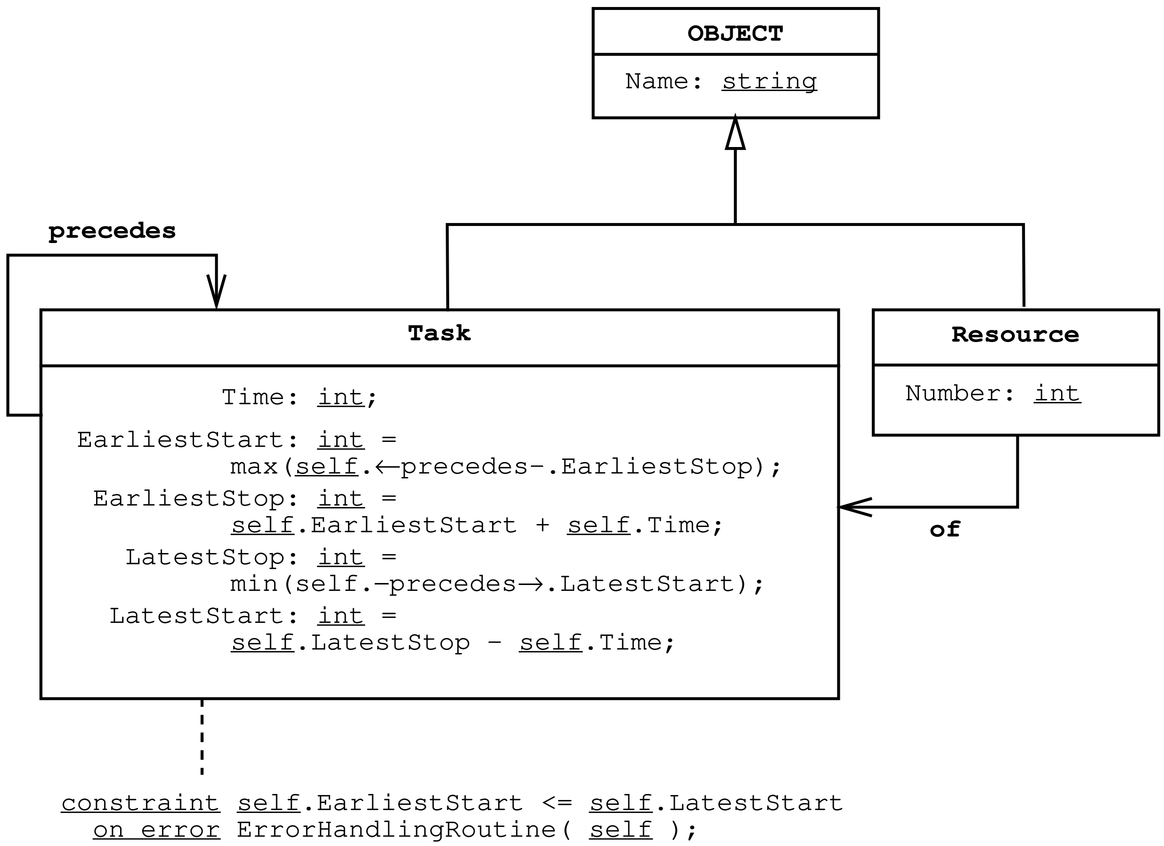


Figure 12 This graph schema defines the structure of the HostGraph used in the precedence-network tool of Figure 11. Two edge labels are used: of **and** precedes**. An** of **edge connects a** Resource **node to a** Task **node, and represents the assignment of the resource to the task. A** precedes **edge connects one** Task **node to another.**
Node labels are arranged in a class hierarchy. At the top of the hierarchy, OBJECT **nodes have a** Name **attribute; this attribute is inherited by both** Task **and** Resource **nodes.** Resource **nodes have an additional** Number **attribute.** Task **nodes have five additional attributes, four of which are derived attributes.** EarliestStart**, the earliest possible start time of the task, is computed by traversing all incoming** Precedes **edges in reverse direction and using the maximum** EarliestStop **value at those nodes. The computation of** LatestStop **is similar, but outgoing** Precedes **edges are traversed.**
Task **nodes have a run-time constraint that** EarliestStart <= LatestStart**. Violation of this constraint means the precedence network cannot be scheduled in the given period of time. This causes the error-handling routine to be called.**

Each editing operation for precedence networks is defined by a graph rewrite rule. Rules for adding and deleting tasks were presented in the previous section; those rules omitted a few details about the manipulation of Time attributes. The rewrite rules for detecting and eliminating resource scheduling conflicts are more complex. Figure 13 shows one example of such a rule, as viewed in the PROGRES editor. As mentioned, the complete PROGRES source code is available on the web [17].

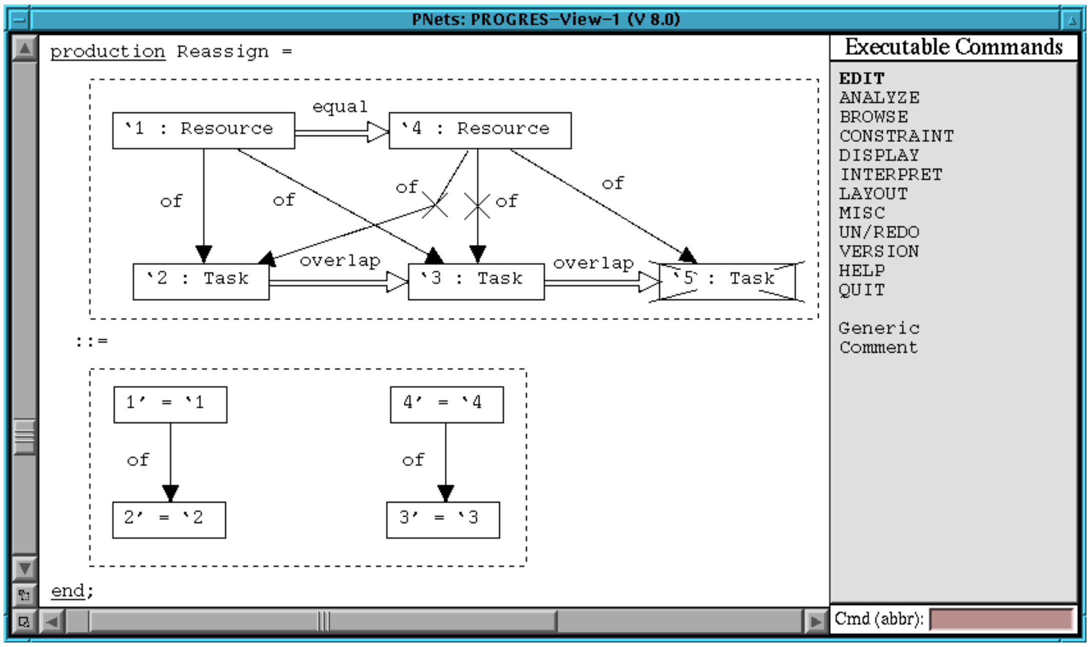


Figure 13 A graph rewrite rule for eliminating an inconsistent assignment of resources. LHS nodes '2 and '3 are two tasks whose execution times must overlap in any schedule. Both of these tasks are assigned the use of resource '1. Node '4 is an equal resource which is not currently assigned to '2 or '3 (as indicated by the crossed-out of edges); in addition, resource '4 is not already assigned to another task which overlaps '3 (as indicated by the crossed out '5 node). If a successful match to LHS is found, the resource conflict can be eliminated: RHS arranges that task '3 uses resource '4 in place of resource '1.
The paths overlap and equal, depicted by double arrows, are derived binary relationships between nodes which are defined as predicate logic formulas over the HostGraph's edges and attributes.

5. Graph rewriting for Pattern Recognition

Pattern-recognition problems pervade our society. There are widespread efforts to find automated solutions to these problems. For example, the post office recognizes addresses on envelopes, medical labs classify and count red and white blood cells, and banks read the monetary value of cheques. Graph rewriting is a useful tool in many pattern-recognition applications. Graphs provide a general and flexible mechanism for expressing spatial and logical relations between arbitrary sets of pattern primitives. Graph rewriting is used to transform the initial, primitive-based graph into a final, interpretation-based graph. We illustrate this by describing a particular pattern recognition task: the recognition of a mathematical expression from a document image [9]. The code is available on the web [17]. A similar style of graph rewriting has been used in the recognition of music notation [7].

As shown in Figure 14, a handprinted or typeset mathematical expression is scanned to turn it into an image. Character recognition is applied, resulting in a two-dimensional arrangement of characters and symbols. This is converted into an initial HostGraph, with the node labels chosen according to the identity of the characters, and with node attributes indicating the (x, y) location of the characters. This initial HostGraph is transformed by graph rewriting rules, which are applied in phases. The Build phase adds edges which represent potentially-important spatial relationships among the symbols. The Constrain phase examines a larger context, to remove unneeded spatial relationships. This leaves only semantically-important relationships, which are interpreted during the Parse phase.

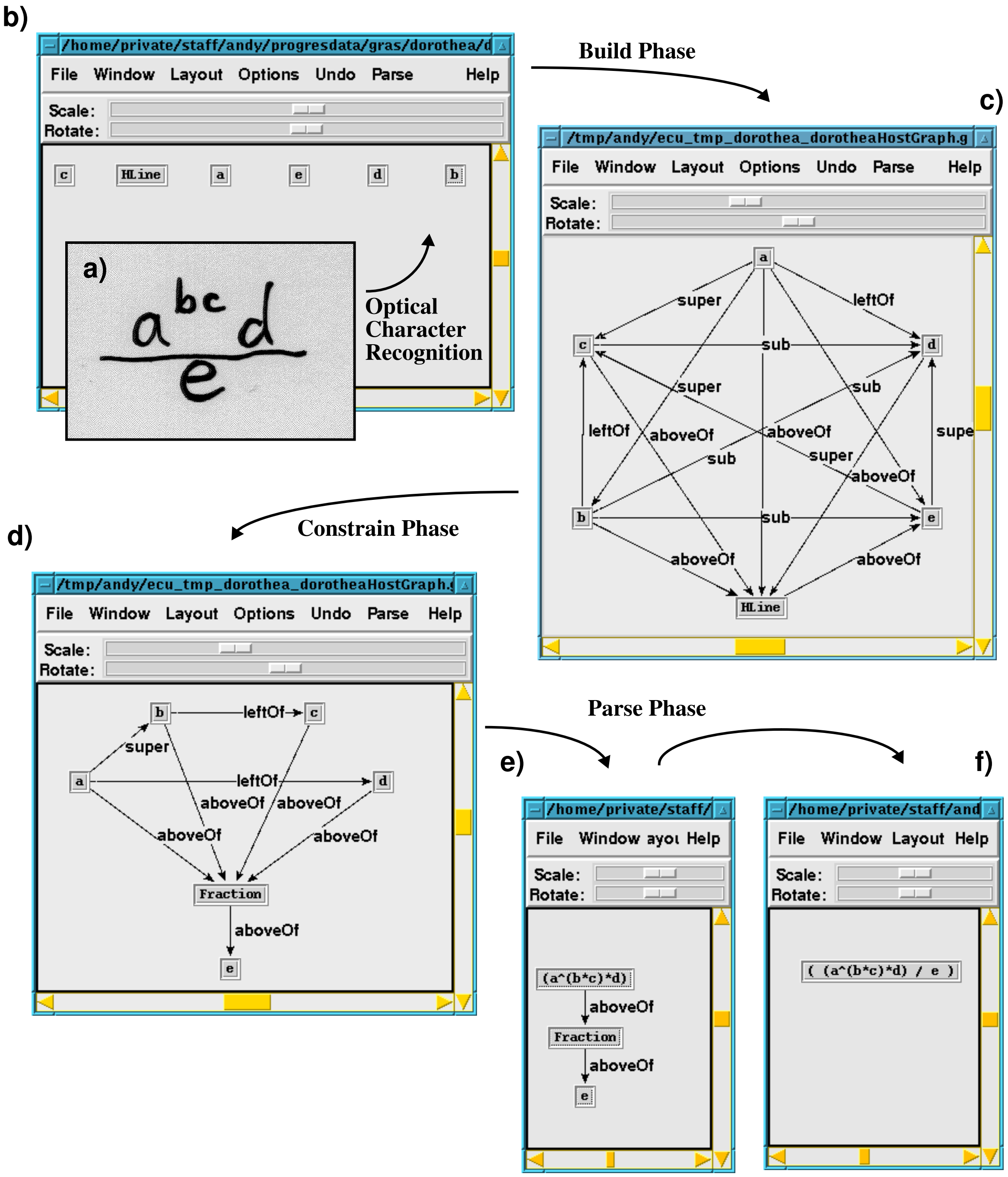


Figure 14 **Recognition of mathematics notation using graph rewriting. A document is scanned to produce image a). After character recognition, the initial graph b) is constructed. This graph contains no edges; each node represents one symbol, and has node attributes to record the (x, y) location of the symbol. Graph c) results from the** Build **phase, and contains edges for potentially-significant spatial relationships. This set of edges is pruned by the** Constrain **phase, resulting in graph d). For example, a** c sub d **edge is created by** Build **and removed by** Constrain. **The** Parse **phase identifies and collapses subgraphs that represent high-precedence operations, as shown in graph e). The final result after the** Parse **phase is the single node f), with a string attribute describing the meaning of the original image.**

Each of the recognition phases, Build, Constrain and Parse, is divided into subphases. This control information is coded as follows. This example uses & for sequential execution, and loop for repeated constructs. Other constructs include and (execute in non-deterministic order) and or (non-deterministically choose one of the actions).

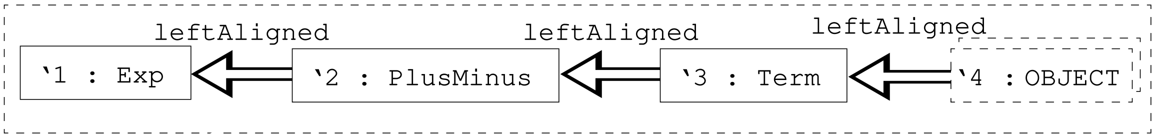
```
transaction ProcessInput =
    ReadScannerOutput    (* Figure 14: a → b                        *)
  & BuildPhase           (* Figure 14: b → c                        *)
  & ConstrainPhase       (* Figure 14: c → d                        *)
  & Parse                (* Figure 14: d → f                        *)
                         (* The Parse transaction is shown below.   *)
end;

transaction Parse =
   ReduceSymbols         (* letters, digits,... → terms *)
 & loop
      ReduceFactors      (* terms → factors *)
    & ReduceExpressions  (* factors → exprs. The ReduceExpressions *)
                         (* transaction is shown below.            *)
    & ReduceFractionLines(* Figure 14: e → f *)
    end
end;

transaction ReduceExpressions =
   ReduceSingleFactor
 & loop
     ReduceMultiply      (* Implied multiplication: A B *)
   end
 & ...
 & loop
     ReducePlusMinus     (* The ReducePlusMinus rule is shown in    *)
                         (* Figure 15.  It reduces expressions      *)
                         (* involving addition and subtraction.     *)
    end
end;
```

The graph rewrite rule in Figure 15 identifies plus and minus operators that are of sufficiently high precedence to be evaluated. The graph modification is quite simple: a spatially aligned sequence, consisting of an expression, addition operator, and term, is replaced by a single expression node. This is analogous to the parsing of a one-dimensional mathematics expression. Complications arise because of the need to ensure that the addition operator is not applied prematurely. In hand-written mathematics expressions, the following type of hasty symbol placement is common.

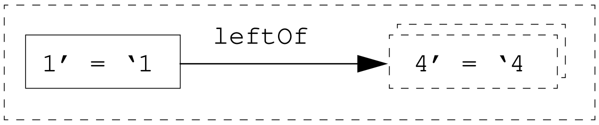
$$\frac{a}{b} + c$$

In this situation, the + operation should not be evaluated yet, even though a + c are in approximate horizontal alignment. However, were the division bar to extend past the c, then the + should be evaluated. The rule in Figure 15 distinguishes these two situations using the leftAligned path shown in Figure 16. This path tests whether two items (such as the a and +) are located spatially above the same set of horizontal bars. The leftAligned path is used by many of the production rules in the Parse phase.

```
production ReducePlusMinus =
```



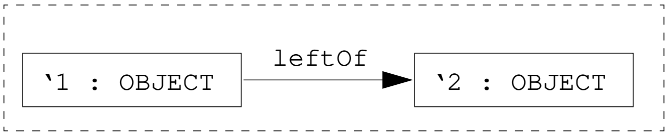

```
::=
```




```
transfer
   1′.meaning := "(" & `1.meaning & `2.meaning & `3.meaning & ")";
end;
```

Figure 15 This rewrite rule processes plus and minus configurations in mathematics notation. Three nodes, Exp, PlusMinus **and** Term, **are replaced by a single** Exp **node. The updated** meaning **attribute of** Exp **node 1' is constructed by concatenating the** meaning **values for node '1 (the first operand), node '2 (a** PlusMinus **node with meaning "+" or "-"), and node '3 (the second operand). The** leftAligned **path (Figure 16) tests that there is a** leftOf **edge between two nodes, and that the nodes are covered by the same set of division bars.**

```
path leftAligned : OBJECT → OBJECT =
  `2 => `1 in
```




```
  condition (`1.←aboveOf- <=> `2.←aboveOf-) and
            (`1.-aboveOf→ <=> `2.-aboveOf→) ;
end;
```

Figure 16 The definition of the leftAligned **path. The existence of the path can be tested between any two** OBJECT **nodes. (This means any two nodes can be tested, since** OBJECT **is the superclass of all node labels.) Node '2 is the origin of the path. The graphical part of the path definition states that there must be a** leftOf **edge between nodes '1 and '2. The first line of the** condition **clause states that the set of nodes above '1 must be the same as the set of nodes above '2. In this line, the notation** `` `1.←aboveOf- `` **describes a set of nodes reached by starting at '1 and following all incoming** aboveOf **edges. Next is the symbol** <=> **which is the equality operator between sets of nodes. The second line of the** condition **is similar, except here the outgoing** aboveOf **edges (notated as** `-aboveOf→`**) are tested.**

6. Summary

Graphs are a popular and versatile data structure in computer programs. Graph rewrite rules are a natural mechanism for expressing modifications to graphs. The visual notation used in graph rewriting languages is more convenient and less error prone than directly coding updates to an incidence matrix or edge list. Graph rewriting provides a useful way to think about a problem, to express the specification of a problem, and to create a prototype.

Graph rewriting has been used in many application areas. Nine applications of these are summarized in Table 1, and two are presented in greater detail. First, we describe an editor for precedence networks. Here, nodes in the HostGraph represent tasks and resources, and edges represent assignments of resources to tasks, as well as precedence constraints among tasks. Graph rewrite rules define the effect of various editor operations. The editor interface itself is automatically generated from this information. Second, we describe a recognizer for mathematical expressions. In this pattern-recognition application the initial HostGraph represents the spatial structure of the symbols in the image. The final HostGraph represents a logical interpretation of the notation found in the image. Computation proceeds via phases of rewrite-rules, with each phase performing some part of the recognition task. In conclusion, graph rewriting is a versatile style of computation, useful in application areas where graphs are a central data structure.

7. References

1. M. Andries, G. Engels, A. Habel, B. Hoffmann, H. Kreowski, S. Kuske, D. Plump, A. Schürr, G. Taentzer, "Graph Transformation for Specification and Programming," submitted to *Science of Computer Programming*, Elsevier Press, May 96.
2. F. Arefi, C. Hughes, D. Workman, "Automatically Generating Visual Syntax-Directed Editors," *Communications of the ACM*, Vol. 33, No. 3, March 1990, pp. 349-360.
3. K. Barthelmann, "Process Specification and Verification," *Proc. Fifth International Workshop on Graph Grammars and Their Application to Computer Science,* Lecture Notes in Computer Science, Vol. 1073, Springer Verlag, 1996, pp. 225-239.
4. T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, MIT Press, 1990.
5. M. Derk and L. DeBrunner, "Reconfiguration Graph Grammar for Massively Parallel, Fault Tolerant Computers," *Proc. Fifth International Workshop on Graph Grammars and Their Application to Computer Science,* Lecture Notes in Computer Science, Vol. 1073, Springer Verlag, 1996, pp. 185-195.
6. M. Nagl (ed.), *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, Lecture Notes in Computer Science, Vol. 1170, Springer Verlag, 1996.
7. H. Fahmy and D. Blostein, "A Graph Grammar Programming Style for Recognition of Music Notation," *Machine Vision and Applications*, Vol. 6, No. 2, 1993, pp. 83-99.
8. Five International Workshops on Graph Grammars and Their Application to Computer Science. Proceedings, Lecture Notes in Computer Science, Vol. 73 (1979), Vol. 153 (1983), Vol. 291 (1987), Vol. 532 (1991), Vol. 1073 (1996), Springer Verlag.
9. 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.
10. F. Gruau, "Automatic Definition of Modular Neural Networks," *Adaptive Behaviour*, Vol. 3, No. 2, 1995, pp. 151-183.
11. H. J. Kreowski, S. Kuske, and A. Schürr, "Nested Graph Transformation Units," *International Journal on Software and Knowledge Engineering,* special issue on Graph Grammar-based Specifications (to appear), World Scientific.
12. L. O'Gorman and R. Kasturi, *Document Image Analysis*, IEEE Computer Society Press, 1995.
13. S. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice Hall, 1987.
14. P. Prusinkiewicz and A. Lindenmayer, *The Algorithmic Beauty of Plant*s, Springer Verlag, 1990.
15. G. Rozenberg, *Handbook of Graph Grammars and Computing by Graph Transformation*, World Scientific, 1997.
16. R. Tamassia, G. Battista, and C. Batini, "Automatic Graph Drawing and Readability of Diagrams," *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 18, No. 1, January 1988, pp. 61-79.
17. Web page http://www-i3.informatik.rwth-aachen.de/private/andy/GraphRewriting/index.html