Chart Parsing of Attributed Structure-Sharing Flowgraphs with Tie-Point Relationships

Rudi Lutz

School of Cognitive and Computing Sciences, University of Sussex
Falmer, Brighton BN1 9QH, England
email: rudil@cogs.susx.ac.uk

Abstract

Many applications make use of diagrams to represent complex objects. In such applications it is often necessary to recognise how some diagram has been pieced together from other diagrams. Examples are electrical circuit analysis, and program understanding in the plan calculus (Rich, 1981). In these applications the recognition process can be formalised as flowgraph parsing, where a flowgraph is a special case of a plex (Feder 1971). Nodes in a flowgraph are connected to each other via intermediate points known as tie-points. Lutz (1986, 1989) generalised chart parsing of context-free string languages (Thompson — Ritchie, 1984) to context-free flowgraph languages, enabling bottom-up and top-down recognition of flowgraphs. However, there are various features of the plan calculus that complicate this - in particular attributes, structure sharing, and relationships between tie-points. This paper will present a chart parsing algorithm for analysing graphs with all these features, suitable for both program understanding and digital circuit analysis. For a *fixed* grammar, this algorithm runs in time polynomial in the number of tie-points in the input graph.

1 Introduction and Motivation

Many applications make use of diagrams to represent complex objects, and we often need to recognise how some diagram has been constructed. Examples are electrical circuit analysis, and program understanding in the plan calculus (Rich, 1981), in which programs are represented by data- and control- flow graphs, and stereotypical programming techniques and algorithms (plans) are represented similarly. Understanding how a program has been built up then amounts to treating plans as forming a grammar, and the understanding process as parsing. Ignoring control flow connections enables us to formalise this as flowgraph parsing. Nodes in a flowgraph consist of labelled boxes with distinguished input and output attaching points (ports), and input ports are connected to output ports via intermediate points known as tie-points, with the restriction that a port is only ever connected to a single tie-point, although fan-out and fan-in is allowed at tie-points. Lutz (1986, 1989) generalised chart parsing of context-free string languages (Thompson — Ritchie, 1984) to context-free flowgraph languages, enabling bottom-up and top-down recognition of flowgraphs. However, there are features of the plan calculus that complicate this:

1. Attributes — control flows in the plan calculus are treated as attributes of the grammar which are propagated during parsing.
2. Data Plans and Overlays — Some plans in the plan calculus allow the introduction of new tie-points not in the input graph. These tie-points either represent aggregate data structures corresponding to collections of other tie-points, or represent a more abstract view of some tie-point (e.g. viewing a list as implementing a set), and act as inputs or outputs of "higher-level" operations. Dealing with this involves using a

second chart storing information about data objects.

3. Structure Sharing — when one component feeds one or more of its outputs to more than one other component (fan-out). In this situation the source component can be viewed as playing more than one role in the structure, and could have been duplicated so that separate copies of the component were responsible for each of these roles. This leads to no change in functionality, although there may be a loss in efficiency as measured by the number of components (digital circuits), or computational effort and code size (plan calculus).

This paper will present a parsing algorithm for analysing graphs with these features, noting that we *permit* structure sharing, but do *not enforce* it. For a *fixed* grammar, this algorithm runs in time polynomial in the number of tie-points in the input graph. We will begin by discussing simple flowgraphs, and then progressively deal with the above features.

2 Notation and Definitions

Flowgraphs and *flow grammars* are special cases of plex languages and plex grammars (Feder, 1971). A *plex* consists of labelled nodes having an arbitrary number, n, of distinct *attaching points*, used to join nodes together. Such a node is called an n-attaching point entity (NAPE). Attaching points of NAPEs do not connect directly, but via intermediate points known as tie-points. A single tie-point may connect two or more attaching points. If the direction of the connections is important then the plex is said to be directed. Many types of graph structure (e.g. webs (Pfaltz — Rosenfeld, 1969; Rosenfeld — Milgram, 1972), directed graphs, and strings) are special cases of directed plexes. We will consider the special case of directed plexes in which each NAPE's attaching points (from now on called *ports*) are subdivided into two mutually exclusive groups, known as *input ports* (restricted to only have incoming connections) and *output ports* (restricted to only have outgoing connections). We further restrict ourselves to the case in which each port of a NAPE is only connected to a single tie-point. This type of plex will be called a *flowgraph* and is a generalisation of Brotsky's (1984) use of the term. See Figure 1 (top) for a simple example.

A production in a string grammar specifies how one string may be replaced by another. However, with flowgraph grammars we encounter a difficulty (due to their 2-dimensional nature) not apparent in the string case. In the string case a production

$$A ==> aXYb$$

applied to a string

$$\ldots dAe \ldots (say)$$

results in the string

$$\ldots daXYbe \ldots$$

and the question of how the replacement string is embedded in the host string never arises because there is a single obvious choice i.e. whatever is to the left of A in the original string is to the left of the replacing string, and similarly on the right. With flowgraphs we no longer have this simple ordering on the NAPEs and embedding becomes much more complicated. Most of the discussion of this is in the web and graph grammar literature (e.g. (Pfaltz — Rosenfeld, 1969; Rosenfeld — Milgram, 1972)), but most of it also applies to flowgraphs. Our approach is to specify with each production which tie-points on the left correspond to which tie-points on the right and then connect everything connecting to one of these left hand tie-points (from the surrounding subgraph) to its corresponding right-hand tie-point.

We define a flowgraph grammar to be a 4-tuple (N,T,P,S) where:

N is a finite non-empty set of NAPEs known as nonterminals.

T is a finite non-empty set of NAPEs known as terminals.

P is a finite set of productions.

S is a special member of N known as the initial, or start, NAPE

where the intersection of N and T is empty.

If we arbitrarily order the input and output ports of a NAPE then each NAPE in a flowgraph can be represented as a triple

$$(NAPE - label, inputlist, outputlist)$$

where NAPE-label is the label on the NAPE, and input list is a list in which the i^{th} entry is the tie-point to which the i^{th} input port is connected. Similarly the output list specifies to which tie-point each of the output ports is connected. Using this convention a flowgraph G can be represented as a set G^c (the component set) of such triples.

With the above conventions the productions in a flowgraph grammar have the general form

$$AL_iL_o ==> CR_iR_o$$

where

- A is known as the left-side structure, represented as a component set
- C is known as the right-side structure, represented as a component set
- L_i is the left-side input tie-point list
- R_i is the right-side input tie-point list
- L_o is the left-side output tie-point list, and
- R_o is the right-side output tie-point list.

L_i and R_i must be of the same length, as must L_o and R_o , and specify how an instance of the right-side structure is to be embedded into a structure W containing an instance of the left-side structure which is being rewritten according to the production. We define the *arity* of the left side of the rule to be the ordered pair $(|L_i|, |L_o|)$ and the arity of the right side of the rule to be the ordered pair $(|R_i|, |R_o|)$. So this requirement simply states that the left- and right-side arities must be the same. The rewriting and embedding is done as follows:

The instance of the left-side structure is removed from W and replaced by an instance of the right-side structure. Now, for each tie-point X in L_i any previous connections from NAPEs in W to X are replaced by connections from the same attaching points of the same NAPEs to the corresponding tie-point in R_i . The same is done for tie-points in L_o and R_o. One can eliminate the need for explicit storing of R_i and R_o by using the same variable names on the left and right hand sides of the production to denote corresponding tie-points.

Just as in the string case, by considering restrictions on X and Y in a production of the form:

$$X ==> Y$$

one can arrive at the notions of context-sensitive, context-free, and regular grammars (Ehrig, 1979). In particular, restricting the productions to have a single NAPE in their left-side structure gives us a context-free flowgraph grammar, and we will restrict ourselves to these from now on. In this case we no longer need to store L_i and L_o since the input and output lists of the single triple on the left of the production already specify this information. See Figure 1 for an example of the notation and of rewriting process.

3 Chart Parsing of Context-free Flowgraphs

In a chart parser, assertions about what has been found by the parser are kept in a "database" known as the *chart*. Such assertions will be called *patches*, and are of two kinds — *complete* patches and *partial* patches. A complete patch asserts that a complete grammatical entity (corresponding to some terminal or non-terminal symbol of the grammar) has been found. Partial patches are assertions that part of some grammatical entity has been found, and about what needs to be found to complete it. One can think of a patch as being a closed loop drawn round some subgraph of the flowgraph, indicating that this subgraph corresponds to all or part of some grammatical entity. Regarding the right-side structures of rules as uninstantiated templates, then complete patches with non-terminal labels correspond to the occurrence of an instantiation of the right-side structure of some rule, thus forming an occurrence of the left-side structure of the rule. Partial patches correspond to partially instantiated instances of the right-side structure of some rule, and thus to partially recognised instances of the left-side structure. Each patch A contains the following information:

1. *label(A)* — the name (one of the terminal or non-terminal symbols) of the grammatical entity corresponding to the patch.

2. *inputs(A)* — a set of input tie-points for the patch.

3. *outputs(A)* — a set of output tie-points for the patch.

4. *components(A)* — a list of the patches involved in making up this patch.

5. *needed(A)* — what else needs to be found to complete the patch. For complete patches this will be empty, and for partial patches this will be a flowgraph structure, represented as a list of triples.

For a partial patch, the input and output tie-points (i.e. those by which the patch connects to the surrounding flowgraph) are each subdivided into two categories — the set of *active* tie-points where the patch still needs other components to attach to these tie-points, and the set of *inactive* tie-points which are those which would be inputs or outputs of the patch were it complete. A NAPE needed by a partial patch will be called *immediately needed* if any of its tie-points are active. The components entry of a patch lists (instantiated versions of) those NAPEs in the right-side structure of the rule which have been completely instantiated, and the needed entry lists uninstantiated (as yet) parts of the rule. Note that some of the *tie-points* in the *needed entry* may be instantiated. These are where the needed NAPEs connect to the ones already found. We will say that a partial patch A is *extendible* by a complete patch B (or that B *can extend* A) in the case where A immediately needs a patch of the same type as B and the instantiated tie-points in this needed patch do not conflict with any instantiations actually occurring in B.

The essence of the chart parsing strategy can then be stated as follows:

Every time a complete patch is added to the chart a search is made for any partial patches immediately needing a patch like the one just added. For each of these partial patches a *new* patch is made extending it by the complete one, and this new patch is then added to an agenda of patches to be processed at some appropriate time. Similarly, every time a partial patch is added to the chart a search is made for complete patches which can extend the partial patch just added, and if any are found new patches are made extending the partial one, and these are added to the agenda to be processed when appropriate. Note that patches are only ever added to the chart. They are never removed, thus avoiding duplication of previous effort.

The basic operation of the algorithm is joining a complete patch to a partial patch to make a new enlarged patch. Figure 2 shows a partial patch being joined to a complete patch to make a new patch (the enclosing box). The resulting patch has the same items in its *components entry* as the original partial patch plus the complete patch. Its *needed entry* is equal to that of the original partial patch minus the needed patch corresponding to the complete patch. Note that the matching of a needed patch to an actual complete patch may introduce further instantiations of tie-points in the *needed entry* of the new patch. On connecting the two patches all the inactive tie-points of the partial patch remain inactive. Some of its active tie-points will correspond to tie-points of the complete patch (this is where the two patches actually join). Other active tie-points remain active in the new patch since it is still looking for other patches to attach to them. Of the complete patch's (input and output) tie-points some have already been mentioned i.e. those connecting directly to the partial patch. Others will become new inactive tie-points of the resulting patch since it will not be looking for anything to attach to them. However other (input and output) tie-points of the complete patch may now become active (viewed as belonging to the new patch) since it may now expect other patches to attach to them in order to complete itself. Provided all these distinctions are kept clear there is no great difficulty in implementing the joining operation.

With this joining operation a limited type of structure sharing happens automatically. This is illustrated in Figure 3. If we wish to prevent this, then, when trying to extend a partial patch P by a complete patch C, the parser must check (recursively!) that none of the components of P have any sub-components in common with C, thus preventing structure sharing at any level. This check will be referred to as the *no-sharing check.*

The initialisation of the agenda will now be described. Initially a complete patch is added to the agenda for each of the terminal NAPEs in the original graph. If the algorithm is to run top-down then an additional step is needed in which partial patches with empty *components entries*

are made for every rule in the grammar whose left-side structure is labelled by the start symbol of the grammar. Each such rule leads to several such empty patches, one for each permutation of the input tie- points of the input graph. The *inactive-inputs* and *active-outputs* entries for these patches are the permuted inputs. The *needed entry* is the right-side structure of the rule with appropriate instantiations of the tie points occurring in it. These patches are also added to the agenda. The complete algorithm is shown below:

```
initialise chart and agenda;
until the agenda is empty do
    pick a patch A from the agenda;
    unless A is already in the chart then
        add A to the chart;
        if A is complete then
        for each partial patch B in chart extendible by A do
                make a new patch extending B with A and put on agenda;
            endfor;
            if bottom-up then
                for each rule R in P such that rhs(R) has an input NAPE  labelled by
                                        label(A) do
                  for each such NAPE X in R do
                    make new empty patch B with label(B)=lhs(R) and
                        needed(B)=rhs(R) with instantiations dependent on match between
                                              X and A and
                        inputs(B)=inputs(A) and
                        active-outputs(B)=inputs(A);
                     add B to agenda;
                  endfor;
                endfor;
            endif;
        else
            for each complete patch B in chart which can extend A do
                make a new patch extending A with B and put on agenda;
            endfor;
            if top-down then
                for each object C immediately needed by A do
                    for each rule R in P with lhs(R)=label(C) do
                          make new empty patch B with label(B)=label(C) and
                              needed(B)=rhs(R) with instantiations dependent on match
                                        between C and lhs(R) and
                              inputs(B)=inputs(C) and
                              active-outputs(B)=inputs(C);
                          add B to agenda;
                    endfor
                endfor
            endif
        endif
     endunless
enduntil;
```

$$A_{N+1} \begin{cases} = & A_N - 1 \quad \textit{if patch is already present in the chart} \\ \leq & A_N - 1 + AQT^{K+M-1} + QR \quad \textit{if patch partial and not in chart} \\ \leq & A_N - 1 + (K+M) \cdot R2^Q T^{K+M+A-1} \quad \textit{if patch complete and not in chart} \end{cases}$$

$$C_{N+1} \begin{cases} = & C_N \quad \textit{if patch chosen is already present in the chart} \\ = & C_N \quad \textit{if patch chosen is partial and not in the chart} \\ = & C_N + 1 \quad \textit{if patch chosen is complete and not in the chart} \end{cases}$$

$$P_{N+1} \begin{cases} = & P_N \quad \textit{if patch chosen is already present in the chart} \\ = & P_N + 1 \quad \textit{if patch chosen is partial and not in the chart} \\ = & P_N \quad \textit{if patch chosen is complete and not in the chart} \end{cases}$$

Figure 14: The top-down case, the $N+1^{th}$ iteration

On termination the parsing is successful if the chart contains a complete patch for S whose *inputs* and *outputs* entries are the input and output tie-points of the input graph.

How can we organise the chart for efficient searching? The chart is divided into two parts, one for complete patches, and one for partial. The part for complete patches is organised as two arrays, one for indexing each patch by its inputs, and one for indexing by its outputs. So each complete patch is entered several times into the chart, once for each of its inputs and outputs. For further efficiency each of the elements in these arrays is a hash table and the patches are actually entered into these hashed by their label. So the entries in the hash table are actually lists of patches with the same label which share a given input or output tie-point. This enables efficient retrieval of all patches with a particular label at a particular tie-point. The treatment of partial patches is slightly more complicated. For each of their immediately needed NAPEs partial patches are entered into their part of the chart indexed by the active inputs and outputs of the needed NAPE, and hashed by the labels of each of these NAPEs. This structure for the chart enables a complete patch to easily find partial patches immediately needing it, and enables partial patches to easily find complete patches that they immediately need.

A similar technique can be used to store the grammar rules in order to enable efficient retrieval of appropriate rules.

4 Complexity Analysis

4.1 A Polynomial Bound

In this section a relatively informal argument will be given to show that, for a *fixed grammar*, the algorithm runs in time polynomial in the number of tie-points T of the input graph (if the grammar is allowed to vary and is therefore regarded as part of the input to the parsing problem, then Wills (1992) has shown that the problem becomes NP-complete). We will not give a tight upper bound on the running time, but simply show that it is polynomial. Let:

G =number of NAPEs in the graph

T =number of tie-points in the graph

K =maximum number of inputs to a NAPE

M =maximum number of outputs from a NAPE

L =number of possible labels

R =number of rules in the grammar

Q =maximum number of NAPEs in the right-side structure of a rule

A =maximum possible number of active tie-points in a partial patch.

Note that K, M, L, R, Q, and A all depend on the grammar, and are independent of the input graph.

$$A_{N+1} \begin{cases} = & A_N - 1 \quad \textit{if patch chosen is already present in the chart} \\ \leq & A_N - 1 + A \cdot Q \cdot T^{K+M-1} \quad \textit{if patch partial and not in chart} \\ \leq & A_N - 1 + (K+M) \cdot R2^Q T^{K+M+A-1} + QR \quad \textit{if patch complete and not in chart} \end{cases}$$

$$C_{N+1} \begin{cases} = & C_N \quad \textit{if patch chosen is already present in the chart} \\ = & C_N \quad \textit{if patch chosen is partial and not in the chart} \\ = & C_N + 1 \quad \textit{if patch chosen is complete and not in the chart} \end{cases}$$

$$P_{N+1} \begin{cases} = & P_N \quad \textit{if patch chosen is already present in the chart} \\ = & P_N + 1 \quad \textit{if patch chosen is partial and not in the chart} \\ = & P_N \quad \textit{if patch chosen is complete and not in the chart} \end{cases}$$

Figure 15: The bottom-up case, the $N+1^{th}$ iteration

For the purposes of adding new patches to the chart, patches are only distinguished according to some of the information contained in them, rather than strict equality being necessary. Complete and partial patches will be dealt with separately.

Complete patches are distinguished which differ in at least one of their input tie-points, their output tie-points, or their label. The maximum number of inputs and outputs in a patch is determined by the grammar, as is the number of possible labels. So the number of possible complete patches in the chart is bounded above by the product of L and the number of possible ways of selecting at most K out of T tie-points, and the number of possible ways of choosing at most M out of T tie-points. This gives us $O(L \cdot T^{K+M})$ complete patches altogether. A similar argument shows that at a given set of K (input) tie-points, there are *at most* $O(T^M)$ complete patches with a given label.

Partial patches are distinguished which differ in at least one of their inactive input tie-points, their inactive output tie-points, their label, or in what they need in order to complete themselves (their *needed* entry). Now, a partial patch represents the partially recognised right side structure of a rule. The rule used determines the label, and there are at most 2^Q subsets of the (at most) Q NAPEs in the rule that could still be needed. Each such subset determines a set of (at most) A active tie-points for the patch. So there can be at most

$$O(R \cdot 2^Q \cdot T^A \cdot T^K \cdot T^M) = O(R \cdot 2^Q \cdot T^{A+K+M})$$

partial patches altogether. In fact there will be very much less than this, as this includes complete patches with nothing needed, and (more importantly) ignores completely any additional constraints implied by the connectivity of the graph.

Since the basic operation of the chart parsing algorithm involves extending partial patches by complete ones, we need to know, for a given partial patch, the largest number of complete patches that could possibly extend it. A partial patch can be extended at any of its (at most) A active tie-points, and any complete patch which could extend it must join at least one of these tie-points, and must share a label with at least one of the NAPEs immediately required by the partial patch. So there are *at most* $O(A \cdot Q \cdot T^{K+M-1})$ such complete patches. Similarly, given a complete patch, there are at most $O((K+M) \cdot R \cdot 2^Q \cdot T^{K+M+A-1})$ possible matching partial patches.

We can use these upper bounds to demonstrate that the algorithm terminates, and does so in polynomial time. Let N denote the number of iterations of the main loop of the algorithm while it is running, and let:

C_N =number of complete patches in the chart after iteration N

P_N =number of partial patches in the chart after iteration N

A_N =length of agenda after iteration N

Then in the top-down case we have:[1]

$$\begin{aligned} A_0 &= G + R_S \cdot P_G \\ C_0 &= 0 \\ P_0 &= 0 \end{aligned}$$

The equation for the $(N+1)^{th}$ iteration is in Figure 14. In the bottom-up case we have

$$\begin{aligned} A_0 &= G \\ C_0 &= 0 \\ P_0 &= 0 \end{aligned}$$

and for the $(N+1)^{th}$ iteration see Figure 15.

So, in both the bottom-up and top-down cases, C_N and P_N are monotonic functions of N. As discussed earlier both are bounded above. Therefore after some number of iterations they must both have reached their maximum value (normally *much less* than the crude estimates above). Once this happens all patches on the agenda must be already present in the chart and A_N decreases by one on each subsequent iteration until it reaches 0 (an empty agenda), and the algorithm terminates. Now on each iteration it can be seen that either:

1. both C_N and P_N remain constant (in which case A_N decreases), or
2. P_N increases by 1, and items are possibly added to the agenda, or
3. C_N increases by 1, and items are possibly added to the agenda.

From the above, at most $O(L \cdot T^{K+M})$ iterations involve adding a complete patch to the chart and add some items to the agenda, and at most $O(R \cdot 2^Q \cdot T^{A+K+M})$ iterations involve adding a partial patch to the chart and add some items to the agenda. All other iterations simply remove items from the agenda. So how many items get added to the agenda?

This is given by:

(no. of items in initial agenda)
+(no. added for complete patches)
+(no. added for partial patches)

In the top down case this bounded by

$$A0 + O(L \cdot T^{K+M}) \cdot O((K+M) \cdot R \cdot 2^Q \cdot T^{K+M+A-1}) + O(R \cdot 2^Q \cdot T^{A+K+M}) \cdot (O(A \cdot Q \cdot T^{K+M-1})) + Q \cdot R$$

and in the bottom-up case this is bounded by

$$A0 + O(L \cdot T^{K+M}) \cdot (O((K+M) \cdot R \cdot 2^Q \cdot T^{K+M+A-1}) + Q \cdot R) + O(R \cdot 2^Q \cdot T^{A+K+M}) \cdot O(A \cdot Q \cdot T^{K+M-1})$$

which are both clearly polynomial.

So, in both cases the number of items added to the agenda, which is the same as the number of iterations performed, is polynomially bounded. How much work is done on each of these iterations? The cost of seeing if a patch is already in the chart can be done in polynomial time. This is because (even with no clever indexing) there are at most a polynomial number of patches in the chart that need to be checked. If the no-sharing check is included then the cost of checking if one patch is extendible by another can be done in time at worst O(G) since both the partial and complete patches are each ultimately made up of at most G NAPEs (at lowest level), and checking for intersection of these two sets can be done in linear time. If the no-sharing check is omitted then the cost of checking if one patch is extendible by another can be done in constant time (since it depends on checking that the instantiated tie-points of the patches are compatible with each other, and the number of tie-points involved depends on the grammar), as can the cost of making a new patch. All the costs involved in checking rules etc. are purely a function of the grammar. So the total cost of the algorithm is easily seen to have an upper bound which is a polynomial function of T.

4.2 Finding All Parses

Athough the algorithm performs flowgraph *recognition* in polynomial time, it does not find *all parses* in polynomial time. This is because for some flowgraphs and some grammars there may well be an exponential number of parses (this is even true of Earley's algorithm operating on strings!). The algorithm will however find a parse if one exists. If an application requires all possible parses, then the algorithm can be modified to

[1] R_S is the number of rules for S (the start symbol); P_G is the number of permutations of inputs of graph.

store any *complete* patch which is equal to one already in the chart in terms of its inputs, outputs, and label, but *not* equal in terms of its *components*, in an auxiliary data structure. At the end of the parsing there is then enough information around in the chart and the auxiliary data structure to easily compute additional parses, by simply adding all the patches in the auxiliary structure to the agenda, and letting the parsing continue with the test for equality of patches now being strict equality (i.e. all the components must be equal as well) rather than just the partial equality used earlier.

5 Dealing With Attributes

As stated earlier graphs and rules in the plan calculus also have a second type of connection between NAPEs - control flow arcs. These are handled as attributes of the graphs, where the attribute for a non-terminal NAPE is calculated from the attributes of its components. Details of this method of handling the control flows can be found in Wills (1986, 1990, 1992), and a generalisation can be found in Lutz (1992). For our purposes we will assume that each NAPE in the original graph is annotated with initial values for the attributes, and we will also assume that each rule has annotations describing how each attribute for the left-hand side of the rule is computed from the attributes of the NAPEs on its right-hand side. These annotations have the general form:

$$A_{lhs} = f_{Rule}(A_1, \ldots, A_k)$$

where A_{lhs} represents an attribute of the left hand side of the rule, f_{Rule} represents the rule specific computation, and $A_1, \ldots, A_k$ represent the attributes of the NAPEs on the right.

Computing the attributes is straightforward. Each patch is given an extra field for each of its attributes. When a *complete* patch (corresponding to some rule of the grammar) whose components have attributes $A_1, \ldots, A_k$, is added to the chart, $f_{Rule}(A1, \ldots, Ak)$ is computed, and stored in the appropriate field for the attribute in the patch. The initial patches receive their attribute values from the original graph.

6 Dealing With Tie-Point Relationships

In order to capture implementation decisions, and data abstractions, the plan calculus contains what Rich (1981) calls data plans and data overlays. So far as the grammatical formalism is concerned, these can be viewed as allowing rules to express named functional relationships that hold between tie-points. To handle these our grammatical formalism is extended to allow annotations (following the keyword *where*) of the form:

$$t_i = F(t_{j_1}, \ldots, t_{j_k}) \quad for\ k \geq 1$$

where t_i represents either any of the tie-points occurring in the NAPEs of the rule, or an additional new tie-point, $t_{j_1}, \ldots, t_{j_k}$ represent any of the tie-points occurring in the NAPEs of the rule or any new tie-points mentioned on the left of other relationships in the rule (this must be non-recursive!), and F is the name of the functional relationship involved. The set of these will be referred to as the *tie-point relationships* of the rule.

We will only discuss the changes to deal with tie-point relationships for the parser running in bottom-up mode. Dealing with them in top-down mode is rather complicated and will not be described further in this paper.

Conside the rules *bump+update* and *bump+update->push* (Figures 4 and 5), which cause problems for the algorithm. Flowgraph grammar rules as described earlier have the same arities for their left- and right-hand sides, and this is true for *bump+update*. However, the left-hand side of *bump+update->push* has arity (2,1), while the right hand side has arity (3,2). Furthermore, although the tie-point t3 occurs as input on both sides of the overlay, this is not true of t6 and any tie-point of the *bump+update* plan. It does not even correspond to the compound object (the *upper-segment*) represented by t1 and t2. It corresponds to the *upper-segment* viewed as a *list* (via a function *upper-segment->list*). To cope with these features the basic bottom-up chart parser presented earlier is modified as follows:

1. The rule format is modified to include the left-hand side inputs and outputs, since these may now be distinct from those on the right. Correspondingly, each patch now

has two extra fields — *left-hand-ins*, and *left-hand-outs*, in addition to the two fields *inputs* and *outputs* (corresponding to inputs and outputs of the right hand side of the rule). Complete patches are stored in the chart indexed by their *left-hand-ins* and *left-hand-outs*, rather than by their inputs and outputs as before. Partial patches are stored as before.

2. A second chart is added. This chart (the *tie-point chart*) stores the functional relationships between tie-points discovered during parsing. It contains entries with the form:

$$T = F(S_1, \ldots, S_k)$$

where T and $S_1, \ldots, S_k$ are known (i.e. instantiated) tie-points. This chart is organised similarly to the earlier (complete) chart, in that it is split into two parts, one used for storing relationships indexed by the left hand side tie-point (T) and by the relationship name (F), and the other used for storing the relationships indexed by the right-hand side tie-points ($S_1, \ldots, S_k$), and by the relationship name.

3. Two new fields are added to each patch. These are:

   (a) *relations-needed* — When an empty patch is created this is initialised to the set of tie-point relationships of the rule involved in creating the patch.

   (b) *relations-found* — When an empty patch is created this is initialised to empty.

4. When a new patch is created, either by extending a partial patch, or by creating a new empty patch for some rule, any instantiations for the tie-point variables occurring in the patch are also propagated into the relations-needed entry. The following is then repeated until there is no change to the patch:

   (a) If some relationship in the relations-needed entry is fully instantiated (i.e. no tie-point *variables* occur on either its left or right hand sides) then it is moved from the relations-needed entry to relations-found.

   (b) If some relationship in the relations-needed entry has a fully instantiated right-hand side i.e. is of the form:

$$V = F(S_1, \ldots, S_k)$$

   where V is a tie-point variable, and $S_1, \ldots, S_k$ are all known tie- points, then the tie-point chart is consulted to see if there is an entry of the form:

$$T = F(S_1, \ldots, S_k)$$

   where T must be a known tie-point. If there is, then V is instantiated to T, and this instantiation is propagated throughout the patch (including its relations-needed entry). If not, then a *new* tie-point T is *created*, the assertion:

$$T = F(S_1, \ldots, S_k)$$

   is added to the tie-point chart, V is instantiated to T, and this instantiation is propagated throughout the patch (including its relations-needed entry).

   (c) If some relationship in the relations-needed entry has an instantiated left-hand side i.e. is of the form:

$$S = F(T_1, \ldots, T_k)$$

   where S is a known tie-point, and $T_1, \ldots, T_k$ are *either* known tie-points *or* tie-point variables, then the tie-point chart is consulted to see if there is an entry of the form:

$$S = F(S_1, \ldots, S_k)$$

   where $S_1, \ldots, S_k$ are all known tie-points. Matching $T_1, \ldots, T_k$ against $S_1, \ldots, S_k$ either succeeds, in which case any variables in $T_1, \ldots, T_k$ get instantiated, and these instantiations are propagated throughout the patch. If the match fails (because two *different* known tie- points are being matched against each other) then the whole patch is invalid, and is rejected (i.e. is not added to the chart).

5. A patch is only considered complete if *both* its needed entry *and* its relations-needed entry are empty. If they are, then the patch is added to the chart as normal. If not, then the patch is considered partial, and is stored in the chart indexed as before, but also indexed by the relationship names and instantiated tie-points of any immediately needed (in the obvious generalised sense of the term) relationships in the relations-needed entry.

6. When a relationship is added to the tie-point chart, the (patch) chart is consulted to see if there are any partial patches waiting for a tie- point relationship compatible with the one just added. If so, the patch is extended by the relationship, and added to the agenda.

Figure 6 illustrates this for the above rules.

Now consider rules like those in Figure 7, which includes a rule (for A) with a "straight-through" arc, and a graph like that in Figure 8. This can be recognised as forming an S, by the following sequence of events:

1. The NAPEs labelled b and c in Figure 8 are recognised as forming a partial A, which still has:

$$t1 = iterator(1, t4)$$

$$t2 = iterator(3, t4)$$

in its relations-needed entry.

2. The NAPEs labelled d and e in Figure 8 are recognised as forming a complete B, with input given by a em new tie-point 8, and output given by a *new* tie-point 9, satisfying:

$$8 = iterator(3, 4)$$

$$9 = iterator(6, 7)$$

These relationships are added to the tie-point chart.

3. When the relationship 8=iterator(3, 4) is added to the tie- point chart, the main chart is consulted to see if there are any partial patches immediately needing a relationship matching this one. The partial A discovered earlier is found, and on matching

$$8 = iterator(3, 4)$$

and

$$t2 = iterator(3, t4)$$

t4 gets instantiated to 4, and t2 gets instantiated to 8. The patch is therefore extended, and it now has only the single relationship:

$$t1 = iterator(1, 4)$$

in its relations-needed entry. This causes the creation of a new tie-point 10 to which t1 is instantiated, and an assertion:

$$10 = iterator(1, 4)$$

is added to the tie-point chart. As a result of all this we now have a complete A patch (with input 10 and output 8) which gets added to the chart. This causes the creation of an empty S patch (with input 10) immediately needing an A (with input 10) to be added to the chart.

4. This patch is extended first by the A patch, and then again by the B patch, giving us a complete S patch with input 10 and output 9, where:

$$10 = iterator(1, 4)$$

and

$$9 = iterator(6, 7)$$

This illustrates very nicely the role of the second chart.

7 Chart Parsing of Structure-Sharing Flowgraphs

As stated earlier we are also interested in the case where structure sharing is allowed. However, for reasons discussed in Lutz (1992), we do not want to allow *any* two NAPEs sharing the same inputs to be collapsed, but only NAPEs with appropriate labels. To make this more precise we define a slightly more general notion:

A *restricted structure sharing flowgraph grammar* (RSSFG) is a 5-tuple (N, T, P, S, R) where N, T, P, S are the same as for ordinary context

free flowgraph grammars, and $R \subseteq N \cup T$. R is the set of NAPEs for which collapsing is allowed. Such a grammar has an additional rewriting rule which will be described below. We define a relation *R-collapses* on the set of flowgraphs over $N \cup T$ by:

> G2 *R-collapses* G1 iff G1 and G2 are flowgraphs, and $G1^c$ contains two triples of the form $T1 = (A,(t_1,\ldots,t_n),(x_1,\ldots,x_m))$ and $T2 = (A,(t_1,\ldots,t_n),(y_1,\ldots,y_m))$, where A $\in R$ and $G2^c$ can be obtained from $G1^c$ by removing these two triples and replacing them by a single triple of the form $T3 = (A,(t_1,\ldots,t_n),(z_1,\ldots,z_m))$ and then replacing all occurrences of $x_1,\ldots,x_m$ and $y_1,\ldots,y_m$ by $z_1,\ldots,z_m$ respectively throughout the remaining triples.
>
> In other words, G2 *R-collapses* G1 iff G1 contains two instances of some NAPE A (whose label is in R) which have the same inputs, and G2 is identical to G1 except that the two instances of A have been replaced by a single instance of A (with the same inputs) and all NAPES which originally connected to the outputs of one or other of the two instances of A now connect to the single instance (in G2). This amounts to identifying the two instances of A and their corresponding tie-points.

The reflexive, transitive, symmetric closure of *R-collapses* is then an equivalence relation (*R*-share-equivalence) on the set of flowgraphs, and we want any parsing algorithm which can recognise some graph G to also be able to recognise any flowgraphs R-share-equivalent to G. We also want the grammar to be able to generate not only the flowgraphs derivable directly from the grammar, but also all R-share-equivalent flowgraphs. This can be done if we allow at any point in the generation of a flowgraph the replacement of the graph so far generated (G1) by any graph G2 for which *either* G1 *R-collapses* G2 or G2 *R-collapses* G1. If $R = \emptyset$ then the grammar is an ordinary flowgraph grammar, and if $R = N \cup T$ then we get a (full) structure sharing flowgraph grammar as defined in Lutz (1989). Figure 9 illustrates several phenomena that can occur with RSSFGs, and which motivated the above definition.

To see how the parsing algorithm can be modified to cope with RSSFGs it should first be noted that for any flowgraph G there is a *smallest* flowgraph G_{min} which is R-share-equivalent to G. Secondly, the right-side structure of any rule in an RSSFG can be replaced by any flowgraph R-share-equivalent to it without altering the generative capacity of the grammar. We therefore define a canonical form for an RSSFG in which each rule of the form:

$$A ==> B$$

has been replaced by the rule:

$$A ==> B_{min}$$

So the first change to the algorithm is to put the grammar into canonical form. The second change is to the action of adding a complete patch to the chart. Previously the only check that was done was to see if the patch was already in the chart. Now the algorithm must additionally check if the label of the patch is in R and if there is another patch in the chart with the same label and the same inputs. If so, the algorithm must collapse the new patch and the one already present into a single patch, by identifying the output tie-points of the two patches. Provided tie- points in the various triples making up the patches are represented as pointers to pointers to tie-points (rather than storing the tie-points directly in the triples) then simply changing the values of the second set of pointers will implement the identification universally throughout all patches in the chart. This can lead to "chains" of pointers which need to be fully dereferenced in order to actually access the tie-points themselves (this is similar to the way variables are handled in many implementations of Prolog). After collapsing an additional step is needed since there may be patches in the chart indexed by the tie-point which has been effectively removed by the identification. These patches must now be stored in the chart indexed by their new output tie-points. If the information that collapsing has been done is needed by an application the algorithm can make a note this fact either by annotating the tie- points involved

or by an assertion held separately. Finally, the no-sharing check must be omitted.

If the grammar also has attributes, then we need to specify how to compute the attribute A_{res} of a NAPE resulting from collapsing two patches with attributes A_1 and A_2. This specification takes the form (for each attribute):

$$A_{res} = f_{collapse}(A_1, A_2)$$

where $f_{collapse}$ is a function which computes the value of the attribute for the new patch from the values for the two collapsed patches. Two NAPEs are only collapsed when an attempt is made to add a complete patch P_2 (with attribute A_2) to the chart, and there is already a patch P_1 (with attribute A_1) present in the chart with the same label and inputs. In this case P_1 is left in the chart (i.e. it is the output tie-points in P_2 which are identified with those in P_1). P_1 then has the value of its attribute set to $f_{collapse}(A_1, A_2)$. If this new value for its attribute is different from its previous value, then any complete patches in the chart which have P_1 as one of their components must also have their attributes recalculated, and any of these patches whose attributes change must also have their attributes recalculated, and so on recursively. To facilitate this, each complete patch P needs an extra field *partof* which holds a list of all complete patches of which P is a component. To maintain this field, whenever a complete patch P is added to the chart, P is added to the *partof* field of each of its component patches. The initial patches (corresponding to the original graph) all have this field initially set to empty.

8 Applications

The algorithm just described forms the basis of the program understanding process described in Lutz (1989b, 1991, 1992). However, there are other domains, in particular digital circuit analysis, in which a similar ability to parse flowgraphs is useful. Consider Figure 10 which shows a circuit for addition of 3-bit numbers. The grammar shown in Figure 11 is capable of generating this circuit. Adding a rule like that shown in Figure 12 enables the parser to recognise the circuit in Figure 10 as being equivalent to Figure 13 i.e. to recognise the circuit as adding two numbers, with the tie-point chart holding information about how the numbers have been "implemented".

9 Conclusions

This paper has presented a polynomial time chart parsing algorithm for context-free flowgraph languages, capable of handling all the features of the plan calculus (Rich, 1981), and which is also applicable to digital circuit analysis.

Although there is a large literature on the generative abilities of various types of graph grammar formalisms (see e.g. (Ehrig, 1979; Feder, 1971; Fu, 1974; Gonzalez — Thomason, 1978; Pfaltz — Rosenfeld, 1969; Rosenfeld — Milgram, 1972)), there is relatively little on parsing strategies, except for restricted classes of graph and web grammars (e.g. Della Vigna — Ghezzi (1978)). In its top-down strictly left-to-right form chart parsing of context-free string languages corresponds to Earley's algorithm (Earley, 1970), which was generalised by Brotsky (1984) to parsing flowgraphs of the kind described here, except that his algorithm could not cope with fan-out at tie-points, or with tie-point relationships. However the approach taken here can also run bottom-up, which is particularly useful in applications in which we want to recognise as much as possible even though full recognition may be impossible (because of errors in the graph, or because the grammar is necessarily incomplete). Wills (1986, 1990) modified Brotsky's algorithm to cope with fan-out, but her algorithm only runs in a pseudo-bottom-up fashion by starting it running top-down looking for every possible non-terminal at every possible place in the graph. More recently, Wills (1992) developed an algorithm heavily based on the chart parsing algorithm described here and in (Lutz, 1986, 1989), which is also capable of dealing with attributes and tie-point relationships. However her algorithm and graph representations make no mention of tie-points, but deal directly with the edges connecting NAPEs. This makes it harder to deal elegantly with the tie-point relationships.

More recently there have been several papers in the visual language literature which have adopted a chart parsing approach. In particular, Wittenburg et al. (1991) and Golin (1991) have both developed bottom up parsers for 2-

dimensional languages, while O'Gorman (1992), Costagliola et al. (1991), and Wittenburg (1992) have developed top-down parsers. This work is all similar in spirit to that presented here, although differences in representation, and application, make it very different in detail. Indeed this seems to be a general problem with work on 2-dimensional languages - there is no known general method suitable for conveniently representing all the different classes of language, and this leads to algorithms for one domain being very different from those in another. Of course, some kind of definite clause encoding could be used for all of these, but this is not always natural, and does not always lend itself to the development of efficient algorithms. In this connection it should be noted that the flowgraph languages discussed in this paper can be encoded in the Datalog formalism (Abitoul — Vianu, 1988) for which it is known that parsing can be performed in polynomial time. However, a special-purpose algorithm like the one presented can be particularly efficient and adaptable (cf. the control of structure sharing).

A particular advantage of a chart parser is that it keeps a record of all partial patches. This is useful when we do not just wish to analyse how some graph has been generated, but also to make suggestions based on "near-miss" information about how to correct the graph. As such this algorithm is being used as the basis of an intelligent debugging system for Pascal programs (Lutz, 1992).

The algorithms described in this paper have been implemented in Pop-11, running under the POPLOG™ system.

References

Abitoul, S. — V. Vianu (1988) "Datalog Extensions for Database Updates and Queries". I.N.R.I.A. Technical Report No. 715

Brotsky, D.C. (1984) "An Algorithm for Parsing Flow Graphs". AI-TR-704. MIT Artificial Intelligence Laboratory.

Costagliola, G. — M. Tomita — S.-K. Chang (1991) "A Generalised Parser for 2-D Languages" In: *Proceedings of IEEE Workshop on Visual Languages* 98 - 104.

Della Vigna, P. — C. Ghezzi (1978) "Context Free Graph Grammars". In: *Information and Control* Volume(37), 207 - 233.

Earley, J. (1970) "An Efficient Context-Free Parsing Algorithm". In: *Communications of the ACM* Volume(13), 94 - 102.

Ehrig, H. (1979) "Introduction to the Algebraic Theory of Graph Grammars (A Survey)". In: Claus, V. & H. Ehrig & G. Rozenberg, (Eds): *Graph Grammars and their Application to Computer Science and Biology* Lecture Notes in Computer Science. Springer-Verlag.

Feder, J. (1971) "Plex Languages". In: *Information Sciences* Volume(3) 225 - 241.

Fu, K.S. (1974) *Syntactic Methods in Pattern Recognition* New York: Academic Press.

Golin, E.J. (1991) "Parsing Visual Languages with Picture Layout Grammars" In: *Journal of Visual Languages and Computing* Volume(2), 371 - 393.

Gonzalez, R.C. — M.G. Thomason (1978) *Syntactic Pattern Recognition: An Introduction* Addison-Wesley.

Lutz, R.K. (1986) "Diagram Parsing - A New Technique for Artificial Intelligence". CSRP-054. School of Cognitive and Computing Sciences, University of Sussex.

Lutz, R.K. (1989a) "Chart Parsing of Flowgraphs". In: *Proceedings of 11th Joint International Conference on AI, Detroit, USA*

Lutz, R.K. (1989b) "Debugging Pascal Programs Using a Flowgraph Chart Parser". In: Proceedings of 2nd Scandinavian conference on AI, Tampere, Finland.

Lutz, R.K. (1991) "Plan Diagrams as the Basis for Understanding and Debugging Pascal Programs". In: Eisenstadt, M. & T. Rajan & M. Keane, (Eds) *Novice Programming Environments* London: Lawrence Erlbaum Associates.

Lutz, R.K. (1992) "Towards an Intelligent Debugging System for Pascal Programs: On the Theory and Algorithms of Plan Recognition in Rich's Plan Calculus". Ph.D. Thesis The Open University, Milton Keynes, England.

O'Gorman, L. (1992) "Image and Document Processing Techniques for the RightPages Electronic Library System" In: *Proceedings 11th IAPR International Conference on Pattern Recognition* Volume(2), 260 - 263.

Pfaltz, J.L. — A. Rosenfeld (1969) "Web Grammars". In: *Proceedings of 1st International Joint Conference on AI* 609 - 619.

Rich, C. (1981) "Inspection Methods in Programming". AI-TR-604 MIT Artificial Intelligence Laboratory.

Rosenfeld, A. — D.L. Milgram (1972) "Web Automata and Web Grammars". In: Meltzer, B. & D. Michie (Eds): *Machine Intelligence* 7, 307 - 324. Edinburgh University Press.

Thompson, H. — G. Ritchie (1984) "Implementing Natural Language Parsers". In: O'Shea, T. & M. Eisenstadt (Eds) *Artificial Intelligence: Tools, Techniques, and Applications* 245 - 300 Harper and Row.

Wills, L.M. (1986) "Automated Program Recognition". MSc Thesis. MIT Electrical Engineering and Computer Science.

Wills, L.M. (1990) "Automated Program Recognition: A Feasibility Demonstration". In: *Artificial Intelligence* Volume(45), 113 - 171.

Wills, L.M. (1992) "Automated Program Recognition by Graph Parsing". Ph.D. Thesis. MIT, Boston, Mass.

Wittenburg, K. (1992) "Earley-style Parsing for Relational Languages" In: *Proceedings of IEEE Workshop on Visual Languages* 192 – 199.

Wittenburg, K. — L. Weitzman — J. Talley (1991) "Unification-based Grammars and Tabular Parsing for Graphical Languages" In: *Journal of Visual Languages and Computing* Volume(2), 347 – 370.

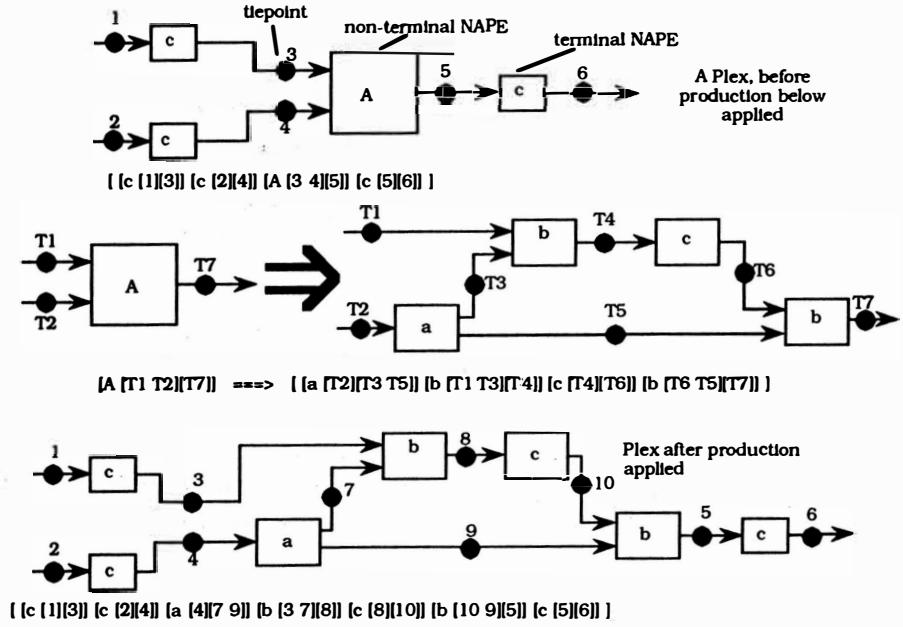


Figure 1.
Simple Flowgraph and Rules

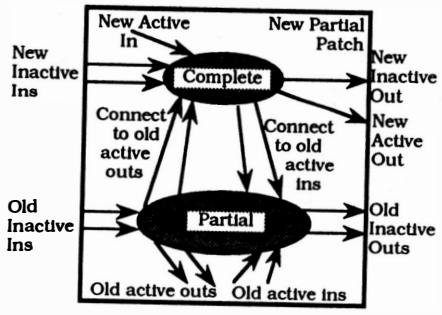
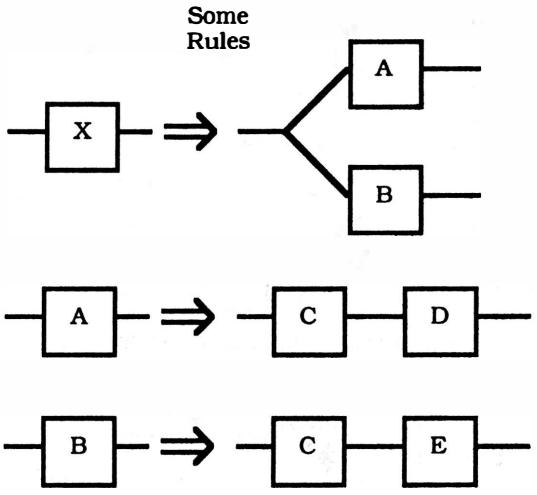


Figure 2
The Joining Operation



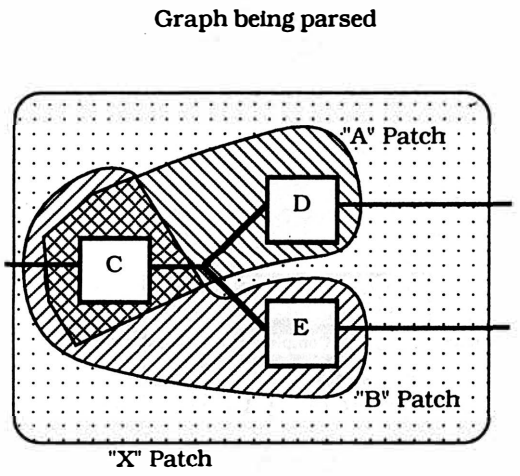
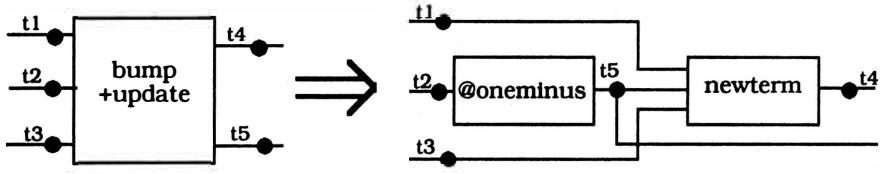
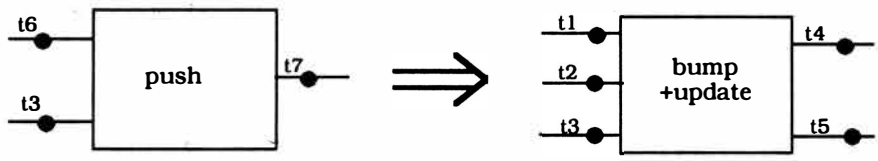


Figure 3
Occurence of Structure Sharing Without No Sharing Check



where t6=upper-segment(t1,t2) and
t7=upper-segment(t4,t5)

Figure 4
Bump+update



where
t6=upper-segment->list(t8) and
t7=upper-segment->list(t9) and
t8=upper-segment(t1,t2) and
t9=upper-segment(t4,t5)

Figure 5
Bump+update->push

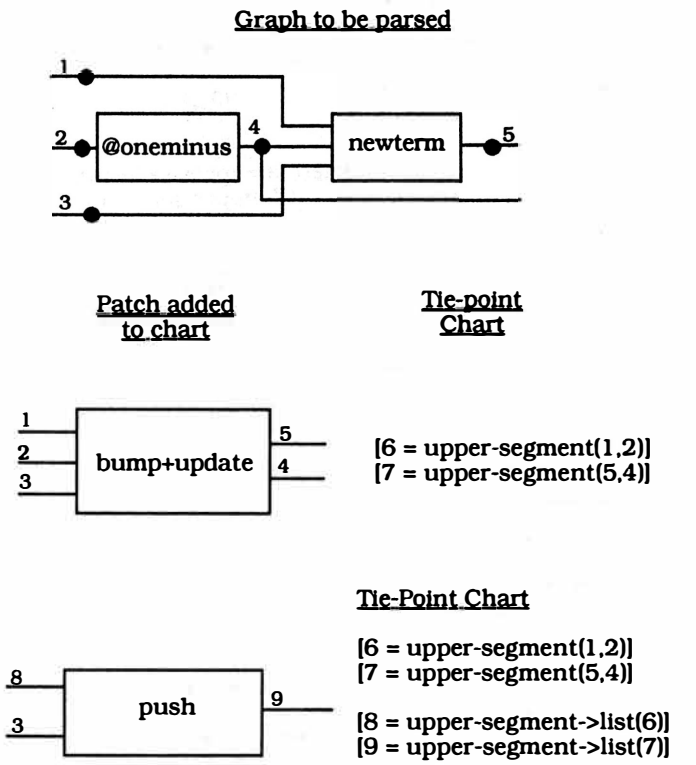
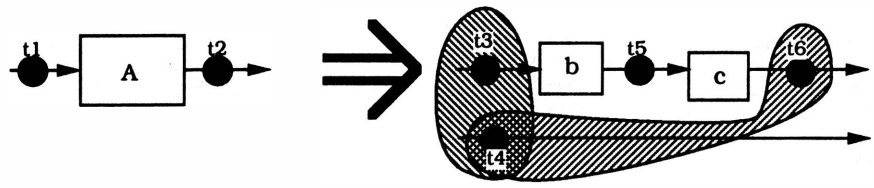
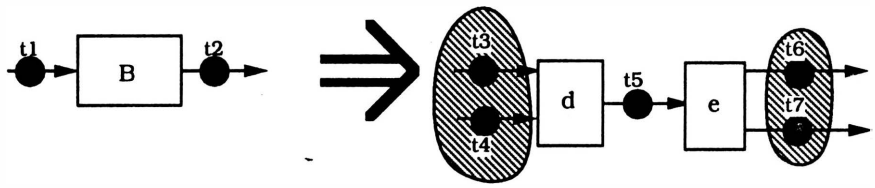


Figure 6
Use of Tie-point Chart



where t1=iterator(t3, t4)
and t2=iterator(t6, t4)



where t1=iterator(t3, t4)
and t2=iterator(t6, t7)

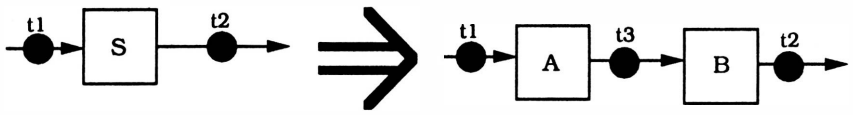


Figure 7
Some Rules (with a "straight-through" arc)

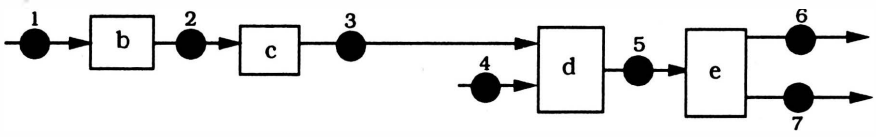


Figure 8
Graph to be Parsed

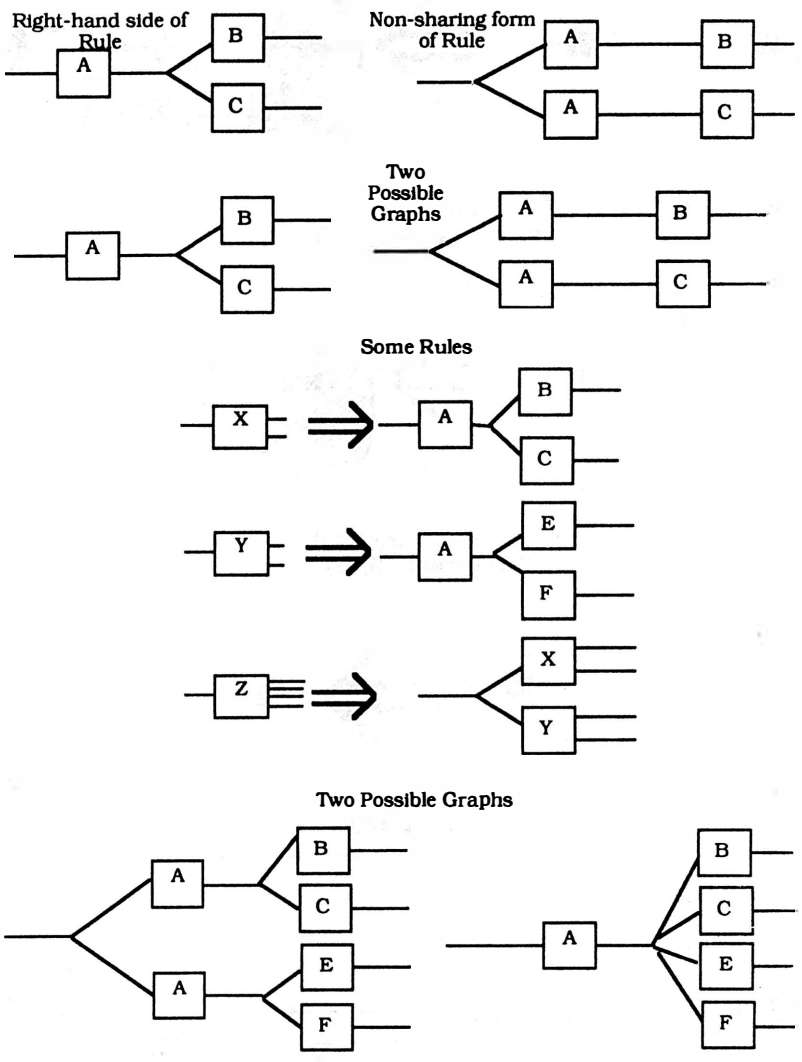


Figure 9
Structure Sharing and Collapsing Phenomena

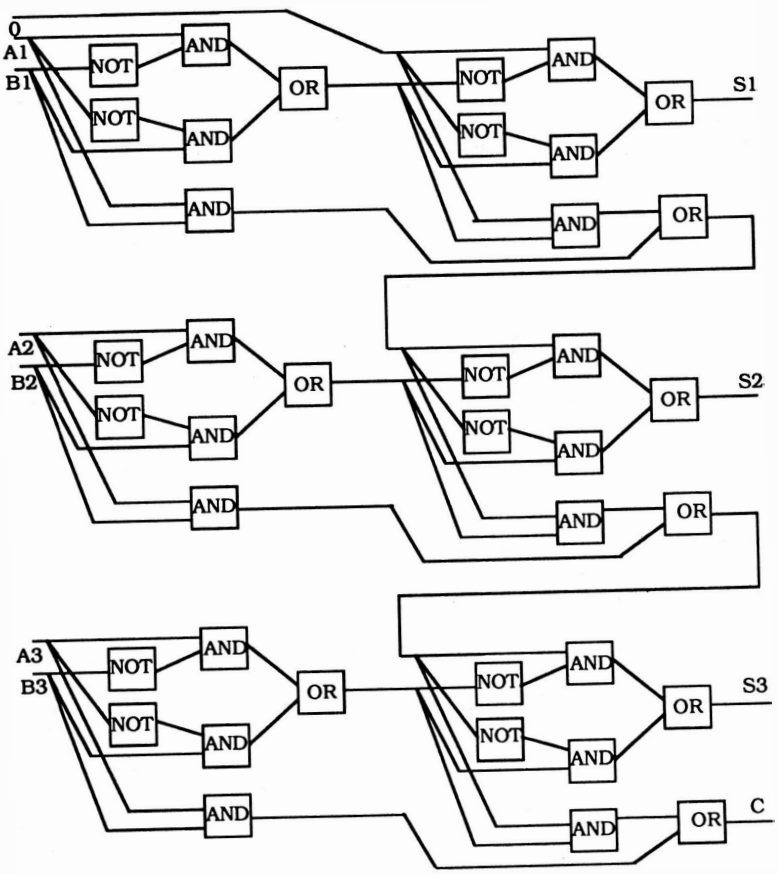


Figure 10
A 3-Bit Addition Circuit

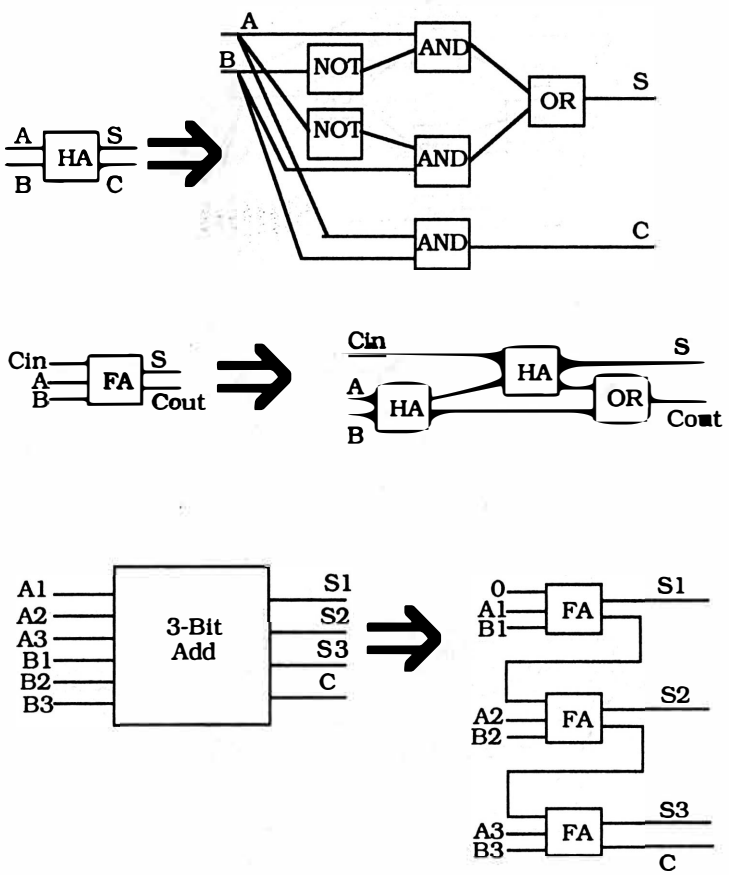
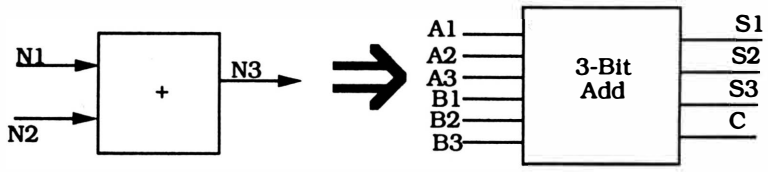
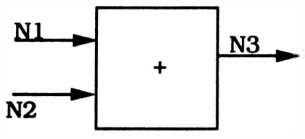


Figure 11
Addition Circuit Grammar



where
Binary1=3-bits(A1,A2,A3) and
Binary2=3-bits(B1,B2,B3) and
Binary3=4-bits(S1,S2,S3,C) and
N1=3-bits->integer(Binary1) and
N2=3-bits->integer(Binary2) and
N3=4-bits->integer(Binary3)

Figure 12
Integer Addition Rule



where
Binary1=3-bits(A1,A2,A3) and
Binary2=3-bits(B1,B2,B3) and
Binary3=4-bits(S1,S2,S3,C) and
N1=3-bits->integer(Binary1) and
N2=3-bits->integer(Binary2) and
N3=4-bits->integer(Binary3)

Figure 13
High-Level Description of Figure 10