Parallel Evaluation in Attribute Grammar-Based Systems

Alan K. Zaring
Ph.D Thesis

90-1149
August 1990

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

PARALLEL EVALUATION IN ATTRIBUTE GRAMMAR-BASED SYSTEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Alan Keith Zaring

August 1990

PARALLEL EVALUATION IN ATTRIBUTE GRAMMAR-BASED SYSTEMS

Alan Keith Zaring, Ph.D.
Cornell University 1990

Attributed context-free grammars provide a rigorous basis for the semantic analysis and translation of tree-structured objects and have been used to build a variety of systems. A number of programming language compilers, compiler generators, and language-based editor generators employing attribute grammars have been described in the literature. Many of these systems make use of *l-ordered attribute grammars*, attribute grammars for which particularly efficient methods for attributing derivation trees have been described.

Derivation trees representing constructs of only moderate size may contain thousands of nodes and tens of thousands of attribute instances, and attribution of such trees on uniprocessor systems may require a significant amount of time. One possibility for reducing this time is to find techniques that exploit opportunities for parallelism in the attribution process and allow attribution to be performed on multiprocessor systems. Such techniques would permit attribute grammars to serve as a rigorous foundation for the development of parallel compilation systems and other parallel applications.

We present several methods for the parallel attribution of trees derived from l-ordered attribute grammars. These methods take advantage of parallelism implicit in the attribution process and, thus, do not require any special considerations to be taken when constructing grammars. Methods appropriate for use on tightly- and loosely-coupled multiprocessor architectures and for use when complete and incremental tree attribution are required are presented.

We present preliminary performance results obtained from implementations of some of the methods on a simple shared-memory multiprocessor simulator embedded within an attribute grammar-based editor generator system. The results suggest that the methods may provide useful reductions in attribution time in some cases.

BIOGRAPHICAL SKETCH

Alan Keith Zaring was born on July 5, 1954. He received an A.B. degree in Computer Science and Mathematics from Indiana University in May, 1976. After pursuing graduate studies in Computer Science at The University of Texas at Austin, he went on to receive an M.S. in Computer Science from Cornell University in January, 1985.

To Laurie, without whose love and devotion I would never have made it

ACKNOWLEDGMENTS

I would like to thank Tim Teitelbaum, my advisor, for his help and for sticking with me over the course of what turned into a drawn-out process. I also thank John Bowers and Keshav Pingali for serving on my committee. In addition, my association with the Cornell Synthesizer Generator Project and its many members were invaluable in completing this dissertation. The administrative work of Elizabeth Maxwell was much appreciated.

I would like to thank Dennis Gannon, Franklin Prosser, and Edward Robertson at Indiana University for providing technical help and a place to work during my leave from Cornell. Special thanks go to David Plaisier for his efforts to keep alive computing machinery that would have otherwise died long ago.

The skillful counsel of William Chestnut is also gratefully acknowledged.

I also thank the National Science Foundation and the Office of Naval Research for their generous support under NSF/ONR Grant DCR-8514862 and ONR Grant N00014-88-K-0594.

TABLE OF CONTENTS

LIST OF FIGURES

1. Introduction

1.1 Syntax and Semantics of Hierarchical Objects

Objects having hierarchical structure are most commonly and naturally represented through the use of *trees*. In particular, the structure of a phrase belonging to a language described by a *context-free grammar* is typically represented by a *derivation tree*. However, while context-free grammars and derivation trees rigorously capture the notion of the form (syntax) of such objects, they provide no formal notion of the meaning (semantics) of objects. For example, a context-free grammar for a typical computer programming language can describe all necessary syntactic properties of programs (i.e., where certain punctuation marks are required, where identifiers are required, what sequences of characters may constitute identifiers, etc.) but cannot describe even simple semantic aspects of programs (e.g., all variables must be declared and programs must be type-correct).

1.2 Attribute Grammars

One approach to rigorously specifying both the form and the meaning of hierarchical objects is to begin by formulating a context-free grammar and then to extend it to an *attributed context-free grammar* (often referred to simply as an *attribute grammar*) [Knuth 1968]. Such an extension requires two steps. First, each nonterminal symbol in the grammar is given a set of *attributes*, making the nonterminal a composite object similar to a "record" variable in a Pascal-like programming language [Jensen & Wirth 1985]. Second, each production in the grammar is given a set of *semantic equations*, each equation defining the value of a single attribute of a nonterminal in terms of the values of other attributes of nonterminals.

In use, conceptually, one first constructs a derivation tree using the context-free productions. Then, one uses the semantic equations corresponding to those productions to *consistently attribute* the tree, that is, to assign values to all the attributes in all the nonterminal nodes in the tree. After attribution, the values of the attributes of the root node of the tree give the defined meaning of the object represented by the tree.

Attribute grammars have been used as a basis in a variety of applications [Deransart et al 1988], including programming language compilers [Aho et al 1986, Lewis et al 1974, Waite & Goos 1984], compiler generators [Farrow 1989, Jourdan et al 1990, Kastens et al 1982], and editor generators [Johnson & Fischer 1985, Reps & Teitelbaum 1989a].

1.3 Attribution and Parallelism

Attribution of a tree is guided by a partial ordering relation among attributes that is derived from the semantic equations. On uniprocessor computer systems, any topological ordering of the attributes based on the partial order can be used to produce a consistent sequential attribution of the tree. On multiprocessor systems, it is often the case that many attributes' values can be calculated concurrently, resulting in a decrease in the time needed for attribution.

This thesis examines methods for exploiting opportunities for parallel execution in attributing trees. We concentrate on trees derived from the class of *l-ordered attribute grammars* [Engelfriet & Filè 1982], a subclass of attribute grammars for which particularly efficient serial attribution

methods are known. A variety of attribution methods for both tightly- and loosely-coupled parallelism are explored. Simulation results suggest that significant reductions in attribution times are possible using these methods on multiprocessor machines.

1.4 Overview of the Thesis

Chapters 2 and 3 of this thesis present background material. Chapter 2 provides a detailed introduction to attribute grammars and l ordered attribute grammars (a summary of notation and terminology is provided at the end of the chapter). Chapter 3 gives a brief presentation of the simple model of multiprocessor architecture assumed in the thesis.

Chapter 4 explores parallel attribution assuming a loosely-coupled (or *asynchronous*) model of parallelism. Asynchronous parallelism is, to some degree, in conflict with those properties of l ordered attribute grammars that allow efficient tree attribution. The non-trivial aspects of developing asynchronous parallel attribution methods are largely concerned with resolving this conflict. In particular, preventing, in an acceptably efficient manner, attempts to use the values of attributes before those values have been determined requires care when asynchronous parallelism is employed.

Chapter 5 continues with a presentation of parallel attribution under tightly-coupled (or *synchronous*) parallelism. While synchronous parallelism may be intuitively more compatible with the desirable properties of l-ordered attribute grammars, problems requiring somewhat intricate handling still occur. In particular, the static hierarchical structure of a derivation tree and the dynamic hierarchical control structure of a group of synchronous processes being used to attribute that tree will not always coincide in an obvious way; coordinating these two hierarchies in order to achieve a correct tree attribution can be difficult.

Chapter 6 then discusses the notion of *incremental attribution* and ways the methods of Chapters 4 and 5 can be adapted to operate in an incremental fashion. This chapter, as well as Chapters 4 and 5, includes comparisons with previous, related work.

Chapter 7 presents the results of experiments conducted using a simple simulated multiprocessor system. The results suggest that both asynchronous and synchronous parallel attribution maybe employed to achieve useful reductions in attribution time, with synchronous parallelism being more advantageous for use with attribute grammars having a greater degree of inherent parallelism and asynchronous parallelism being more advantageous for use with grammars that are more inherently serial.

Chapter 8 presents some conclusions and suggestions for further research.

2. Attribute Grammars

This chapter introduces concepts, terminology, and notation concerning attribute grammars, l-ordered attribute grammars [Engelfriet & Filè 1982], and ordered attribute grammars [Kastens 1980]. Those already familiar with these topics may skip to §2.4 for a summary of terminology and notation.

2.1 Basic Concepts

An attribute grammar is an extension of a context-free grammar. To each nonterminal we assign a (possibly empty) set of *attributes*. To each production we assign a set of *semantic equations*. Each semantic equation defines the value of some single attribute of a nonterminal occurring in the production. For example, Figure 2.1 shows a context-free grammar (derived from an example in [Knuth 1968]) defining the set of binary integers as strings of zeros and ones.

```
BinaryInt → BitString
BitString → Bit
BitString → BitString Bit
Bit → '0'
Bit → '1'
```

Figure 2.1 - A Grammar for Binary Integers

To define the meaning of a binary integer thus represented to be the equivalent nonnegative decimal integer value represented by the bit-string, we extend the above grammar to an attribute grammar. First, we define the attributes of the nonterminals:

BinaryInt has an attribute "meaning."

BitString has attributes "meaning" and "position."

Bit has attributes "meaning" and "position."

Then we associate semantic equations with each production, as in Figure 2.2. The notation "X.b" denotes attribute b of nonterminal X. In

```
p0 : BinaryInt → BitString
   {  BitString.position = 0
      BinaryInt.meaning = BitString.meaning }
p1 : BitString → Bit
   {  Bit.position = BitString.position
      BitString.meaning = Bit.meaning }
p2 : BitString → BitString Bit
   {  Bit.position = BitString$1.position
      BitString$2.position = BitString$1.position + 1
      BitString$1.meaning = BitString$2.meaning + Bit.meaning }
p3 : Bit → '0'
   {  Bit.meaning = 0 }
p4 : Bit → '1'
   {  Bit.meaning = 2^(Bit.position) }
```

Figure 2.2 - An Attribute Grammar for Binary Integers

productions with more than one occurrence of nonterminal X, "X$n" $(n \geq 1)$ denotes the n^{th} occurrence of nonterminal X, where the leftmost occurrence is denoted by X$1.

Each occurrence of a nonterminal in a production has its own *occurrences* of each of the nonterminal's attributes. In a tree derived using a grammar, each tree node represents an *instance* of the corresponding nonterminal, each with its own *attribute instances*. The semantic equations associated with the productions determine the values of each of the

attribute instances in the tree. If a subtree U of tree T was derived using production p (i.e., U is an *instance* of production p), the semantic equations associated with p are used to determine values for attribute instances of the root node of U and its child nodes. For example, Figure 2.3 shows the derivation tree for the binary integer 1101.

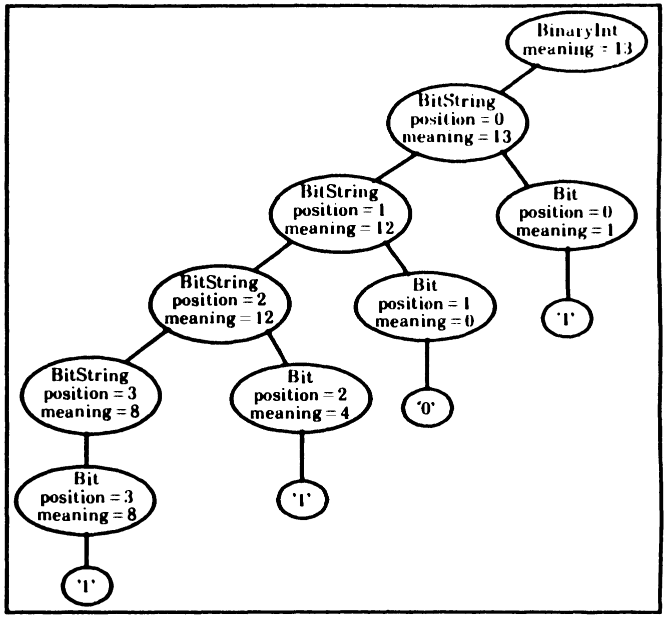

Figure 2.3 - An Attributed Derivation Tree

Note that for each tree node n that is an instance of the nonterminal BitString, the value of BitString.position is defined by a semantic equation associated with the production used to derive n's parent node. That is, the value of BitString.position is always defined by a semantic equation used in a context in which BitString is a right-hand side nonterminal occurrence in some production. We say that such attributes are *inherited*. In contrast, the values of instances of BitString.meaning are always defined by a semantic equation associated with productions in which BitString is a left-hand side occurrence; such attributes are termed *synthesized*.

More formally, let $G = (N, T, P, S)$ be a context-free grammar where

N is the set of nonterminal symbols,

T is the set of terminal symbols,

P is the set of productions, and

$S \in N$ is the starting nonterminal.

We construct an attribute grammar $H = (N, T, P, S, AI, AS, R)$ from G by defining for each $X \in N$

$AI(X)$ a set of inherited attributes of X and

$AS(X)$ a set of synthesized attributes of X.

We denote $AI(X) \cup AS(X)$, the set of all attributes of X, by $A(X)$.

For all X, $AI(X)$ and $AS(X)$ are disjoint. By extension, we will also use $A(X)$, $AI(X)$, and $AS(X)$ to denote the analogous sets of attribute occurrences for an occurrence of nonterminal X as well as the analogous set of attribute instances for an instance of X (i.e., a derivation tree node).

We note at this point that, while we have referred to attributes using simple names like "position," the name of an attribute (or attribute instance or attribute occurrence, respectively) is in reality always qualified by a nonterminal (or nonterminal occurrence or nonterminal instance, respectively) to which the attribute belongs. Thus, while nonterminals BitString and Bit both have attribute "position," the attributes in question are actually BitString.position and Bit.position. The qualification will often be assumed rather than shown explicitly when the qualification is

derivable from the context in which the attribute name appears. The qualification will always be given explicitly in semantic equations.

For each production $p: X_0 \rightarrow X_1 \ldots X_n$ we define a set of semantic equations $R(p)$, each equation of the form

$$X_i.b = f(\ldots, Y.c, \ldots)$$

where $Y \in \{ X_0 \ldots X_n \}$, $c \in A(Y)$, $b \in A(X_i)$, and f is some partial recursive function. (although when discussing the termination properties of particular attribution algorithms, we will assume that all semantic functions are total recursive.) A semantic equation *defines* attribute occurrence X.b if X.b is the left-hand side of the equation.

We define the set of *output attribute occurrences* of p to be

$$Out(p) = AS(X_0) \cup AI(X_1) \cup \ldots \cup AI(X_n)$$

and the set of *input attribute occurrences* of p to be

$$In(p) = AI(X_0) \cup AS(X_1) \cup \ldots \cup AS(X_n).$$

Production p is said to be *well-formed* if and only if every attribute occurrence in $Out(p)$ is defined by exactly one semantic equation in $R(p)$.

An attribute grammar $G = (N, T, P, S, AI, AS, R)$ is *well-formed* if and only if

(1) $AI(S) = \emptyset$

(2) every $p \in P$ is well-formed.

We will deal exclusively with well-formed attribute grammars.

For each production p we define the *local dependency graph* $DP(p)$ to be the directed graph

$DP(p) = (V, E)$ where

$V = A(X_0) \cup \ldots \cup A(X_n)$

$E = \{ (X_i.b, X_j.c) \mid X_j.c = f(\ldots, X_i.b, \ldots) \text{ is an element of } R(p) \}.$

That is, there is an edge from occurrence $X_i.b$ to occurrence $X_j.c$ if and only if the value of $X_i.b$ is used directly in determining the value of $X_j.c$. Production p is said to be *locally noncircular* if DP(p) is acyclic.

We define a similar graph for trees derived from attribute grammars. The *tree dependency graph* DT(T) for tree T is the directed graph

DT(T) = (V, E) where

V = { b | b is an attribute instance of some tree node (i.e., nonterminal instance) in T }

E = { (b, c) | the value of c depends directly on the value of b }.

An attribute grammar G is *well-defined* (or *noncircular*) if and only if for all trees T derived using G, DT(T) is acyclic. Determining the noncircularity of a grammar requires time exponential in the size of the grammar [Jazayeri et al 1975]. In the remainder of this thesis, we will assume all grammars are noncircular.

2.2 Attribute Evaluation

Assume that a derivation tree T has been constructed and that the attribute instances in T have yet to be assigned values. It is necessary to *consistently attribute* T; that is, each attribute instance in the tree must be given the value defined for it by its defining semantic equation.

Attribution of T is necessarily subject to the dependencies represented by the edges in DT(T). If Q is the set of all attribute instances upon whose values the value of attribute instance b depends, the value of b cannot be determined until all attribute instances in Q have been assigned their respective defined values. At *any* point in time after all elements of Q have

been assigned values, the value of b can be determined using the appropriate semantic equation.

2.2.1 Dynamic Evaluation Order Attribution

A straightforward way to honor the dependencies in DT(T) is to drive the attribution process by a topological ordering of DT(T), as does the algorithm in Figure 2.4. This algorithm assumes that each attribute

```
Evaluate(tree T) :
    (V, E) := DT(T);
    W := { b | b∈V & indegree(b) = 0 };
    while W ≠ ∅ do
        remove some b from W;
        value(b) := value defined by appropriate semantic fn;
        forall (b, c)∈E do
            indegree(c) := indegree(c) – 1;
            if indegree(c) = 0
                W := W ∪ { c }
            fi
        od
    od
end Evaluate;
```

Figure 2.4 - A Dynamic Order Evaluator

instance has a component initialized to the indegree of that instance with respect to DT(T).

Using this procedure, the order in which attribute instances in T are evaluated is, in general, determined only during the actual attribution process. Attribution methods having this property are said to use *dynamic evaluation orders*. Unfortunately, attribution techniques of this type in general require the explicit attribution-time presence of the tree dependency graph, and, consequently, have both storage and time requirements of $O(|DT(T)|)$.

2.2.2 Static Evaluation Order Attribution

An alternative is to base tree attribution on a traversal of the derivation tree rather than on a traversal of the dependency graph. For such a method to succeed, it must be possible to statically determine for each production p in a grammar an order in which to evaluate the attribute instances of any instance of p in any tree derived using the grammar. For any T, the evaluation order must obey all dependencies in DT(T) while not requiring the explicit presence of DT(T) during attribution.

The existence of such an ordering can be guaranteed only if restrictions are placed on the local dependencies permitted in the productions of a grammar. Various restrictions define various *subclasses* of the class of well-formed attribute grammars. Many such subclasses, together with associated tree-traversal-based attribution procedures, have been defined [Bochmann 1976, Jazayeri & Walter 1976, Kastens 1980, Kennedy & Warren 1976, Lewis et al 1974].

A simple example is the class of *S-attributed grammars* (SAG's) [Jazayeri & Walter 1976]. A well-formed attribute grammar G is an SAG if and only if no nonterminal has any inherited attributes (i.e., $AI(X) = \emptyset$ for all $X \in N$).

The grammar in Figure 2.5 is an S-attributed version of the grammar for binary integers. Given identical terminal strings, the meaning of a binary integer (i.e., the value of BinaryInt.meaning) is the same for the


```
p0 : BinaryInt → BitString
     { BinaryInt.meaning = BitString.meaning }
p1 : BitString → Bit
     { BitString.meaning = Bit.meaning }
p2 : BitString → BitString Bit
     { BitString$1.meaning = 2 × BitString$2.meaning + Bit.meaning }
p3 : Bit → '0'
     { Bit.meaning = 0 }
p4 : Bit → '1'
     { Bit.meaning 1 }
```

Figure 2.5 - An S-attributed Grammar for Binary Integers

grammars of Figure 2.2 and Figure 2.5, but the meanings of the various individual interior nonterminals (i.e., their respective attributes "meaning") are not necessarily the same for the two grammars. Inherited attributes BitString.position and Bit.position have necessarily been removed from this SAG version of the grammar.

Let G be an SAG. Every semantic equation of every production $p: X_0 \to X_1 \ldots X_n$ is necessarily of the form

$$X_0.b = f(\ldots, X_i.c, \ldots)$$

where $i \geq 0$, since $Out(p) = AS(X_0)$. For any instance of p in any derivation tree, we can first evaluate the instances of the synthesized attributes of the tree nodes corresponding to $X_1, \ldots, X_n$, and then, according to a topological ordering of the projection of $DP(p)$ onto $A(X_0)$, evaluate the attribute instances of the node corresponding to X_0.

To do this, we produce for each production a *plan* that can be used by a tree-walking attribute evaluator. A plan is an ordered sequence of *plan elements* of the form

[EVAL b] — which tells the evaluator to use the appropriate semantic equation to give a value to the instance of attribute b of the current tree node,

[VISIT_CHILD j] — which tells the evaluator to descend to the j^{th} child ($j>0$) of the current tree node to continue attribution,

[VISIT_PARENT] — which tells the evaluator to ascend to the parent of the current tree node to continue attribution, or

[SKIP] — which is a no-op for the evaluator.

The plan for production p will be a sequence of plan elements of the form

$$<e_1 \ldots e_n \text{[EVAL } c_1] \ldots \text{[EVAL } c_m]\text{[VISIT\_PARENT]}>$$

where each of e_k is either a [VISIT_CHILD k] or [SKIP] element and where $c_1, \ldots, c_m$ is a topological ordering of $A(X_0)$ ($=AS(X_0)$) with respect to DP(p). (Consecutive items enclosed in $<>$ denote an ordered sequence containing the items.)

If we assume that the productions in P are numbered from 0 to $|P|-1$, we can use the algorithm given in Figure 2.6 to produce a sequence of plans ordered by production number for an SAG. In this (and subsequent) algorithms, we let

$<i_0 \ldots i_{n-1}>$ — denote an ordered sequence of items,

$s[j]$ — denote selection of the j^{th} item in sequence s, where the first item in s is denoted by s[0],

$s_1 \| s_2$ — denote the concatenation of sequences s_1 and s_2, and


```
Let G = (N, T, P, S) be an S-attributed grammar.

PLANS := < >;
for i := 0 .. |P|-1 do
   (X0 → X1 . . . Xn) := pi;
   plan := < >;
   for j := 1 to n do
      if Xj ∈ N
         plan := plan || < [VISIT_CHILD j] >
      else
         plan := plan || < [SKIP] >
         fi
      od;
   D := DP(pi) / A(X0);
   W := { b | b ∈ D & indegree(b) = 0 };
   while W ≠ ∅ do
      remove some b from W;
      plan := plan || < [EVAL b] >;
      forall (b, c) ∈ D do
         indegree(c) := indegree(c) - 1;
         if indegree(c) = 0
            W := W ∪ { c }
            fi
         od
      od;
   plan := plan || < [VISIT_PARENT] >;
   PLANS := PLANS || < plan >
   od
```

Figure 2.6 - An S-attributed Grammar Planning Algorithm

G / V — denote the projection of digraph G onto vertex set V.

Figure 2.7 shows the sequence of plans that would be produced for the binary integers grammar of Figure 2.5.

```
< <[VISIT_CHILD 1][EVAL meaning][VISIT_PARENT]>
  <[VISIT_CHILD 1][EVAL meaning][VISIT_PARENT]>
  <[VISIT_CHILD 1][VISIT_CHILD 2][EVAL meaning]
        [VISIT_PARENT]>
  <[SKIP][EVAL meaning][VISIT_PARENT]>
  <[SKIP][EVAL meaning][VISIT_PARENT]>
  >
```

Figure 2.7 - Plans for an S-attributed Grammar for Binary Integers

Associated with a derivation tree node must be (at least) its attribute instances, a pointer to each of its child nodes, and the number of the production from which the subtree rooted at that node was derived. We can construct a grammar-independent recursive tree-walking attribute evaluator that, provided with a sequence of plans constructed by the algorithm of Figure 2.6, can attribute any tree derived from an SAG. Such an algorithm is given in Figure 2.8. To attribute a tree, one calls RecursiveEvaluate on the root node of the tree.

If we construct each tree node so that it also contains its own child number (i.e., the position it occupies as a sibling among the children of its parent node) and a pointer to its parent node, we can construct an iterative, finite-automaton-style evaluator that requires run-time space constant in the size of the grammar whose trees are to be attributed (assuming constant space for semantic function evaluation). An evaluator of this sort is presented in Figure 2.9. Note that the evaluator relies on the fact that the algorithm of Figure 2.8 produces plans that cause the children of a node to be visited and/or skipped in order from left to right


```
RecursiveEvaluate(treenode n):
    i := 0;
    while PLANS[prod_num(n)][i] ≠ [VISIT_PARENT] do
        case PLANS[prod_num(n)][i] of
            [EVAL b]:
                value(n.b) := value defined by appropriate semantic fn;
            [VISIT_CHILD j]:
                RecursiveEvaluate(child(j,n));
            [SKIP]:
            esac;
        i := i + 1
        od
    end RecursiveEvaluate;
```

Figure 2.8 - A Recursive S-attributed Tree Evaluator

2.2.3 Ordered Attribute Grammars

A particularly useful subclass of attribute grammars for which static evaluation orders can be determined is the class of *ordered attribute grammars* (OAG's) [Kastens 1980]. The class of OAG's is relatively large in that it properly includes many other statically orderable subclasses. Empirical evidence suggests that it is also a *natural* subclass in that many "intuitive" grammars fall into the class of OAG's and that grammars not falling into the class can often be relatively easily "massaged" into OAG's. (This is roughly analogous to claims made for the LALR(1) subclass of context-free languages [DeRemer 1969, Johnson 1978].)

The plans corresponding to the productions of an OAG allow for a more general type of tree traversal than did those of S-attributed grammars in that each node may visit its parent and child nodes many times. Plan

```
FAEvaluate(treenode n):
    i := 0;
    while PLANS[prod_num(n)][i] ≠ [VISIT_PARENT]
              | n ≠ root of tree do
        case PLANS[prod_num(n)][i] of
            [EVAL b] : {
                value(n.b) := value defined by appropriate semantic fn;
                i := i + 1
                };
            [VISIT_CHILD j] : {
                i := 1;
                n := child(j, n)
                };
            [VISIT_PARENT] : {
                i := child_num(n) + 1;
                n := parent(n)
                };
            [SKIP] :
                i := i + 1
            esac
        od
    end FAEvaluate;
```

Figure 2.9 - A Finite Automaton S-attributed Tree Evaluator

elements for OAG plans are generalizations of the SAG plan elements and have the form

[EVAL j b]	which tells the evaluator to evaluate attribute instance b of the j^{th} child of the current tree node (a node is considered to be its own zeroth child),

[VISIT_CHILD j k]	which tells the evaluator to descend to the j^{th} child ($j > 0$) of the current node for the k^{th} time ($k > 0$), or
[VISIT_PARENT k]	which tells the evaluator to ascend to the parent of the current node for the k^{th} time ($k \geq 0$).

(The [SKIP] element is not used.)

The intent is that after executing a [VISIT_CHILD j k] in the plan for node n, the tree-walking evaluator will execute a portion of the plan for n's j^{th} child and eventually resume execution of n's plan with the element immediately following the [VISIT_CHILD j k] element. Likewise, after executing a [VISIT_PARENT k], the evaluator will execute a portion of the plan for n's parent and eventually resume execution of n's plan with the element immediately following that [VISIT_PARENT k].

In order to produce plans for a grammar, we first determine if the grammar is indeed an OAG. If it is, information obtained while making the orderedness determination can subsequently be used to generate plans.

2.2.3.1 Determining Membership in the Class of OAG's

Given a well-formed attribute grammar G, for each production p and associated local dependency graph DP(p) = (V, E), we define the *induced dependency graph* IDP(p) to be

IDP(p) = (V', E') where

V' = V

$$E' = E \cup \{ (X_i.b, X_i.c) \mid X_i \text{ is a nonterminal occurrence in p, } Y \text{ occurs in some production q, } X_i = Y, \text{ and } (Y.b, Y.c) \text{ is an edge in } IDP(q)^+ \}.$$

(D^+ represents the non-reflexive transitive closure of directed graph D.) Figure 2.10 shows the IDP graphs for the grammar of Figure 2.2. Note that

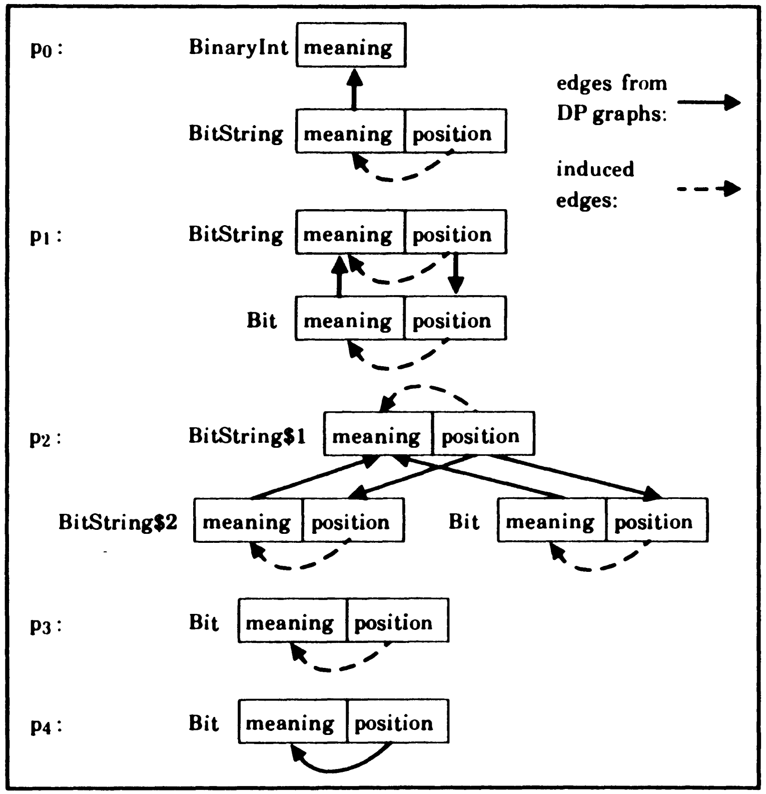

Figure 2.10 - IDP Graphs for the Binary Integer Grammar

the collection of IDP graphs for the productions of a grammar is defined in a mutually recursive manner.

IDP(p) is equal to DP(p) with the addition of any intra-symbol dependency edges (i.e., edges representing dependencies between attribute occurrences of a single nonterminal occurrence) that can be transitively inferred from any productions in the grammar. IDP(p) is *conservative* in the sense that, while containing edges representing all possible intra-symbol dependencies that can occur in derivation trees, IDP(p) may contain additional edges representing dependencies that cannot occur in any derivation tree.

If, for any production p, IDP(p) contains a cycle, G is not an OAG.

If nonterminal X occurs in production p as X_i and in production q as X_j, IDP(p) projected onto $A(X_i)$ is identical to IDP(q) projected onto $A(X_j)$. This notion is captured by the *induced dependency graph for a nonterminal* X IDS(X):

$$IDS(X) = (V, E) \text{ where}$$
$$V = A(X)$$
$$E = \{ (b, c) \mid X \text{ occurs as } X_i \text{ in some production p and } (X_i.b, X_i.c) \text{ is an edge in } IDP(p) \}.$$

IDS(X) includes all dependencies that can exist between attribute instances of any instance of nonterminal X in any derivation tree. Observe that, by construction of the IDP graphs, IDS(X) is transitively closed. Observe further that if IDP(p) is acyclic for all p, IDS(X) must be acyclic for all X. Figure 2.11 shows the IDS graphs for the grammar of Figure 2.2.

Recall that our goal is to produce plans consisting of EVAL, VISIT_CHILD, and VISIT_PARENT elements. As part of this process we

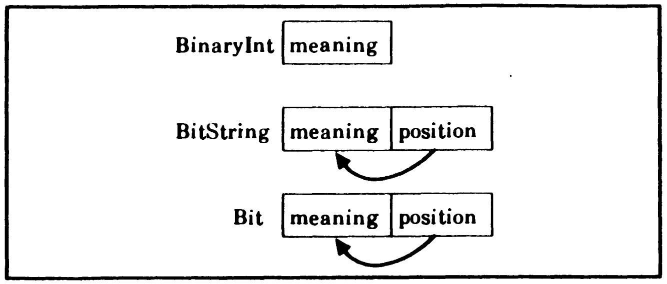

Figure 2.11 - IDS Graphs for the Binary Integer Grammar

must determine, for each nonterminal X, an order in which the attribute instances of any instance of X in any derivation tree may be evaluated, irrespective of context. Having defined IDS we can do this.

We define successive *partitions* of A(X)

$$A_1(X) = \{ b \in AS(X) \mid outdegree(b) = 0 \text{ or for all } (b, c) \text{ in } IDS(X), c \in A_1(X) \}$$

$$A_{2n}(X) = \{ b \in AI(X) \mid b \notin A_1(X) \cup \ldots \cup A_{2n-1}(X) \text{ and for all } (b, c) \text{ in } IDS(X), c \in A_m(X), m < 2n \}$$

$$A_{2n+1}(X) = \{ b \in AS(X) \mid b \notin A_1(X) \cup \ldots \cup A_{2n}(X) \text{ and for all } (b, c) \text{ in } IDS(X), c \in A_m(X), m < 2n+1 \}$$

until all elements of A(X) are assigned to a partition. We define m(X) to be the number of partitions thus defined for X. Figure 2.12 shows the partitions for the grammar of Figure 2.2.

Having partitioned A(X), we define DS(X), the *dependency graph for a nonterminal* X, to be

$$(V, E') = DS(X) \text{ where}$$
$$IDS(X) = (V, E)$$
$$E' = E \cup \{ (b,c) \mid b \in A_k(X), c \in A_{k-1}(X), 2 \le k \le m(X) \}.$$

X	m(X)	$A_2(X)$	$A_1(X)$
BinaryInt	1	–	{ meaning }
BitString	2	{ position }	{ meaning }
Bit	2	{ position }	{ meaning }

Figure 2.12 - Attribute Partitions for the Binary Integer Grammar

DS(X) is simply IDS(X) with additional edges reflecting the partitioning of A(X). For the grammar of Figure 2.2, the partitioning step adds no new edges, so DS(X) = IDS(X) for each X.

For each nonterminal X, DS(X) embodies a proposed context-independent order in which to evaluate the attribute instances of any instance of X in a derivation tree. For each nonterminal, it must be determined if the order is compatible with each occurrence of the nonterminal in the productions of the grammar. For each production $p: X_0 \rightarrow X_1 \ldots X_n$, we define the *extended dependency graph* EDP(p):

$$(V, E') = EDP(p) \text{ where}$$
$$DP(p) = (V, E)$$
$$E' = E \cup \{ (X_i.b, X_i.c) \mid 0 \le i \le n, X_i = X, (b, c) \text{ is in } DS(X) \}.$$

For the grammar of Figure 2.2, for each production p, EDP(p) = IDP(p).

Attribute grammar G is an OAG if and only if, for all productions p of G, EDP(p) is acyclic.

2.2.3.2 Plan Generation for Ordered Attribute Grammars

Having determined that G is an OAG, we can produce a plan for each production $p: X_0 \rightarrow X_1 \ldots X_n$ in G using EDP(p) and the partitions defined for the nonterminals of G. Each attribute occurrence in Out(p) will be represented in the plan by an EVAL element. Each partition of inherited attributes of X_0 will correspond to a VISIT_PARENT element. Each partition of synthesized attributes of $X_1 \ldots X_n$ will correspond to a VISIT_CHILD element in the plan. Tree nodes corresponding to terminal symbols are not visited since terminal symbols have no attributes.

In the following discussion, we assume that attribution takes place only after the derivation tree is fully constructed. If we wish to interleave tree construction with attribution, plan generation is somewhat different [Kastens 1980].

For convenience, we define for each nonterminal X

$f(X)$ = the smallest even integer such that $f(X) \geqq m(X)$

$nv(X) = f(X)/2$.

We also define MAPVS, a mapping from nodes of EDP(p) to plan elements:

$$
\begin{aligned}
MAPVS(p, X_j.b) = {} & [\text{EVAL } j \ b] \\
& \quad \text{if } X_j.b \in Out(p), \\
& [\text{VISIT\_CHILD } j \ k-1] \\
& \quad \text{if } j>0,\ X_j = Y,\ b \in A_i(Y) \cap In(p), \\
& \quad k = (f(Y) - i + 1) \text{ div } 2, \text{ and } k>0, \\
& [\text{VISIT\_PARENT } j \ k-1] \\
& \quad \text{if } j=0,\ X_j = Y,\ b \in A_i(Y) \cap In(p), \\
& \quad k = (f(Y) - i + 1) \text{ div } 2, \text{ and } k>0, \\
& \text{undefined} \\
& \quad \text{if } X_i = Y,\ b \in A_i(Y) \cap In(p), \text{ and} \\
& \quad 0 = (f(Y) - i + 1) \text{ div } 2, \text{ and}
\end{aligned}
$$

$$
\begin{aligned}
& \text{undefined} \\
& \quad \text{otherwise.}
\end{aligned}
$$

Finally, we define the *visit-sequence graph* VS(p), for each production $p{:}X_0 \rightarrow X_1 \ldots X_n$, to be

$$
\begin{aligned}
VS(p) = (V, E) \text{ where} \\
V = \{ & [\text{EVAL } j \ b] \mid 1 \leq j \leq n,\ b \in AI(X_j) \} \\
& \cup \{ [\text{EVAL } 0 \ b] \mid b \in AS(X_0) \} \\
& \cup \{ [\text{VISIT\_CHILD } j \ k-1] \mid 1 \leq j \leq n,\ 1 \leq k \leq nv(X_j) \} \\
& \cup \{ [\text{VISIT\_PARENT } k-1] \mid 1 \leq k \leq nv(X_0) \} \\
E = \{ & (MAPVS(p, X_i.b), MAPVS(p, X_j.c)) \mid 1 \leq i \leq n,\ 1 \leq j \leq n, \\
& \quad MAPVS(p, X_i.b) \text{ and } MAPVS(p, X_j.c) \text{ are defined,} \\
& \quad \text{and } (X_i.b, X_j.c) \text{ is an edge in } EDP(p) \} \\
& \cup \{ (v, [\text{VISIT\_PARENT } nv(X_0)-1]) \mid v \in V, \\
& \quad \text{and } v \neq [\text{VISIT\_PARENT } nv(X_0)-1] \}.
\end{aligned}
$$

The visit-sequence graphs (with edges implied by transitivity omitted) for the grammar of Figure 2.2 are given in Figure 2.13. Plans for the grammar are produced by topologically ordering the vertices of the visit sequence graphs. Figure 2.14 shows the sequence of plans generated for the grammar of Figure 2.2.

A minor complication may arise. For a nonterminal X, partition $A_1(X)$ may be empty. This occurs if all members of AI(X) apparently depend on one or more elements of AS(X). If this is so, whenever X occurs on the right-hand side of a production as, say, X_i, the plan for that production will not contain a [VISIT_CHILD i nv(X) − 1] element.

This missing VISIT_CHILD element can cause no additional attribute instances of an instance of X_i to be given values (since they must

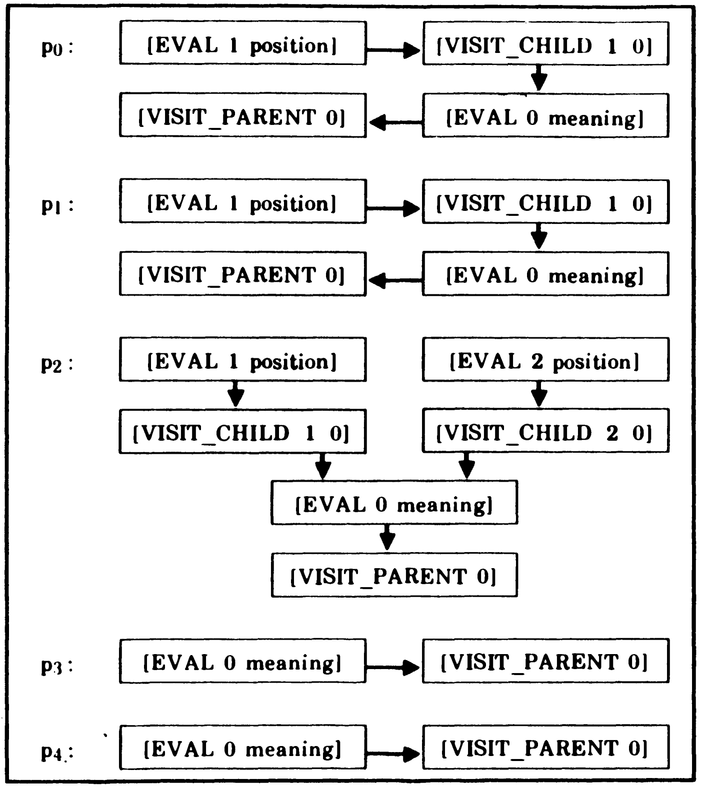

Figure 2.13 - VS Graphs for the Binary Integer Grammar

all already have been assigned values) but might cause attribute instances belonging to some of the children of an instance of X_i to be given values. Although omitting the visit could result in an incompletely attributed tree, this does no harm in systems that use a pure attribute grammar discipline: the values of any of the child attributes left unevaluated cannot be


```
< <[EVAL 1 position] [VISIT_CHILD 1 0] [EVAL 0 meaning]
    [VISIT_PARENT 0] >
  <[EVAL 1 position] [VISIT_CHILD 1 0] [EVAL 0 meaning]
    [VISIT_PARENT 0] >
  <[EVAL 1 position] [VISIT_CHILD 1 0] [EVAL 2 position]
    [VISIT_CHILD 2 0] [EVAL 0 meaning] [VISIT_PARENT 0] >
  <[EVAL 0 meaning] [VISIT_PARENT 0] >
  <[EVAL 0 meaning] [VISIT_PARENT 0] >
  >
```

Figure 2.14 - Plans for the Binary Integer Grammar

propagated upward, and thus cannot be used to determine attribute values in other parts of the derivation tree.

However, some systems, such as language-based editors [Reps & Teitelbaum 1989a], may make use of the value of some attribute instance b even though no other attribute instance depends on the value of b. For example, the value of b may be displayed on a terminal screen. In systems with this property, it is necessary to ensure that all attribute instances are given their defined values. When generating plans for such systems, any missing VISIT_CHILD elements can be placed, in arbitrary order, immediately before the final VISIT_PARENT element in the plan.

2.2.3.3 Attribution of OAG Derivation Trees

Given that plans for an OAG are produced as described in the previous section, we can exhibit grammar-independent tree-walking evaluators that use a set of plans to attribute trees derived from the grammar. A simple recursive attribution algorithm is given in Figure 2.15. This algorithm assumes, as did the recursive S-attributed grammar attribution algorithm,

```
RecursiveEvaluate(treenode n, integer i):
    while PLANS[prod_num(n)][i] ≠ [VISIT_PARENT *] do
        case PLANS[prod_num(n)][i] of
            [EVAL j b]:
                value(child(j,n).b) :=
                    value defined by appropriate semantic fn;
            [VISIT_CHILD j k]:
                RecursiveEvaluate(
                    child(j, n),
                    MAPDOWN(k, prod_num(child(j, n)))
                    )
            esac;
        i := i+1
        od
    end RecursiveEvaluate;
```

Figure 2.15 - A Recursive Visit-Sequence Tree Evaluator

that each derivation tree node contains a pointer to each child node, and the number of the production from which the subtree rooted at the node was derived. The routine also requires the existence of the mapping MAPDOWN, defined as

MAPDOWN(k, p)

= the index of the first plan element to be executed on the k^{th} VISIT_CHILD to a node derived from production p

= 0 if k = 0

i+1 if k > 0, where i is the index of the plan element [VISIT_PARENT k − 1] in the plan for production p

MAPDOWN can be statically determined by examining the plans produced for a grammar. To attribute tree T, call RecursiveEvaluate(*root of* T, 0).

As we did for S-attributed grammars, we can also construct a finite-automaton-style attribution routine. We again require that each tree node contains its own child number and a pointer to its parent.. We also require the existence of an additional mapping MAPUP:

MAPUP(k, j, p)

= the index of the first plan element to be executed on the k^{th} VISIT_PARENT to the root node of a subtree derived from production p from its j^{th} child

= i+1 where i is the index of the plan element [VISIT_CHILD j k] in the plan for production p.

MAPUP can be statically determined, as was MAPDOWN, and both MAPUP and MAPDOWN require constant space with respect to a given OAG. Figure 2.16 shows the finite-automaton-style OAG attribution routine. To attribute tree T, call FAEvaluate(*root of* T, 0).

2.3 L-Ordered Attribute Grammars

The class of OAG's is, in fact, properly contained within the class of *l-ordered attribute grammars* [Engelfriet & Filè 1982] (also called *partitionable attribute grammars* in [Waite & Goos 1984]), a class first characterized in [Kastens 1978]. The class of l-ordered attribute grammars (l-OAG's) is the largest class of grammars for which visit-sequences suitable for use by evaluators of the type presented in §2.2.3.3 can be constructed.

```
FAEvaluate(treenode n, integer i) :
    while PLANS[prod_num(n)][i] ≠ [VISIT_PARENT *]
            | n ≠ root of tree do
        case PLANS[prod_num(n)][i] of
            [EVAL j b] : {
                value(child(j,n).b) : =
                    value defined by appropriate semantic fn;
                i : = i + 1
                };
            [VISIT_CHILD j k] : {
                n : = child(j, n);
                i : = MAPDOWN(k, prod_num(n))
                };
            [VISIT_PARENT k] : {
                i : = MAPUP(k, child_num(n), prod_num(parent(n)));
                n : = parent(n)
                }
            esac
        od
    end FAEvaluate;
```

Figure 2.16 - A Finite Automaton Visit-Sequence Evaluator

Unfortunately, determining membership in the class of l-OAG's is an NP-complete problem [Engelfriet & Filè 1982]. The use of the IDP and IDS graphs in determining attribute instance partitions for OAG's can be viewed as a heuristic that permits polynomial-time membership testing for a subclass of l-OAG's. Other heuristics lead to other subclasses of l-OAG's [Jourdan et al 1990].

Although we will occasionally resort to OAG-specific terminology for simplicity, the parallel attribution algorithms presented in Chapters 4-6 of

this thesis are suitable for use with visit-sequence graphs produced from any l-OAG.

2.4 Summary of Notation

miscellaneous notation:

$<i_0 \ldots i_{n-1}>$	ordered sequence of items,
$s[j]$	j^{th} item in sequence s, with the first item in s denoted by s[0],
$s_1 \| s_2$	concatenation of sequences s_1 and s_2, and
G / V	projection of graph G onto vertex set V

abbreviations:

AG	attribute grammar,
l-OAG	l-ordered attribute grammar,
OAG	ordered attribute grammar,
SAG	S-attributed attribute grammar, and
VS graph	visit-sequence graph

attribute grammar notation:

G = (N,T,P,S)	context-free grammar G, with set of nonterminal symbols N, set of terminal symbols T, set of productions P, and starting nonterminal S,
$p: X_0 \rightarrow X_1 \ldots X_n$	production p, with left-hand nonterminal X_0 and right-hand side $X_1 \ldots X_n$,

G=(N,T,P,S,AI,AS,R)	attributed context-free grammar G, with set of nonterminal symbols N, set of terminal symbols T, set of productions P, starting nonterminal S, inherited attribute.mapping AI, synthesized attribute mapping AS, and semantic equation mapping R.
X.b	attribute b of nonterminal X,
X$n	n^{th} occurrence of nonterminal X in a given production,
$X_j.b = f(\ldots, Y.c, \ldots)$	semantic equation defining the value of output attribute occurrence $X_j.b$ in terms of a function of (at least) the value of attribute occurrence Y.c,
A(X)	set of attributes of nonterminal X,
AI(X)	set of inherited attributes of nonterminal X,
AS(X)	set of synthesized attributes of nonterminal X,
DP(p)	local dependency graph for production p,
DT(T)	tree dependency graph for tree T,
In(p)	set of input attribute occurrences of production p, and
Out(p)	set of output attribute occurrences of production p

l-OAG and/or OAG notation:

$A_j(X)$	j^{th} partition of attributes of nonterminal X,
DS(X)	dependency graph for nonterminal X,

EDP(p)	extended dependency graph for production p,
IDP(p)	induced dependency graph for production p,
IDS(X)	induced dependency graph for nonterminal X, and
VS(p)	visit-sequence graph for production p,

static evaluation order plan elements:

EVAL	determine the value of a specified attribute instance of a specified tree node by using the appropriate semantic equation,
SKIP	a no-op,
VISIT_CHILD	descend to a specified child of the current tree node to continue attribution, and
VISIT_PARENT	ascend to the parent of the current tree node to continue attribution

3. Parallel Computation

A variety of models and architectures for parallel computation have been developed [Duncan 1990, Fortune & Wyllie 1978, Snir 1982]. While the work to be presented here is not strictly dependent on any particular underlying model or architecture, it is most convenient to view the work within the framework of a *shared memory* model of parallel computation.

3.1 Shared Memory Parallel Architecture

A shared memory MIMD processor [Flynn 1966] can be thought of as a group of processors with a common memory accessible to each of the processors, as depicted in Figure 3.1. The common memory may in fact be

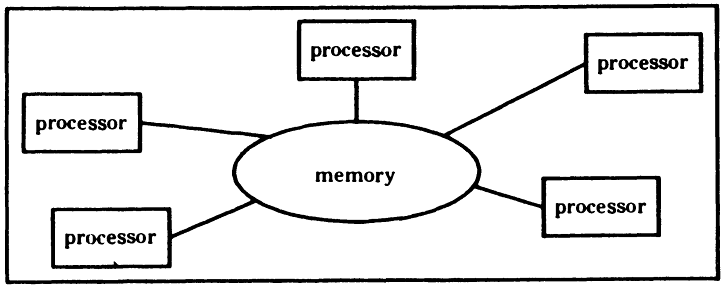

Figure 3.1 - A Shared Memory MIMD Processor

implemented in any of a variety of ways. It may be implemented, for example, with a shared bus structure (like that shown in Figure 3.2), with multi-ported memory modules (Figure 3.3), or with a processor-memory interconnection network (Figure 3.4).

Irrespective of the actual hardware realization, we require that the shared-memory MIMD processor together with its operating system support several features. First, there must appear to be a common memory

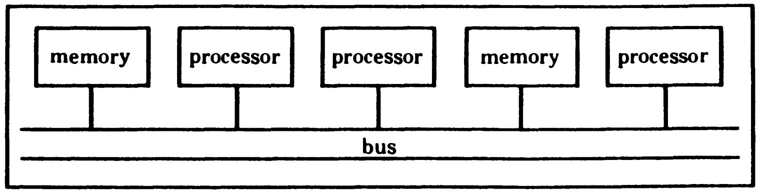

Figure 3.2 - A Shared Bus Architecture

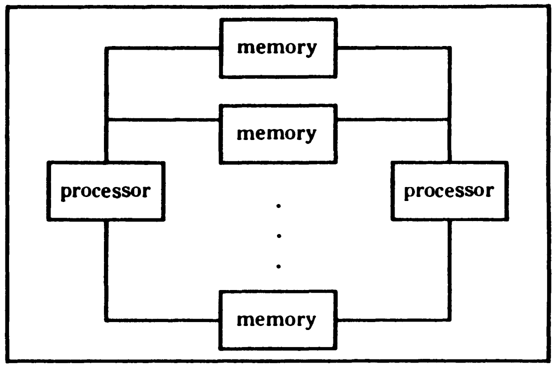

Figure 3.3 - A Multi-Ported Memory Architecture

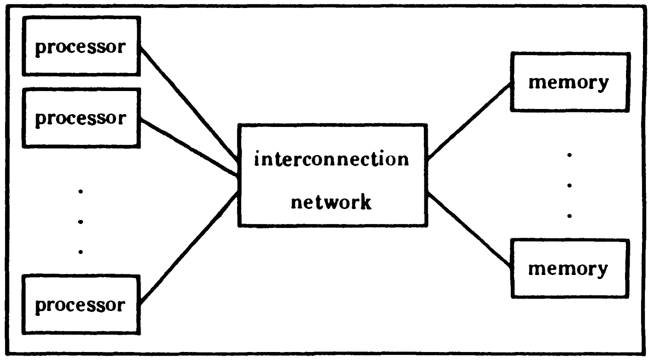

Figure 3.4 - An Interconnection Network Architecture

address space. That is, a given cell in shared memory must apparently have a single address used by all processors to refer to that cell. All processors must appear to have direct access to all cells in the shared memory.

Second, the system must support a number of *process synchronization operations* [Almasi & Gottlieb 1989]. In particular, we are interested in the following operations on shared memory cells:

lock(c)	Set the contents of memory cell c to a non-zero value.
wait(c)	If memory cell c contains a non-zero value, suspend the current process; otherwise, do nothing.
unlock(c)	Set the contents of memory cell c to zero and awaken any processes that are suspended having done a wait(c).
atomic_set(c)	Set the contents of memory cell c to the Boolean value true and return the previous value in c, all as a single atomic operation.
atomic_add(c_1, c_2)	Add the contents of memory cell c_2 to the contents of memory cell c_1, leave the sum in cell c_1, and return the new value in c_1, all as a single atomic operation.
atomic_add1(c)	Add one to contents of memory cell c and return the new value in c, all as a single atomic operation.

atomic_sub1(c)	Subtract one from the contents of memory cell c and return the new value in c, all as an atomic operation.
atomic_monus1(c)	If the initial value of memory cell c is positive, subtract one from the contents of c and return the new value in c, all as an atomic operation; otherwise, set c to zero and return zero, all as an atomic operation.

It is not required that all these operations be directly supported by the hardware. On typical MIMD systems it is often possible to easily simulate some or all of these operations through software. However, software simulation of certain of these operations may incur unacceptably large space or time overhead on some systems.

3.2 Styles of Parallel Algorithms

The algorithms we will develop will be categorized by the style of parallelism they exhibit. We will broadly categorize parallelism as being either *asynchronous* or *synchronous*.

3.2.1 Asynchronous Parallelism

Asynchronous parallel algorithms achieve parallelism by the simultaneous execution of *asynchronous processes*. These processes typically execute independently of one another. On occasion, however, a process may find that before it can continue it must wait for another process to complete some task. That is, the first process must temporarily *synchronize* its execution with the other process. The various synchronization operations described in §3.1 may be used for this.

In addition, an asynchronous process may find that it must itself create a new asynchronous process responsible for carrying out some task. The specific details of process creation may vary significantly across different implementations, so we introduce a high-level language construct for asynchronous process creation.

The statement

spawn S

is assumed to cause the creation of an asynchronous process that will eventually perform statement S and then terminate. (Statement S will often be a procedure call.) The spawned process is given private copies of variables whose values are used in S and references to variables which may be side-effected by S.

The execution of the spawn statement itself is complete as soon as the asynchronous process has been created: the spawning process does not wait for the execution of the newly-created process to finish. It is assumed that the spawning process may, if necessary, detect whether any (directly or indirectly) spawned processes that have yet to be completed.

An example of an asynchronous parallel algorithm is presented in Figure 3.5. This algorithm performs dynamic order tree attribution in an asynchronous parallel manner.

3.2.2 Synchronous Parallelism

Synchronous parallel algorithms also achieve parallelism by the concurrent execution of processes, but the individual processes are more


```
Evaluate(tree T):
    (V, E) := DT(T);
    forall b∈ V s.t. indegree(b) = 0 do
        spawn EvaluateAttribute(b)
        od;
    wait until all spawned asynchronous processes have terminated
    end Evaluate;

EvaluateAttribute(attribute_instance b):
    value(b) := value defined by appropriate semantic fn;
    forall (b, c)∈ E do
        if atomic_sub1(indegree(c)) = 0
          spawn EvaluateAttribute(c)
          fi
        od
    end EvaluateAttribute;
```

Figure 3.5 - An Asynchronous Parallel Dynamic Order Evaluator

highly synchronized with one another. Typically, at some stage in the execution of a synchronous algorithm, a process will determine that some number of tasks should be executed concurrently. The process will initiate the concurrent execution of the tasks and continue on itself only when all the tasks have been completed.

The low-level synchronization operations that allow a process to wait for the completion of all the tasks mentioned above are typically hidden within high-level-language constructs (e.g., cobegin-coend [Dijkstra 1968]). Parallelism implemented using such constructs is less general and has a more static structure than does asynchronous parallelism. Conversely, synchronous algorithms can be easier to construct since low-level synchronization need not be considered explicitly in the algorithm.

In the algorithms we develop, we will use the statement

pforall *binding_expression* do

S_1; S_2; . . . ; S_m

od

to implement synchronous parallelism. *binding_expression* will define a set of n values for some variable. The body of the pforall (i.e., the statement list S_1; S_2; . . . ; S_m) will be executed once for each of the n values. All n executions of the body may proceed concurrently. The execution of the pforall statement itself is complete only when all n executions of the body are complete.

Figure 3.6 shows a synchronous parallel version of the dynamic order attribution algorithm. It should be noted that the dynamic order algorithm

```
Evaluate(tree T):
    (V, E) := DT(T);
    W := { b | b∈V & indegree(b) = 0 };
    U := Ø;
    while W = Ø do
        pforall b∈W do
            value(b) := value defined by appropriate semantic fn;
            foreach (b, c)∈E do
                if atomic_sub1(indegree(c)) = 0
                    U := U ∪ { c }
                    fi
                od
            od;
        W := U
        od
    end Evaluate;
```

Figure 3.6 - A Synchronous Parallel Dynamic Order Evaluator

does not naturally fit the synchronous model of parallelism. An attribute instance b may be forced to wait, directly or indirectly, for the value of some attribute instance c to be determined even though b does not depend, directly or indirectly, on the value of c. This synchronous version is presented largely to allow comparison with the asynchronous algorithm of Figure 3.5.

4. Asynchronous Parallel Attribution

In §3.2.1 and §3.2.2 we presented parallel dynamic order attribution algorithms. It is also possible to develop parallel static order attribution algorithms. In this chapter we develop asynchronous-style parallel attribution schemes for trees derived from l-ordered attribute grammars.

We begin by discussing basic issues in synchronizing processes using locks. We continue by developing attribution algorithms, first considering only the asynchronous execution of EVAL elements and then considering the asynchronous execution of other element types. We conclude with a discussion of practical aspects of asynchronous algorithms and a consideration of related research.

4.1 Basic Issues

4.1.1 Synchronization Using Attribute Instance Locking

We can exploit parallelism in attributing a tree derived using an l OAG by allowing the execution of certain plan elements to overlap in time. In particular, we need not always wait for the execution of one plan element to be completed before beginning the execution of another. An attribution scheme can accomplish this by spawning asynchronous processes to execute plan elements. However, some care must be taken to synchronize processes in certain situations.

Suppose the value of attribute instance b is used in determining the value of attribute instance c. Further suppose that concurrently executing processes q_1 and q_2 are evaluating the semantic functions defining values for b and c, respectively. Since we cannot, in general, make any guarantees regarding the relative speeds at which q_1 and q_2 execute, q_2 may make

progress faster than q_1. In particular, the semantic function defining the value of c may attempt to read the value of b before q_1 has finished determining b's value. We must prevent this (and other similar situations) and will do so through locking protocols on attribute instances.

We make use of the following locking operations:

lockAttrInst(n.b)	places a lock on attribute instance b of tree node n
unlockAttrInst(n.b)	removes the lock.

In the asynchronous algorithms we present, certain attribute instances are locked at a point in the attribution process before any other process attempts to use their respective values. An instance is not unlocked until its value has been determined. Semantic functions attempting to read the value of a locked attribute instance suspend execution and wait until the instance is unlocked before continuing on. The act of unlocking an attribute instance must notify any processes waiting for the value of that instance that they may proceed. While many processes may attempt to read the value of an instance, only one process will place a lock on the instance, and only one process will unlock the instance.

4.1.2 Semantic Function Strictness

The above protocol allows (continuing the example from §4.1.1) processes q_1 and q_2 to execute concurrently but, correctly, does not allow q_2 to get too far ahead of q_1 and attempt to use the value of b prematurely. The protocol also allows the attribution process to take advantage of certain cases of non-strictness by requiring an attribute evaluation process to wait only if it actually attempts to read a locked attribute instance

In the context of tree attribution, if a semantic function is strict in its arguments, it can only be evaluated after all its argument attribute instances have been given values and been unlocked. This has the advantage of requiring a semantic function to examine each argument attribute instance's lock exactly once per invocation.

If a semantic function is non-strict, it may begin work immediately upon being invoked, querying the lock status of an argument attribute instance only if and when the value of that instance is actually needed. This has the possible advantage of increasing the amount of concurrency. However, this requires, in the worst possible case, that a semantic function query the lock status of an argument attribute instance each time the value of that instance is used in the body of the function.

If a semantic function is not strict and, in the course of determining a value for an attribute instance b, finds that it need not read the value of some attribute instance c, the attribute evaluation process need not wait for c's value to be determined, regardless of any statically-determined dependencies of b on c. A common way this situation can arise is when *conditional expressions* are allowed in semantic functions.

We consider conditional expressions of the form

$$expression_1 \ ? \ expression_2 \ : \ expression_3$$

where $expression_1$ must produce a Boolean value. The value of a conditional expression is determined in the following manner. First, $expression_1$ is evaluated. If $expression_1$ has a true value, $expression_2$ is evaluated and its value returned as the value of the conditional expression. If the value of $expression_1$ is instead false, $expression_3$ is evaluated and its value returned as the value of the conditional expression. Only one of

$expression_2$ and $expression_3$ will be evaluated for each evaluation of the conditional expression.

Consider the production and associated semantic equation

$$p\colon W \rightarrow X\ Y\ Z \ \ \{ W.b = X.c > 0 \ ? \ Y.d : Z.e \}$$

EDP(p) would contain both an edge from Y.d to W.b and an edge from Z.e to W.b. The plan constructed for p would take these dependencies into account. An asynchronous process evaluating some instance of W.b might have to wait for the value of the appropriate instance of either Y.d or Z.e, but would never have to wait for the values of *both* Y.d and Z.e.

The algorithms presented in the remainder of this chapter work correctly for either strict or non-strict semantic functions. The matter of strictness will be considered once again in Chapter 6, where it will assume greater importance.

4.1.3 Asynchronous Attribution and Static Order Attribution

It might be argued that the locking regimen discussed above largely negates the alleged time and space advantages of using a static order attribution scheme over a dynamic order scheme. Lock setting and querying appears to lead to an attribution style quite similar in practice to dynamic order attribution based on the tree dependency graph.

The primary reason for our staying within the static order framework of l-OAG's is that it easily extends to permit efficient incremental attribution [Reps 1984, Yeh 1983], as discussed in Chapter 6. Other reasons include that the order in which processes are spawned by a static order asynchronous parallel attribution algorithm is necessarily reasonable (in the sense that processes are always spawned in an order that makes it possible that few semantic function invocations will ever be blocked awaiting unlocking of attribute instances) and that not all of the space and time advantages of static evaluation order schemes are lost.

4.1.4 Development and Presentation of Attribution Algorithms

Our approach is to use the plans generated for the non-parallel (or *serial*) attribution routine and, as mentioned above, to spawn processes to execute various plan elements. We will examine the asynchronous execution of EVAL, VISIT_CHILD, and VISIT_PARENT elements individually. Then, we will present attribution algorithms combining the asynchronous execution of different types of these elements.

We will adopt a naming convention for attribution procedures that indicates the type of element(s) being performed in a concurrent manner. Parallel attribution algorithm names ending in the letters "E," "C," and "P" respectively perform EVAL, VISIT_CHILD, and VISIT_PARENT elements in a parallel fashion. An attribution algorithm whose name ends in a combination of these letters performs the corresponding collection of plan elements types using parallel techniques. For example, the procedure EvaluateAsynchECP performs all types of plan elements in parallel.

We assume that if a given type of plan element is to be executed asynchronously, then all elements of that type in all plans will be executed asynchronously. Later, we will discuss relaxing this assumption.

4.2 Asynchronous Execution of EVAL Elements

Perhaps the most obvious approach to asynchronous parallel attribution is to execute EVAL elements with asynchronous processes.

Each time such an attribution algorithm encounters an EVAL element, it locks the attribute instance in question, spawns a process to perform the semantic function evaluation (and subsequent value assignment to and unlocking of the attribute instance), and then continues on to the execution of the next element in the plan. Figure 4.1 gives a modified version of the serial attribution algorithm of Figure 2.16 that does this. To attribute tree T, one would perform

EvaluateAsynchE(*root of* T, 0);

wait for all asynchronous processes to terminate

We assume for all the asynchronous algorithms that the code for (possibly) waiting on the value of a locked attribute instance is contained within the code for the semantic function.

A question that must be addressed concerning this algorithm (and subsequent asynchronous algorithms) is whether or not the individual attribute instances are locked and unlocked correctly. That is, for every derivation tree T, do all attribute instances get locked before any semantic functions attempt to read their respective values, and are all attribute instances eventually unlocked, allowing complete attribution of the tree?

EvaluateAsynchE encounters plan elements in exactly the same order as does the serial attribution routine FAEvaluate (Figure 2.16). Thus, when the EVAL element responsible for determining the value of an attribute instance b is encountered, no semantic function using the value of b can yet have been evaluated (since the plans were constructed so that FAEvaluate would honor any dependencies in DT(T)). Therefore, locking b when its respective EVAL element is encountered assures that no semantic

```
EvaluateAsynchE(treenode n, integer i):
    while PLANS[prod_num(n)][i] ≠ [VISIT_PARENT *]
            | n ≠ root of tree do
        case PLANS[prod_num(n)][i] of
            [EVAL j b]: {
                lockAttrInst(child(j, n).b);
                spawn EvaluateAttribute(n, i);
                i := i + 1
                };
            [VISIT_CHILD j k]: {
                n := child(j, n);
                i := MAPDOWN(k, prod_num(n))
                };
            [VISIT_PARENT k]: {
                i := MAPUP(k, child_num(n), prod_num(parent(n)));
                n := parent(n)
                }
            esac
        od
    end EvaluateAsynchE;

EvaluateAttribute(treenode n, integer i):
    [EVAL j b] := PLANS[prod_num(n)][i];
    value(child(j, n).b) := value defined by appropriate semantic fn;
    unlockAttrInst(child(j, n).b)
    end EvaluateAttribute;
```

Figure 4.1 - An Asynchronous EVAL Attribution Algorithm

function will attempt to read b's value before b has been locked. The same argument holds for any attribute instance in T.

After b is locked, EvaluateAttribute is called, and b is unlocked after its defining semantic function determines the value for b. Assuming

that all semantic functions are total, b's defining function could fail to eventually find a value for b only if it and at least one other semantic function invocation are mutually deadlocked, each function invocation blocked awaiting the unlocking of the instance the other is trying to determine the value for. Such deadlock cannot occur, however, since it would imply a circularity in DT(T), and all trees derived from I OAG's are noncircular.

4.3 Asynchronous Execution of VISIT_CHILD Elements

A different approach to asynchronous parallel attribution is to execute VISIT_CHILD plan elements asynchronously. Each time the attribution routine encounters a VISIT_CHILD element, the routine can spawn a process to perform the work to be done by that element and then proceed. If the attribution routine were currently focused on an element [VISIT_CHILD j k] in the plan associated with node n, the work to be done by that element corresponds to the execution of plan elements e_h through e_i of the plan for the j^{th} child of n where

$< \ldots e_h \ldots e_i \ldots > =$ the plan for child(j, n)

p = prod_num(child(j, n))

h = MAPDOWN(k, p)

i = the index of the element [VISIT_PARENT k] in the plan for the production numbered p

Execution of the VISIT_PARENT element with index i would cause termination of the asynchronous process.

We must also develop a locking strategy. In this case, things are complicated by the fact that plan elements may no longer be encountered in the same chronological order as they were during serial attribution. In particular, with respect to a given tree node, the j^{th} element of a plan may be executed before any or all of elements 0 through $j-1$ of that plan. Incorrectly handled, such a situation could lead to attribute instance values being used before those values are actually calculated. This will be illustrated by the grammar of Figure 4.2 (the plans for which appear in

```
AI(S) = ∅, AS(S) = { ss1 }
AI(Z) = ∅, AS(Z) = ∅
AI(X) = { xi1, xi2 }, AS(X) = { xs1, xs2 }
AI(Y) = { yi1, yi2 }, AS(Y) = { ys1, ys2 }

p0:     S → X Z
             {  X.xi1 = 0;
                X.xi2 = X.xs1;
                S.ss1 = X.xs2 }
p1:     S → X
             {  X.xi1 = 0;
                X.xi2 = 0;
                S.ss1 = X.xs2 }
p2:     X → Y
             {  Y.yi1 = X.xi1;
                X.xs1 = Y.ys1;
                Y.yi2 = X.xi2;
                X.xs2 = Yy.s1 + Y.ys2 + X.xs1 + Y.yi1 }
p3:     Y → '0'
             {  Y.ys1 = Y.yi1;
                Y.ys2 = Y.yi1 + Y.yi2 }
p4:     Z → '1'
```

Figure 4.2 - An Example Grammar

Figure 4.3, with subscripts appended to each element for reference) and the

<< [EVAL 1 xi1]$_0$ [VISIT_CHILD 1 0]$_1$ [EVAL 1 xi2]$_2$
[VISIT_CHILD 1 1]$_3$ [EVAL 0 ss1]$_4$ [VISIT_PARENT 0]$_5$ >
< [EVAL 1 xi1]$_6$ [VISIT_CHILD 1 0]$_7$ [EVAL 1 xi2]$_8$
[VISIT_CHILD 1 1]$_9$ [EVAL 0 ss1]$_{10}$ [VISIT_PARENT 0]$_{11}$ >
< [EVAL 1 yi1]$_{12}$ [VISIT_CHILD 1 0]$_{13}$ [EVAL 0 xs1]$_{14}$
[VISIT_PARENT 0]$_{15}$ [EVAL 1 yi2]$_{16}$ [VISIT_CHILD 1 1]$_{17}$
[EVAL 0 xs2]$_{18}$ [VISIT_PARENT 1]$_{19}$ >
< [EVAL 0 ys1]$_{20}$ [VISIT_PARENT 0]$_{21}$ [EVAL 0 ys2]$_{22}$
[VISIT_PARENT 1]$_{23}$ >
<>
>

Figure 4.3 - Plans for the Example Grammar

derivation tree shown in Figure 4.4.

```
n0:  S
     |
n1:  X
     |
n2:  Y
     |
n3:  '0'
```

Figure 4.4 - A Derivation Tree

We wish to attribute this tree with VISIT_CHILD elements being done asynchronously. Suppose, in the course of attribution, the plan for node n_0 is being executed by some process q_0. q_0 would first execute the [EVAL 1 xi1]$_6$ element, thus evaluating n_1.xi1. q_0 would then encounter the [VISIT_CHILD 1 0]$_7$ element and spawn a process (call it q_1) to perform the visit. q_0 would then move on to [EVAL 1 xi2]$_8$. Supposing that q_1 has not yet begun to execute. Figures 4.5 and 4.6 show the state of

process / node	previous element executed	element being executed	next element to execute
q_0 / n_0	$[\text{VISIT\_CHILD 1 0}]_7$	$[\text{EVAL 0 xi2}]_8$	$[\text{VISIT\_CHILD 1 1}]_9$
q_1 / n_1	---	---	$[\text{EVAL 1 yi1}]_{12}$

Figure 4.5 - Process State of Tree Attribution

n_0.ss1:	unevaluated		
n_1.xi1:	evaluated	n_2.yi1:	unevaluated
n_1.xi2:	under evaluation	n_2.yi2:	unevaluated
n_1.xs1:	unevaluated	n_2.ys1:	unevaluated
n_1.xs2:	unevaluated	n_2.ys2:	unevaluated

Figure 4.6 - Attribute Instance State of Tree Attribution

the attribution at this point.

The semantic function defining n_1.xi2 in this case is the constant function 0, so evaluation of n_1.xi2 depends on no attribute instance values and therefore need not wait on q_1, irrespective of any locking protocol. q_0 would then encounter $[\text{VISIT\_CHILD 1 1}]_9$, spawn a process (call it q_2) to perform this visit, and move on to $[\text{EVAL 0 ss1}]_{10}$. Supposing that q_2 has not yet begun to execute and that q_1 still has not yet begun to execute, Figures 4.7 and 4.8 show the state of the attribution at this point.

Suppose that q_2 begins to execute while q_1 remains "stuck." q_2 first encounters and executes $[\text{EVAL 1 yi2}]_{16}$. Since n_2.yi2 depends only on the already evaluated n_1.xi2, the element can be executed with no waiting. q_2

process / node	previous element executed	element being executed	next element to execute
q_0 / n_0	$[\text{VISIT\_CHILD 1 1}]_9$	$[\text{EVAL 0 ss1}]_{10}$	$[\text{VISIT\_PARENT 0}]_{11}$
q_1 / n_1	---	---	$[\text{EVAL 1 yi1}]_{12}$
q_2 / n_1	---	---	$[\text{EVAL 1 yi2}]_{16}$

Figure 4.7 - Process State of Tree Attribution

n_0.ss1:	under evaluation		
n_1.xi1:	evaluated	n_2.yi1:	unevaluated
n_1.xi2:	evaluated	n_2.yi2:	unevaluated
n_1.xs1:	unevaluated	n_2.ys1:	unevaluated
n_1.xs2:	unevaluated	n_2.ys2:	unevaluated

Figure 4.8 - Attribute Instance State of Tree Attribution

then encounters $[\text{VISIT\_CHILD 1 1}]_{17}$, spawns process q_3 to perform the visit, and moves on to the $[\text{EVAL 0 xs2}]_{18}$ element.

Now suppose that, with q_1 still not having ever executed, q_3 starts executing. q_3 begins to execute $[\text{EVAL 0 ys2}]_{22}$ in order to determine the value of n_2.ys2. Figures 4.9 and 4.10 show the state of the attribution at this point.

The plan elements for nodes n_1 and n_2 are not being encountered or executed consecutively from the beginning of their respective plans as they would be under serial attribution. At this point, q_3 cannot be allowed to invoke the semantic function for n_2.ys2 since the value of n_2.yi1 has not yet been determined. Similarly, if q_2 proceeds with execution of

process / node	previous element executed	element being executed	next element to execute
q_0/n_0	[VISIT_CHILD 1 1]$_9$	[EVAL 0 ss1]$_{10}$	[VISIT_PARENT 0]$_{11}$
q_1/n_1	---	---	[EVAL 1 yi1]$_{12}$
q_2/n_1	[VISIT_CHILD 1 1]$_{17}$	[EVAL 0 xs2]$_{18}$	[VISIT_PARENT 1]$_{19}$
q_3/n_2	---	[EVAL 0 ys2]$_{22}$	[VISIT_PARENT 1]$_{23}$

Figure 4.9 - Process State of Tree Attribution

n_0.ss1:	under evaluation		
n_1.xi1:	evaluated	n_2.yi1:	unevaluated
n_1.xi2:	evaluated	n_2.yi2:	evaluated
n_1.xs1:	unevaluated	n_2.ys1:	unevaluated
n_1.xs2:	under evaluation	n_2.ys2:	under evaluation

Figure 4.10 - Attribute Instance State of Tree Attribution

[EVAL 0 xs2]$_{18}$, the semantic function defining n_1.xs2 cannot yet be invoked since the values of n_2.ys1, n_1.xs1, and n_2.yi1 have yet to be determined. Whatever locking strategy we adopt must prevent premature semantic function invocation.

The above example illustrates that if we execute the j^{th} element of a plan for a node derived from some production p before executing one or more of elements 0 through $j-1$ of the plan, attribute instances that are given values due to the execution to one of elements 0 through $j-1$ may not yet have been given values. To prevent the premature use of these instances, they must be locked prior to the execution of the j^{th} element. A simple way

to guarantee this in general is to lock every attribute instance in the derivation tree prior to the start of attribution.

Figure 4.11 gives an obvious algorithm for locking a derivation tree in an asynchronous parallel manner. To lock tree T, one calls lockTree on

```
lockTree(treenode n):
    forall attribute instances b of n do
        lockAttrInst(n.b)
        od;
    forall children m of n do
        spawn lockTree(m)
        od
    end lockTree;
```

Figure 4.11 - An Asynchronous Tree Locking Algorithm

the root node of T and wait for all spawned processes to terminate.

The use of a preceding locking phase would also have worked for EvaluateAsynchE. However, when possible, the use of a separate locking phase is best avoided since this technique does not extend well to allow optimal incremental re-attribution (discussed in Chapter 6). The effectively identical technique of originally constructing the derivation tree with all attribute instances locked suffers from the same drawback.

Having first locked all attribute instances in a tree, the algorithm of Figure 4.12 can be used to attribute the tree. This algorithm is again a modification of the serial LOAG attribution routine. To attribute tree T one would use the code

lockTree(*root of* T);

EvaluateAsynchC(*root of* T, 0);

wait for all asynchronous processes to terminate

```
EvaluateAsynchC(treenode n, integer i) :
    while PLANS[prod_num(n)][i] ≠ [VISIT_PARENT *] do
        case PLANS[prod_num(n)][i] of
            [EVAL j b] : {
                value(child(j, n).b) :=
                    value defined by appropriate semantic fn;
                unlockAttrInst(child(j, n).b);
                i := i + 1
                };
            [VISIT_CHILD j k] : {
                spawn EvaluateAsynchC(
                        child(j, n),
                        MAPDOWN(k, prod_num(child(j, n))));
                i := i + 1
                }
            esac
        od
    end EvaluateAsynchC;
```

Figure 4.12 - An Asynchronous VISIT_CHILD Attribution Algorithm

For an additional cost of one bit per tree node, we can avoid the use of a preceding locking phase. Each node has a bit "visited?" initially set to zero. Each time a VISIT_CHILD element is executed, the value of this bit for the child node in question is checked. If it is zero (indicating that this will be the chronologically first visit to the child node), all attribute instances of each child node of the child node to be visited are locked. If the bit is instead one, no locking is done.

This gives rise to the algorithm of Figure 4.13, which assumes the existence of a procedure

lockChildAttrInsts(n)


```
EvaluateAsynchC(treenode n, integer i) :
    while PLANS[prod_num(n)][i] ≠ [VISIT_PARENT *] do
        case PLANS[prod_num(n)][i] of
            [EVAL j b] : {
                value(child(j, n).b) :=
                    value defined by appropriate semantic fn;
                unlockAttrInst(child(j, n).b);
                i := i + 1
                };
            [VISIT_CHILD j k] : {
                if ¬atomic_set(visited?(child(j, n)))
                    lockChildAttrInsts(child(j, n))
                    fi;
                spawn EvaluateAsynchC(
                        child(j, n),
                        MAPDOWN(k, prod_num(child(j, n))));
                i := i + 1
                }
            esac
        od
    end EvaluateAsynchC;
```

Figure 4.13 - An Asynchronous VISIT_CHILD Attribution Algorithm

that locks all attribute instances of each child of node n. (lockChildAttrInsts can be done with the aid of a mapping determined during plan generation or by interpreting the respective plans for n and its children.) Assuming that the tree has been built with visited? set to zero for each node, the code lockChildAttrInsts(*root of* T);

EvaluateAsynchC(*root of* T, 0);

wait for all asynchronous processes to terminate

may be used to attribute a tree.

4.4 Asynchronous Execution of VISIT_PARENT Elements

We can also consider performing tree attribution with VISIT_PARENT plan elements executed asynchronously. This method of attribution is, in essence, a dual to the method employing asynchronous execution of VISIT_CHILD elements.

Each time a VISIT_PARENT element is encountered during attribution, an asynchronous process is spawned to perform the work done during that visit to the current tree node's parent. Determining the work to be done by a given VISIT_PARENT element in the plan for some node is less straightforward than it was in the case of asynchronous VISIT_CHILD elements.

In serial attribution, if the current element is [VISIT_PARENT k] in the plan for node n, the work performed on that visit corresponds to the execution of plan elements e_h through e_i in the plan for n's parent, where

$< \ldots e_h \ldots e_i \ldots >$ = the plan for parent(n)

p = prod_num(parent(n))

j = child_num(n)

h = MAPUP(k, j, p)

i = the index of the element [VISIT_CHILD j k + 1], if it exists, otherwise the index of the last element in production p's plan.

However, by this definition, for a particular node, a single plan element may be claimed as part of the work done by VISIT_PARENT elements in the respective plans for more than one node, as illustrated by the following example.

Consider the following fragment of the plan for some node n:

...[VISIT_CHILD 1 4][EVAL 2 b][VISIT_CHILD 2 7]
[EVAL 1 c][VISIT_CHILD 1 5][EVAL 2 d]
[VISIT_CHILD 2 8]...

By the definition above, the elements [EVAL 2 b], [VISIT_CHILD 2 7], [EVAL 1 c], and [VISIT_CHILD 1 5] are part of the work done by the [VISIT_PARENT 4] element in the plan for the first child of n. However, the elements [EVAL 1 c] and [VISIT_CHILD 1 5] are also considered part of the work done by the [VISIT_PARENT 8] element in the plan for the second child of n. In the serial case, this causes no problems, since there is only a single process performing plan execution. In the case of multiple plan execution processes, for efficiency, we should insure that each plan element is part of the work assigned to exactly one VISIT_PARENT element execution.

We redefine the work to be done by the execution of a [VISIT_PARENT k] element in the plan for node n to be the execution of elements e_h through e_i in the plan for the parent of n, where

$< \ldots e_h \ldots e_i \ldots >$ = the plan for parent(n)

p = prod_num(parent(n))

j = child_num(n)

h = MAPUP(k, j, p)

i = the index of the next VISIT_CHILD element with index · h
in the plan for production p, if such an element exists;
the index of the last element in the plan, if no such element
exists.

Using this definition and returning to the previous example, the [VISIT_PARENT 4] element of the first child of n now claims only the elements [EVAL 2 b] and [VISIT_CHILD 2 7], while the [VISIT_PARENT 8] element of the second child of n claims only [EVAL 1 c] and [VISIT_CHILD 1 5].

An asynchronous process spawned to perform the work of a VISIT_PARENT terminates upon encountering a VISIT_CHILD element. Upon encountering the final element in a plan (always a VISIT_PARENT element), the process must either spawn a process to do the work of that element and then terminate or do the work of that element itself (thus saving the cost of a process creation). Since it is convenient to be able to easily distinguish the final VISIT_PARENT in a plan, we will terminate all plans by a special VISIT_PARENT element of the form [LAST_VISIT_PARENT k]. Note that execution of the LAST_VISIT_PARENT element in the plan for the root node must not cause a process to be spawned.

While the 1-OAG attribution algorithms thus far presented have attributed trees in a top-down fashion, the asynchronous execution of VISIT_PARENT elements leads to a bottom-up style of attribution. Under asynchronous VISIT_PARENT execution, upon encountering such an element, a process is spawned to execute a portion of the parent node's plan. Process spawning and, hence, attribute evaluation tend to proceed from the

frontier of the tree up to the root. Bottom-up attribution schemes require that slightly different plans be generated [Kastens 1980].

Again we must consider a locking strategy. There are similarities between the serial execution of VISIT_CHILD and VISIT_PARENT elements. Noting this, we can construct grammars analogous to that of Figure 4.2 in which the asynchronous execution of VISIT_PARENT elements can lead to premature attribute instance use that can be prevented by locking the entire derivation tree prior to the start of attribution. Once again, the use of this method will preclude the easy extension of this particular attribution scheme to handle incremental re-attribution.

Figure 4.14 presents an attribution algorithm based on the preceding discussion. To use the algorithm, one would execute the code

lockTree(*root of* T);
foreach *leaf node* n *of* T do
    spawn EvaluateAsynchP(n, 0)
    od;
wait for all asynchronous processes to terminate

A reasonable implementation of this algorithm would require fast access to the leaves of derivation trees. As with EvaluateAsynchC, visited? bits could be used to eliminate the need for the preliminary locking phase.

4.5 Asynchronous Execution of Visit Elements

It is also possible to combine the asynchronous execution of more than one type of plan element within a single attribution algorithm. A particularly interesting algorithm results from considering the

```
EvaluateAsynchP(treenode n, integer i):
   while PLANS[prod_num(n)][i] ≠ [VISIT_CHILD * *] do
      case PLANS[prod_num(n)][i] of
         [EVAL j b]: {
            value(child(j, n).b) :=
               value defined by appropriate semantic fn;
            unlockAttrInst(child(j, n).b);
            i := i + 1
            };
         [VISIT_PARENT k]: {
            spawn EvaluateAsynchP(
                  parent(n),
                  MAPUP(k, child_num(n), prod_num(parent(n))));
            i := i + 1
            };
         [LAST_VISIT_PARENT k]: {
            spawn EvaluateAsynchP(
                  parent(n),
                  MAPUP(k, child_num(n), prod_num(parent(n))));
            return
            }
         esac
      od
   end EvaluateAsynchP;
```

Figure 4.14 - An Asynchronous VISIT_PARENT Attribution Algorithm

asynchronous execution of both VISIT_CHILD and VISIT_PARENT elements.

When executing VISIT_CHILD elements asynchronously, a process engaged in attribution does not halt upon encountering a VISIT_CHILD element. Rather, it spawns a process to handle the child visit and goes on with the execution of the plan for the current node. Likewise, the

asynchronous execution of a VISIT_PARENT element causes the spawning of a process to perform the parent visit, but does not cause termination of the process that encountered the VISIT_PARENT. Combining these two characteristics leads to a simple attribution scheme.

The execution of an element of the form [VISIT_CHILD j 0] (i.e., the initial visit to the j^{th} child of the current node) causes the spawning of a process to execute the plan of the j^{th} child. All other elements of the form [VISIT_CHILD j k], where $k > 0$, are treated as no-ops. All VISIT_PARENT elements other than VISIT_PARENT elements terminating plans will also act as no-ops. The execution of a plan-terminating VISIT_PARENT element (again to be represented with a LAST_VISIT_PARENT element) causes termination of the process encountering that element.

This approach to plan execution results in a one-to-one correspondence between tree nodes and plan execution processes. Once spawned, a process executes every element in a node's plan before terminating. This means that, as in EvaluateAsynchE and unlike in EvaluateAsynchC and EvaluateAsynchP, all elements in the plan for each node are initiated in order from first to last chronologically. With respect to any single node, the corresponding plan's elements are encountered in the same order as by the serial l-OAG attribution algorithm. This behavior means that complete locking of the tree prior to the start of attribution is certainly not necessary.

Locking is not the same is as done by EvaluateAsynchE, however. Suppose that, as a result of encountering a [VISIT_CHILD j 0] element in the plan for node n, process q_1 spawns process q_2. q_1 then may race ahead of

q_2 and attempt to use the value of some synthesized attribute instance that q_2 has yet to determine. Conversely, q_2 may progress more quickly than q_1 and attempt to use some inherited attribute instance value before q_1 has produced it. Both of these situations must be prevented.

To prevent q_1 from using synthesized values before q_2 calculates them, prior to spawning q_2, q_1 should lock all synthesized attribute instances that q_2 will eventually evaluate. Similarly, to prevent q_2 from using inherited attribute instances before q_1 has evaluated them, q_1 must lock all inherited attribute instances of the j^{th} child of n that have not yet been evaluated.

A simple way to accomplish the locking just described is, immediately prior to spawning a process to execute the plan for tree node n, lock all attribute instances that the plan for n will ultimately evaluate. That is, all attribute instances corresponding to EVAL elements in the plan for n (i.e., all output attribute instances of n) are locked. Since visiting a node for the first time requires locking only output attribute instances belonging to the production instance rooted at that node, this is a slightly simpler locking strategy than was necessary under EvaluateAsynchC.

Given the above, we present in Figure 4.15 an attribution algorithm that executes both types of visit elements asynchronously. The algorithm makes use of the procedure

lockOutputAttrInsts(n)

that locks all output attribute instances of the production instance rooted at node n. (lockOutputAttrInsts is quite similar to lockChildAttrInsts from Figure 4.13.) To attribute a tree T with this routine, one would use the code


```
EvaluateAsynchCP(treenode n, integer i):
    while PLANS[prod_num(n)][i] ≠ [LAST_VISIT_PARENT *] do
        case PLANS[prod_num(n)][i] of
            [EVAL j b]: {
                value(child(j, n).b) :=
                    value defined by appropriate semantic fn;
                unlockAttrInst(child(j, n).b);
                i := i + 1
                };
            [VISIT_CHILD j k]: {
                if k = 0
                    lockOutputAttrInsts(child(j, n));
                    spawn EvaluateAsynchCP(
                            child(j, n),
                            MAPDOWN(k, prod_num(child(j, n))))
                    fi;
                i := i + 1
                };
            [VISIT_PARENT k]:
                i := i + 1
            esac
        od
    end EvaluateAsynchCP;
```

Figure 4.15 - An Asynchronous Visit Element Attribution Algorithm

lockOutputAttrInsts(*root of* T);

EvaluateAsynchCP(*root of T*, 0);

wait for all asynchronous processes to terminate

4.6 Asynchronous Execution of All Plan Elements

Finally, we consider attribution wherein all types of plan elements are executed asynchronously. Such an algorithm is simply a combination of the features of EvaluateAsynchE with those of EvaluateAsynchCP, resulting in the algorithm of Figure 4.16. Observe that this algorithm

```
EvaluateAsynchECP(treenode n, integer i):
    while PLANS[prod_num(n)][i] ≠ [LAST_VISIT_PARENT *] do
        case PLANS[prod_num(n)][i] of
            [EVAL j b]: {
                spawn EvaluateAttribute(n, i);
                i := i + 1
                };
            [VISIT_CHILD j k]: {
                if k = 0
                    lockOutputAttrInsts(child(j, n));
                    spawn EvaluateAsynchECP(
                            child(j, n),
                            MAPDOWN(k, prod_num(child(j, n))))
                fi;
                i := i + 1
                };
            [VISIT_PARENT k]:
                i := i + 1
            esac
        od
    end EvaluateAsynchECP;
```

Figure 4.16 - An Asynchronous Attribution Algorithm

employs the same locking strategy as EvaluateAsynchCP.

4.7 Some Practical Considerations

4.7.1 Process Spawning Characteristics of the Algorithms

The various asynchronous attribution algorithms exhibit significantly different behavior with regard to the number of processes spawned, the amount of work done by typical processes, the points at which processes are spawned, and the order in which processes are spawned. Each of these characteristics may affect the execution time required by actual implementations of the algorithms.

Process spawning requires time, and on many systems, a non-trivial amount of time. Obviously, it is desirable to attempt to minimize time overhead due to spawning. While the architecture of the actual multiprocessor and various characteristics of its operating system greatly affect this overhead, the manner in which an attribution algorithm (or any piece of software) spawns processes contributes to the overhead as well.

Consider a derivation tree T containing k nodes having at least one attribute instance. Suppose further that T contains a total of m attribute instances. Clearly, $k \leq m$, but typically $k < m$.

On such a tree, EvaluateAsynchE will spawn a total of m processes, one per EVAL element per plan per node.

EvaluateAsynchC will spawn one process per VISIT_CHILD element per plan per node. At worst, this will be O(m) processes, but for a typical grammar, EvaluateAsynchC will spawn significantly fewer than m processes. EvaluateAsynchP exhibits similar behavior.

EvaluateAsynchCP spawns O(k) processes, essentially one process for each node. EvaluateAsynchECP, as may be imagined, would spawn O(m + k) processes.

While the number of process creations done by an algorithm is a factor in determining its performance, an attempt to predict the relative performance of the various attribution algorithms based solely on this factor would be ill-advised. Other factors must be considered.

One such factor is the *granularity* of the processes. That is, one must consider the amount of work done by the processes. For example, if the work done by a process takes less time than the time it took to spawn the process, there is likely an unnecessary slowdown in the attribution process. By this reasoning, we should favor larger-grained processes over smaller-grained processes in the absence of other considerations.

EvaluateAsynchE spawns fairly small processes. Each process is responsible for evaluating one semantic function, assigning a value to an attribute instance, unlocking that attribute instance, and, usually, signaling any waiting processes. Often, semantic functions take relatively little time to evaluate. Assigning the function value to the attribute instance and unlocking the attribute instance are relatively inexpensive operations. Also, there are often few waiting processes to be signaled. Thus, EVAL processes can tend to be short-lived and do little work.

The processes created by the algorithms that execute either or both VISIT_PARENT and VISIT_CHILD elements asynchronously tend to perform more work and live longer. Such processes often cause the evaluation of multiple EVAL and visit elements. EvaluateAsynchCP and EvaluateAsynchECP generally spawn the largest processes.

One subject not dealt with here is the possible use of heuristics either at plan-generation time or during tree attribution to decide whether or not a given plan element should be performed "in-line" or by an

asynchronous process. Conceivably, it would be possible in many cases to estimate that the time required to create and initiate an asynchronous process would be greater than the time required to perform the element in a synchronous, non-parallel manner. For example, nothing would likely be gained from using individual asynchronous processes to evaluate *copy rules*, semantic equations of the form

$$Y.b = Z.c$$

Making such estimations would be quite simple for EVAL elements that invoke very simple semantic functions. EVAL elements invoking more complicated semantic functions (using recursion and/or conditional expressions), VISIT_CHILD elements, and VISIT_PARENT elements would be more difficult to provide reasonable estimates for, depending on the the attribution algorithm in question.

4.7.2 Concurrency Characteristics of the Algorithms

Another factor affecting the performance of an attribution algorithm is the amount of concurrency one can expect Clearly, the dependencies among attributes influence this to a great extent, but certain characteristics of the algorithms themselves also have an affect. In particular, consider the pattern in which processes are spawned by the various algorithms.

EvaluateAsynchE encounters plan elements serially, in the same chronological order as does the serial attribution algorithm. While the evaluation processes may execute concurrently with each other and with EvaluateAsynchE, all other operations, including attribute instance locking and process spawning, are done serially.

Under EvaluateAsynchC, the situation changes. Each process spawned as the result of encountering a VISIT_CHILD element may execute EVAL elements and other VISIT_CHILD elements. Thus, locking, spawning, attribute evaluation, and plan execution may be done concurrently by multiple processes. EvaluateAsynchP, EvaluateAsynchCP, and EvaluateAsynchECP have similar behavior.

EvaluateAsynchC and EvaluateAsynchP do not execute EVAL elements asynchronously. This causes any serial work done by these algorithms (e.g., plan index incrementing, while-loop execution, or case splitting on the plan element type) to be held up awaiting the completion of the execution of an EVAL element. Under EvaluateAsynchE, the serial part of the work takes place in parallel with at least semantic function evaluation and attribute instance unlocking.

EvaluateAsynchCP shows behavior similar to that of EvaluateAsynchC and EvaluateAsynchP with regard to the types of operations that may take place concurrently. Again, since EVAL elements are not done asynchronously, arbitrarily large amounts of serial work may be held up by EVAL element execution. EvaluateAsynchECP, by spawning processes for EVAL elements, can usually proceed with at least the serial portions of the work.

4.7.3 Space Considerations

The asynchronous attribution algorithms have greater space requirements than has the serial attribution algorithm. Additional space is required for the attribute instance locks and for the asynchronous processes themselves.

A lock is required for each attribute instance whose value may be used to determine the value of another attribute instance. Since this includes the vast majority of attribute instances, we assume that all attribute instances require locks.

A lock may be represented by a single bit, with one value representing "locked" and the other value representing "unlocked". Alternatively, we could reserve a special universal attribute instance value to have the meaning "locked." Locking an instance would amount to assigning this value to the instance, while unlocking would take place when any other value was assigned to the instance.

Any attribute instance requiring a lock may, during attribution, have processes waiting for its value to be determined. After the attribute instance is unlocked, the waiting processes must be allowed to continue. The processes themselves may determine when they may proceed by executing busy-waits, but this may waste a great deal of processor time. The alternative is to suspend waiting processes, place them on a list of waiting processes, and awaken the processes when it is appropriate for them to proceed.

A simple solution is to associate a wait queue with each attribute instance. Under a linked-list implementation of queues, this would require additional constant space (for a pointer to the head element of the queue) for each attribute instance. If this requirement is unacceptable, alternatives are possible. One possibility would be to use distinct attribute instance identifiers as keys into a fast search table [Knuth 1973]. Each table entry would consist of a list of processes waiting on the attribute instance associated with the key.

Finally, the asynchronous processes themselves require attribution-time space. Each process requires some constant amount of storage to maintain various private values (including the current tree node, current plan element index, and current semantic function instruction) and a variable amount of storage for one or more stacks, used for semantic function evaluation, expression evaluation, procedure calling, etc.

4.8 Related Work

An early use of asynchronous processes for attribute evaluation appears in [Fang 1972]. In this work, productions in arbitrary well-formed attribute grammars are associated with *semantic rules* (composed in an ALGOL-like language). Statements in semantic rules can be explicitly designated by a grammar designer as *parallel statements*, to be executed by asynchronous processes. Processes attempting to access not-yet determined attribute instance values block until the values become available. Incremental attribution (to be discussed in Chapter 6) is not possible in the system.

[Boehm & Zwaenepoel 1987] describe a hybrid attribute evaluation scheme for trees derived from OAG's. Derivation trees are carved into a collection of disjoint subtrees ("bottom trees") and the remainder of the tree (the "top tree"). Within each bottom tree, attribution is performed by a sequential visit-sequence evaluator. In the top tree, attribution among the bottom trees is coordinated by a dynamic order attribution scheme. Parallelism is achieved by essentially having each bottom tree attributed by a separate asynchronous process. The grammar designer must provide

explicit clues as to how a derivation tree may be split. The scheme does not provide for incremental attribution.

A very different approach, placing emphasis on the static assignment of asynchronous processes to processors, is given in [Kuiper 1989]. The attribution scheme presented is essentially a dynamic order scheme with attribute instance evaluation processes statically assigned to individual processors. Again, this scheme does not provide for incremental attribution.

5. Synchronous Parallel Attribution

We continue with the development of parallel attribution schemes for trees derived using l-ordered attribute grammars. In this chapter we develop attribution algorithms utilizing synchronous-style parallelism.

We begin by considering a style of synchronous attribution based on sets of mutually independent plan elements to be executed concurrently. We continue by considering a style of attribution based on converting visit-sequence graphs into series-parallel graphs. For both attribution styles, we discuss the concept of optimal plans and consider means by which suitable optimal plans may be generated.

5.1 Introduction

In the preceding chapter we developed algorithms that achieved parallel execution of plan elements by spawning and not (immediately) waiting for the completion of asynchronous processes executing plan elements. As we demonstrated, such processes occasionally need to be coordinated to preclude the use of attribute instances whose values have not yet been determined. This coordination was accomplished through a locking protocol implemented by the inclusion of explicit locking and unlocking operations in the attribution algorithms.

An alternative is to overlap the execution of plan elements in time while guaranteeing that whenever execution of a plan element e in the plan for an instance of production p is initiated, all plan elements (as opposed to all attribute instance values) upon which element e depends, with respect to VS(p), have been previously completed. In this way, it is assured that whenever an EVAL element is encountered, all attribute instance values

needed by the corresponding semantic function will have already been determined.

We can accomplish this by constructing plans that can be executed using synchronous parallel language constructs. In particular, we wish to construct plans to be carried out by attribution routines making use of the *pforall* statement described in §3.2.2. In this way, no explicit locking or unlocking of attribute instances will be necessary.

5.2 Synchronous Sets

One scheme for utilizing synchronous parallel language constructs stems from a simple observation about visit-sequence graphs. Consider production p and two plan elements a and b from VS(p). We say that a and b are *mutually independent* if and only if there are no dependencies, direct or transitively implied, between a and b in VS(p). We generalize this concept by saying that any set W of plan elements from VS(p) is mutually independent if and only if the elements of W are pairwise mutually independent.

Let W be such a mutually independent set of plan elements. If, during attribution, all plan elements determining any of the elements in W have been executed previously, execution of all elements in W can take place concurrently. This leads to an obvious approach to synchronous parallel attribution.

Plans are produced that are, as before, topological orderings of visit-sequence graph nodes and that, when possible, manifest sets of mutually independent elements. When such a set is encountered during plan evaluation, all plan elements in the set are executed in parallel. After the executions of all elements in the set are complete, plan evaluation continues with the portion of the plan immediately following the set. We call a mutually independent set of plan elements executed in this fashion a *synchronous set* and plans containing such sets *synchronous-set plans.*

Due the necessary properties of l-OAG partitioning algorithms, some brief observations concerning synchronous sets can be made. First, no two visit-sequence elements [VISIT_CHILD j k_1] and [VISIT_CHILD j k_2] can be in the same synchronous set (since all visits to a given child are totally ordered within a VS graph). Second, no two visit-sequence elements [VISIT_PARENT k_1] and [VISIT_PARENT k_2] can be in the same synchronous set (since all visits to the parent are totally ordered within a VS graph). Finally, the final VISIT_PARENT (represented by a LAST_VISIT_PARENT element) can never appear in a synchronous set (since it depends on all other vertices in the VS graph).

5.2.1 Synchronous-Set Plan Elements

We introduce the following additional types of plan elements to represent synchronous sets:

[SYNCHSET n]	which tells the evaluator that the next n ($n \geq 2$) consecutive plan elements constitute a synchronous set,
[PEVAL j b]	
[PVISIT_CHILD j k]	
[PVISIT_PARENT k]	which are similar to EVAL, VISIT_CHILD, and VISIT_PARENT elements, respectively, but are used for plan elements included in synchronous sets.

For example, given the visit-sequence graph of Figure 5.1, a

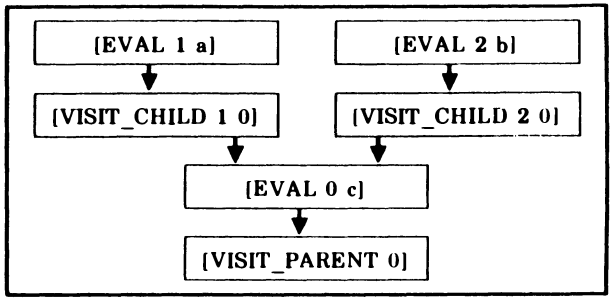

Figure 5.1 - A Visit-Sequence Graph

possible synchronous set plan is

< [SYNCHSET 2][PEVAL 1 a][PEVAL 2 b]
[SYNCHSET 2][PVISIT_CHILD 1 0][PVISIT_CHILD 2 0]
[EVAL 0 c]
[VISIT_PARENT 0]
>

An alternative synchronous set plan is

< [EVAL 1 a]
[SYNCHSET 2][PVISIT_CHILD 1 0][PEVAL 2 b]
[VISIT_CHILD 2 0]
[EVAL 0 c]
[VISIT_PARENT 0]
>

5.2.2 Generating Synchronous-Set plans

The preceding example illustrates that there need not be a unique synchronous-set plan for a given visit-sequence graph. Since our overall goal is to reduce the time required for tree attribution, it makes sense to attempt to produce a plan that is in some sense optimal. There are numerous possible optimality criteria that may be considered.

One distinction to be made at the outset is that between *local optimality* (considering the optimality of the plan for each production in isolation from all other productions) and *global optimality* (considering the entire set of plans for all productions for a grammar). We will consider only local optimality in most of this chapter, returning briefly to the issue of global optimality in §5.2.2.3. (Note that the use of the word "local" is something of a misnomer, since the visit-sequence graph for a production is constructed using information distilled from the entire grammar.)

The task of producing a locally optimal synchronous-set plan for a production can be idealized as the following problem:

Given dag $G=(V,E)$ with unique sink $v_s \in V$, lengths $l \in V \rightarrow N^{>0}$, and deadline $D \in N^{\geq 0}$, find dag $G'=(V, E \cup E')$ such that

$$\left(\sum_{j=0}^{depth(G')} \max\{l(v) \mid v \in V \wedge depth(v)=j\}\right) \leq D$$

where

depth(v) = the maximal length among all paths to v from all sources of G'

depth(G') = max{depth(v) | v∈V}

(Note that we use $N^{>0}$ to designate the set of positive integers and $N^{\geq 0}$ to designate the set of nonnegative integers.) Intuitively, G corresponds to the VS graph for production p, and the length l(v) of each element v in V corresponds to an estimate of the time required to execute that element of the VS graph.

If such a G' can be found, a synchronous-set plan meeting deadline D can be generated in time O(|G'|). By iterating over values of D, we can find the minimum possible deadline.

We first consider an obvious special case, that in which l(n) = 1 for all n∈V. We refer to this special case as *the optimal unit-length synchronous-set planning problem.*

5.2.2.1 Optimal Unit-Length Plans

If the length (i.e., execution time) of each visit-sequence graph node is 1, a locally optimal synchronous-set plan is easily constructed.

Define the *k^{th} level* of G to be {v | v∈V and depth(v) = k}. An optimal plan for production p will consist of h synchronous sets, where h is the depth of VS(p). The first synchronous set contains all vertices on level zero of VS(p), the second set contains all vertices on level one of VS(p), and so on. Such a plan can be produced using the algorithm of Figure 5.2.

A "level-by-level" plan as generated by this algorithm corresponds to finding a schedule σ that meets deadline

$$D = depth(VS(p)) + 1.$$

No smaller deadline can be met by a synchronous-set plan without violating the partial order represented by the edges of VS(p). Thus, level-by-level plans are optimal under the unit-length assumption.

As will be discussed in §5.2.3, it may be necessary or convenient to forbid certain vertices of a VS-graph from being included in any

```
GenerateSynchSetPlan(production p):
    plan := < >;
    (V,E) := VS(p);
    W := {v | v∈V and indegree(v) = 0};
    while W ≠ ∅ do
        newW := ∅;
        if |W| > 1 then
            plan := plan || <[SYNCHSET |W|]>
            fi;
        forall v∈W do
            forall (v,u)∈E do
                indegree(u) := indegree(u) − 1;
                if indegree(u) = 0 then
                    newW := newW ∪ {u}
                    fi
                od;
            if |W| > 1 then
                plan := plan || <"P" version of v>
            else
                plan := plan || <v>
                fi
            od;
        W := newW
        od;
    return(plan)
    end GenerateSynchSetPlan;
```

Figure 5.2 - A Synchronous-Set Planning Algorithm

synchronous set. In this situation, a relatively simple procedure may be used to produce locally-optimal unit-length plans.

For production p, let $(V, E) = VS(p)$ and define

$$V_f = \{v \mid v \in V \wedge v \text{ is a forbidden node}\},$$

$$V_a = V - V_f,$$
$$VS_f(p) = VS(p)/V_f, \text{ and}$$
$$VS_a(p) = VS(p)/V_a.$$

Conceptually, a planning algorithm must simultaneously produce and merge two subplans: a level-by-level synchronous set subplan for $VS_a(p)$ and a serial subplan for $VS_f(p)$. A procedure accomplishing this is given in Figure 5.3.

A plan generated by this procedure obeys all dependencies in VS(p) and corresponds to a schedule σ that meets deadline

$$D = |V_f| + depth(VS_a(p)) + 1$$

Any smaller deadline could be achieved only by violating the dependencies in VS(p) or by including one or more elements of V_f in a synchronous set.

5.2.2.2 Optimal Synchronous-Set Plans

If we abandon the unit-length assumption, we face two problems. The first is how to determine an estimate for the length of each vertex of a given VS graph (since semantic equations may include partial recursive functions, finding the exact length of a vertex is, in general, undecidable). The second is how to produce an optimal synchronous-set plan given those length estimates.

The difficulty of estimating the length of a VS graph vertex using heuristics depends greatly on the allowable right-hand sides for semantic equations. If semantic equations contain only simple total functions (e.g., addition, subtraction, etc.), the task is fairly straightforward. If semantic equations may contain conditional expressions, the task is more complicated.

Note that we are considering only local optimality criteria. Therefore, the bulk of the work falls on estimating the lengths of EVAL elements. In isolation from all other productions, estimating the lengths of VISIT_CHILD and VISIT_PARENT elements essentially degenerates to assigning predetermined constant lengths to such elements.

```
GenerateSynchSetPlan(production p, predicate forbidden):
    plan := < >;
    (V,E) := VS(p);
    W := {v | v∈V and indegree(v) = 0};
    while W ≠ Ø do
        synchSetW := Ø;
        while W ≠ Ø do
            newW := Ø;
            forall v∈W do
                if ¬forbidden(v) then
                    synchSetW := synchSetW ∪ {v}
                else
                    forall (v,u)∈E do
                        indegree(u) := indegree(u) – 1;
                        if indegree(u) = 0 then
                            newW := newW ∪ {u}
                        fi
                    od;
                    plan := plan || <v>
                fi
            od;
            W := newW
        od;
(continued)
```

Figure 5.3 - Synchronous-Set Planning with Forbidden Nodes

Figure 5.3 (Continued)

```
        W := synchSetW;
        if |W| > 1 then
            plan := plan || <(SYNCHSET |W|)>
        fi;
        forall v∈W do
            forall (v,u)∈E do
                indegree(u) := indegree(u) – 1;
                if indegree(u) = 0 then
                    newW := newW ∪ {u}
                fi
            od;
            if |W| > 1 then
                plan := plan || <"P" version of v>
            else
                plan := plan || <v>
            fi
        od;
        W := newW
    od;
return(plan)
end GenerateSynchSetPlan;
```

Assuming lengths have been assigned to all vertices of VS(p) for production p, it remains to produce an optimal synchronous-set plan. Unlike the unit-length case, this is a non-trivial task; optimization problems of this sort often fall into the set of NP-complete problems. While it is unclear whether the problem of generating optimal non-unit-length plans is NP-complete, aspects of the problem suggest it may be. The following example illustrates this.

Consider the VS graph depicted in Figure 5.4, in which individual vertices have been labeled with their respective lengths. Considering only

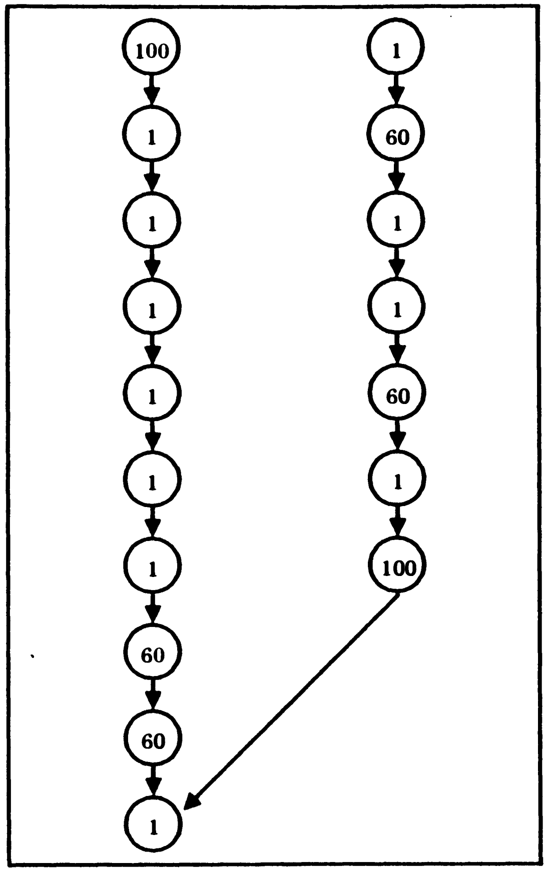

Figure 5.4 - A VS Graph

the vertex lengths, the level-by-level plan generated from this graph can be executed in time 444 (i.e., meets deadline 444). By adding the dashed edge

shown in Figure 5.5, a graph for which a level-by-level plan that can be executed in time 351 is created. Adding any other single edge to the graph

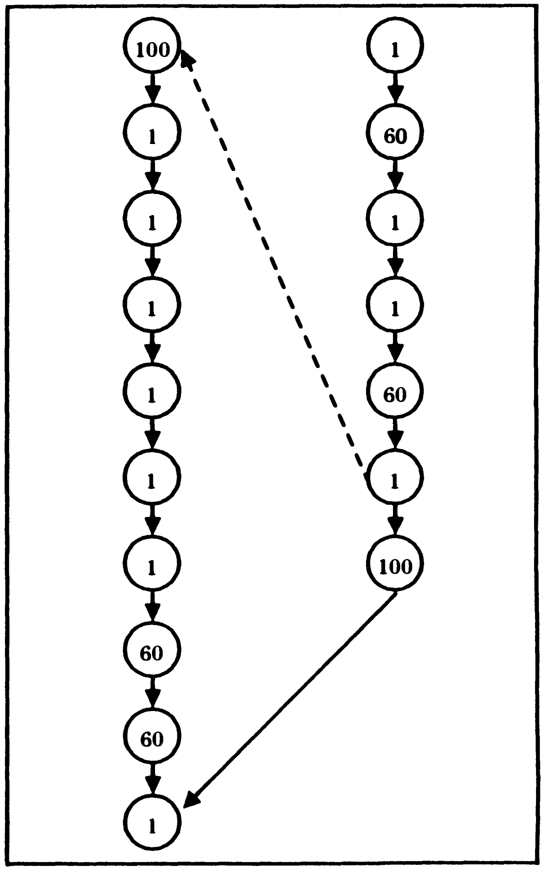

Figure 5.5 - A VS Graph With a Single Added Edge

of Figure 5.4 results in a level-by-level plan requiring greater time to execute.

However. the optimal plan that can be generated from the VS graph in Figure 5.4 requires time 327 to execute and corresponds to adding the two dashed edges shown in Figure 5.6. Further, the edge added in Figure

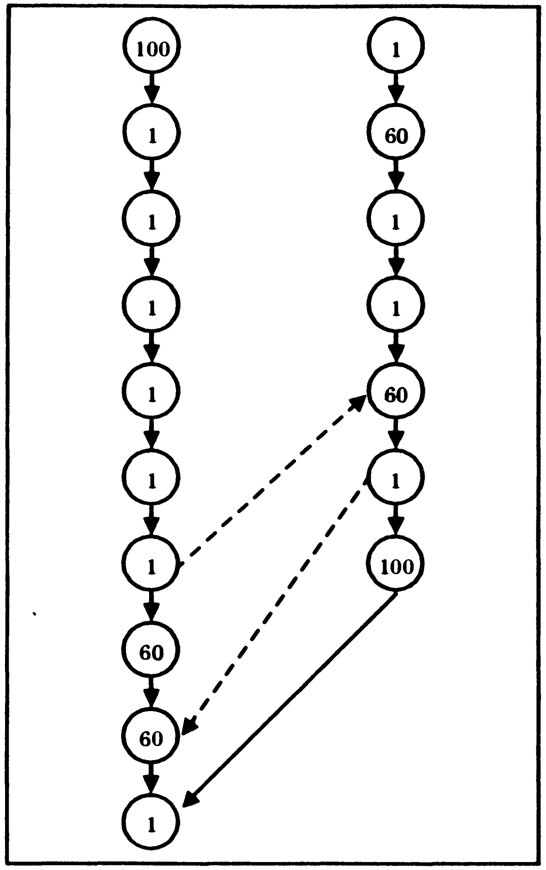

Figure 5.6 - An Optimal VS Graph

5.5 precludes arriving at this optimal solution (i.e., adding additional edges

to the graph of Figure 5.5 will never result in a dag meeting the optimal deadline). Thus, no simple "greedy" method can be used for optimal synchronous-set planning without incurring the danger of combinatorial explosion.

It is likely that there exist heuristic methods (possibly based on branch-and-bound techniques) that can produce "reasonably optimal" plans. Whether such methods can produce truly optimal plans in non-exponential time remains an open question.

5.2.2.3 Global Optimality

Producing an optimal synchronous-set plan for a production p using only information local to p may be sufficient in simple cases. Often, however, attribution time may be further reduced by taking into account information *global* to p when generating the plan for p.

For example, we have previously assumed the length assigned to each VISIT_CHILD and VISIT_PARENT plan element was some predetermined constant. A better estimate of the length of visit elements might allow us to generate better plans. However, the information needed to produce a better estimate can only come from other productions.

Consider plan element e = [VISIT_CHILD j k] in VS(p) for production $p{:}X_0 \rightarrow X_1 \ldots X_n$. Rather than assume e has arbitrary a priori constant length, we wish to estimate how long it might actually take to perform the work associated with element e. One such estimate could be determined by looking at all productions having X_j as the left-hand side.

Let p' be a production having X_j as the left-hand side. The work associated with e is roughly the work associated with all vertices on all paths in VS(p') from the vertex [VISIT_PARENT k – 1] to the vertex [VISIT_PARENT k]. Assuming the lengths of all vertices on all such paths are known, a conservative estimate of the length of element e might be the minimal length-sum over all such paths.

The preceding suggests a length-assigning procedure that begins by finding the lengths of VS graph vertices for *leaf productions* (i.e., productions having no nonterminals on their respective right-hand sides) and works "upward," eventually finding the lengths for VS graph vertices of *root productions* (i.e., productions having the starting nonterminal as their respective left-hand sides). Care and reasonable heuristic techniques would be required to deal with self-recursive productions and groups of mutually-recursive productions.

Other uses of global information are certainly possible. Techniques employing flow analysis [Muchnik & Jones 1981] or execution profiling [Pettis & Hansen 1990] for code optimization could be applicable here. For example, productions whose instances are likely to appear often at deep levels of a derivation tree might merit special attention in the same way as do often-executed inner loops [Waite 1976].

5.2.2.4 External Factors Influencing Plan Optimality

Optimality can be significantly influenced by factors completely unrelated to an attribute grammar itself. In particular, architectural properties of the computing engine (e.g., number of physical processors, inter-processor communication protocols, and memory hierarchies) can easily have a great effect on actual attribution times obtained. These

matters and ways in which the planning process might take such factors into account are beyond the scope of this work.

5.2.3 Synchronous Set Attribution

In presenting algorithms for synchronous set attribution, we will proceed by restricting the contents of synchronous sets in various ways. First, we present a tree attribution algorithm for plans whose synchronous sets contain only EVAL elements. Next we present an algorithm suitable when synchronous sets may contain only EVAL and/or VISIT_CHILD elements. Finally, we consider the general case in which synchronous sets may be mixtures of EVAL, VISIT_CHILD, and/or VISIT_PARENT elements. Plans usable by the various algorithms could, for example, be constructed with the procedure given in Figure 5.3 by appropriately defining the predicate forbidden().

5.2.3.1 Synchronous Sets Containing EVAL Elements Only

By making all VISIT_CHILD and VISIT_PARENT elements in a grammar forbidden, we can produce a collection of plans in which any synchronous sets contain only EVAL elements. Such synchronous sets are particularly easy to execute in a synchronous parallel fashion, as shown in Figure 5.7. This algorithm is a modest modification of the serial l-OAG attribution algorithm.

Note that in an actual implementation, the code for the PEVAL element case might be moved into the body the pforall statement, thereby avoiding an unnecessary level of procedure calls. The manner in which the procedure in Figure 5.7 was written was chosen in an attempt to emphasize

```
EvaluateSynchSetE(treenode n, integer i):
    while PLANS[prod_num(n)][i] ≠ [LAST_VISIT_PARENT *]
            | n ≠ root of tree do
        case PLANS[prod_num(n)][i] of
            [SYNCHSET m]: {
                pforall j in 1..m do
                    EvaluateSynchSetE(n, i+j)
                    od;
                i := i+m+1
                };
            [PEVAL j b]: {
                value(child(j, n).b) := value of appropriate semantic fn;
                return
                };
            [EVAL j b]: {
                value(child(j, n).b) := value of appropriate semantic fn;
                i := i+1
                };
            [VISIT_CHILD j k]: {
                n := child(j, n);
                i := MAPDOWN(k, prod_num(n));
                };
            [LAST_VISIT_PARENT k]:
            [VISIT_PARENT k]: {
                i := MAPUP(k, child_num(n), prod_num(parent(n)));
                n := parent(n)
                }
            esac
        od
    end EvaluateSynchSetE;
```

Figure 5.7 - An Attribution Algorithm for Synchronous Sets with EVAL's Only

the similarities among the various attribution algorithms presented in this chapter.

5.2.3.2 Synchronous Sets Containing EVAL and VISIT_CHILD Elements Only

By making all VISIT_PARENT elements forbidden, plans may contain synchronous sets containing EVAL and/or VISIT_CHILD elements. PEVAL elements in such sets can be dealt with as they were by the procedure in the preceding section. PVISIT_CHILD elements require different but still straightforward handling, as follows.

Upon encountering a PVISIT_CHILD element in a synchronous set, the attribution routine must cause the work for that element to be performed. Recall that if an attribution routine is currently focused on the element [VISIT_CHILD j k] in the plan for tree node n, the work associated with that element is the execution of elements e_h through e_i in the plan for the j^{th} child of n, where

$< \ldots e_h \ldots e_i \ldots >$ = the plan for child(j, n)

p = prod_num(child(j, n))

h = MAPDOWN(k, p)

i = the index of the element [VISIT_PARENT k] in the plan for the production numbered p

The synchronous parallel step executing a [PVISIT_CHILD j k] element should be terminated when element e_i(= [VISIT_PARENT k]) is executed.

This last statement leads to a small complication regarding the execution of VISIT_PARENT elements. In some contexts, a VISIT_PARENT element in the plan for an instance of production p may represent the end of the work corresponding to a VISIT_CHILD element in its parent's plan. In other contexts, the same VISIT_PARENT element instead represents the end of the work corresponding to a PVISIT_CHILD element in the parent's plan. In the former context, execution of the VISIT_PARENT should cause a change of focus to the parent node via MAPUP. In the latter context, execution of the VISIT_PARENT should simply cause the termination of the process synchronous step executing it.

For example, consider the grammar of Figure 5.8, from which the

```
AI(S) = ∅, AS(S) = { ss }
AI(X) = { xi }, AS(X) = { xs }
AI(Y) = { yi }, AS(Y) = { ys }
AI(Z) = { zi }, AS(Z) = { zs }

p0:     S → X Y
            {   X.xi ← 0;
                Y.yi ← 1;
                S.ss ← X.xs + Y.ys }
p1:     X → Z Z
            {   Z$1.zi ← X.xi + 1;
                Z$2.zi ← Z$1.zs + 1;
                X.xs ← Z$1.zs + Z$2.zs }
p2:     Y → Z Z
            {   Z$1.zi ← Y.yi + 1;
                Z$2.zi ← Y.yi + 1;
                Y.ys ← Z$1.zs + Z$2.zs }
p3:     Z → 'a'
            {   Z.zs ← Z.zi + 1 }
```

Figure 5.8 - An Example Grammar

plans shown in Figure 5.9 are produced. The sole tree derivable from this


```
<<[EVAL 1 xi][EVAL 2 yi]
   [SYNCHSET 2][PVISIT_CHILD 1 0][PVISIT_CHILD 2 0]
   [EVAL 0 ss][VISIT_PARENT 0]>
 <[EVAL 1 zi][VISIT_CHILD 1 0][EVAL 2 zi]
   [VISIT_CHILD 2 0][EVAL 0 xs][VISIT_PARENT 0]>
 <[EVAL 1 zi][EVAL 2 zi]
   [SYNCHSET 2][PVISIT_CHILD 1 0][PVISIT_CHILD 2 0]
   [EVAL 0 ys][VISIT_PARENT 0]>
 <[EVAL 0 zs][VISIT_PARENT 0]>
 >
```

Figure 5.9 - Plans for the Example Grammar

grammar (shown in Figure 5.10) contains four instances of nonterminal Z.

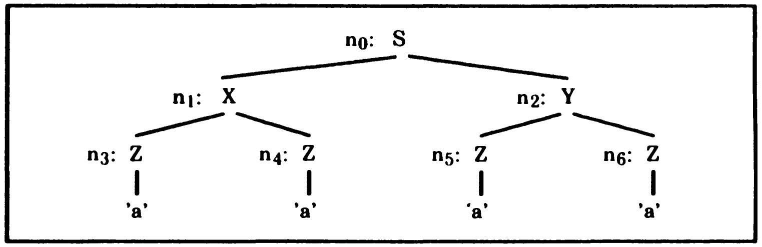

Figure 5.10 - The Derivation Tree

Two of these, nodes n_3 and n_4, will be visited serially while two others, nodes n_5 and n_6, will be visited in parallel. The [VISIT_PARENT 0] element in the plan for Z must respond differently depending on the type (i.e., serial or parallel) of VISIT_CHILD caused that element to be executed.

This difficulty can be dealt with through the use of the predicate

going_up_from_parallel_visit(k, j, p)

that returns true if the k^{th} parent visit up to a node derived from production p from its j^{th} child ends the work done by a PVISIT_CHILD and returns false if the k^{th} parent visit ends the work done by a VISIT_CHILD. This predicate can be implemented as a statically determined mapping (derivable from MAPUP) or as a routine that examines plans during attribution (making use of the distinction between VISIT_CHILD and PVISIT_CHILD elements). Figure 5.11 presents an attribution algorithm using this solution.

```
EvaluateSynchSetEC(treenode n, integer i):
   while PLANS[prod_num(n)][i] ≠ [LAST_VISIT_PARENT *]
         | n ≠ root of tree do
      case PLANS[prod_num(n)][i] of
         [SYNCHSET m]: {
            pforall j in 1 .. m do
                EvaluateSynchSetEC(n, i + j)
                od;
            i := i + m + 1
            };
         [PEVAL j b]: {
            value(child(j, n).b) := value of appropriate semantic fn;
            return
            };
         [EVAL j b]: {
            value(child(j, n).b) := value of appropriate semantic fn;
            i := i + 1
            };
(continued)
```

Figure 5.11 - An Attribution Algorithm for Synchronous Sets with EVAL and/or VISIT_CHILD Elements

Figure 5.11 (Continued)

```
         [PVISIT_CHILD j k]:
         [VISIT_CHILD j k]: {
            n := child(j, n);
            i := MAPDOWN(k, prod_num(n));
            };
         [LAST_VISIT_PARENT k]:
         [VISIT_PARENT k]:
            if going_up_from_parallel_visit(
                  k, child_num(n), prod_num(parent(n)))
               return
            else
               i := MAPUP(k, child_num(n), prod_num(parent(n)));
               n := parent(n)
               fi
         esac
      od
   end EvaluateSynchSetEC;
```

5.2.3.3 General Synchronous-Set Plans

Finally, we consider the case in which synchronous sets may contain any and all types of plan elements. Having treated the synchronous parallel execution of EVAL and VISIT_CHILD elements in previous sections, it remains only to add the capability to perform VISIT_PARENT elements as members of synchronous sets. This turns out to be less straightforward than it might first appear.

5.2.3.3.1 Problems Posed by VISIT_PARENT Elements in Synchronous Sets

The means by which EVAL elements may be executed as members of synchronous sets is quite intuitive. Executing VISIT_CHILD elements as members of synchronous sets is somewhat more involved, but the technique is still fairly obvious since the control structures involved (i.e., synchronous process spawning/waiting) has a very "treelike" nature: a parent node (via a pforall statement) spawns processes for some or all of its children, these processes then performing work affecting only the subtrees rooted at their respective child nodes and eventually yielding control back to the parent. These properties of spawning processes "downward" only and of processes terminating and yielding control "upward" only are lost when VISIT_PARENT elements may appear in synchronous sets.

Informally, problems occur when an "upward" spawning (resulting from spawning a process to execute a VISIT_PARENT element) and a "downward" spawning (resulting from spawning a process to execute a VISIT_CHILD element) collide. When this occurs, there is no longer a trivial correspondence between data structure (i.e., the derivation tree being attributed) and control structure (i.e., the pforall statement).

Suppose the tree pictured in Figure 5.12 is being attributed using the plans from Figure 5.13 (whose elements have been appended with subscripts for reference).

Let q_0 be the initial process performing attribution. q_0 begins executing n_0's plan. After executing the [EVAL 1 b]$_0$ element, q_0 encounters the [VISIT_CHILD 1 0]$_1$ element and, in executing it, changes its focus down to node n_1, where it begins to execute the plan for that node.

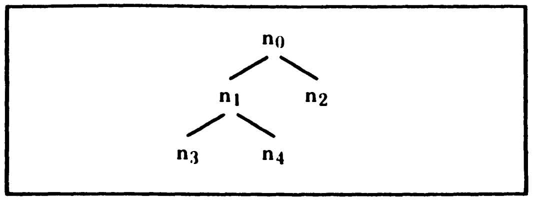

Figure 5.12 - An Example Derivation Tree

```
n0:<[EVAL 1 b]0
     [VISIT_CHILD 1 0]1
     [SYNCHSET 2]2 [PEVAL 1 c]3 [PEVAL 2 d]4
     [SYNCHSET 2]5 [PVISIT_CHILD 1 1]6 [PVISIT_CHILD 2 0]7
     [EVAL 0 e]8
     [VISIT_PARENT 0]9>
n1:<[SYNCHSET 3]10 [PEVAL 1 f]11 [PEVAL 0 g]12 [PEVAL 2 h]13
     [SYNCHSET 3]14 [PVISIT_CHILD 1 0]15
          [PVISIT_CHILD 2 0]16 [PVISIT_PARENT 0]17
     [EVAL 0 i]18
     [VISIT_PARENT 1]19>
```

Figure 5.13 - Plans for Selected Nodes of the Derivation Tree

After executing the set containing the elements [PEVAL 1 f]$_{11}$, [PEVAL 0 g]$_{12}$, and [PEVAL 2 h]$_{13}$, q_0 encounters the [SYNCHSET 3]$_{14}$ element. The state of attribution at this point is depicted in Figures 5.14 and 5.15.

In executing the [SYNCHSET 3]$_{14}$ element, q_0 will spawn (under the control of a pforall statement) three synchronous processes, one to execute each element in the set. Suppose that execution of the [PVISIT_CHILD 1 0]$_{15}$ finishes uneventfully and execution of the [PVISIT_CHILD 2 0]$_{16}$ element, by a process q_1, proceeds. The process

process / node	previous element executed	element being executed	next element to execute
q_0 / n_1	[SYNCHSET 3]$_{10}$	[SYNCHSET 3]$_{14}$	[EVAL 0 i]$_{18}$

Figure 5.14 - Process State of Tree Attribution

n_0.e:	unevaluated	n_1.i:	unevaluated
n_1.b:	evaluated	n_2.d:	unevaluated
n_1.c:	unevaluated	n_3.f:	evaluated
n_1.g:	evaluated	n_4.h:	evaluated

Figure 5.15 - Attribute Instance State of Tree Attribution

executing the [PVISIT_PARENT 0]$_{17}$ element, call it q_2, starts the visit, shifts its attention up to node n_0, and successfully executes the synchronous set represented by the [SYNCHSET 2]$_2$ element, which contains the [EVAL 1 c]$_3$ and [EVAL 2 d]$_4$ elements. The state of the attribution at this point is shown in Figures 5.16 and 5.17.

process / node	previous element executed	element being executed	next element to execute
q_0 / n_1	[SYNCHSET 3]$_{10}$	[SYNCHSET 3]$_{14}$	[EVAL 0 i]$_{18}$
q_1 / n_1	---	[PVISIT_CHILD 2 0]$_{16}$	---
q_2 / n_0	---	[SYNCHSET 2]$_2$	[SYNCHSET 2]$_5$

Figure 5.16 - Process State of Tree Attribution

n_0.e:	unevaluated	n_1.i:	unevaluated
n_1.b:	evaluated	n_2.d:	evaluated
n_1.c:	evaluated	n_3.f:	evaluated
n_1.g:	evaluated	n_4.h:	evaluated

Figure 5.17 - Attribute Instance State of Tree Attribution

It is important to remark at this point that MAPDOWN and MAPUP must be altered for use in this attribution scheme. In particular, a visit to a node should never start or resume execution of a plan in the midst of a synchronous set. If a particular MAPDOWN or MAPUP entry would specify a member of a synchronous set, that entry is adjusted to specify instead the first element following that set in the given plan.

Process q_2 now encounters and executes the [SYNCHSET 2]$_5$ element. Processes to execute the [PVISIT_CHILD 1 1]$_6$ and [PVISIT_CHILD 2 0]$_7$ elements are created. Assume that execution of the [PVISIT_CHILD 2 0]$_7$ element terminates uneventfully. Process q_3 begins to execute the [PVISIT_CHILD 1 1]$_6$.

In executing the [PVISIT_CHILD 1 1]$_6$ element, process q_3 will shift its focus down to node n_1 and attempt to execute the [EVAL 0 i]$_{18}$ element. The state of the entire attribution is shown in Figures 5.18 and 5.19.

At this point, two problems emerge. The semantic function evaluated in the course of performing the [EVAL 0 i]$_{18}$ may require the value of some synthesized attribute instance of node n_4 that q_1 has yet to determine, so q_3 must not proceed until the value of any such instance is certain to be known. This must be done in the absence of locking on attribute instances. In addition, q_0 is also slated to execute the element

process / node	previous element executed	element being executed	next element to execute
q_0 / n_1	[SYNCHSET 3]$_{10}$	[SYNCHSET 3]$_{14}$	[EVAL 0 i]$_{18}$
q_1 / n_1	---	[PVISIT_CHILD 2 0]$_{16}$	---
q_2 / n_0	---	[SYNCHSET 2]$_5$	[EVAL 0 e]$_8$
q_3 / n_1	---	[EVAL 0 i]$_{18}$	[VISIT_PARENT 1]$_{19}$

Figure 5.18 - Process State of Tree Attribution

n_0.e:	unevaluated	n_1.i:	under evaluation
n_1.b:	evaluated	n_2.d:	evaluated
n_1.c:	evaluated	n_3.f:	evaluated
n_1.g:	evaluated	n_4.h:	evaluated

Figure 5.19 - Attribute Instance State of Tree Attribution

[EVAL 0 i]$_{18}$ at some point, resulting in more than one execution of that element.

These problems can be dealt with in this situation by recognizing that q_3 need not actually perform the visit, since the work of the visit will eventually be done by q_0 and since the work of the visit can not be done until q_0 is finished executing the [SYNCHSET 3]$_{14}$ element. However, if n_1 were to be replaced by a node whose plan had instead a [VISIT_PARENT 0] (rather than a [PVISIT_PARENT 0]), a process executing the [PVISIT_CHILD 1 1]$_6$ element would have to actually perform the visit.

One possibility for finding an attribution algorithm that can deal with problems like those illustrated in the above example lies in exploiting the complementary nature of parent visits and child visits and in explicitly noting, at attribution time, when a node's plan is in the midst of executing a SYNCHSET element.

5.2.3.3.2 Sketch of a Solution

Let node n_1 be the j^{th} child of node n_0 in some derivation tree. The *completing visit* (or simply, the *completer*) for a [VISIT_CHILD j k] element in n_0's plan is the element [VISIT_PARENT k] in n_1's plan. The completing visit for the element [VISIT_PARENT k] in n_1's plan is the [VISIT_CHILD j k + 1] element in n_0's plan. The final VISIT_PARENT element in a plan has no completing VISIT_CHILD and is assumed to complete itself. An element of the form [VISIT_CHILD j 0] completes no VISIT_PARENT.

A visit element is considered *incomplete* with respect to a tree node when that element has been encountered during execution of that node's plan and the completing visit has yet to be encountered. A visit element becomes *complete* when its completing visit element is encountered during attribution. A SYNCHSET element is said to be complete if and only if all visit elements in the associated synchronous set are complete.

The algorithm for attribution with general synchronous-set plans will make use of these concepts by guaranteeing that execution of the plan for a node cannot proceed if any element in the node's plan is incomplete. The attribution scheme will also prevent visits to nodes whose plans are currently in the midst of executing a synchronous set. These properties will prevent both premature plan element execution and multiple execution of any plan element.

Figure 5.20 outlines an algorithm incorporating these properties.

```
EvaluateSynchSet(treenode n, integer i):
  while PLAN[prod_num(n)][i] ≠ [LAST_VISIT_PARENT *])
      | n ≠ root of tree do
    case PLAN[prod_num(n)][i] of
      [SYNCHSET m]: {
        executing_synch_set(n) := true;
        pforall j in 1 .. m do
          EvaluateSynchSet(n, i + j)
          od;
        executing_synch_set(n) := false;
        if any visit element in n's plan is incomplete
          return
        else
          i := i + m + 1
          fi
        };
      [PEVAL j b]: {
        value(child(j, n).b) := value of appropriate semantic fn;
        return
        };
      [EVAL j b]: {
        value(child(j, n).b) := value of appropriate semantic fn;
        i := i + 1
        };
      [VISIT_CHILD j k]:
      [PVISIT_CHILD j k]: {
        mark this visit element as incomplete;
(continued)
```

Figure 5.20 - An Attribution Algorithm for General Synchronous Sets

Figure 5.20 (Continued)

```
        if k ≠ 0
          mark the completed VISIT_PARENT element in
            child(j, n)'s plan as complete
          fi;
        if executing_synch_set(child(j, n))
            | any visit element in child(j, n)'s plan is incomplete
          return
        else
          n := child(j, n);
          i := MAPDOWN(k, prod_num(n))
          fi
        };
      [LAST_VISIT_PARENT k]:
      [VISIT_PARENT k]:
      [PVISIT_PARENT k]: {
        if this visit element isn't a LAST_VISIT_PARENT element
          mark this visit element as incomplete
          fi;
        mark the completed VISIT_CHILD element in parent(n)'s
          plan as complete;
        if executing_synch_set(parent(n))
            | any visit element in parent(n)'s plan is incomplete
          return
        else
          i := MAPUP(k, child_num(n), prod_num(parent(n)));
          n := parent(n)
          fi
        }
      esac
    od
end EvaluateSynchSet;
```

The algorithm requires that tree nodes, in addition to their usual contents, contain both a bit used to note when a SYNCHSET element is under execution in that node's plan and bits used to note when visit elements are complete (more will be said concerning these bits later). The algorithm also requires (as mentioned previously) that both MAPUP and MAPDOWN be constructed to prevent plan execution from being resumed in the midst of a parallel group.

5.2.3.3.3 Implementing the Algorithm

As mentioned previously, the algorithm of Figure 5.20 requires that each tree node contain two additional groups of bits: a single bit used to keep track of whether or not a synchronous set in the node's plan is currently under execution and a collection of bits used to determine the incompleteness of visit elements in the node's plan. The bit for keeping track of synchronous set execution, designated by

executing_synch_set(n)

in Figure 5.20, can be implemented in an obvious fashion. The collection of bits for keeping track of incomplete visit elements requires more consideration.

A naive approach would implement the collection as a bit vector with one bit for each visit element in the given tree node's plan. During attribution, statically determined mappings (similar to MAPUP and MAPDOWN) would be used to determine the bit corresponding to the visit element completed by another visit element in a child or parent node.

However, the use of bit vectors implies that it is necessary to know *which* particular visit elements within the plan for a node are incomplete.

This is, in fact, not the case. It is sufficient to know *how many* visit elements in the plan are incomplete. Unless architectural properties of a particular multiprocessor make the use of bit vectors particularly advantageous, simple counters may be used in place of bit vectors.

Each tree node n requires a counter of size

$$\lceil \log(1 + |\textit{largest synchronous set in n's plan}|) \rceil$$

which is, in the worst case,

$$\lceil \log(2 + (\textit{number of children of } n)) \rceil$$

bits. The total maximum number of bits required for derivation tree T is then

$$\sum_{n \in T} \lceil \log(2 + (\textit{number of children of } n)) \rceil \leq \sum_{n \in T} (1 + (\textit{number of children of } n))$$

$$< 2\,(\textit{number of nodes in } T)$$

Marking a visit element incomplete corresponds to incrementing the appropriate node's counter by one, while marking a visit complete corresponds to decrementing the correct counter. Determining if a node has any incomplete visits is done by comparing that node's counter to zero. Mutual exclusion is needed for all these operations, but atomic fetching, incrementing, and decrementing are commonly provided operations on parallel machines.

If we instead provide a counter of size

$$\lceil \log(3 + (\textit{number of children of } n)) \rceil$$

with node n, we can do away with the separate bit

executing_synch_set(n)

and use the counter for that purpose as well.

Note that typical machines do not provide arithmetic operations on arbitrary-sized strings of bits. Thus, an actual implementation using counters would choose some convenient but adequate size of counter, say 8 or 16 bits, for all nodes. This would restrict a system using this implementation to nodes having no more than 253 or 65533 children, respectively, which is not an unreasonable restriction in practice.

Figure 5.21 presents a counter-driven implementation of the algorithm of Figure 5.20. Note the use of the atomic operators introduced in §3.1 and the use of

incomplete_counter(n)

to designate the counter allotted to node n.

5.2.3.3.4 Correctness of the Algorithm

In order to show the algorithm of Figure 5.21 is correct, we must show that it produces a *consistent* attribution (i.e., attribute instances are evaluated in an order consistent with the tree dependency graph) and a *complete* attribution (i.e., every attribute instance is eventually given a value). In addition, it is desirable to show that the algorithm is efficient in the sense that it is *non-redundant* (i.e., no plan element is performed more than once with respect to any derivation tree node).

We first give the following definitions:

A visit element in the plan for node n is *executed prematurely* (or, more simply, a visit element is *premature*) with respect to n if it begins execution before the visit element it completes has finished executing. (We categorize visit elements whose

synchronous processes are simply awaiting termination via a *return* statement as having finished execution.)

The plan for node n is *started prematurely* with respect to n if any element in the plan begins execution before the

```
EvaluateSynchSet(treenode n, integer i):
   while PLAN[prod_num(n)][i] ≠ [LAST_VISIT_PARENT *])
         | n ≠ root of tree do
      case PLAN[prod_num(n)][i] of
         [SYNCHSET m]: {
            atomic_add1(incomplete_counter(n));
            pforall j in 1 .. m do
               EvaluateSynchSet(n, i + j)
               od;
            atomic_sub1(incomplete_counter(n));
            if incomplete_counter(n) ≠ 0
               return
            else
               i := i + m + 1
               fi
            };
         [PEVAL j b]: {
            value(child(j, n).b) := value of appropriate semantic fn;
            return
            };
         [EVAL j b]: {
            value(child(j, n).b) := value of appropriate semantic fn;
            i := i + 1
            };
(continued)
```

Figure 5.21 - Attribution Algorithm for General Synchronous Sets

Figure 5.21 (Continued)

```
        [VISIT_CHILD j k]:
        [PVISIT_CHILD j k]: {
            atomic_add1(incomplete_counter(n));
            if k ≠ 0
                atomic_sub1(incomplete_counter(child(j,n)))
                fi;
            if incomplete_counter(child(j,n)) ≠ 0
                return
            else
                n := child(j, n);
                i := MAPDOWN(k, prod_num(n))
                fi
            };
        [LAST_VISIT_PARENT k]:
        [VISIT_PARENT k]:
        [PVISIT_PARENT k]: {
            if PLAN[prod_num(n)][i] ≠ [LAST_VISIT_PARENT k]
                atomic_add1(incomplete_counter(n))
                fi;
            atomic_sub1(incomplete_counter(parent(n)));
            if incomplete_counter(parent(n)) ≠ 0
                return
            else
                i := MAPUP(k, child_num(n), prod_num(parent(n)));
                n := parent(n)
                fi
            }
        esac
    od
end EvaluateSynchSet;
```


[VISIT_CHILD child_num(n) 0] element in parent(n)'s plan has finished executing.

The plan for node n *has no gaps* (or is *gap-free*) with respect to n if all elements preceding the rightmost (i.e., highest-indexed) executing elements of the plan are both complete (in the case of visit elements) and have finished executing. (Elements within a single synchronous set are not considered to precede or follow one another.)

We first show that as the algorithm proceeds, it simultaneously preserves the properties of freedom from gaps, premature visits, premature starting of plans, and redundant execution of plan elements.

In the lemmas and theorems below, we consider time to be passing in discrete steps. Thus, when we speak of a property holding "prior to time t," we mean that the property is true after the execution of all algorithmic steps occurring at time-steps 0 through $t-1$. When we speak of a property holding "at time t," we mean that the property holds after the execution of all algorithmic steps taking place in time-step t.

Lemma 5.2.3.3.4.1:

During attribution of a derivation tree by the algorithm of Figure 5.21, if, prior to time t, all plans are gap-free, there have been no premature visits, no plans have been started prematurely, and there have been no redundant plan element executions with respect to any nodes, then all plans are gap-free at time t.

Proof: (by contradiction)

Suppose execution of plan element e in the plan for node n is started at time t and that this creates a gap in n's plan. We proceed by case analysis, showing that this assumption must be incorrect in all cases.

Case 1:

Assume e is not a member of a synchronous set. e must be preceded in n's plan by an unstarted, unfinished, or incomplete element e'.

Case 1.1:

If n's plan is of the form

$$< \ldots e' \ldots [\text{EVAL } j\ b]\ e \ldots >$$

then e would be performed by the same process that performed the [EVAL j b] element, after that element was finished. Since there was no gap before e was started, e' must have already been finished before the EVAL element was started. Thus, the execution of e at time t creates no gap.

Case 1.2:

If n's plan is of the form

$$< \ldots e' \ldots [\text{VISIT\_CHILD } j\ k]\ e \ldots >$$

or

$$< \ldots e' \ldots [\text{VISIT\_PARENT } k]\ e \ldots >$$

then e would be performed by the same process that performed the VISIT_CHILD or VISIT_PARENT element's completing visit. Further, the VISIT_CHILD or VISIT_PARENT element preceding e must not have begun execution prior to time t or there would have been a gap prior to time t. This would mean, however, that the completer of the VISIT_CHILD or VISIT_PARENT element was premature prior to time t, which is a contradiction.

Case 1.3:

If n's plan is of the form

$$< \ldots e' \ldots [\text{SYNCHSET } m]\ e_1 \ldots e_m\ e \ldots >$$

we consider two subcases.

If each of e_1 through e_m is an EVAL element, the same process that performs the SYNCHSET element will afterwards perform e. Since the plan had no gaps prior to the time e was started, e' must be finished before e begins.

If at least one of e_1 through e_m is a VISIT_CHILD or VISIT_PARENT element, then e will be executed either by the same process that performed the completing visit to one of e_1 through e_m (in which case the execution of e could not create a gap, by the reasoning of Case 1.2) or by the same process that performed the SYNCHSET element (in which case e's execution could not cause a gap, by the same reasoning as in the first part of Case 1.3).

Case 1.4:

If n's plan is of the form

$$< \ldots e'\ e \ldots >$$

then by reasoning similar to that of Cases 1.1-1.3, execution of e at time t could not produce a gap.

Case 2:

Assume e is a member of a synchronous set. The process performing e must be one spawned by the process performing the SYNCHSET element to which e belongs (recall that MAPUP and MAPDOWN have been altered to prevent a visit into the midst of a synchronous set). Since there were no gaps prior to time t, execution of e by this same process could not cause a gap at time t.

□

Lemma 5.2.3.3.4.2:

During attribution of a derivation tree by the algorithm of Figure 5.21, if, prior to time t, all plans are gap-free, there have been no premature visits, no plans have been started prematurely, and there have been no redundant plan element executions with respect to any nodes, then there are no premature visits at time t.

Proof: (by contradiction)

Assume that execution of VISIT_CHILD or VISIT_PARENT element e_1 at time t in the plan for node n_a is premature. e_1 therefore must not be a [VISIT_CHILD j 0] element. We proceed by cases.

Case 1:

Assume that e_1 is not a [VISIT_PARENT 0] element. The visit element e_1 completes in the plan for node n_b (call it e_1') must not have finished execution yet. Assume e_1' completed visit element e_1'' (which must be in n_a's plan and must be complete and have finished execution). By virtue of the nature of l-OAG planning algorithms, e_1 and e_1'' must be distinct plan elements. Let e_2 be the element immediately following e_1'' in n_a's plan. We can picture this as

n_a's plan: $< \ldots e_1'' \; e_2 \ldots e_1 \ldots >$

n_b's plan: $< \ldots e_1' \ldots >$

in the situation in which e_1 is a VISIT_CHILD.

Case 1.1:

Assume e_1'' is not a member of a synchronous set. By Lemma 5.2.3.3.4.1, n_a's plan must be gap-free at time t. This means that e_2 must have finished execution before time t. Further, since e_2 was performed by the same process that performed e_1', e_1' must also have finished execution prior to time t. Therefore, e_1 could not be premature.

Case 1.2:

Assume e_1'' is a member of a synchronous set. Without loss of generality, also assume that e_1'' is the rightmost (i.e., highest-indexed) element of its respective synchronous set. By Lemma 5.2.3.3.4.1, e_2 must be finished before e_1 starts executing. If e_2 is performed by the same process as was e_1' (as in Case 1.1), e_1's execution can not be premature.

If e_2 is performed by the same process that performed the SYNCHSET element associated with e_1'', then the execution of e_1' must have completed e_1'' and then finished execution before the SYNCHSET process moved on to execute e_1. So, again, e_1 can not be premature.

Finally, if e_2 is performed by a process p that performed some other completing visit element of another member of the synchronous set to which e_1'' belongs, then e_1' must have completed e_1'' and finished execution before process p would have been allowed to start e_1.

Case 2:

Assume e_1 is a premature [VISIT_PARENT 0] element. This means that e_1' must be a [VISIT_CHILD child_num(n_b) 0] element. We picture the situation as

n_b's plan: $< \ldots e_1' \ldots >$

n_a's plan: $< e_2 \ldots e_1 \ldots >$

By the same reasoning as in Case 1.1 and since n_a's plan can not have been started prematurely prior to time t, e_1's execution can not be premature.

□

Lemma 5.2.3.3.4.3:

During attribution of a derivation tree by the algorithm of Figure 5.21, if, prior to time t, all plans are gap-free, there have been no premature visits, no plans have been started prematurely, and there have been no redundant plan element executions with respect to any nodes, then there are no prematurely started plans at time t.

Proof:

Trivial. The process that begins executing the plan for the j^{th} child of node n does so only after it has completed execution of the element [VISIT_CHILD j 0] in n's plan. It follows immediately from Lemma 5.2.3.3.4.1 that the plan can not be started prematurely.

□

Theorem 5.2.3.3.4.4:

The algorithm of Figure 5.21 does no redundant plan element executions.

Proof:

Similar to the proofs of Lemmas 5.2.3.3.4.1 and 5.2.3.3.4.2.

□

Theorem 5.2.3.3.4.5:

The algorithm of Figure 5.21 produces consistent attribution of derivation trees.

Proof:

It is necessary only to show that no attribute instance value is calculated before the values of all determining attribute instances have been calculated. This follows immediately from the necessary properties of l-OAG plan generation and Lemmas 5.2.3.3.4.1 through 5.2.3.3.4.3.

□

Having proved consistency and efficiency, it remains only to prove that the algorithm produces a complete attribution.

Theorem 5.2.3.3.4.6:

The algorithm of Figure 5.21 produces a complete attribution.

Proof: (by contradiction)

Assume that for some derivation tree T the algorithm does not produce a complete attribution. By the nature of the algorithm, this means that at least one plan has at least one incomplete visit element with respect to some node. We proceed by cases.

Case 1:

The element e_1 =[VISIT_CHILD j k] in the plan for node n_1 is incomplete. By Theorem 5.2.3.3.4.5, the nature of the algorithm, and the necessary properties of l-OAG plan construction, it follows that node n_2, the j^{th} child of n_1, must have one or more incomplete visit elements. Further, by Lemma 5.2.3.3.4.2, n_2 must have only incomplete VISIT_CHILD elements. Immediately re-applying this reasoning, we can determine that some child node n_3 of node n_2 must also have at least one incomplete VISIT_CHILD element.

We can continue this chain of reasoning until we eventually determine that there must exist a leaf node of T with an incomplete VISIT_CHILD element. This is a contradiction, since the plans for leaf nodes can not contain VISIT_CHILD elements.

Case 2:

The element e_1 = [VISIT_PARENT k] in the plan for node n_1 is incomplete (e_1 can not be a LAST_VISIT_PARENT element). This case is handled similarly to Case 1. Eventually, we determine that the root of T must have an incomplete VISIT_PARENT element, which can not happen since the plan for a root node has only a single LAST_VISIT_PARENT element and no VISIT_PARENT elements.

□

One minor detail remains to be dealt with. The algorithm might appear to have a race condition in the VISIT_CHILD and VISIT_PARENT cases concerning the use of nodes' counters. For example, it seems as if the code

```
if k ≠ 0
    atomic_sub1(incomplete_counter(child(j,n)))
    fi;
if incomplete_counter(child(j,n)) ≠ 0
    return
else
    n := child(j, n);
```



```
        i := MAPDOWN(k, prod_num(n))
        fi
```

(from the VISIT_CHILD case) contains a race involving the setting and subsequent testing of

incomplete_counter(child(j,n))

by multiple processes.

There is, in fact, no harmful race condition. This can be verified by observing two facts. First, if a node n is *asleep* (i.e., no element in the plan for node n is being executed), then at most one of n's parent and child nodes can currently be *awake* (i.e., some process is executing an element of that node's plan). Thus, if node n is asleep, then there can not be multiple simultaneous attempts to visit n. The proof of this is based on the semantics of visit elements in l-OAG plans and the manner in which visit elements and synchronous sets are executed; the proof is left to the interested reader.

Second, if node n is awake and there are multiple simultaneous attempts to visit n, then n must currently be executing a SYNCHSET element. If n is executing a synchronous set, the value of its incomplete counter will be large enough that all of the attempted visits will be thwarted (i.e., their respective processes will execute a *return* statement). The proof of this, which is somewhat similar to the proof of the first fact above, is also left to the reader.

5.3 Series-Parallel Groups

Synchronous-set plans permit only a somewhat degenerate type of synchronous parallel attribution. The "flat" nature of synchronous sets does not allow us to take advantage of the rich structure present in many visit-sequence graphs.

For example, reconsider the VS graph of Figure 5.1 (repeated here as Figure 5.22). Synchronous-set planning allows us only two basic options.

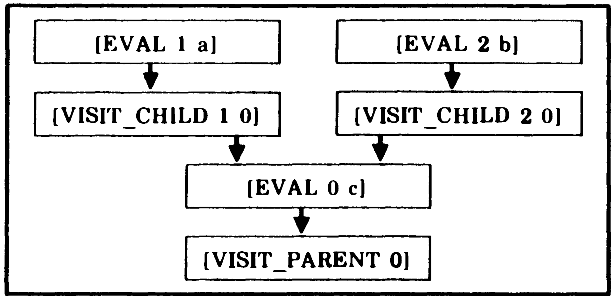

Figure 5.22 - A Visit-Sequence Graph

One option is to perform the two EVAL elements in parallel and then perform the two VISIT_CHILD elements in parallel. The other option is to perform one of the EVAL elements by itself, then perform the other EVAL element and one of the VISIT_CHILD elements in parallel, and then, finally, perform the remaining VISIT_CHILD by itself.

A third option, available if synchronous sets are abandoned, is to perform the two EVAL-VISIT_CHILD *sequences* in parallel. That is, the sequence of plan elements

[EVAL 1 a][VISIT_CHILD 1 0]

would be executed concurrently with the sequence

[EVAL 2 b][VISIT_CHILD 2 0]

After both sequences were completely executed, execution of the [EVAL 0 c] element could begin. Disregarding any additional time overhead due to

plan structure and given a sufficient number of processors, this third option would require no more and possibly significantly less time than either of the first two options.

To allow this new type of synchronous parallel execution, we need to construct and execute plans with a more complex structure than synchronous-set plans. A central concept in this is that of *series-parallel dags* [Valdes et al 1979].

5.3.1 Series-Parallel Dags

The *transitive reduction* of a directed acyclic graph G is the unique minimal dag G' having the same transitive closure as does G. A dag is said to be *minimal series-parallel* (or MSP) if it can be constructed according to the following rules:

(1) A dag having a single vertex and no edges is MSP.

(2) If $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are MSP, then $G = (V_1 \cup V_2, E_1 \cup E_2)$ is MSP.

(3) If $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are MSP, then $G = (V_1 \cup V_2, E_1 \cup E_2 \cup (A \times B))$, where A is the set of all sinks of G_1 and B is the set of all sources of G_2, is MSP.

Finally, a dag is *series-parallel* (or SP) if its transitive reduction is minimal series-parallel.

For example, the dags in Figure 5.23 are both MSP, one formed by applying rule (3) to node a and node b and the other formed by applying the same rule to nodes c and d. Combining these dags, using rule (2), results in a single MSP dag, shown in Figure 5.24. Using rule (3) to combine this dag with node e results in the MSP dag of Figure 5.25.

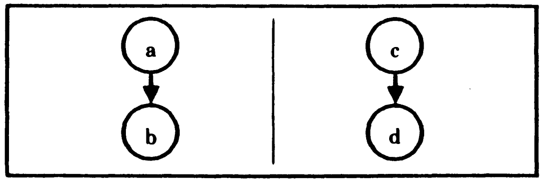

Figure 5.23 - Two MSP Dags

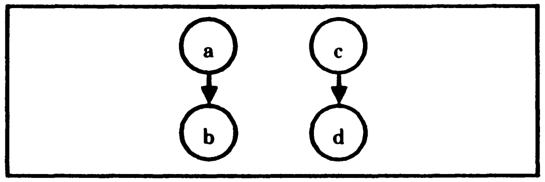

Figure 5.24 - A Single MSP Dag

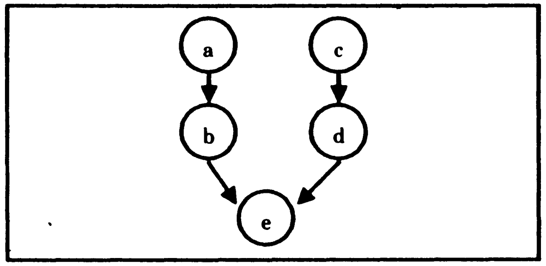

Figure 5.25 - A Single MSP Dag

In contrast, the dag depicted in Figure 5.26 is not MSP (and, hence, not SP).

The structure of SP dags corresponds well with the notion of synchronous parallel execution. Our intent is to construct for each production p in a grammar a plan whose structure corresponds to an SP dag

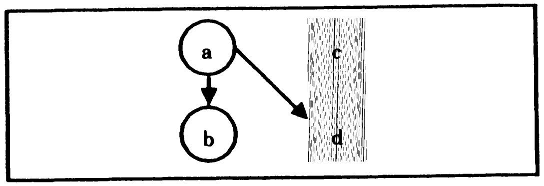

Figure 5.26 - A Non-MSP Dag

capturing all the dependencies in VS(p). We refer to such plans as *series-parallel plans* or *SP plans*.

5.3.2 SP Plan Elements and Structure

SP plans are composed of sequences of the following types of plan elements:

[EVAL j b]

[LAST_VISIT_PARENT k]

[VISIT_CHILD j k]

[VISIT_PARENT k] which have the same meaning and cause the same actions as they did under serial attribution,

[PARALLEL m l] which tells the evaluator that the next l consecutive plan elements constitute a group of m (m > 1) series of plan elements to be executed in parallel (Immediately following this element are m consecutive SERIES_START elements.), [SERIES_START l] which tells the evaluator that the first plan element in this particular series occurs l elements past this element, and

[SERIES_END] which tells the evaluator that it has reached the end of a series of plan elements being executed in parallel with other series.

We will refer to a [PARALLEL m l] element together with the l elements following it as a *parallel group*.

These elements allow us to represent a series-parallel-structured plan as a linear sequence of plan elements. For example, a possible SP plan for the VS graph of Figure 5.22 is

```
< [PARALLEL 2 8][SERIES_START 2][SERIES_START 4]
    [EVAL 1 a][VISIT_CHILD 1 0][SERIES_END]
    [EVAL 2 b][VISIT_CHILD 2 0][SERIES_END]
  [EVAL 0 c]
  [LAST_VISIT_PARENT 0]
  >
```

This representation allows for nested parallel groups. For the VS graph of Figure 5.27, a possible SP plan is

```
< [EVAL 1 a]
  [PARALLEL 2 15][SERIES_START 2][SERIES_START 4]
    [VISIT_CHILD 1 0][EVAL 0 c][SERIES_END]
    [EVAL 2 b][VISIT_CHILD 2 0]
     [PARALLEL 2 6][SERIES_START 2][SERIES_START 3]
        [EVAL 0 d] [SERIES_END]
        [EVAL 0 e] [SERIES_END]
```


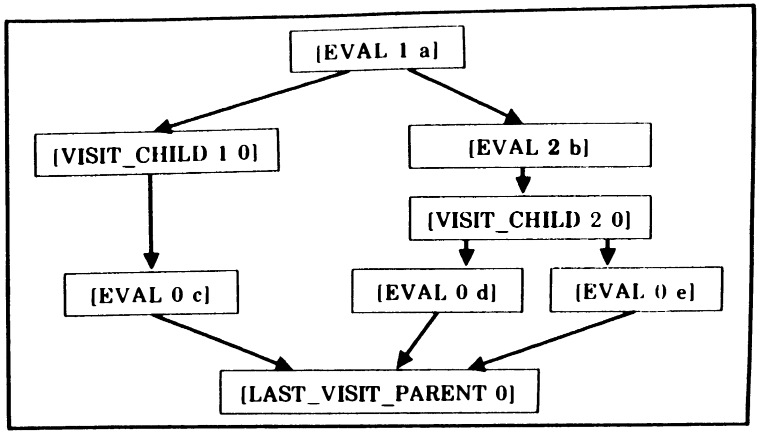

Figure 5.27 - A Visit-Sequence Graph

```
      [SERIES_END]
  [LAST_VISIT_PARENT 0]
  >
```

5.3.3 Generating SP Plans

As was the case with synchronous-set plans, it would be preferable to generate optimal SP plans. Optimality issues (local vs. global optimality, etc.) that pertained to synchronous-set plan generation may also be applicable to SP planning, and we will again restrict ourselves to considering only local optimality criteria. Producing optimal SP plans, as might be expected, is more involved than producing optimal synchronous-set plans.

The production of optimal SP plans can be idealized as the following problem:

Given dag $G = (V, E)$ with unique sink $v_s \in V$, lengths $l \in V \rightarrow N^{>0}$, and deadline $D \in N^{\geq 0}$, find $G' = (V, E \cup E')$ such that

(1) G' is series-parallel

(2) $\max\{cost(p) \mid p$ is a path from a source of G' to $v_s\} \leq D$

where $cost(p) =$ (the sum of $l(n)$ for all nodes n on path p)

If such a G' can be found, an SP plan meeting deadline D can be generated from G' in time $O(|V| + |E \cup E'|)$.

As has been the case before, we may also wish to make some types of plan elements forbidden, in this case meaning that such plan elements may not appear in parallel groups. PARALLEL, SERIES_START, and SERIES_END elements will, obviously, never be forbidden.

It is unclear whether optimal SP-planning is an NP-complete problem. Intuitively, it appears to be at least as difficult a problem as the generation of optimal synchronous-set plans.

Currently, we employ a simple heuristic method to generate reasonable SP plans. Initially, an SP plan P for production p consists of a sequence of parallel groups, each group corresponding to a single level of the VS graph for production p; when the algorithm below mentions a "level of P" or "level i of P," this is a reference to a parallel group in the plan. The basic concepts of the method are illustrated in Figure 5.28.

Each of the more significant steps in the algorithm can be implemented in a variety of ways. The method by which the value of set R is selected , the manner in which elements are combined in a series with elements on the preceding level, etc. all allow for variation in the particulars of the algorithm. If $G = (V,E)$ is the VS graph for production p, it


```
p: = a production;
P: = the SP plan analogous to the optimal unit-length synchronous-set plan
     for production p;
for i: = (# of levels in P) – 1 .. 1 do
     R: = some set of elements from level i of P such that including these
          elements in a series with their respective predecessor element(s)
          on level i – 1 of P will decrease the overall number of elements
          whose execution must wait for the completion of unrelated
          elements;
     remove the elements in R from level i of P;
     include each element in R in a series with its predecessor element(s)
          on level i – 1 of P
     od
```

Figure 5.28 - An SP-Planning Heuristic

is reasonable to expect a concrete implementation of the algorithm to take no more than $O(|V|^3)$ time.

5.3.4 Series-Parallel Attribution

We proceed by first assuming that, of the primary plan element types (i.e., EVAL, VISIT_CHILD, and VISIT_PARENT), only EVAL elements may be included in parallel groups. Next, we assume that VISIT_CHILD elements may also be done in parallel. Finally, we allow VISIT_PARENT elements to be done in parallel as well. As in previous cases, we assume that each derivation tree node contains at least its attribute instances, pointers to its parent and child nodes, the number of the production from which the subtree rooted at that node was derived, and the node's own child number.

The mappings MAPUP and MAPDOWN, constructed during plan generation, will be used as before. As will be seen, no special modifications will be needed to cause MAPUP and MAPDOWN to "skip" to the end of parallel groups, as was necessary under synchronous set attribution (§5.2.3.3.1). The responsibility for such "skipping" is instead shifted (when necessary at all) to the SERIES_END elements. The presence of SERIES_END elements will also eliminate the need for the going_up_from_parallel_visit predicate of §5.2.3.2.

5.3.4.1 SP Attribution with Parallel EVAL and VISIT_CHILD Elements

SP plans in which only EVAL elements may occur inside parallel groups can be executed in a very straightforward manner, as shown by the algorithm in Figure 5.29. What is perhaps surprising at first (and an obvious departure from the synchronous set case) is that the same algorithm will correctly execute SP plans in which *both* EVAL and VISIT_CHILD elements may appear inside parallel groups.

The function served by the going_up_from_parallel_visit predicate used during synchronous set attribution has been "compiled" into the SP plans in the form of SERIES_END elements. The code for processing EVAL, VISIT_CHILD, and VISIT_PARENT elements is identical to that used under serial attribution (§2.2.3.3). At the cost of increasing the size of plans, the inclusion of SERIES_END elements simplifies the algorithm by making the static structure of the plan more explicitly represent the flow of control that takes place during synchronous parallel attribution.


```
EvaluateSeriesParallelEC(treenode n, integer i):
  while PLAN[prod_num(n)][i] ≠ [LAST_VISIT_PARENT *])
        | n ≠ root of tree do
     case PLAN[prod_num(n)][i] of
        [PARALLEL m l]: {
           pforall j in 1 .. m do
              EvaluateSeriesParallelEC(n, i + j)
              od;
           i := i + l + 1
           };
        [SERIES_START l]:
           i := i + l;
        [SERIES_END]:
           return;
        [EVAL j b]: {
           value(child(j, n).b) := value of appropriate semantic fn;
           i := i + 1
           };
        [VISIT_CHILD j k]: {
           n := child(j, n);
           i := MAPDOWN(k, prod_num(n))
           };
        [LAST_VISIT_PARENT k]:
        [VISIT_PARENT k]: {
           i := MAPUP(k, child_num(n), prod_num(parent(n)));
           n := parent(n)
           }
        esac
   od
  end EvaluateSeriesParallelEC;
```

Figure 5.29 - An SP Attribution Algorithm for Parallel Groups Containing EVAL and VISIT_CHILD Elements

5.3.4.2 SP Attribution with Parallel EVAL, VISIT_CHILD, and VISIT_PARENT Elements

As was the case when VISIT_PARENT elements were permitted in synchronous sets (§5.2.3.3), the inclusion of VISIT_PARENT elements in parallel groups in SP plans introduces the possibility of non-hierarchical (with respect to the tree being attributed) control flow among the synchronous processes performing attribution. We deal with the problem by generalizing the solution described in §5.2.3.3.3.

With each derivation tree node n we associate a collection of counters. Conceptually, we need one counter for each PARALLEL element in N's plan. Actually, we need only as many counters as there are PARALLEL elements that may be active simultaneously in n's plan. A counter must be at least of size

$$\lceil \log(2 + (\textit{number of series for that PARALLEL element})) \rceil$$

bits.

When a [PARALLEL m l] element in the plan for node n is executed, it first initializes its associated counter to the value 1 + m. It then spawns m synchronous processes to execute the m series in the parallel group. When control returns to the element after termination of all m processes, it performs the same steps as does a SERIES_END element.

When a SERIES_END element is executed, it decrements by one the counter associated with its parallel group. It then acts based on the value of the counter. If the counter is non-zero, the process terminates by returning. If the counter is zero, the process continues execution with the first element following the parallel group to which the SERIES_END element belongs.

To make design of an attribution algorithm using this method simpler, we introduce modified versions of the PARALLEL and SERIES_END elements:

[PARALLEL m l c] which tells the evaluator that the next l consecutive plan elements constitute a parallel group of m series and that counter c is associated with this group

[SERIES_END l c] which tells the evaluator that this is the end of a series in a parallel group associated with counter c and that the next l consecutive plan elements are part of the same parallel group

Figure 5.30 gives an attribution algorithm driven by SP plans constructed using such elements. The notation

counters(n)

designates the sequence of counters associated with node n for use during the execution of parallel groups in n's plan.

A proof that the algorithm of Figure 5.30 produces a consistent, complete attribution is similar to the proof of Theorem 5.2.3.3.4.6 in §5.2.3.3.4 and is omitted here.

5.4 Related Work

[Schell 1979] discusses static order evaluation making use of constructs quite similar to synchronous sets containing EVAL elements only and synchronous sets containing VISIT_CHILD elements only. The possibilities for including VISIT_PARENT elements in synchronous sets and mixing types of plan elements within synchronous sets are not discussed. Neither SP attribution nor incremental attribution is covered.

```
EvaluateSeriesParallelECP(treenode n, integer i):
    while PLAN[prod_num(n)][i] ≠ [LAST_VISIT_PARENT *])
            | n ≠ root of tree do
        case PLAN[prod_num(n)][i] of
            [PARALLEL m l c]: {
                counters(n)[c] := 1 + m;
                pforall j in 1 .. m do
                    EvaluateSeriesParallelECP(n, i + j)
                    od;
                if atomic_decrement(counters(n)[c]) ≠ 0
                    return
                else
                    i := i + l + 1
                    fi
                };
            [SERIES_START l]:
                i := i + l;
            [SERIES_END l c]:
                if atomic_decrement(counters(n)[c]) ≠ 0
                    return
                else
                    i := i + l + 1
                    fi;
(continued)
```

Figure 5.30 - An SP Attribution Algorithm for Parallel Groups Containing EVAL, VISIT_CHILD, and VISIT_PARENT Elements

Figure 5.30 (Continued)

```
            [EVAL j b]: {
                value(child(j, n).b) := value of appropriate semantic fn;
                i := i + 1
                };
            [VISIT_CHILD j k]: {
                n := child(j, n);
                i := MAPDOWN(k, prod_num(n))
                };
            [LAST_VISIT_PARENT k]:
            [VISIT_PARENT k]: {
                i := MAPUP(k, child_num(n), prod_num(parent(n)));
                n := parent(n)
                }
            esac
        od
    end EvaluateSeriesParallelECP;
```

6. Parallel Incremental Attribution

This chapter presents the concept of incremental (re-)attribution of trees derived using l-OAG's. We show how many of the asynchronous parallel attribution methods presented in Chapter 4 can be adapted to perform optimal incremental attribution. We then show how many of the synchronous algorithms from Chapter 5 may also be modified for incremental use. Finally, we discuss some related research.

6.1 Incremental Attribution

In certain systems (e.g., editors, compilers, and program transformation systems), once a tree has been constructed and consistently attributed, it may be necessary to alter the tree. All such alterations may be characterized in terms of one or more *subtree replacements*; that is, with respect to attribute grammar G, some subtree U rooted at node n within consistently attributed derivation tree T is replaced with free-standing, consistently attributed subtree U' (U' was previously "clipped" out of some consistently attributed tree) rooted at node n' to obtain new derivation tree T'. Specifically, if n is the j^{th} child of its parent node m, we make n' the j^{th} child of m and make m the parent of n'. The attribute instance values in T – U and U' are left undisturbed.

After substituting U' for U, it is possible, even likely, that T' is inconsistently attributed. We wish to adjust the attribute instance values in T' so that T' is once again consistently attributed. A naive remedy is to treat T' as if it is entirely unattributed and completely re-attribute it. While this will certainly produce a consistent attribution, much of the work performed during this complete re-attribution may be unnecessary.

Conceivably, many attribute instances in T – U and U' may have the same values both before and after re-attribution of T'.

Let A(T') be the set of all attribute instances in T' and let AFFECTED be the subset of A(T') whose values change as a result of the consistent re-attribution of T'. Since |AFFECTED| may be considerably smaller than |A(T')|, it would be highly desirable to re-attribute T' in time O(|AFFECTED|) rather than in time O(|A(T')|), the time necessary for complete attribution of T'. It is important to note that both time bounds are with respect to a given grammar and assume constant-time semantic function evaluation.

An algorithm that can consistently re-attribute any such T' is said to be capable of *incremental attribution*. An algorithm that can incrementally attribute any such T' in time O(|AFFECTED|) is said to perform *optimal incremental attribution*.

It has been shown that any tree derived from any non-circular AG can be optimally incrementally attributed if the cost of certain bookkeeping operations is amortized over local cursor movements through the tree [Reps 1984]. It requires considerably less bookkeeping to perform optimal incremental attribution in the case of trees derived from l-OAG's [Reps & Teitelbaum 1989a, Yeh 1983].

6.2 Optimal Incremental Attribution and l-OAG's

Suppose we replace subtree U with root n of consistently attributed tree T with consistently attributed tree U' with root n'. The attribute instances of T' which may be immediately recognized as possibly having inconsistent values are those belonging to n', the parent of n', the children of n', and the siblings of n'. Changes in the values of any of these attribute instances may cause a "ripple" of changes to propagate outward from this initial region of inconsistency.

It is sufficient to begin re-attribution by simulating the execution of an element [VISIT_CHILD child_num(parent(n')) 0] in the plan for the "grandparent" of n'. In a complete serial attribution of T', all attribute instances whose values are determined prior to the execution of this plan element cannot directly or transitively depend on any attribute instances in U or U'. Thus, such attribute instances could not have differing respective values before and after re-attribution of T', and, therefore, cannot be members of AFFECTED.

It is also necessary to determine when to cease re-attribution. That is, it is necessary to recognize a point at which all attribute instances in AFFECTED have been given their new (and different) values and at which no more than O(|AFFECTED|) total work has been performed. This is easily done, as follows.

During incremental attribution, we maintain a set of the derivation tree nodes that are *active*. Initially only n' and its parent are considered active. If an input attribute instance of a production instance rooted at a node is observed to have changed in value as a result of re-attribution, that node is made active. Attempts to visit *inactive* nodes are treated as no-ops. Further, if the incremental attribution algorithm encounters a LAST_VISIT_PARENT element whose execution would cause a visit up to an inactive node, attribution ceases and every instance in AFFECTED is guaranteed to have been re-evaluated at that point.

A simple implementation of this scheme requires that each tree node n contain a bit designated by

active(n)

that is set to true if and only if n has been made active during incremental attribution. Derivation trees must be constructed with active(n) set to false for each node n. Similarly, incremental attribution must eventually leave active(n) set to false for all nodes n.

Figure 6.1 presents a serial optimal incremental attribution algorithm using this method. Note that each active node is made inactive when the LAST_VISIT_PARENT element in its plan is executed, leaving

```
IncrFAEvaluate(treenode n, integer i) :
   while active(n) do
      case PLANS[prod_num(n)][i] of
         [EVAL j b] : {
            if j ≠ 0
               candidate := child(j, n)
            elseif n ≠ root of tree
               candidate := parent(n)
            else
               candidate := none
               fi;
            previous_value := value(child(j, n).b);
            value(child(j,n).b) := value of appropriate semantic fn;
            if candidate ≠ none cand previous_value ≠ value(child(j, n).b)
               active(candidate) := true
               fi;
            i := i + 1
            };
(continued)
```

Figure 6.1 - A Serial Optimal Incremental 1-OAG Tree Evaluator

Figure 6.1 (Continued)

```
        [VISIT_CHILD j k]:
            if active(child(j, n)
                n := child(j, n);
                i := MAPDOWN(k, prod_num(n))
            else
                i := i + 1
                fi;
        [VISIT_PARENT k]:
            if active(parent(n))
                i := MAPUP(k, child_num(n), prod_num(parent(n)));
                n := parent(n)
            else
                i := i + 1
                fi;
        [LAST_VISIT_PARENT k]: {
            active(n) := false;
            if n ≠ root of tree cand active(parent(n))
                i := MAPUP(k, child_num(n), prod_num(parent(n)));
                n := parent(n)
                fi
            }
        esac
    od
end IncrFAEvaluate;
```

the tree in a state in which subsequent optimal re-attributions may be performed. To re-attribute tree T' after replacing subtree U by attributed tree U' with root n', execute the steps

active(n') := true;

active(parent(n')) := true;

IncrFAEvaluate(parent(n'), 0)

For brevity, the algorithm makes use of the *conditional-and* operator *cand*, similar to operators found in a variety of programming languages. An expression using this operator has the form

$$expression_1 \text{ cand } expression_2$$

(where $expression_1$ and $expression_2$ have Boolean values) and the following operational semantics. First, $expression_1$ is evaluated. If its value is false, the value of the conditional-and expression is false, and $expression_2$ is not evaluated. If the value of $expression_1$ is true, $expression_2$ is then evaluated, and its value is returned as the value of the entire conditional-and expression. (The dual *conditional-or* operator *cor* will also appear in subsequent algorithms.)

The definition of optimal incremental attribution relies on the assumption of constant-time semantic function evaluation. Examining the EVAL element case in Figure 6.1, we see that we must also assume constant-time copying and equality testing of attribute instance values. If attribute instance values may be arbitrarily complicated structures, it will be necessary to use a technique similar to that of *hashed constructors* [Goto 1974] to actually meet such assumptions.

Various optimizations may be employed in an effort to reduce the constant factor in the O(|AFFECTED|) time for incremental attribution [Reps & Teitelbaum 1989a]. As an example, suppose an output attribute instance b is given a new and different value during re-attribution as a result of the execution of some EVAL element in the plan for node n. This would usually cause either some child of n or the parent of n to be made active. However, if the production instance rooted at either the parent of n or the appropriate child of n makes no use of that input attribute instance (i.e., the corresponding input attribute occurrence does not appear in any semantic equation of the relevant production), this change in attribute instance value need not cause activation of any node.

This observation can be introduced as an optimization through the use of a predicate

has_successors(n, j, b)

that determines if attribute b of child j of n appears as an input attribute occurrence in any semantic equation in the production used to derive either parent(n) (if $j=0$) or child(j, n) (if $j \neq 0$). The predicate can be implemented with the aid of a mapping constructed during l-OAG membership determination. A possible use to prevent both unnecessary node activation as well as unnecessary attribute instance value equality testing is to modify the EVAL element case of the algorithm of Figure 6.1 as shown in Figure 6.2.

6.3 Asynchronous Parallel Optimal Incremental Attribution

Certain asynchronous parallel attribution algorithms from Chapter 4 can be modified to perform optimal incremental attribution. In this section we present incremental versions of some of those asynchronous parallel algorithms.

6.3.1 Asynchronous Execution of EVAL Elements

To extend the algorithm of §4.1 to perform optimal incremental attribution, we need to incorporate node activation and deactivation into the algorithm. Node activation is performed in this case by asynchronous


```
[EVAL j b]:{
  if j ≠ 0
     candidate := child(j, n)
  elseif n ≠ root of tree
     candidate := parent(n)
  else
     candidate := none
     fi;
  previous_value := value(child(j, n).b);
  value(child(j,n).b) := value of appropriate semantic fn;
  if candidate ≠ none cand has_successors(n, j, b)
        cand previous_value ≠ value(child(j,n).b)
     active(candidate) := true
     fi;
  i := i + 1
  };
```

Figure 6.2 - An Optimization of the Serial Optimal Incremental Evaluator

EVAL processes. This creates certain difficulties concerning the execution of visit elements and the deactivation of nodes.

6.3.1.1 Node Activation

Consider the sequence of elements

...[EVAL j b][EVAL j c][VISIT_CHILD j k]...

in the plan for some node n with j^{th} child node n'. In non-incremental asynchronous parallel attribution, an asynchronous process is spawned for each EVAL element. Execution of the VISIT_CHILD element can proceed without regard to whether the EVAL processes have terminated.

Now consider execution of the same sequence of elements during incremental attribution. The following situation may arise: n' is inactive, the two EVAL processes have been spawned but not completed, and the VISIT_CHILD element has been encountered. Correct execution of the VISIT_CHILD element depends on knowing if n' is active (in which case the visit is actually performed) or inactive (in which case the visit is skipped), and it is impossible to know if n' will become active until one of the EVAL processes activates n' or both of the EVAL processes terminate without activating n'.

Since n' is inactive when the EVAL processes are spawned and either process may cause activation of n', we say that each of these EVAL elements is a *potential activator* of node n' for this instance of tree attribution. We present two of a variety of approaches for coordinating visit elements and potential activators.

One approach is to require that potential activators be done in a non-asynchronous fashion. In the above example, when the [EVAL j b] element is encountered with n' inactive, that element is a potential activator of n' and would be done non-asynchronously. If, however, n' were already active, the [EVAL j b] element is not a potential activator of n', and an synchronous EVAL process would be spawned.

This method solves the problem at the cost of a possible loss of concurrency. In the worst case, all the EVAL elements in n's plan defining input attribute instance values for n' would be done asynchronously. Use of the has_successors predicate (as in §6.2) could restore some concurrency by reducing the numbers of potential activators for tree nodes.

An alternate approach involves mutual cooperation between EVAL elements and visit elements. With each tree node n we associate a counter of size

$$\lceil \log(1 + |\text{In}(\textit{production}\ \text{prod\_num}(n))|) \rceil$$

bits which is assumed to hold a value of zero at the start of incremental attribution. When an EVAL process is spawned to give a value to an input attribute instance of the production instance rooted at node n, n's counter is incremented by one. When the EVAL process terminates, n's counter is decremented by one. Strictly speaking, only asynchronous processes executing potential activators of a node need increment or decrement that node's counter.

Attempting a visit to an inactive node n requires examining n's counter. If the counter holds zero, no potential activators of n can be running, and the visit can be skipped. If, however, the counter is non-zero, the attribution routine must wait for the counter to take on a value of zero (as it eventually must, assuming all semantic functions are total) and then actually visit n only if n has become active.

6.3.1.2 Node Deactivation

An active node can be made inactive at any time after its plan has been completely executed. In the case of serial incremental attribution, a node can be deactivated when the LAST_VISIT_PARENT element in its plan is executed. In the parallel case in which nodes are made active through the actions of asynchronous EVAL processes, deactivation can be more complicated.

The difficulty arises since a naive attribution algorithm could attempt to activate a node a second time after the node had already been deactivated during the course of an incremental re-attribution of a derivation tree. Such a situation is possible since an asynchronous EVAL process defining an input attribute instance for a production instance rooted at a node n might not terminate until after the LAST_VISIT_PARENT element in n's plan has been completed. If nodes are deactivated as a result of the execution of LAST_VISIT_PARENT elements, "tardy" EVAL processes might attempt to "reactivate" nodes. Such reactivations could lead to non-optimal re-attribution following subsequent subtree replacements.

If all potential activators of all nodes are done non-asynchronously, as suggested in the preceding section, this problem does not occur: no asynchronous EVAL process is ever responsible for activating a node. If the possible loss of concurrency associated with this solution is unacceptable, and the potential activators counter method is used instead, the reactivation problem can still be dealt with. We present several possibilities.

The simplest solution is to do no node deactivation during re-attribution, but instead to perform all node deactivation after re-attribution is completed. This makes attribution a two-phase procedure. However, the deactivation phase is quite simple and typically has high potential concurrency.

A variation of the preceding method is to define active(n) to be an integer variable rather than a single bit Each act of subtree replacement is assigned a unique integer. Typically, the j^{th} act of subtree replacement is assigned the integer j. When a node n is made active, active(n) is assigned the integer associated with the most recent subtree replacement. Node n is considered active only if active(n) has a value equal to the number of the most recent subtree replacement. All active nodes are effectively

deactivated when the integer value identifying the most recent subtree replacement is changed.

Since the amount of storage allotted to active(n) will usually be finite (to permit constant-time assignment and comparison) this scheme places a limit on the number of subtree replacements allowed. If this limit is reached (which may be unlikely for many applications, given reasonably-sized active(n) fields), one could either reset the subtree replacement identifying value and reset the active(n) value of each node in the derivation tree, or one could let the subtree replacement value "wrap-around" (assuming modular arithmetic) and not reset any active(n) values.

The first approach above leads to a series of optimal re-attributions regularly interspersed with a tree-wide resetting of active(n) values. The latter approach leads to re-attributions that would be consistent, but whose optimality characteristics are probabilistic after the subtree replacement value begins to take on previously-held values.

A final alternative is to again have nodes deactivated as a result of the execution of the relevant LAST_VISIT_PARENT element, but to postpone execution of this element in the plan for node n until all EVAL processes which might attempt to (re)activate n have been spawned and have terminated. This method is simple to implement and combines easily with the use of the potential activators counter method for postponing visits (discussed in §6.3.1.1). However, such a method may again reduce concurrency.

6.3.1.3 Algorithms

If all potential activators are performed non-asynchronously, the serial optimal incremental attribution algorithm is readily adapted to provide asynchronous parallel execution of EVAL elements. The algorithm of Figure 6.3 illustrates this. After replacing subtree U of derivation tree T

```
IncrEvaluateAsynchE(treenode n, integer i) :
  while active(n) do
    case PLANS[prod_num(n)][i] of
      [EVAL j b] : {
        if j ≠ 0
          candidate := child(j, n)
        elseif n ≠ root of tree
          candidate := parent(n)
        else
          candidate := none
          fi;
        if candidate = none cor active(candidate)
            cor ¬has_successors(n, j, b)
          lockAttrInst(child(j, n).b);
          spawn EvaluateAttribute(n, j, b)
        else
          previous_value := value(child(j, n).b);
          value(child(j,n).b) := value of appropriate semantic fn;
          if previous_value ≠ value(child(j, n).b)
            active(candidate) := true
            fi
          fi;
        i := i + 1
        };
(continued)
```

Figure 6.3 - Incremental I OAG Tree Evaluator with Asynchronous Parallel EVAL Execution Using Non-Asynchronous Potential Activators

Figure 6.3 (Continued)

```
      [VISIT_CHILD j k] :
        if active(child(j, n)
          n := child(j, n);
          i := MAPDOWN(k, prod_num(n))
        else
          i := i + 1
          fi;
      [VISIT_PARENT k] :
        if active(parent(n))
          i := MAPUP(k, child_num(n), prod_num(parent(n)));
          n := parent(n)
        else
          i := i + 1
          fi;
      [LAST_VISIT_PARENT k] : {
        active(n) := false;
        if n ≠ root of tree cand active(parent(n))
          i := MAPUP(k, child_num(n), prod_num(parent(n)));
          n := parent(n)
          fi
        }
      esac
    od
  end IncrEvaluateAsynchE;

EvaluateAttribute(treenode n, integer j, attribute b) :
  value(child(j, n).b) := value of appropriate semantic fn;
  unlockAttrInst(child(j, n).b)
  end EvaluateAttribute;
```

by subtree U' with root node n' to produce tree T', T' would be re-attributed with the steps

```
active(n') := true;
active(parent(n')) := true;
IncrEvaluateAsynchE(parent(n'), 0);
wait for all asynchronous processes to terminate;
```

If we instead choose to maintain a count of executing potential activators for each node, we may employ any of the techniques discussed in §6.3.1.2 to prevent node reactivation. The algorithm shown in Figure 6.4 deactivates nodes using a second pass over the tree, given in Figure 6.5. After subtree replacement, T' is re-attributed by the steps

```
active(n') := true;
active(parent(n')) := true;
IncrEvaluateAsynchE(parent(n'), 0);
wait for all asynchronous processes to terminate;
Deactivate(parent(n'));
wait for all asynchronous processes to terminate;
```

The counter designated by

pendingActivators(n)

at any moment holds the number of executing potential activators for n.

For convenience, the algorithm uses an extension of the *wait* operation of the form

wait(*boolean_expression*)

that allows a process to continue only when *boolean_expression* is true. In general, such a statement may be quite difficult to implement efficiently. We will limit its use to expressions for which such waiting can be efficiently performed.

The next attribution algorithm is given in Figure 6.6. This version


```
IncrEvaluateAsynchE(treenode n, integer i) :
  done := false;
  while ¬done do
    case PLANS[prod_num(n)][i] of
      [EVAL j b] : {
        if j ≠ 0
          candidate := child(j, n)
        elseif n ≠ root of tree
          candidate := parent(n)
        else
          candidate := none
          fi;
        lockAttrInst(child(j, n).b);
        if candidate = none cor active(candidate)
              cor ¬has_successors(n, j, b)
          spawn EvaluateAttribute(n, j, b, candidate, false)
        else
          atomic_add1(pendingActivators(candidate));
          spawn EvaluateAttribute(n, j, b, candidate, false)
          fi;
        i := i + 1
        };
      [VISIT_CHILD j k] : {
        wait(pendingActivators(child(j, n)) = 0 | active(child(j, n)));
        if active(child(j, n)
          n := child(j, n);
          i := MAPDOWN(k, prod_num(n))
        else
          i := i + 1
          fi
        };
(continued)
```

Figure 6.4 - Incremental I OAG Tree Evaluator with Asynchronous Parallel EVAL Execution using Potential Activator Counters

Figure 6.4 (Continued)

```
        [VISIT_PARENT k] : {
            wait(pendingActivators(parent(n)) = 0 | active(parent(n)));
            if active(parent(n))
                i := MAPUP(k, child_num(n), prod_num(parent(n)));
                n := parent(n)
            else
                i := i + 1
                fi
            };
        [LAST_VISIT_PARENT k] :
            if n ≠ root of tree
                wait(pendingActivators(parent(n)) = 0 | active(parent(n)));
                if active(parent(n))
                    i := MAPUP(k, child_num(n), prod_num(parent(n)));
                    n := parent(n)
                else
                    done := true
                    fi
            else
                done := true
                fi
        esac
    od
  end IncrEvaluateAsynchE;

EvaluateAttribute(treenode n, integer j, attribute b, treenode candidate,
                  boolean activator) :
  previous_value := value(child(j, n).b);
  value(child(j, n).b) := value of appropriate semantic fn;
(continued)
```


Figure 6.4 (Continued)

```
  if activator
    if previous_value ≠ value(child(j, n).b)
        active(candidate) := true
        fi;
    atomic_sub1(pendingActivators(candidate))
    fi;
  unlockAttrInst(child(j, n).b)
  end EvaluateAttribute;
```

```
Deactivate(treenode n) :
    active(n) := false;
    if n ≠ root of tree cand active(parent(n))
        spawn Deactivate(parent(n))
        fi;
    forall child nodes n' of n do
        if active(n')
            spawn deactivate(n')
            fi
        od
    end Deactivate;
```

Figure 6.5 - Node Deactivation Procedure

uses the unique subtree replacement number approach to preventing node reactivation. Trees are re-attributed using the steps

active(n') := *unique value associated with most recent subtree replacement*;

active(parent(n')) := active(n');

IncrEvaluateAsynchE(parent(n'), 0, active(n'));

wait for all asynchronous processes to terminate;

The final version, shown in Figure 6.7, uses the pendingActivators(n) counter in each node n both to delay visits until relevant node activations are performed and to delay execution of LAST_VISIT_PARENT elements in order to prevent node reactivation. Re-attribution of tree T' is done by

```
IncrEvaluateAsynchE(treenode n, integer i, integer replacement_id) :
    done := false;
    while ¬done do
        case PLANS[prod_num(n)][i] of
            [EVAL j b] : {
                if j ≠ 0
                    candidate := child(j, n)
                elseif n ≠ root of tree
                    candidate := parent(n)
                else
                    candidate := none
                    fi;
                lockAttrInst(child(j, n).b);
                if candidate = none cor active(candidate) = replacement_id
                        cor ¬has_successors(n, j, b)
                    spawn EvaluateAttribute(n, j, b, candidate, false,
                            replacement_id)
                else
                    atomic_add1(pendingActivators(candidate));
                    spawn EvaluateAttribute(n, j, b, candidate, false,
                            replacement_id)
                    fi;
                i := i + 1
                };
(continued)
```

Figure 6.6 - Incremental I-OAG Tree Evaluator with Asynchronous Parallel EVAL Execution using Potential Activator Counters and Unique Subtree Replacement Numbers

Figure 6.6 (Continued)

```
            [VISIT_CHILD j k] : {
                wait(pendingActivators(child(j, n)) = 0
                        | active(child(j, n)) = replacement_id);
                if active(child(j, n) = replacement_id
                    n := child(j, n);
                    i := MAPDOWN(k, prod_num(n))
                else
                    i := i + 1
                    fi
                };
            [VISIT_PARENT k] : {
                wait(pendingActivators(parent(n)) = 0
                        | active(parent(n)) = replacement_id);
                if active(parent(n)) = replacement_id
                    i := MAPUP(k, child_num(n), prod_num(parent(n)));
                    n := parent(n)
                else
                    i := i + 1
                    fi
                };
            [LAST_VISIT_PARENT k] :
                if n ≠ root of tree
                    wait(pendingActivators(parent(n)) = 0
                            | active(parent(n)) = replacement_id);
                    if active(parent(n)) = replacement_id
                        i := MAPUP(k, child_num(n), prod_num(parent(n)));
                        n := parent(n)
                    else
                        done := true
                        fi
(continued)
```

Figure 6.6 (Continued)

```
            else
                done := true
                fi
        esac
    od
  end IncrEvaluateAsynchE;

EvaluateAttribute(treenode n, integer j, attribute b, treenode candidate,
                  boolean activator, integer replacement_id) :
  previous_value := value(child(j, n).b);
  value(child(j, n).b) := value of appropriate semantic fn;
  if activator
     if previous_value ≠ value(child(j, n).b)
        active(candidate) := replacement_id
        fi;
     atomic_sub1(pendingActivators(candidate))
     fi;
  unlockAttrInst(child(j, n).b)
  end EvaluateAttribute;
```

```
active(n') := true;
active(parent(n')) := true;
IncrEvaluateAsynchE(parent(n'), 0);
wait for all asynchronous processes to terminate;
```

6.3.2 Asynchronous Execution of VISIT_CHILD Elements

As discussed in §4.3, the asynchronous execution of VISIT_CHILD elements can result in elements in the plan for a node n being encountered and executed out of sequence with respect to that plan. That is, some element e_{j+k} ($j \geq 0$, $k > 0$) in n's plan can be encountered and executed before


```
IncrEvaluateAsynchE(treenode n, integer i) :
  while active(n) do
    case PLANS[prod_num(n)][i] of
      [EVAL j b] : {
         if j ≠ 0
            candidate := child(j, n)
         elseif n ≠ root of tree
            candidate := parent(n)
         else
            candidate := none
            fi;
         lockAttrInst(child(j, n).b);
         if candidate = none cor active(candidate)
                cor ¬has_successors(n, j, b)
            spawn EvaluateAttribute(n, j, b, candidate, false)
         else
            atomic_add1(pendingActivators(candidate));
            spawn EvaluateAttribute(n, j, b, candidate, false)
            fi;
         i := i + 1
         };
      [VISIT_CHILD j k] : {
         wait(pendingActivators(child(j, n)) = 0 | active(child(j, n)));
         if active(child(j, n)
            n := child(j, n);
            i := MAPDOWN(k, prod_num(n))
         else
            i := i + 1
            fi
         };
(continued)
```

Figure 6.7 - Incremental 1-OAG Tree Evaluator with Asynchronous Parallel EVAL Execution using Potential Activator Counters

Figure 6.7 (Continued)

```
        [VISIT_PARENT k] : {
            wait(pendingActivators(parent(n)) = 0 | active(parent(n)));
            if active(parent(n))
                i := MAPUP(k, child_num(n), prod_num(parent(n)));
                n := parent(n)
            else
                i := i + 1
                fi
            };
        [LAST_VISIT_PARENT k] : {
            if n ≠ root of tree
                wait(pendingActivators(parent(n)) = 0 |active(parent(n)));
                if active(parent(n))
                    spawn IncrEvaluateAsynchE(
                        parent(n),
                        MAPUP(k, child_num(n), prod_num(parent(n)))
                        )
                    fi
                fi;
            wait(pendingActivators(n) = 0);
            active(n) := false
            }
        esac
    od
  end IncrEvaluateAsynchE;

EvaluateAttribute(treenode n, integer j, attribute b, treenode candidate,
                boolean activator) :
  previous_value := value(child(j, n).b);
  value(child(j, n).b) := value of appropriate semantic fn;
(continued)
```


Figure 6.7 (Continued)

```
  if activator
    if previous_value ≠ value(child(j, n).b)
      active(candidate) := true
      fi;
    atomic_sub1(pendingActivators(candidate))
    fi;
  unlockAttrInst(child(j, n).b)
  end EvaluateAttribute;
```

element e_j is encountered. In the non-incremental case in §4.3, we dealt with this problem either by locking all attribute instances in the entire derivation tree prior to the start of attribution or by locking all attribute instances of all children of each node n the first time (chronologically) n is visited during attribution. Such approaches will not work for incremental attribution: complications exist concerning when and if a given attribute instance may be locked, as illustrated below.

Consider the I-OAG grammar fragment shown in Figure 6.8, the plans generated for that fragment (shown in Figure 6.9), and the subtree in Figure 6.10. (These figures are in much the same vein as Figures 4.2, 4.3, and 4.4 in §4.3.) Assume that node n_0 has been activated and its plan started from the beginning.

Consider the following possible sequence of events (listed in chronological order):

(1) $[\text{EVAL 1 xi1}]_8$ is executed by process q_0, causing activation of node n_1.

(2) $[\text{VISIT\_CHILD 1 0}]_9$ is executed by process q_0, spawning process q_1 to perform the visit to node n_1.

AI(W) = {wi1, wi2}, AS(W) = {ws1, ws2}
AI(X) = {xi1, xi2}, AS(X) = {xs1, xs2}
AI(Y) = {yi1}, AS(Y) = {ys1}
A(Z) = ∅

p_0: W → X Z
{ X.xi1 = f_1(W.wi1);
W.ws1 = f_2(X.xs1);
X.xi2 = f_3(W.wi2);
W.ws1 = f_4(X.xs2) }

p_1: W → X
{ X.xi1 = f_5(W.wi1);
W.ws1 = f_6(W.wi1);
X.xi2 = f_7(W.wi1);
W.ws1 = f_8(X.xs2) }

p_2: X → Y
{ Y.yi1 = f_9(X.xi1);
X.xs1 = f_{10}(Y.ys1);
X.xs2 = f_{11}(Y.ys1, X.xi2) }

p_3: Y → '0'
{ Y.ys1 = f_{12}(Y.yi1) }

p_4: Z → '1'

Figure 6.8 - A Grammar Fragment

(3) [EVAL 0 xws1]$_{10}$ through [EVAL 1 xi2]$_{12}$ are executed sequentially by process q_0.

(4) [VISIT_CHILD 1 1]$_{13}$ is executed by process q_0, spawning process q_2 to perform the visit to node n_1.

(5) [EVAL 0 xs2]$_{20}$ is encountered by process q_2.

At this point, should q_2 be permitted to determine a (new) value for attribute instance n_1.xs2? The answer is not entirely obvious.

<<[EVAL 1 xi1]$_0$ [VISIT_CHILD 1 0]$_1$
[EVAL 0 ws1]$_2$ [VISIT_PARENT 0]$_3$
[EVAL 1 xi2]$_4$ [VISIT_CHILD 1 1]$_5$
[EVAL 0 ws1]$_6$ [LAST_VISIT_PARENT 1]$_7$>
<[EVAL 1 xi1]$_8$ [VISIT_CHILD 1 0]$_9$
[EVAL 0 ws1]$_{10}$ [VISIT_PARENT 0]$_{11}$
[EVAL 1 xi2]$_{12}$ [VISIT_CHILD 1 1]$_{13}$
[EVAL 0 ws1]$_{14}$ [LAST_VISIT_PARENT 1]$_{15}$>
<[EVAL 1 yi1]$_{16}$ [VISIT_CHILD 1 0]$_{17}$
[EVAL 0 xs1]$_{18}$ [VISIT_PARENT 0]$_{19}$
[EVAL 0 xs2]$_{20}$ [LAST_VISIT_PARENT 1]$_{21}$>
<[EVAL 0 ys1]$_{22}$ [LAST_VISIT_PARENT 0]$_{23}$>
<>
>

Figure 6.9 - Plans for the Grammar Fragment

n_0: W
|
n_1: X
|
n_2: Y
|
n_3: '0'

Figure 6.10 - A Subtree

If node n_2 will never be activated at any point in the course of this particular incremental attribution, q_2 could certainly proceed at this point. If node n_2 will be activated at a later time, q_2 must instead wait until the (new) value of attribute instance n_2.ys1 is determined before proceeding.

Whether q_2 is allowed to proceed apparently depends on the future actions of process q_1. It would seem that an appropriate locking protocol could be employed to delay q_2 when necessary.

Unfortunately, the situation is more complicated. Assume instead that node n_0's plan was started at the element $[\text{EVAL } 1 \text{ xi2}]_{12}$ rather than at the beginning of the plan (as might happen under incremental re-attribution).

Consider the following possible sequence of events:

(1) $[\text{EVAL } 1 \text{ xi2}]_{12}$ is executed by process q_0, causing activation of node n_1.

(2) $[\text{VISIT\_CHILD } 1 \ 1]_{13}$ is executed by process q_0, spawning process q_2 to perform the visit to node n_1.

(3) $[\text{EVAL } 0 \text{ xs2}]_{20}$ is encountered by process q_2.

In this situation, q_2 may safely calculate the value of instance n_1.xs2 since the value of instance n_2.ys1 cannot have been changed by re-attribution.

From the point of view of process q_2, both sequences of events above appear identical. In particular, q_2 is the first process chronologically to begin work on the plan for node n_1. Yet, in one case q_2 must wait for another process to complete possibly crucial work before proceeding, while in the other case q_2 must not wait on any other process

It appears that no simple attribute instance locking scheme can resolve this problem in all cases. The specific difficulty lies in knowing when and if instance n_2.ys1 should be locked. Should n_2.ys1 be locked in a situation in which node n_2 is not subsequently activated, the lock will never be removed, and process q_2 will be blocked forever. Should n_2.ys1 not be

locked when, in fact, node n_2 will be activated, the value given to n_1.xs2 may be inconsistent.

Conceivably, a locking scheme could be devised to deal with this problem (perhaps a method involving keeping explicit track of dependencies among asynchronous processes). However, such a scheme would likely defeat some of the advantages of abandoning dynamic evaluation orders in favor of static evaluation orders for AG's. This being the case, we will consider the asynchronous execution of VISIT_CHILD (or, analogously, VISIT_PARENT) elements alone as unsuitable where optimal incremental attribution is to be performed.

6.3.3 Asynchronous Execution of Visit Elements

The view of asynchronous execution of both VISIT_CHILD and VISIT_PARENT elements taken in §4.5 extends readily to allow optimal incremental attribution.

Recall that the locking scheme adopted in §4.5 was to lock all output attribute instances of the production instance rooted at node n prior to spawning the asynchronous process to execute n's plan. This was done by calling the procedure

lockOutputAttrInsts(n)

just prior to the initial visit to node n. In the non-incremental case, this initial visit was always accomplished by an element of the form [VISIT_CHILD j 0].

To allow optimal incremental attribution, two modifications are made to this scheme. First, the first visit (chronologically) to each node n during incremental re-attribution must be detected. In order to detect this first visit, each node must contain a bit designated by

visited(n)

and initially set to zero. This bit is set to one by the process spawning the asynchronous process that will execute whatever portion of n's plan is to be performed.

A second (related) modification is a simple extension of the lockOutputAttrInsts procedure to the procedure

lockRemainingOutputAttrInsts(n, i)

that locks all output attribute instances evaluated by elements in n's plan having plan indices greater than or equal to i. As always, this procedure could be based on either static tables derived during plan generation or by interpreting n's plan at attribution time.

6.3.3.1 Node Activation

Unlike in §6.3.1, the coordination of node activation and node visits is not a problem here. Since EVAL elements are executed non-asynchronously and since elements within any given node's plan are executed sequentially, whenever an element attempting a visit to a node n is encountered, the active/inactive status of n is certain.

6.3.3.2 Node Deactivation

Deactivating nodes in a manner that precludes node reactivation is again a non-trivial problem. Two of the methods presented in §6.3.1.2 (i.e., the separate deactivation phase and the unique subtree replacement number methods) are easily adapted for use with asynchronous visits. The

method employing a counter of unterminated potential activators may also be used, but requires significant modification.

We have heretofore considered methods applicable to attribution in the face of non-strict semantic functions. Assuming strict semantic functions would have had few or no ramifications for the algorithms presented previously. With visits performed asynchronously, forcing all semantic functions to be strict eliminates the possibility of node reactivation, at the cost of some concurrency.

6.3.3.3 Attribution Assuming Strict Semantic Functions

As seen in Figure 6.11, this algorithm is a straightforward hybrid of the serial optimal incremental attribution algorithm and the algorithm of §4.5. To re-attribute tree T', again constructed by replacing subtree U of T by subtree U' with root node n', one could use the steps

active(n') := true;
active(parent(n')) := true;
visited(parent(n')) := true;
lockRemainingOutputAttrInsts(parent(n'), 0);
wait for all asynchronous processes to terminate;

Note that the order in which the active(n) and visited(n) bits are examined in the visit cases and are set to false in the LAST_VISIT_PARENT case is carefully determined in order to prevent race conditions.

6.3.3.4 Attribution Assuming Non-Strict Semantic Functions

While non-strict semantic functions complicate matters somewhat, if we employ a separate deactivation phase, we obtain the algorithm depicted

```
IncrEvaluateAsynchCP(treenode n, integer i) :
    while active(n) do
        case PLANS[prod_num(n)][i] of
            [EVAL j b] : {
                if j ≠ 0
                    candidate := child(j, n)
                elseif n ≠ root of tree
                    candidate := parent(n)
                else
                    candidate := none
                    fi;
                previous_value := value(child(j, n).b);
                value(child(j,n).b) := value of appropriate semantic fn;
                if candidate ≠ none cand has_successors(n, j, b)
                        cand previous_value ≠ value(child(j,n).b)
                    active(candidate) := true
                    fi;
                unlockAttrInst(child(j, n).b);
                i := i + 1
                };
            [VISIT_CHILD j k] : {
                if ¬visited(child(j, n)) cand active(child(j, n)
                    visited(child(j, n)) := true;
                    lockRemainingOutputAttrInsts(
                        child(j, n), MAPDOWN(k, prod_num(n)));
                    spawn IncrEvaluateAsynchCP(
                        child(j, n), MAPDOWN(k, prod_num(n)))
                    fi;
                i := i + 1
                };
(continued)
```

Figure 6.11 - Incremental 1-OAG Tree Evaluator with Asynchronous Parallel Visit Execution Assuming Strict Semantic Functions

Figure 6.11 (Continued)

```
            [VISIT_PARENT k] : {
                if ¬visited(parent(n)) cand active(parent(n))
                    visited(parent(n)) := true;
                    lockRemainingOutputAttrInsts(
                        parent(n),
                        MAPUP(k, child_num(n), prod_num(parent(n)))
                        );
                    spawn IncrEvaluateAsynchCP(
                        parent(n),
                        MAPUP(k, child_num(n), prod_num(parent(n)))
                        )
                    fi;
                i := i + 1
                };
            [LAST_VISIT_PARENT k] : {
                active(n) := false;
                visited(n) := false;
                if n ≠ root of tree cand ¬visited(parent(n))
                        cand active(parent(n))
                    visited(parent(n)) := true;
                    lockRemainingOutputAttrInsts(
                        parent(n),
                        MAPUP(k, child_num(n), prod_num(parent(n)))
                        );
                    i := MAPUP(k, child_num(n), prod_num(parent(n)));
                    n := parent(n)
                    fi
                }
            esac
        od
    end IncrEvaluateAsynchCP;
```

in Figure 6.12. This is arguably the simplest of all the asynchronous parallel optimal incremental attribution algorithms. While requiring two passes over the affected region of the derivation tree, the second pass (performed by the procedure shown in Figure 6.13) is again quite simple and highly concurrent. Derivation tree T' can be re-attributed by executing

```
active(n') := true;
active(parent(n')) := true;
visited(parent(n')) := true;
lockRemainingOutputAttrInsts(parent(n'), 0);
IncrEvaluateAsynchCP(parent(n'), 0);
wait for all asynchronous processes to terminate;
Deactivate(n');
wait for all asynchronous processes to terminate
```

We can again make use of unique subtree replacement identification numbers. The need to maintain both active/inactive status and visited/unvisited status requires, conceptually, that both active(n) and visited(n) be set to the number of the most recent subtree replacement when necessary. The subtree replacement number must not be allowed to take on previously held values unless all nodes' visited(n) values are first reset. Failure to do so could lead to incomplete re-attribution.

Figure 6.14 illustrates this method. Tree T' would be re-attributed by the steps

```
active(n') := unique value associated with the most
        recent subtree replacement;
active(parent(n')) := active(n');
visited(parent(n')) := active(n');
```



```
IncrEvaluateAsynchCP(treenode n, integer i):
  done := false;
  while ¬done do
    case PLANS[prod_num(n)][i] of
      [EVAL j b]: {
        if j ≠ 0
          candidate := child(j, n)
        elseif n ≠ root of tree
          candidate := parent(n)
        else
          candidate := none
          fi;
        previous_value := value(child(j, n).b);
        value(child(j,n).b) := value of appropriate semantic fn;
        if candidate ≠ none cand has_successors(n, j, b)
            cand previous_value ≠ value(child(j, n).b)
          active(candidate) := true
          fi;
        unlockAttrInst(child(j, n).b);
        i := i + 1
        };
      [VISIT_CHILD j k]: {
        if ¬visited(child(j, n)) & active(child(j, n)
          visited(child(j, n)) := true;
          lockRemainingOutputAttrInsts(
            child(j, n), MAPDOWN(k, prod_num(n)));
          spawn IncrEvaluateAsynchCP(
            child(j, n), MAPDOWN(k, prod_num(n)))
          fi;
(continued)
```

Figure 6.12 - Incremental l OAG Tree Evaluator with Asynchronous Parallel Visit Execution Assuming Non-Strict Semantic Functions Using a Separate Deactivation Phase

Figure 6.12 (Continued)

```
            i := i + 1
            };
        [VISIT_PARENT k] : {
            if ¬visited(parent(n)) & active(parent(n))
                visited(parent(n)) := true;
                lockRemainingOutputAttrInsts(
                    parent(n),
                    MAPUP(k, child_num(n), prod_num(parent(n)))
                    );
                spawn IncrEvaluateAsynchCP(
                    parent(n),
                    MAPUP(k, child_num(n), prod_num(parent(n)))
                    )
                fi;
            i := i + 1
            };
        [LAST_VISIT_PARENT k] :
            if n ≠ root of tree
                    cand ( ¬visited(parent(n)) & active(parent(n)))
                visited(parent(n)) := true;
                lockRemainingOutputAttrInsts(
                    parent(n),
                    MAPUP(k, child_num(n), prod_num(parent(n)))
                    );
                i := MAPUP(k, child_num(n), prod_num(parent(n)));
                n := parent(n)
            else
                done := true
                fi
        esac
    od
end IncrEvaluateAsynchCP;
```



```
Deactivate(treenode n) :
    active(n) := false;
    visited(n) := false;
    if n ≠ root of tree cand active(parent(n))
        spawn Deactivate(parent(n))
        fi;
    forall child nodes n' of n do
        if active(n')
            spawn Deactivate(n')
            fi
        od
    end Deactivate;
```

Figure 6.13 - Deactivation Procedure

lockRemainingOutputAttrInsts(parent(n'), 0);

IncrEvaluateAsynchCP(parent(n'), 0);

wait for all asynchronous processes to terminate

Finally, we can adopt the strategy of delaying execution of the LAST_VISIT_PARENT element for a node (and the associated deactivation of the node) until all EVAL elements that might attempt to activate or reactivate the node have been executed. Implementing this strategy is more complicated than was the case in §6.3.1, owing to the fact that some potential activators and reactivators of a node may not yet have been encountered when the node's LAST_VISIT_PARENT element is encountered and executed. Fortunately, this can be managed by the following method.

Each node is allotted an initially-zero counter

pendingActivators(n)

of at least size

$$\lceil \log(1 + |In(\textit{production}\ prod\_num(n))|) \rceil$$

Whenever a visit in the plan for a node n spawns a process to execute the plan for a node n', two counter operations occur. First, pendingActivators(n) is incremented by the number of input attribute instances of the production instance rooted at n that are given values in the yet-to-be-executed portion of the plan for n'. Second, pendingActivators(n')

```
IncrEvaluateAsynchCP(treenode n, integer i, integer replacement_id):
  done := false;
  while ¬done do
    case PLANS[prod_num(n)][i] of
      [EVAL j b]: {
        if j ≠ 0
          candidate := child(j, n)
        elseif n ≠ root of tree
          candidate := parent(n)
        else
          candidate := none
          fi;
        previous_value := value(child(j, n).b);
        value(child(j,n).b) := value of appropriate semantic fn;
        if candidate ≠ none cand has_successors(n, j, b)
            cand previous_value ≠ value(child(j, n).b)
          active(candidate) := replacement_id
          fi;
        unlockAttrInst(child(j, n).b);
        i := i + 1
        };
(continued)
```

Figure 6.14 - Incremental I-OAG Tree Evaluator with Asynchronous Parallel Visit Execution Assuming Non-Strict Semantic Functions Using Unique Subtree Replacement Numbers

Figure 6.14 (Continued)

```
      [VISIT_CHILD j k]: {
        if visited(child(j, n)) ≠ replacement_id
            & active(child(j, n) = replacement_id
          visited(child(j, n)) := replacement_id;
          lockRemainingOutputAttrInsts(
            child(j, n), MAPDOWN(k, prod_num(n)));
          spawn IncrEvaluateAsynchCP(
            child(j, n), MAPDOWN(k, prod_num(n)))
          fi;        i := i + 1
        };
      [VISIT_PARENT k]: {
        if visited(parent(n)) ≠ replacement_id
            & active(parent(n)) = replacement_id
          visited(parent(n)) := replacement_id;
          lockRemainingOutputAttrInsts(
            parent(n),
            MAPUP(k, child_num(n), prod_num(parent(n)))
            );
          spawn IncrEvaluateAsynchCP(
            parent(n),
            MAPUP(k, child_num(n), prod_num(parent(n)))
            )
          fi;
        i := i + 1
        };
      [LAST_VISIT_PARENT k]:
        if n ≠ root of tree
            cand (visited(parent(n)) ≠ replacement_id
                & active(parent(n)) = replacement_id)
(continued)
```

Figure 6.14 (Continued)

```
            visited(parent(n)) := replacement_id;
            lockRemainingOutputAttrInsts(
                parent(n),
                MAPUP(k, child_num(n), prod_num(parent(n)))
                );
            i := MAPUP(k, child_num(n), prod_num(parent(n)));
            n := parent(n)
        else
            done := true
            fi
    esac
  od
end IncrEvaluateAsynchCP;
```

is initialized to the number of input attribute instances of the production instance rooted at n' that will be given values in the remaining portion of the plan for n. Whenever an EVAL element is executed, the appropriate counter (if any) is decremented by one.

This technique is illustrated in the algorithm given in Figure 6.15. The pair of mappings numRemainingInputAttrsFromParent and numRemainingInputAttrsFromChild are used to provide the values needed to correctly initialize and increment the pendingActivators(n) counter for each node n. Re-attribution of tree T' after subtree U is replaced by subtree U' with root n' is done by

```
active(n') := true;
active(parent(n')) := true;
visited(parent(n')) := true;
lockRemainingOutputAttrInsts(parent(n'), 0);
```



```
IncrEvaluateAsynchCP(parent(n'), 0);
wait for all asynchronous processes to terminate
```

```
IncrEvaluateAsynchCP(treenode n, integer i):
  while active(n) do
    case PLANS[prod_num(n)][i] of
      [EVAL j b]: {
        if j ≠ 0
          candidate := child(j, n)
        elseif n ≠ root of tree
          candidate := parent(n)
        else
          candidate := none
          fi;
        previous_value := value(child(j, n).b);
        value(child(j,n).b) := value of appropriate semantic fn;
        if candidate ≠ none cand has_successors(n, j, b)
            cand previous_value ≠ value(child(j, n).b)
          active(candidate) := true
          fi;
        if candidate ≠ none
            cand (visited(candidate) & has_successors(n, j, b))
          atomic_sub1(pendingActivators(candidate)))
          fi;
        unlockAttrInst(child(j, n).b);
        i := i + 1
        };
(continued)
```

Figure 6.15 - Incremental l-OAG Tree Evaluator with Asynchronous Parallel Visit Execution Assuming Non-Strict Semantic Functions Using Pending Activators Counters to Delay Node Deactivation

Figure 6.15 (Continued)

```
[VISIT_CHILD j k]:{
    if ¬visited(child(j, n)) cand active(child(j, n))
        visited(child(j, n)) := true;
        atomic_add(
            pendingActivators(n),
            numRemainingInputAttrsFromChild(
                k, j, prod_num(n))
        );
        pendingActivators(child(j, n)) :=
            numRemainingInputAttrsFromParent(
                k, prod__num(child(j, n)));
        lockRemainingOutputAttrInsts(
            child(j, n), MAPDOWN(k, prod_num(n)));
        spawn IncrEvaluateAsynchCP(
            child(j, n), MAPDOWN(k, prod_num(n)))
        fi;
    i := i + 1
    };
[VISIT_PARENT k]:{
    if ¬visited(parent(n)) cand active(parent(n))
        visited(parent(n)) := true;
        atomic_add(
            pendingActivators(n),
            numRemainingInputAttrsFromParent(
                k, prod_num(n))
        );
        pendingActivators(parent(n)) :=
            numRemainingInputAttrsFromChild(
                k + 1, child_num(n), prod_num(parent(n)));
```
(continued)

Figure 6.15 (Continued)

```
        lockRemainingOutputAttrInsts(
            parent(n),
            MAPUP(k, child_num(n), prod_num(parent(n)))
            );
        spawn IncrEvaluateAsynchCP(
            parent(n),
            MAPUP(k, child_num(n), prod_num(parent(n)))
            )
        fi;
    i := i + 1
    };
[LAST_VISIT_PARENT k]:{
    if n ≠ root of tree cand ¬visited(parent(n))
            cand active(parent(n))
        visited(parent(n)) := true;
        lockRemainingOutputAttrInsts(
            parent(n),
            MAPUP(k, child_num(n), prod_num(parent(n)))
            );
        spawn IncrEvaluateAsynchCP(
            parent(n),
            MAPUP(k, child_num(n), prod_num(parent(n)))
            )
        fi;
    wait(pendingActivators(n) = 0);
    active(n) := false;
    visited(n) := false
    }
  esac
od
end IncrEvaluateAsynchCP;
```

Typically, the order in which the active(n) bit, the visited(n) bit, and the pendingActivators(n) counter are examined and altered must be carefully arranged to prevent race conditions.

6.3.4 Asynchronous Execution of EVAL and Visit Elements

We can combine features of the algorithms of §6.3.1 and §6.3.3 to produce algorithms that perform optimal incremental attribution while performing all types of visit-sequence elements asynchronously.

6.3.4.1 Node Activation

Since EVAL elements are again to be done asynchronously, coordinating node activation with node visiting is again necessary. The solutions proposed in §6.3.1.1 are applicable here.

6.3.4.2 Node Deactivation

It is also again necessary to deactivate nodes while preventing node reactivation. The methods used in §6.3.1.2 and §6.3.3.2 can be readily adapted for use here. Since EVAL elements are to be performed by asynchronous processes, operator strictness is not an issue.

6.3.4.3 Algorithms

We present only a few of the possible algorithms combining the features of §6.3.1 and §6.3.3. Figure 6.16 presents the simplest solution, that of performing potential activators non-asynchronously and using the procedure Deactivate from Figure 6.13 to perform node deactivation in a subsequent pass over the relevant region of the derivation tree. To

re attribute T' after performing a subtree replacement, one would use the steps

```
IncrEvaluateAsynchECP(treenode n, integer i) :
  done := false;
  while ¬done do
     case PLANS[prod_num(n)][i] of
        [EVAL j b] : {
           if j ≠ 0
              candidate := child(j, n)
           elseif n ≠ root of tree
              candidate := parent(n)
           else
              candidate := none
              fi;
           if candidate ≠ none cand has_successors(n, j, b)
                 cand ¬active(candidate)
              previous_value := value(child(j, n).b);
              value(child(j,n).b) := value of appropriate semantic fn;
              unlockAttrInst(child(j, n).b);
              if previous_value ≠ value(child(j, n).b)
                 active(candidate) := true
                 fi
           else
              spawn EvaluateAttribute(n, j, b)
              fi;
           i := i + 1
           };
(continued)
```

Figure 6.16 - Incremental I OAG Tree Evaluator with Asynchronous Parallel Element Execution Performing Potential Activators Non-Asynchronously and Using a Separate Deactivation Phase

Figure 6.16 (Continued)

```
[VISIT_CHILD j k]:{
    if ¬visited(child(j, n)) & active(child(j, n)
        visited(child(j, n)) := true;
        lockRemainingOutputAttrInsts(
            child(j, n), MAPDOWN(k, prod_num(n)));
        spawn IncrEvaluateAsynchECP(
            child(j, n), MAPDOWN(k, prod_num(n)))
        fi;
    i := i + 1
    };
[VISIT_PARENT k]:{
    if ¬visited(parent(n)) & active(parent(n))
        visited(parent(n)) := true;
        lockRemainingOutputAttrInsts(
            parent(n),
            MAPUP(k, child_num(n), prod_num(parent(n)))
            );
        spawn IncrEvaluateAsynchECP(
            parent(n),
            MAPUP(k, child_num(n), prod_num(parent(n)))
            )
        fi;
    i := i + 1
    };
[LAST_VISIT_PARENT k]:
    if n ≠ root of tree
            cand ( ¬visited(parent(n)) & active(parent(n)))
        visited(parent(n)) := true;
        lockRemainingOutputAttrInsts(
            parent(n),
            MAPUP(k, child_num(n), prod_num(parent(n)))
            );
(continued)
```


Figure 6.16 (Continued)

```
                i := MAPUP(k, child_num(n), prod_num(parent(n)));
                n := parent(n)
            else
                done := true
                fi
        esac
    od
end IncrEvaluateAsynchECP;

EvaluateAttribute(treenode n, integer j, attribute b):
    value(child(j, n).b) := value of appropriate semantic fn;
    unlockAttrInst(child(j, n).b)
    end EvaluateAttribute;
```

```
active(n') := true;
active(parent(n')) := true;
visited(parent(n')) := true;
lockRemainingOutputAttrInsts(parent(n'), 0);
IncrEvaluateAsynchECP(parent(n'), 0);
wait for all asynchronous processes to terminate;
Deactivate(n');
wait for all asynchronous processes to terminate
```

Figure 6.17 presents a version of the algorithm again performing potential activators non-asynchronously, but delaying node deactivation by LAST_VISIT_PARENT elements through the use of the pendingActivators(n) counter.

Finally, Figure 6.18 presents a version of the algorithm using a count of the executing potential activators of a node to both delay visits to the

```
IncrEvaluateAsynchECP(treenode n, integer i):
    while active(n) do
        case PLANS[prod_num(n)][i] of
            [EVAL j b]: {
                if j ≠ 0
                    candidate := child(j, n)
                elseif n ≠ root of tree
                    candidate := parent(n)
                else
                    candidate := none
                    fi;
                if candidate ≠ none cand has_successors(n, j, b)
                    if ¬active(candidate)
                        previous_value := value(child(j, n).b);
                        value(child(j,n).b) := value of appropriate semantic fn;
                        unlockAttrInst(child(j, n).b);
                        if previous_value ≠ value(child(j, n).b)
                            active(candidate) := true
                            fi
                    else
                        if visited(candidate)
                            atomic_sub1(pendingActivators(candidate))
                            fi;
                        spawn EvaluateAttribute(n, j, b)
                        fi
                else
                    spawn EvaluateAttribute(n, j, b)
                        fi;
                 i := i + 1
                 };
(continued)
```

Figure 6.17 - Incremental I-OAG Tree Evaluator with Asynchronous Parallel Element Execution Performing Potential Activators Non-Asynchronously and Using Counters to Delay Node Deactivation

Figure 6.17 (Continued)

```
            [VISIT_CHILD j k]: {
                if ¬visited(child(j, n)) cand active(child(j, n)
                    visited(child(j, n)) := true;
                    atomic_add(
                        pendingActivators(n),
                        numRemainingInputAttrsFromChild(
                            j, k, prod_num(n))
                        );
                    pendingActivators(child(j, n)) :=
                        numRemainingInputAttrsFromParent(
                            k, prod_num(child(j, n)));
                    lockRemainingOutputAttrInsts(
                        child(j, n), MAPDOWN(k, prod_num(n)));
                    spawn IncrEvaluateAsynchECP(
                        child(j, n), MAPDOWN(k, prod_num(n)))
                    fi;
                i := i + 1
                };
            [VISIT_PARENT k]: {
                if ¬visited(parent(n)) cand active(parent(n))
                    visited(parent(n)) := true;
                    atomic_add(
                        pendingActivators(n),
                        numRemainingInputAttrsFromParent(
                            k, prod_num(n))
                        );
                    pendingActivators(parent(n)) :=
                        numRemainingInputAttrsFromChild(
                            k + 1, child_num(n), prod_num(parent(n)));
                    lockRemainingOutputAttrInsts(
                        parent(n),
                        MAPUP(k, child_num(n), prod_num(parent(n)))
                        );
(continued)
```

Figure 6.17 (Continued)

```
            spawn IncrEvaluateAsynchECP(
                parent(n),
                MAPUP(k, child_num(n), prod_num(parent(n)))
                )
            fi;
        i := i + 1
        };
    [LAST_VISIT_PARENT k] : {
        if n ≠ root of tree cand ¬visited(parent(n))
                cand active(parent(n))
            visited(parent(n)) := true;
            lockRemainingOutputAttrInsts(
                parent(n),
                MAPUP(k, child_num(n), prod_num(parent(n)))
                );
            spawn IncrEvaluateAsynchECP(
                parent(n),
                MAPUP(k, child_num(n), prod_num(parent(n)))
            fi;
        wait(pendingActivators(n) = 0);
        active(n) := false;
        visited(n) := false
        }
    esac
od
end IncrEvaluateAsynchECP;

EvaluateAttribute(treenode n, integer j, attribute b) :
    value(child(j, n).b) := value defined by appropriate semantic fn;
    unlockAttrInst(child(j, n).b)
    end EvaluateAttribute;
```

node (when necessary) and to delay deactivation of a node until all possible reactivators have terminated.

Note that all three versions have quantitatively and qualitatively varying concurrency characteristics. The granularity of the asynchronous processes, the overhead of the *spawn* statement, the overhead of the *wait* operation, and the overhead associated with atomic operations on shared memory locations are all factors possibly affecting the actual performance of implementations of the algorithms. It seems doubtful that analytical models could provide a great deal of insight regarding the superiority of one version over others, particularly given the large variety of possible

```
IncrEvaluateAsynchECP(treenode n, integer i) :
    while active(n) do
        case PLANS[prod_num(n)][i] of
            [EVAL j b] : {
                if j ≠ 0
                    candidate := child(j, n)
                elseif n ≠ root of tree
                    candidate := parent(n)
                else
                    candidate := none
                    fi;
                if candidate ≠ none cand has_successors(n, j, b)
                    if ¬visited(candidate)
                        atomic_add1(pendingActivators(candidate))
                        fi;
                    spawn EvaluateAttribute(n, j, b, candidate, true)
                else
                    spawn EvaluateAttribute(n, j, b, candidate, false)
                        fi;
                  i := i + 1
                  };
(continued)
```

Figure 6.18 - Incremental 1-OAG Tree Evaluator with Asynchronous Parallel Element Execution Using Possible Activators Counters

Figure 6.18 (Continued)

```
    [VISIT_CHILD j k] : {
      wait(active(child(j, n)) | pendingActivators(child(j, n)) = 0);
      if ¬visited(child(j, n)) cand active(child(j, n)
        visited(child(j, n)) := true;
        atomic_add(
          pendingActivators(n),
          numRemainingInputAttrsFromChild(
            j, k, prod_num(n))
          );
        atomic_add(
          pendingActivators(child(j, n)),
          numRemainingInputAttrsFromParent(
            k, prod_num(child(j, n)));
        lockRemainingOutputAttrInsts(
          child(j, n), MAPDOWN(k, prod_num(n)));
        spawn IncrEvaluateAsynchECP(
          child(j, n), MAPDOWN(k, prod_num(n)))
        fi;
      i := i + 1
      };
    [VISIT_PARENT k] : {
      wait(active(parent(n)) | pendingActivators(parent(n)) = 0);
      if ¬visited(parent(n)) cand active(parent(n))
        visited(parent(n)) := true;
        atomic_add(
          pendingActivators(n),
          numRemainingInputAttrsFromParent(
            k, prod_num(n))
          );
        atomic_add(
          pendingActivators(parent(n)),
          numRemainingInputAttrsFromChild(
            k + 1, child_num(n), prod_num(parent(n)));
```

(continued)

Figure 6.18 (Continued)

```
          lockRemainingOutputAttrInsts(
            parent(n),
            MAPUP(k, child_num(n), prod_num(parent(n)))
            );
          spawn IncrEvaluateAsynchECP(
            parent(n),
            MAPUP(k, child_num(n), prod_num(parent(n)))
            )
          fi;
        i := i + 1
        };
      [LAST_VISIT_PARENT k] : {
        if n ≠ root of tree
          wait(active(parent(n)) | pendingActivators(parent(n)) = 0);
          if ¬visited(parent(n)) cand active(parent(n))
            visited(parent(n)) := true;
            lockRemainingOutputAttrInsts(
              parent(n),
              MAPUP(k, child_num(n), prod_num(parent(n)))
              );
            spawn IncrEvaluateAsynchECP(
              parent(n),
              MAPUP(k, child_num(n), prod_num(parent(n)))
              )
            fi
          fi;
        wait(pendingActivators(n) = 0);
        active(n) := false;
        visited(n) := false
        }
      esac
    od
  end IncrEvaluateAsynchECP;
```

(continued)

Figure 6.18 (Continued)

```
EvaluateAttribute(treenode n, integer j, attribute b, treenode candidate,
                  boolean activator):
    previous_value := value(child(j, n).b);
    value(child(j, n).b) := value of appropriate semantic fn;
    unlockAttrInst(child(j, n).b);
    if activator
        if previous_value ≠ value(child(j,n).b)
            active(candidate) := true
            fi;
        atomic_sub1(pendingActivators(candidate)))
        fi
    end EvaluateAttribute;
```

multiprocessor architectures. Rather, the use of empirical evidence and simulation will likely be necessary to judge among various algorithms.

6.4 Synchronous Parallel Optimal Incremental Attribution

The synchronous parallel attribution algorithms of Chapter 5 are more readily adapted to permit optimal incremental attribution. Since the necessary synchronization among processes is managed largely by high-level programming language constructs (e.g., *pforall*), the algorithms are much less intricate than their asynchronous counterparts.

6.4.1 Synchronous Sets

6.4.1.1 Attribution With Synchronous Sets Containing EVAL Elements Only

An optimal incremental version of the algorithm of §5.2.3.1 is quite straightforward. As shown in Figure 6.19, the algorithm combines features of the algorithms of Figures 5.7 and 6.1 in an obvious way. To re-attribute


```
IncrEvaluateSynchSetE(treenode n, integer i):
  while active(n) do
    case PLANS[prod_num(n)][i] of
      [SYNCHSET m]: {
        pforall j in 1 .. m do
          IncrEvaluateSynchSetE(n, i + j)
          od;
        i := i + m + 1
        };
      [PEVAL j b]:
      [EVAL j b]: {
        if j ≠ 0
          candidate := child(j, n)
        elseif n ≠ root of tree
          candidate := parent(n)
        else
          candidate := none
          fi;
        previous_value := value(child(j,n).b;
        value(child(j,n).b) := value of appropriate semantic fn;
        if candidate ≠ none cand has_successors(n, j, b)
            cand previous_value ≠ value(child(j,n).b
          active(candidate) := true
          fi;
        if PLANS[prod_num(n)][i] = [EVAL * *]
          i := i + 1
        else
          return
          fi
        };
(continued)
```

Figure 6.19 - Incremental 1-OAG Tree Evaluator Assuming Synchronous Sets Containing EVAL Elements Only

Figure 6.19 (Continued)

```
    [VISIT_CHILD j k]:
        if active(child(j, n))
            n := child(j,n);
            i := MAPDOWN(k, prod_num(n))
        else
            i := i + 1
            fi;
    [VISIT_PARENT k]:
        if active(parent(n))
            i := MAPUP(k, child_num(n), prod_num(parent(n)));
            n := parent(n)
        else
            i := i + 1
            fi;
    [LAST_VISIT_PARENT k]: {
        active(n) := false;
        if n ≠ root of tree cand active(parent(n))
            i := MAPUP(k, child_num(n), prod_num(parent(n)));
            n := parent(n)
            fi
        }
    esac
  od
end IncrEvaluateSynchSetE;
```

tree T' after a subtree replacement, the steps

active(n') := true;

active(parent(n') := true;

IncrEvaluateSynchSetE(parent(n'), 0)

are performed. Note that an analogous series of steps will be used for tree re-attribution with all the attribution algorithms presented in the remainder of this chapter.

6.4.1.2 Attribution With Synchronous Sets Containing EVAL and VISIT_CHILD Elements Only

The algorithm of §5.2.3.2 requires a number of modifications in order to perform optimal incremental attribution. The majority of these modifications are fairly obvious. The processing of VISIT_PARENT elements requires a subtle change.

In the algorithm of Figure 5.11, there are only two possible courses of action when executing a VISIT_PARENT (or LAST_VISIT_PARENT) element. If the VISIT_PARENT element is completing a PVISIT_CHILD element, the process terminates and no visit to the parent node is made. If, however, the VISIT_PARENT element is completing a VISIT_CHILD element, the visit to the parent node is performed.

There are additional possibilities under incremental attribution. First, the active/inactive status of the parent node must be considered. Second, a VISIT_PARENT element completing a PVISIT_CHILD must actually perform the visit in some circumstances. In particular, if a VISIT_PARENT element corresponds to the first attempt (chronologically) to visit a node during the course of an incremental attribution following a subtree replacement, the visit must be performed irrespective of the type of child visit element completed.

This requires two changes to the attribution scheme. First, we require (as we did in §5.2.3.3.1) that MAPUP be adjusted to never specify the index of a member of a synchronous set. Second, we require a method for detecting if a VISIT_PARENT completing a PVISIT_CHILD must perform a visit rather than terminating its own process.

To detect the above condition, we require that each derivation tree node n contain an additional bit designated by

executing_synch_set(n)

whose value is set to true when n's plan is in the midst of executing a SYNCHSET element and is set to false at all other times. The derivation tree is initially constructed with all such bits set to false.

An algorithm using this mechanism is given in Figure 6.20. The use of the executing_synch_set(n) bit eliminates the need for the going_up_from_parallel_visit predicate used in §5.2.3.2, and, in fact, eliminates the need for distinct serial and parallel versions of the various plan elements (i.e., PEVAL versus EVAL, etc.).

6.4.1.3 General Synchronous Sets

The algorithm of §5.2.3.3.3 can be adapted along the lines of the algorithm of Figure 6.20 to perform optimal incremental attribution when VISIT_PARENT elements may appear in synchronous sets, as shown in Figure 6.21. This algorithm uses the atomic_monus1 operation described in §3.1 to eliminate several special cases.

6.4.2 Series-Parallel Groups

The series-parallel attribution algorithms of §5.3.4 are easily modified to perform optimal incremental attribution. We present only the incremental version of the algorithm of §5.3.4.2 for SP attribution with


```
IncrEvaluateSynchSetEC(treenode n, integer i):
   while active(n) do
      case PLANS[prod_num(n)][i] of
         [SYNCHSET m]: {
            executing_synch_set(n) := true;
            pforall j in 1 .. m do
               IncrEvaluateSynchSetEC(n, i + j)
               od;
            executing_synch_set(n) := false;
            i := i + m + 1
            };
         [PEVAL j b]:
         [EVAL j b]: {
            if j ≠ 0
               candidate := child(j, n)
            elseif n ≠ root of tree
               candidate := parent(n)
            else
               candidate := none
               fi;
            previous_value := value(child(j,n).b;
            value(child(j,n).b) := value of appropriate semantic fn;
            if candidate ≠ none cand has_successors(n, j, b)
                  cand previous_value ≠ value(child(j,n).b
               active(candidate) := true
               fi;
            if PLANS[prod_num(n)][i] = [EVAL * *]
               i := i + 1
            else
               return
               fi
            };
(continued)
```

Figure 6.20 - Incremental l-OAG Tree Evaluator Assuming Synchronous Sets Containing EVAL and VISIT_CHILD Elements Only

Figure 6.20 (Continued)

```
        [PVISIT_CHILD j k]:
        [VISIT_CHILD j k]:
            if active(child(j, n))
                n := child(j,n);
                i := MAPDOWN(k, prod_num(n))
            elseif PLANS[prod_num(n)][i] := [VISIT_CHILD * *]
                i := i + 1
            else
                return
                fi;
        [VISIT_PARENT k]:
            if active(parent(n))
                if ¬executing_synch_set(parent(n))
                    i := MAPUP(k, child_num(n), prod_num(parent(n)));
                    n := parent(n)
                else
                    return
                    fi
            else
                i := i + 1
                fi;
        [LAST_VISIT_PARENT k]: {
            active(n) := false;
            if n ≠ root of tree cand active(parent(n))
                if ¬executing_synch_set(parent(n))
                    i := MAPUP(k, child_num(n), prod_num(parent(n)));
                    n := parent(n)
                else
                    return
                    fi
                fi
            }
        esac
    od
end IncrEvaluateSynchSetEC;
```



```
IncrEvaluateSynchSet(treenode n, integer i):
    while active(n) do
        case PLANS[prod_num(n)][i] of
            [SYNCHSET m]: {
                atomic_add1(incomplete_counter(n));
                pforall j in 1 .. m do
                    IncrEvaluateSynchSet(n, i + j)
                    od;
                atomic_sub1(incomplete_counter(n));
                if incomplete_counter(n) = 0
                    i := i + m + 1
                else
                    return
                    fi
                };
            [PEVAL j b]:
            [EVAL j b]: {
                if j ≠ 0
                    candidate := child(j, n)
                elseif n ≠ root of tree
                    candidate := parent(n)
                else
                    candidate := none
                    fi;
                previous_value := value(child(j,n).b;
                value(child(j,n).b) := value of appropriate semantic fn;
                if candidate ≠ none cand has_successors(n, j, b)
                        cand previous_value ≠ value(child(j,n).b
                    active(candidate) := true
                    fi;
                if PLANS[prod_num(n)][i] = [EVAL * *]
                    i := i + 1
                else
                    return
                    fi
                };
(continued)
```

Figure 6.21 - Incremental l-OAG Tree Evaluator Assuming General Synchronous Sets

Figure 6.21 (Continued)

```
[PVISIT_CHILD j k]:
[VISIT_CHILD j k]:
    if active(child(j, n))
        atomic_add1(incomplete_counter(n));
        atomic_monus1(incomplete_counter(child(j, n)));
        if incomplete__counter(child(j, n)) = 0
            n := child(j,n);
            i := MAPDOWN(k, prod_num(n))
        else
            return
            fi
    elseif PLANS[prod_num(n)][i] := [VISIT_CHILD * *]
        i := i + 1
    else
        return
        fi;
[PVISIT_PARENT k]:
[VISIT_PARENT k]:
    if active(parent(n))
        atomic_add1(incomplete_counter(n));
        atomic_monus1(incomplete_counter(parent(n)));
        if incomplete_counter(parent(n)) = 0
            i := MAPUP(k, child_num(n), prod_num(parent(n)));
            n := parent(n)
        else
            return
            fi
    elseif PLANS[prod_num(n)[i] = [VISIT_PARENT *]
        i := i + 1
    else
        return
        fi;
(continued)
```


Figure 6.21 (Continued)

```
[LAST_VISIT_PARENT k]: {
    active(n) := false;
    if n ≠ root of tree cand active(parent(n))
        atomic_monus1(incomplete_counter(parent(n)));
        if incomplete_counter(parent(n)) = 0
            i := MAPUP(k, child_num(n), prod_num(parent(n)));
            n := parent(n)
        else
            return
            fi
        fi
    }
  esac
 od
end IncrEvaluateSynchSet;
```

EVAL, VISIT_CHILD, and VISIT_PARENT elements permitted in parallel groups. Excluding VISIT_PARENT elements from parallel groups does not simplify the algorithm significantly. Figure 6.22 gives the algorithm.

6.5 Related Work

[Hoover 1987] presents a modified dynamic order incremental attribution scheme that may be parallelized. The scheme is provably not optimal, but, based on empirical evidence, performs well in most cases. The scheme has a number of advantages when attribute instance values may be *aggregate values* ([Hoover & Teitelbaum 1986]) and when using attribute instance values to describe the semantics of identifier declarations and uses in programming languages ([Hoover 1986]).

```
IncrEvaluateSeriesParallelECP(treenode n, integer i):
    while active(n) do
        case PLANS[prod_num(n)][i] of
            [PARALLEL m l c]: {
                counters(n)[c] := 1 + m;
                pforall j in 1 .. m do
                    IncrEvaluateSeriesParallelECP(n, i + j)
                    od;
                atomic_sub1(counters(n)[c]);
                if counters(n)[c] = 0
                    i := i + l + 1
                else
                    return
                    fi
                };
            [SERIES_START l]:
                i := i + l;
            [SERIES_END l c]: {
                atomic_monus1(counters(n)[c]);
                if counters(n)[c] = 0
                    i := i + l + 1
                else
                    return
                    fi
                };
            [EVAL j b]: {
                if j ≠ 0
                    candidate := child(j, n)
                elseif n ≠ root of tree
                    candidate := parent(n)
                else
                    candidate := none
                    fi;
(continued)
```

Figure 6.22 - Incremental l OAG Tree Evaluator Assuming General Series-Parallel Groups

Figure 6.22 (Continued)

```
                previous_value := value(child(j,n).b;
                value(child(j,n).b) := value of appropriate semantic fn;
                if candidate ≠ none cand has_successors(n, j, b)
                        cand previous_value ≠ value(child(j,n).b
                    active(candidate) := true
                    fi;
                i := i + 1
                };
            [VISIT_CHILD j k]:
                if active(child(j, n))
                    n := child(j,n);
                    i := MAPDOWN(k, prod_num(n))
                else
                    i := i + 1
                    fi;
            [VISIT_PARENT k]:
                if active(parent(n))
                    i := MAPUP(k, child_num(n), prod_num(parent(n)));
                    n := parent(n)
                else
                    i := i + 1
                    fi;
            [LAST_VISIT_PARENT k]: {
                active(n) := false;
                if n ≠ root of tree cand active(parent(n))
                    i := MAPUP(k, child_num(n), prod_num(parent(n)));
                    n := parent(n)
                    fi
                }
            esac
        od
    end IncrEvaluateSeriesParallelECP;
```

[Kaplan & Kaiser 1986] discusses a method for incrementally attributing trees in the presence of multiple asynchronous subtree replacements. The scheme is based on the dynamlc order evaluator presented in [Reps 1984] and is targetted for use in multi-user environments in which a derivation tree may be spread over a network of distributed processors.

7. Simulation of Parallel Attribution

To gain insights into the performance of various parallel attribution techniques, a number of simulation studies were carried out. The results suggest that parallel attribution may provide significant reductions in the time required for tree attribution.

7.1 Overview of the Simulator

A simple shared-memory multiprocessor simulator was implemented within the confines of a modified version of the Synthesizer Generator [Reps & Teitelbaum 1989a, Reps & Teitelbaum 1989b]. Structure editors produced by this modified version may be run in such a way that tree attributions are carried out under the control of the multiprocessor simulator with subsequent reporting of a variety of statistics. The behavior of the simulator can be controlled to a degree through the choices of values assigned to a number of parameters.

The simulator assumes that all processors are identical and have equally fast access to shared-memory locations. An arbitrary number of processors may successfully read from a single shared-memory location simultaneously. The architecture is assumed to support the synchronization operations discussed in Chapter 3.

The simulator provides only the most basic notions of process scheduling. As processes are spawned, they enter a FIFO *ready queue*. When a processor becomes available, the first process on the ready queue is given that processor and begins to execute. A running process retains exclusive rights to its assigned processor until such time as the process terminates or the process blocks awaiting the completion of other processes, at which point the process enters a *blocked queue* and then makes its processor available to a ready process. When a blocked process becomes unblocked, it again enters the ready queue.

Within the simulator, time is measured in discrete, unnamed units. The following parameters must be assigned values by the simulator user:

- the times required to execute the various types of plan elements
- the time required to execute the pseudo-code instructions into which semantic functions are compiled
- the time required to spawn new processes
- the time required for a process to change state (e.g., from "ready" to "running")

The simulator has a number of limitations, including

- Any effects of memory contention are ignored (although they may be crudely modeled simply by increasing all time parameters by a common "slowdown" factor).
- The time required for dynamic memory management is ignored.
- Multiple processes may apparently enter and/or leave a queue simultaneously (although specialized hardware realizations of such queues may exist [Leiserson 1983, Cheng 1990]).

Due to such limitations, the simulator should not be considered for use in distinguishing between minor variants of parallel algorithms (e.g., is node reactivation performed during attribution or in a separate pass, are counters or bit-vectors employed to keep track of incomplete visit elements, etc.). Rather, the simulator allows for comparisons at a coarser level (e.g., evaluation of the relative merits of asynchronous vs. synchronous attribution, the gains possible by performing EVAL elements

asynchronously, etc.). Thus, the simulator may be said to allow comparisons at the strategic level as opposed to the tactical level.

7.2 Implementation of Parallel Attribution Algorithms

A selection of algorithms from Chapter 6 were chosen for simulation. The chosen algorithms form a representative sample of both asynchronous and synchronous parallel algorithms. Only incremental attribution algorithms were implemented, primarily since they permit a wider variety of possible experiments and, secondarily, since the relatively coarse nature of the simulator would, to some degree, fail to distinguish significant performance differences between incremental and non-incremental algorithms.

7.2.1 Asynchronous Algorithms

Two versions of asynchronous parallel attribution algorithms were implemented: a version in which all types of plan elements are executed asynchronously and a version in which only VISIT_CHILD and VISIT_PARENT elements are executed asynchronously. In both algorithms

- Potential activators are performed non-asynchronously, with the has_successors predicate used to reduce the number of potential activators.
- Semantic functions are non-strict.
- Node deactivation is assumed to be done using unique subtree replacement numbers, and any time needed for tree-wide resetting of replacement numbers is ignored.

- Any process executing an EVAL element is responsible for unblocking and placing on the ready queue all processes waiting on the attribute instance value calculated by that EVAL.

7.2.2 Synchronous Algorithms

Two synchronous parallel attribution algorithms were implemented as well. One performs synchronous set attribution, while the other performs series-parallel attribution.

The synchronous set algorithm permits EVAL, VISIT_CHILD, and VISIT_PARENT elements to appear in synchronous sets. The plans generated for grammars are optimal considering only local optimality under the unit-length assumption. A process blocks only in the course of executing a SYNCHSET element while waiting for termination of the synchronous processes spawned by that element.

The series-parallel attribution algorithm permits all types of plan elements to appear in parallel groups. SP plans are generated using a simple heuristic that attempts to "improve" optimal unit-length synchronous set plans into SP plans.

Processes never block in this SP implementation. Rather, a process executing a PARALLEL element spawns a process for each of its constituent series and then dies. The single process executing that SERIES_END element which (chronologically) completes the execution of the entire parallel group then goes on to execute the portion of the plan following the parallel group.

7.3 Simulation Experiments

The four parallel attribution algorithm implementations described above, along with a serial attribution algorithm, were used to attribute trees derived using two grammars. A variety of different time parameter values were used.

7.3.1 Experiments With a Contrived Grammar

The first group of simulations involve attributing a full binary tree of 255 nodes generated using the grammar shown in Figure 7.1. This

```
AI(S) = Ø, AS(S) = Ø
AI(X) = { i }, AS(X) = { s }

p0:     S → X
            {    X.i ← 0 }
p1:     X → X X
            {    X$2.i ← X$1.i + 1;
                 X$3.i ← X$1.i + 1;
                 X$1.s ← X$2.s + X$3.s }
p2:     X →
            {    X.s ← X.i + 1; }
```

Figure 7.1 - A Contrived Grammar

grammar was specifically designed to provide obvious opportunities for parallelism. Attributing such a tree can provide an idea of whether and how much inherent parallelism the various attribution algorithms can actually exploit, as well as providing clues as to the effects of various types of overhead on the performance of the algorithms.

Results of the first experiment appear in Figure 7.2. In this

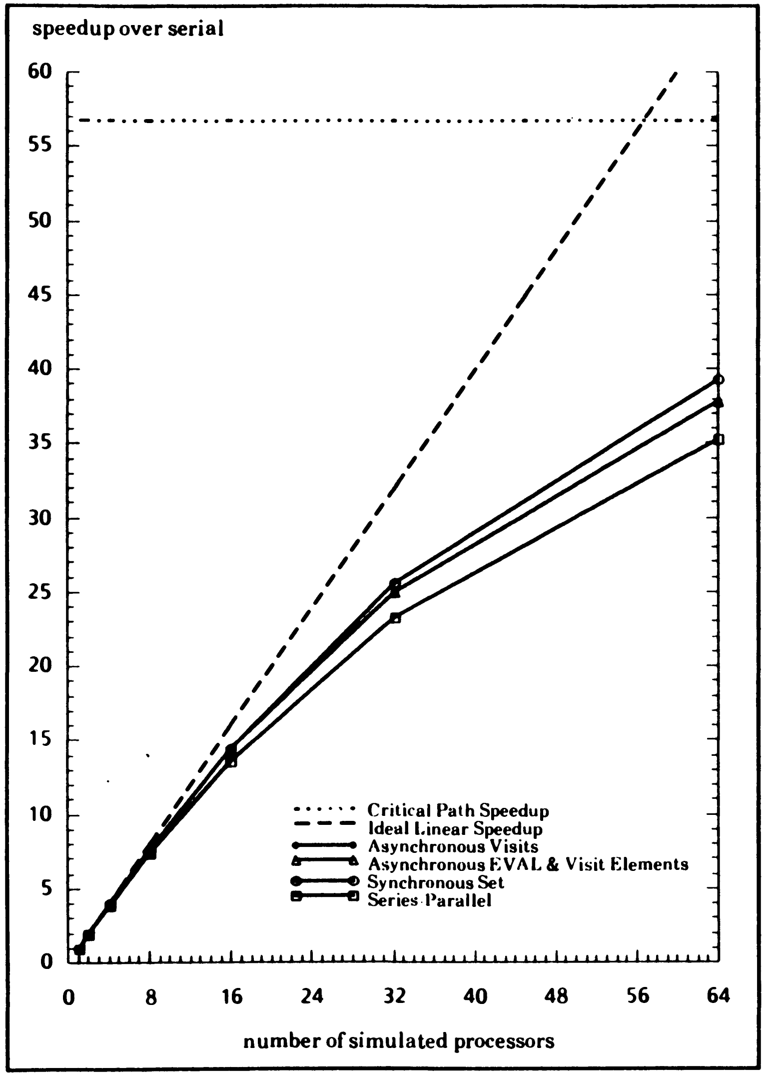

Figure 7.2 - Results When EVAL Time Dominates

experiment, the simulator parameters were chosen such that the time required to execute EVAL elements dominated all other times by several orders of magnitude. This models the ideal circumstances in which all attribute instance values are uniformly expensive to calculate and there is no significant overhead of any kind. The simulation results obtained with such settings are assumed to provide a measure of the inherent parallelism available to the attribution methods for the specific derivation tree and attribute grammar combination used in the experiment.

In the graph of Figure 7.2 (and in all subsequent graphs), we include boundary lines indicating two types of performance ideals. The first boundary represents *ideal linear speedup* (i.e., a speedup equal to the number of processors). The second boundary represents the *critical path speedup*. This represents the speedup over serial attribution time that would be obtained if attribution could be performed in time equal to the cost of the path of maximum cost in the derivation-tree dependency graph. In order to allow a reasonable comparison with the various attribution algorithms, each node in the dependency graph is assumed to have a cost equal to the sum of the times required to spawn a process, to change the state of that process from "ready" to "running," to execute an EVAL element, and to execute the semantic function defining the attribute instance corresponding to that node.

All the parallel algorithms performed quite well in this case. It must be noted that, in this extreme situation only, the design of the simulator unfairly penalizes algorithms performing EVAL elements asynchronously, resulting in the two asynchronous parallel algorithms having essentially identical performance. Given better simulation techniques, it is likely that the algorithm performing both EVAL and visit elements asynchronously would fare much better when compared to the other algorithms.

The graph in Figure 7.3 gives the results of the second experiment. For this experiment, the time required to execute semantic function pseudo-code instructions was assumed to dominate all other times. This models the situation in which most overhead is insignificant but in which individual attribute instance values may take varying amounts of time to calculate (although in the grammar of Figure 7.1, the majority of semantic functions do in fact take identical amounts of time to evaluate). In addition, due to the design of the simulator, some of the overhead of querying the value of a locked attribute instance begins to become significant.

Again, all the algorithms performed well. The synchronous algorithms gave results virtually identical with those of Figure 7.2. The performance of the asynchronous visits algorithm begins to suffer slightly from lock querying overhead. The apparent improvement over the results shown in Figure 7.2 in the performance of the asynchronous EVAL and visit element algorithm is due to the fact that the simulator design does not unduly penalize asynchronously executed EVAL elements in this case.

The third experiment portrays a somewhat more realistic situation. In this case, all simulator time parameters were set to identical values, representing an environment in which neither attribute instance evaluation nor overhead dominates, overhead time is significant, and semantic functions take varying amounts of time to execute. These results are presented in Figure 7.4.

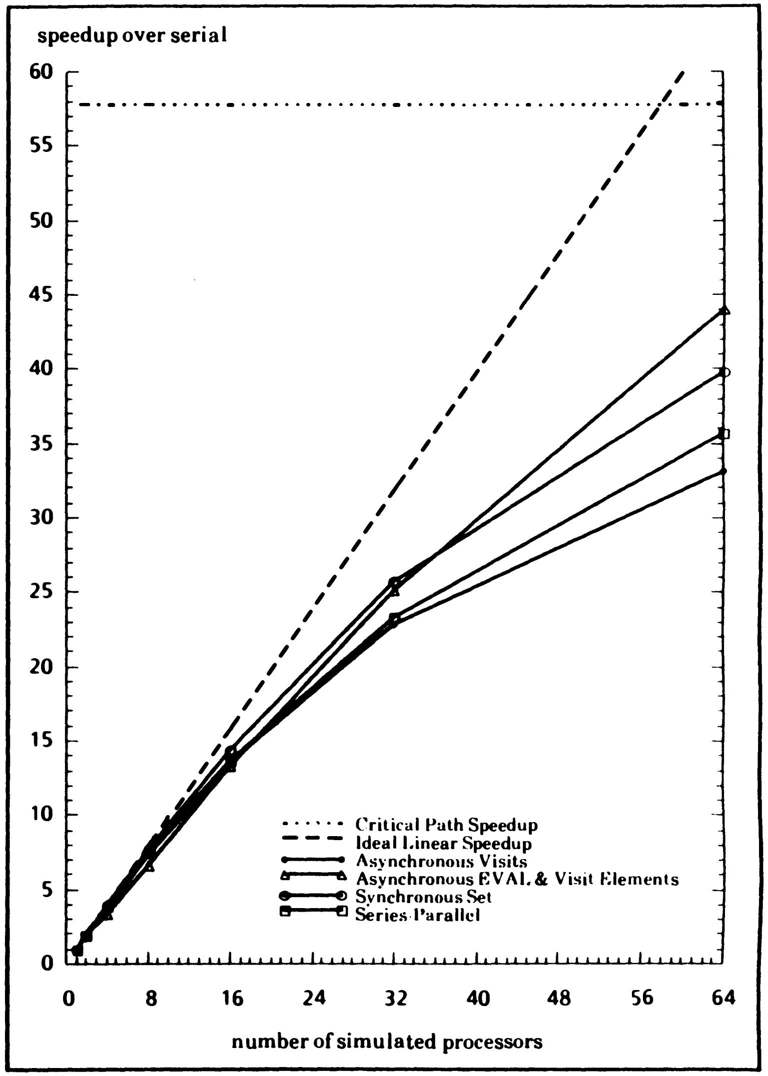

Figure 7.3 - Results When Semantic Function Instruction Time Dominates

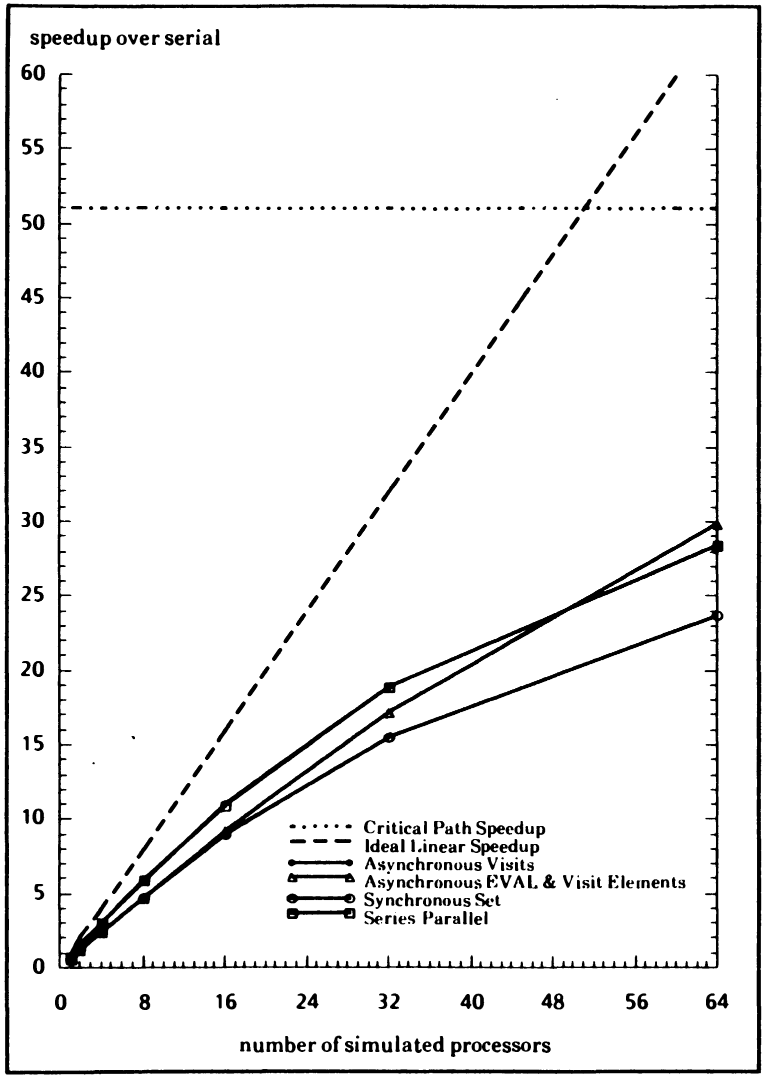

Figure 7.4 - Results When Times are Uniform

All the algorithms perform less well but acceptably. The effect of the increased cost of overhead is most conspicuous in the algorithm performing all types of plan elements asynchronously.

Finally, Figure 7.5 depicts the outcome of the least optimistic set of assumptions. In this case, semantic function instructions required t units of time to execute, plan elements required 5t time units, and process spawning required 10t time units. While the actual times required for such operations would vary significantly across different implementations, we suppose this collection of parameter settings to be most realistic.

The effects of the increased overhead is apparent. The series-parallel algorithm and the asynchronous visit algorithm, by virtue of spawning relatively fewer processes, perform best. The decrease in performance of the algorithm executing EVAL and visit elements asynchronously is dramatic.

7.3.2 Experiments With an Uncontrived Grammar

Having seen that the algorithms can provide significant improvements over serial attribution times when presented with obvious opportunities for parallelism, we conducted a second series of simulations using a more typical attribute grammar. This second grammar is an attribute grammar for the Pascal programming language [Jensen & Wirth 1985] that provides complete static semantic checking. This grammar is distributed with the Synthesizer Generator for demonstration purposes and was not designed with any concern for parallel attribution.

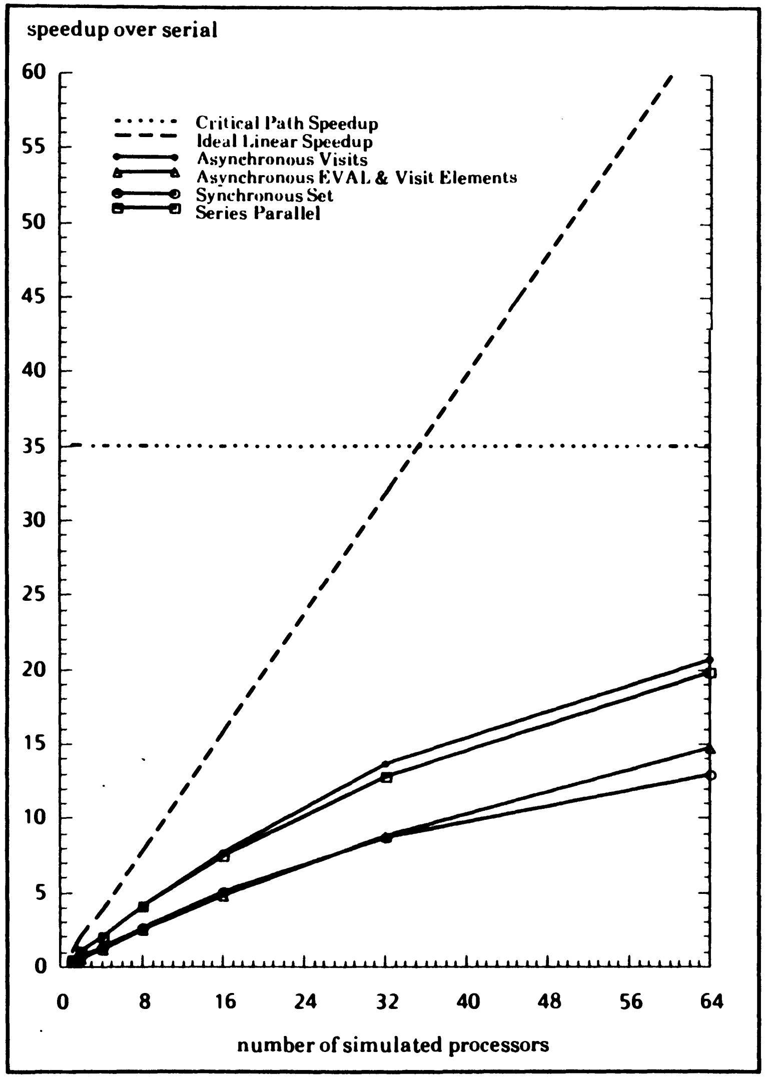

Figure 7.5 - Results Under More Realistic Assumptions

This set of experiments involved incrementally re-attributing a tree representing a simple text-formatting program chosen from [Kernighan & Plauger 1981]. The program consists of approximately 500 lines of source code and contains approximately 40 global constant, type, and variable declarations, as well as 23 procedure and function declarations (averaging 17 lines of source each). The syntactic structure of the program is quite simple in that the program contains no label declarations and none of the procedures or functions contains any local constant, type, or procedure or function declarations.

The derivation tree constructed from this program contains approximately 6000 nodes. Re-attribution of the tree was precipitated by deleting a single global identifier declaration.

Figure 7.6 shows the results of assuming that the time to execute EVAL elements dominate all other times (as was done in the experiment of Figure 7.2). Recall that this assumption penalizes the asynchronous EVAL and visit elements algorithm more than the other algorithms. Note the change in the size of the vertical scale from that used in Figures 7.2 through 7.5.

The graph suggests that the Pascal grammar offers significantly fewer opportunities for parallelism than the grammar of Figure 7.1. Several factors may combine to account for this.

One such factor concerns the *block*, the major syntactic construct in Pascal, which consists of label, constant, type, variable, and procedure and function declaration sections (each of which is optional within any given block) and a single executable compound statement. Blocks may be nested. The ability to attribute multiple blocks concurrently should allow

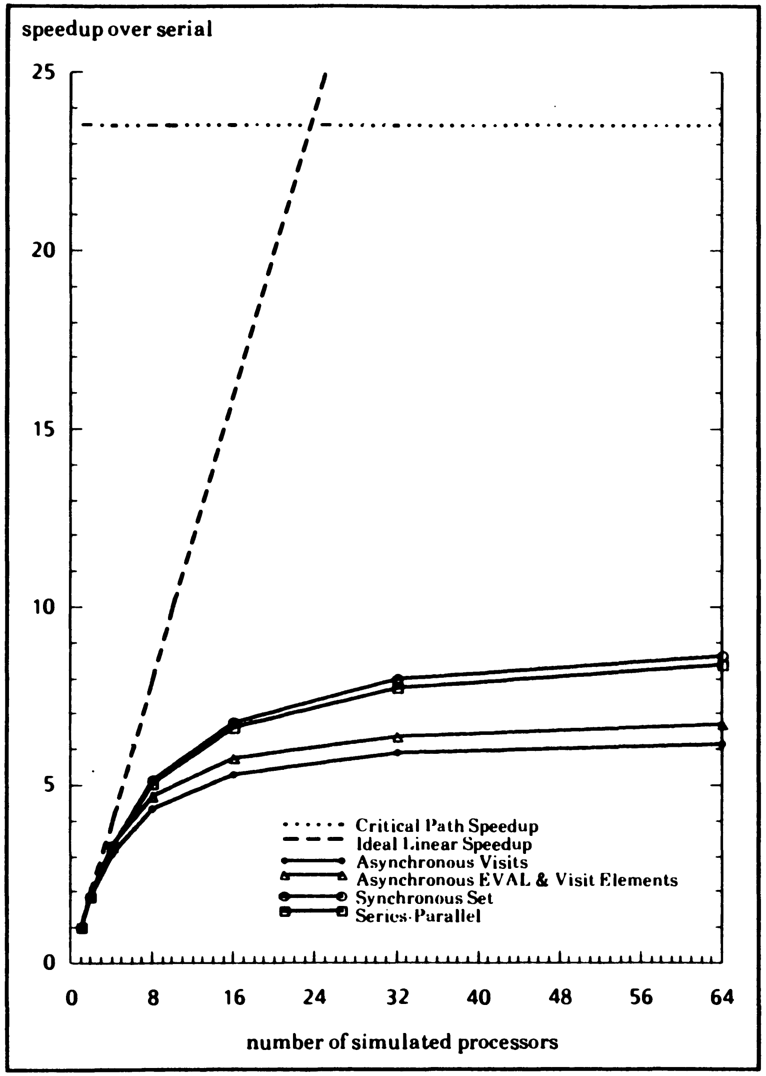

Figure 7.6 - Results When EVAL Time Dominates

reductions in overall attribution time. The dependencies (and, hence, the visit sequences) derived from the grammar in question do indeed permit parallel attribution of blocks.

However, the Pascal program used in this experiment has quite small, simple, non-nested blocks. Relatively little advantage is gained through parallel attribution of such blocks. Casual experiments with programs containing greater numbers of similarly-complex blocks produced similar speedups, while casual experiments with programs containing more complex, nested blocks produced greater speedups, comparable to the speedups reported in §7.3.1.

A second factor possibly contributing to the relative lack of available parallelism implied by Figure 7.6 concerns Pascal's very strict declare-before-use policy for program identifiers. This policy, which is naturally reflected in the attributes and semantic equations in the experimental grammar, tends to serialize the processing of declarations. It is conceivable that more dramatic speedups would be observed for grammars for programming languages with more liberal or complex declaration-order policies.

Finally, the choice of methods for handling potential activators (i.e., performing potential activators serially) had the effect of reducing the degree of parallelism achieved by the algorithm performing both EVAL and visit elements asynchronously. Approximately 25% of the EVAL elements were identified as potential activators and, thus, were not performed asynchronously. More will be said on this later.

The experiments whose results are depicted in Figures 7.7 through 7.9 make use of the same time parameter values as did the experiments of

Figures 7.3 through 7.5, respectively. (Note that the vertical scale has again been expanded.) After introducing non-constant times for attribute instance value calculation in Figure 7.7, increasing the amount of overhead in Figures 7.8 and 7.9 had relatively little additional effect. As overhead increased, the asynchronous methods tended to do a slightly better job of exploiting what little parallelism was available.

The great reduction in speedups depicted in Figure 7.7 suggests that semantic function execution times vary significantly for the given tree and grammar, with many semantic functions requiring very small amounts of time. This apparently combines with the factors mentioned above to effectively create a highly-dominant critical path through the dependency graph for the program's derivation tree.

The relatively meager reductions in attribution time would not recommend the use of the parallel incremental attribution techniques presented in this dissertation when working with this Pascal grammar and this (or any similarly structurally-simple) Pascal program. On more complex grammars and/or programs, the use of such incremental methods is more likely to prove advantageous.

As was mentioned previously, the algorithm performing EVAL and visit elements asynchronously suffered from the relatively high percentage of potential activators encountered during attribution. The following experiment was carried out to determine the extent to which this affected the performance of the algorithm.

In this experiment, the same Pascal program tree and simulator parameter settings used in previous experiments was completely attributed by a non-incremental serial attribution algorithm and a non-incremental

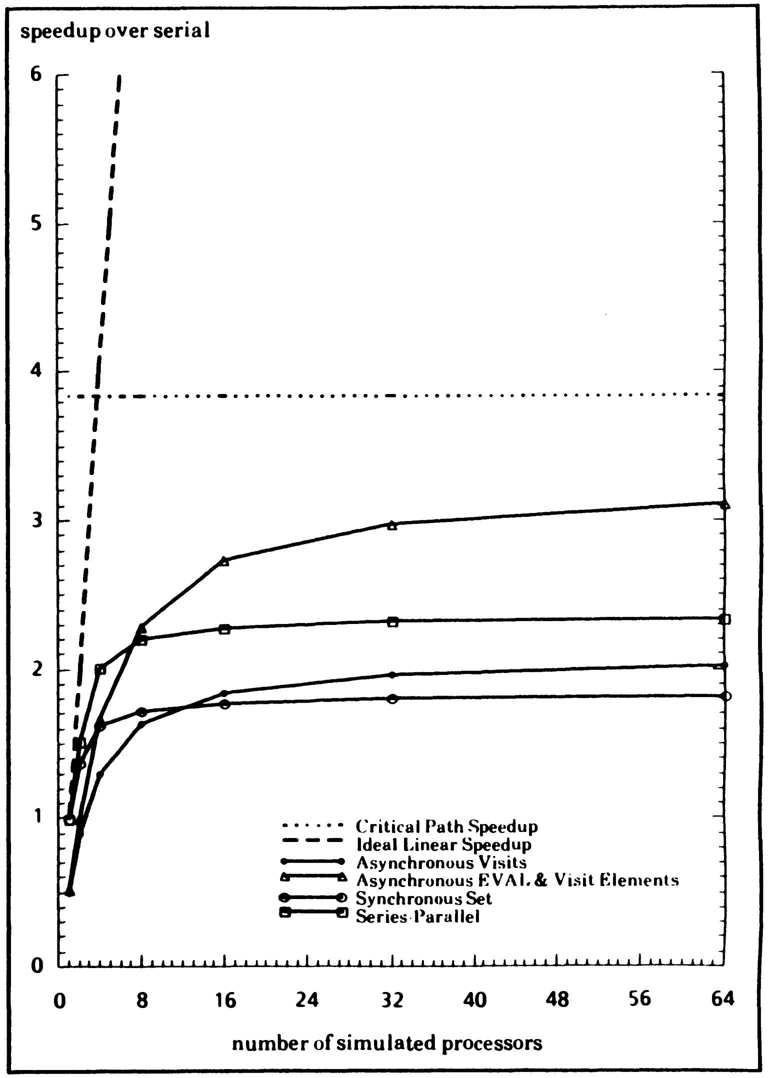

Figure 7.7 - Results When Semantic Function Instruction Time Dominates

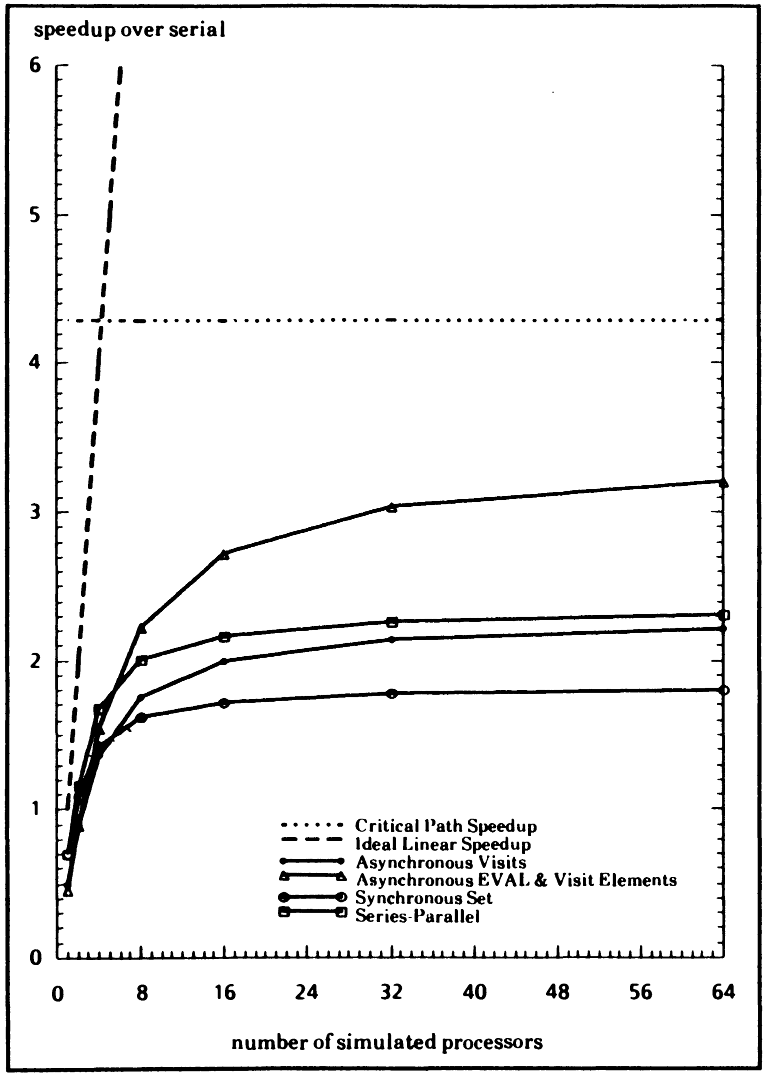

Figure 7.8 - Results When Times are Uniform

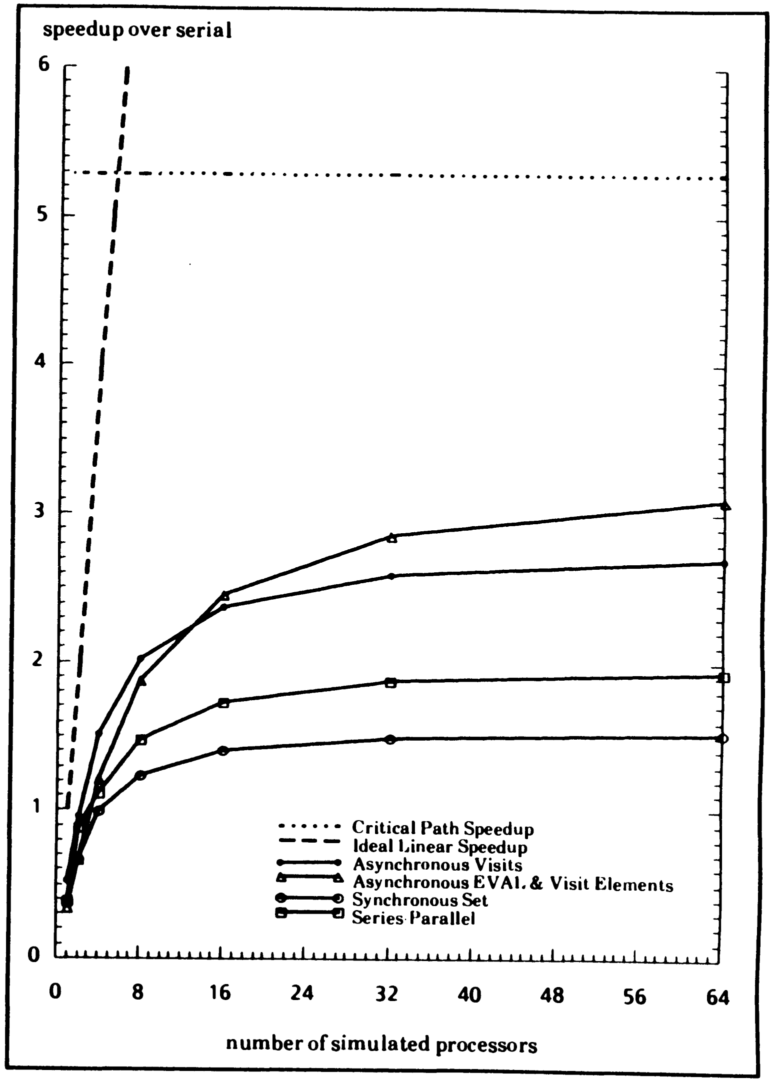

Figure 7.9 - Results Under More Realistic Assumptions

parallel algorithm performing all types of plan elements asynchronously. The results are depicted in Figure 7.10. Note that the speedups obtained by

Simulator Parameter Settings	Critical Path Speedup	Asynchronous EVAL & Visit Elements Speedup (64 processors)
EVAL time dominates	18.2	45.7
semantic function instruction time dominates	6.5	6.6
uniform times	6.8	6.8
more realistic assumptions	6.9	7.0

Figure 7.10 - Results of Non-Incremental Asynchronous Attribution

the asynchronous algorithm actually surpass the supposedly optimal critical path speedup: this is apparently due to the fact that the grammar has a number of semantic equations (over 100) containing conditional expressions and to the fact that such expressions are executed in a non-strict manner by the asynchronous algorithm. The results illustrate that the manner in which potential activators are reckoned with can significantly affect the performance of this algorithm.

7.4 Conclusions

The results of the simulation studies suggest several quite obvious conclusions. First, the parallel attribution methods developed in this dissertation can, in some circumstances, result in useful reductions in tree attribution time. Second, these reductions are highly dependent on the dependency properties of grammars, the care taken to produce optimal plans, and the complexity of the derivation trees being worked with. Third,

the more expensive it is to calculate attribute instance values, the more likely it is that parallel attribution will be advantageous. Finally, even when there are obvious opportunities for parallelism available, process and synchronization overhead must be minimized.

It appears that techniques for optimizing grammars for parallel attribution will be essential. Possible techniques for such optimization include "decorating" grammars with explicit parallel attribution directives [Boehm & Zwaenepoel 1987] and explicitly or implicitly restructuring grammars to increase available parallelism [Kuiper 1989].

Significant related work in optimal plan generation remains to be done. It is clear that many individual plan elements (and groups of plan elements, in the synchronous set and SP-plan cases) in particular grammars are sufficiently inexpensive to execute that they are unsuitable choices for execution by individual processes.

It is certain that actual implementations of parallel attribution algorithms will benefit from reducing various types of overhead [Greenbaum 1989]. The costs of process synchronization, process creation, and process blocking and unblocking must be minimized. In addition, attribute instance locking and unlocking, etc. must be done cheaply. Greater pains to provide intelligent process execution scheduling would also be necessary [Sarkar 1989].

8. Conclusion

8.1 Summary

We have presented a variety of methods for introducing parallelism into the process of tree attribution. We have concentrated on techniques for parallelizing *visit-sequence evaluators* for trees derived from *l-ordered attribute grammars*. For convenience, a *shared-memory model* of parallel architecture was assumed, but was not crucial to the correctness of the parallel techniques..

8.1.1 Asynchronous Parallel Attribution

Asynchronous parallel attribution methods achieve concurrency through the execution of certain plan elements by asynchronous processes. Various combinations of the asynchronous execution of EVAL, VISIT_CHILD, and VISIT_PARENT elements are possible. Algorithms for a number of these combinations were presented.

A central issue for asynchronous attribution techniques is the need to prevent attribute instance values from being used before those values are actually calculated. In order to accomplish this, attribute instance locking/unlocking protocols, which differed depending on the types of plan elements to be executed asynchronously, were developed.

A discussion of practical matters concerning the algorithms was presented. The relative numbers of processes spawned, the relative granularity of the processes spawned, and the relative amounts of space needed by the various algorithms was presented.

8.1.2 Synchronous Parallel Attribution

Synchronous parallel attribution methods avoid the need for explicit locking/unlocking of attribute instances within a derivation tree. Through the use of synchronous parallel programming constructs, processes are assured that all necessary attribute instance values have been previously determined.

We first introduced the concept of collecting mutually-independent plan elements into *synchronous sets*. Various methods for producing *synchronous-set plans* were presented. The probable difficulty of producing *optimal synchronous-set plans* was mentioned.

If the contents of synchronous sets are restricted to only certain types of plan elements, parallel attribution algorithms of differing complexity and characteristics result. In particular, permitting both VISIT_CHILD and VISIT_PARENT elements to appear within synchronous sets causes complications. These complications were resolved by introducing the notion of *incomplete visits*. An algorithm exploiting this notion was given, and a proof that it completely and consistently attributes trees, while performing no redundant plan element executions, was offered.

A second general style of synchronous parallel attribution, *series-parallel attribution*, was then presented. Methods for producing *series-parallel plans* were discussed. A number series-parallel attribution algorithms were given, each based on restricting the types of plan elements allowed within *parallel groups*. Permitting VISIT_PARENT elements within parallel groups produced complications; however, the complications were dealt with by generalizing the solution used to allow VISIT_PARENT elements within synchronous sets.

8.1.3 Parallel Incremental Attribution

Both asynchronous and synchronous parallel attribution can be performed *incrementally*. A selection of algorithms from Chapters 4 and 5 were adapted to perform *optimal incremental attribution* in a parallel manner.

Complications regarding the need to *activate* and *deactivate* derivation-tree nodes during attribution (in order to maintain optimality) arose. A number of rather intricate techniques for handling these complications were presented. For the asynchronous parallel attribution methods, it was sometimes necessary to alter the attribute instance locking protocols used for non-incremental attribution.

The synchronous algorithms were more easily adapted to perform optimal incremental attribution. Only relatively minor changes (largely involving the treatment of VISIT_PARENT elements) were required in both the synchronous-set and series-parallel attribution algorithms.

8.1.4 Simulation of Parallel Attribution

A simple multiprocessor simulator was used to examine the behavior of a variety of parallel incremental attribution algorithms. The level of simulation was sufficiently coarse that the results may be taken only as a suggestion of the performance characteristics of the algorithms.

Two groups of experiments were carried out. The first involved a grammar and a derivation tree specifically designed to provide obvious opportunities for parallel execution of plan elements. All the attribution methods performed well, although detrimental effects due to various kinds

of process overhead were observable. The algorithms that tended to spawn the fewest processes had the best performance under what are assumed to be the most realistic experimental conditions.

This set of experiments showed that the parallel attribution methods presented in this thesis can indeed provide significant speedups in attribution time.

The second set of experiments made use of an attribute grammar for the Pascal programming language and a derivation tree representing a program of approximately 500 lines of Pascal source code. Compared with the first group of experiments, the results from this group showed significantly fewer benefits from the use of using parallel attribution. It was argued that features of the grammar, the derivation tree, and the attribution methods themselves combined to produce these markedly small speedups. A non-incremental asynchronous parallel algorithm performing all types of plan elements asynchronously was shown to achieve more significant speedups.

Thus, while the parallel attribution methods can indeed produce useful improvements in attribution times, the methods are sensitive both to overhead and to characteristics of individual attribute grammars and derivation trees. These facts argue for the importance of finding methods for (manually or automatically) transforming attribute grammars to provide increased opportunities for parallelism, for the importance of constructing optimal plans, and for the importance of minimizing overhead if such parallel methods are to be generally useful.

8.2 Directions for Further Work

As we note above, attribute grammar transformation and optimal plan construction are important topics in which study remains to be done. Reduction in process overhead (an area of persistent interest in the areas of general parallel computing, operating systems, machine architecture, etc.) continues to be a topic of much research.

The value of the parallel attribution methods presented here can be determined only by more detailed examination. A good first step would involve implementing the methods on simulators capable of much greater discrimination than the one used here. Finally, actual implementations on actual multiprocessors would be necessary to determine the practical worth of the methods.

Bibliography

[Aho et al 1986]

Aho, A., Sethi, R., and Ullman, J. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[Alamasi & Gottlieb 1989]

Alamasi, G. and Gottlieb, A. *Highly Parallel Computing.* Benjamin Cummings, Redwood City, CA, 1989.

[Bochmann 1976]

Bochmann, G. Semantic Evaluation From Left to Right. *Communications of the ACM 19*, 2 (February 1976), pp. 55-62.

[Boehm & Zwaenepoel 1987]

Boehm, H. and Zwaenepoel, W. Parallel Attribute Grammar Evaluation. *Proceedings of the Seventh International Conference on Distributed Computing Systems*, Berlin, West Germany, 1987, pp. 347-354.

[Cheng 1990]

Cheng, K. A Simultaneous Access Queue. *Journal of Parallel and Distributed Computing 9*, 1 (May 1990), pp. 83-86.

[Deransart et al 1988]

Deransart, P., Jourdan, M., and Lorho, B. *Attribute Grammars: Definitions, Systems and Bibliography.* Lecture Notes in Computer Science 323, Springer-Verlag, New York, NY, 1988.

[DeRemer 1969]

DeRemer, F. Practical Translators for LR(k) Languages. PhD Dissertation, MIT, Cambridge, MA, 1969.

[Dijkstra 1968]

Dijkstra, E. Cooperating Sequential Processes. In *Programming Languages*, F. Genuys (ed.), Academic Press, New York, NY, 1968.

[Duncan 1990]

Duncan, R. A Survey of Parallel Computer Architectures. *IEEE Computer 23*, 2 (February 1990), pp. 5-16.

[Engelfriet & Filè 1982]

Engelfriet, J. and Filè, G. Simple Multi-Visit Attribute Grammars. *Journal of Computer and Systems Sciences 24*, 3 (June 1982), pp. 283-314.

[Fang 1972]

Fang, I. FOLDS - A Declarative Formal Language Definition System. Technical Report STAN-CS-72-329, Computer Science Dept., Stanford University, Palo Alto, CA, 1972.

[Farrow 1982]

Farrow, R. LINGUIST-86: Yet Another Translator Writing System Based on Attribute Grammars. *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, Boston, MA, June 23-25, 1982, pp. 160-171. (*SIGPLAN Notices 17*, 6 (June 1982)).

[Farrow 1989]

Farrow, R. *The Linguist Translator-Writing System - User's Manual.* Declarative Systems Inc., Palo Alto, CA, 1989.

[Flynn 1966]

Flynn, M. Very High-Speed Computing Systems. *Proceedings IEEE 54*, 1966, pp. 1901-1909.

[Fortune & Wyllie 1978]

Fortune, S. and Wyllie, J. Parallelism in Random Access Machines. *Proceedings of the Tenth ACM Symposium on Theory of Computing*, 1978, pp. 114-118.

[Garey & Johnson 1979]

Garey, M. and Johnson, D. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, CA, 1979.

[Goto 1974]

Goto, E. Monocopy and Associative Structures in an Extended Lisp. Information Sciences Laboratory, University of Tokyo, Tokyo, Japan, 1974.

[Greenbaum 1989]

Greenbaum, A. Synchronization Costs on Multiprocessors. *Parallel Computing 10*, 1 (March 1989), pp. 3-14.

[Hoover 1986]

Hoover, R. Dynamically Bypassing Copy Rule Chains in Attribute Grammars. *Proceedings of the Thirteenth ACM Symposium on Principles of Programming Languages*, St. Petersburg, FL, January 13-15, 1986, pp. 14-25.

[Hoover 1987]

Hoover, R. Incremental Graph Evaluation. Technical Report 87-836, Dept. of Computer Science, Cornell University, Ithaca, NY, 1987.

[Hoover & Teitelbaum 1986]

Hoover, R. and Teitelbaum, T. Efficient Incremental Evaluation of Aggregate Values in Attribute Grammars. *Proceedings of the*

SIGPLAN '86 Symposium on Compiler Construction, Palo Alto, CA, June 25-27, 1986, pp. 39-50. (*SIGPLAN Notices 21*, 7 (July 1986)).

[Jazayeri & Walter 1975]

Jazayeri, M. and Walter, K. Alternating Semantic Evaluator. *Proceedings of the ACM 1975 Annual Conference*, 1975, pp. 230-234.

[Jazayeri et al 1975]

Jazayeri, M., Ogden, W., and Rounds, W. The Intrinsically Exponential Complexity of the Circularity Problem for Attribute Grammars. *Communications of the ACM 18*, 12 (December 1975), pp. 697-706.

[Jensen & Wirth 1985]

Jensen, K. and Wirth, N. *Pascal User Manual and Report, 3rd Ed.* Springer-Verlag, New York, NY, 1985.

[Johnson 1978]

Johnson, S. YACC - Yet Another Compiler-Compiler. Technical Report, Bell Laboratories, Murray Hill, NJ, 1978.

[Johnson & Fischer 1985]

Johnson, G. and Fischer, C. A Meta-Language and System for Nonlocal Incremental Attribute Evaluation in Language-Based Editors. *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, New Orleans, LA, January 1985, pp. 141-151.

[Jourdan et al 1990]

Jourdan, M., Didier, P., Julié, C., Durin, O., and LeBellec, C. Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System. *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 20-22, 1990, pp. 209-222. (*SIGPLAN Notices 25*, 6 (June 1990)).

[Kaplan & Kaiser 1986]

Kaplan, S. and Kaiser, G. Incremental Attribute Evaluation in Distributed Language-Based Editors. *Proceedings of the Fifth ACM Symposium on Principles of Distributed Computing*, 1986, pp. 121-130.

[Kastens 1978]

Kastens, U. Ordered Attribute Grammars. Bericht 7/78, Institut für Informatik II, Universität Karlsruhe, 1978.

[Kastens 1980]

Kastens, U. Ordered Attribute Grammars. *Acta Informatica 13*, 3 (1980), pp. 229-256.

[Kastens et al 1982]

Kastens, U., Hutt, B., and Zimmerman, E. *GAG: A Practical Compiler Generator*. Lecture Notes in Computer Science 141, Springer-Verlag, New York, NY, 1982.

[Kennedy & Warren 1976]

Kennedy, K. and Warren, S. Automatic Generation of Efficient Evaluators for Attribute Grammars. *Proceedings of the Third ACM Symposium on Principles of Programming Languages*, Atlanta, GA, January 19-21, 1976, pp. 32-49.

[Kernighan & Plauger 1981]

Kernighan, B. and Plauger, P. *Software Tools in Pascal*. Addison-Wesley, Reading, MA, 1981.

[Knuth 1968]

Knuth, D. Semantics of Context-Free Languages. *Mathematical Systems Theory 2*, 2 (June 1968), pp. 127-145. Correction. *ibid. 5*, 1 (March 1971), pp. 95-96.

[Knuth 1973]

Knuth, D. *The Art of Computer Programming. Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, MA, 1973.

[Kuiper 1989]

Kuiper, F. Parallel Attribute Evaluation. PhD Dissertation, Dept. of Computer Science, University of Utrecht, Utrecht, The Netherlands, 1989.

[Leiserson 1983]

Leiserson, C. *Area-Efficient VLSI Computation*. MIT Press, Cambridge, MA, 1983.

[Lewis et al 1974]

Lewis, P., Rosenkrantz, D., and Stearns, R. Attributed Translations. *Journal of Computer and Systems Sciences 9*, 3 (December 1974), pp. 279-307.

[Muchnik & Jones 1981]

Muchnik, S. and Jones, N. *Program Flow Analysis: Theory and Applications*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Naini 1985]

Naini, S. Design of Attribute Evaluators and Attribute Machine. Technical Report MS-CIS-86-02, Dept. of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, 1986.

[Pettis & Hansen 1990]

Pettis, K. and Hansen, R. Profile Guided Code Positioning. *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 20-22, 1990, pp. 16-27. (*SIGPLAN Notices 25*, 6 (June 1990)).

[Reps 1984]

Reps, T. *Generating Language-Based Environments*. MIT Press, Cambridge, MA, 1984.

[Reps & Teitelbaum 1989a]

Reps, T. and Teitelbaum, T. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, New York, NY, 1989.

[Reps & Teitelbaum 1989b]

Reps, T. and Teitelbaum, T. *The Synthesizer Generator Reference Manual, Third Edition*. Springer-Verlag, New York, NY, 1989.

[Sarkar 1989]

Sarkar, V. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, 1989.

[Schell 1979]

Schell, R. Methods for Constructing Parallel Compilers for Use in a Multiprocessor Environment. Technical Report UIUCDCS-R-79-958, Dept. of Computer Science, University of Illinois, Urbana, IL, 1979.

[Snir 1982]

Snir, M. On Parallel Search. *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Ottawa, Canada, August 18-20, 1982, pp. 242-253.

[Valdes et al 1979]

Valdes, J., Tarjan. R., and Lawler, E. The Recognition of Series Parallel Digraphs. *Proceedings of the Eleventh ACM Symposium on Theory of Computing*, Atlanta, GA, 1979, pp. 1-12.

[Waite 1976]

Waite, W. Optimization. In *Compiler Construction: An Advanced Course, 2nd Ed.*, F. Bauer and J. Eickel (eds.), Springer-Verlag, New York, NY, 1976.

[Waite & Goos 1984]

Waite, W. and Goos, G. *Compiler Construction*. Springer-Verlag, New York, NY, 1984.

[Yeh 1983]

Yeh, D. On Incremental Evaluation of Ordered Attributed Grammars. *BIT 23* (1983), pp. 308-320.