JOURNAL OF COMPUTER AND SYSTEM SCIENCES 24, 283–314 (1982)

Simple Multi-Visit Attribute Grammars

JOOST ENGELFRIET AND GILBERTO FILÈ

Department of Applied Mathematics, Twente University of Technology, 7500 AE Enschede, The Netherlands

Received October 30, 1980

An attribute grammar is simple multi-visit if each attribute of a nonterminal has a fixed visit-number associated with it such that, during attribute evaluation, the attributes of a node which have visit-number j are computed at the jth visit to the node. An attribute grammar is l-ordered if for each nonterminal a linear order of its attributes exists such that the attributes of a node can always be evaluated in that order (cf. the work of Kastens).

An attribute grammar is simple multi-visit if and only if it is l-ordered. Every noncircular attribute grammar can be transformed into an equivalent simple multi-visit attribute grammar which uses the same semantic operations.

For a given distribution of visit-numbers over the attributes, it can be decided in polynomial time whether the attributes can be evaluated according to these visit-numbers. The problem whether an attribute grammar is simple multi-visit is *NP*-complete.

INTRODUCTION

In [9] Kastens introduced the class of ordered attribute grammars. An attribute grammar (AG) G is ordered if for each nonterminal a linear order of its attributes exists with the property that for every node of a derivation tree of G its attributes can be evaluated in that order. The ordered AG are attractive because (1) they include naturally the multi-pass AG of [2, 8], and (2) they have a simple attribute evaluation method. With regard to point (2), it is shown in [9, 10] that for every ordered AG a tree-walking attribute evaluator can be constructed with the following properties: (i) it does not store any information at the nodes of the tree (apart from the values of the attributes), (ii) it visits each node of the tree a bounded number of times, and, most importantly, (iii) it decides which attributes (of a node) to compute and which sons to visit (in what order) on the basis of the number of the current visit only (i.e., how many times the node has been visited plus one), and this number is passed as a parameter to the evaluation procedure. The ordered AG are a proper subclass of the absolutely noncircular AG of [11] for which a more complicated tree-walking evaluator is given in [11]; this evaluator stores information at the nodes of the tree and uses it to decide which attributes to compute and how to walk through the tree.

In [9] a wrong proof was given to show that it takes polynomial time to decide whether an AG is ordered. To overcome this difficulty the notion of ordered AG was redefined in [10] and called OAG; the class of OAG is a proper subclass of the

0022-0000/82/030283–32$02.00/0

original class of ordered AG which we shall henceforth call *l-ordered* AG. It takes polynomial time to decide whether an attribute grammar is an OAG. In this paper we shall show that *to the contrary*, deciding whether an AG is l-ordered, is an *NP*-complete problem.

As mentioned above, for each l-ordered AG there is a tree-walking evaluator which visits each node n a bounded number of times and computes a (fixed) subset $A_j(F)$ of the attributes of nonterminal F (labeling n) at the jth visit of n. In this paper we shall give a precise definition of AG for which such a tree-walking attribute evaluation strategy exists, calling them *simple multi-visit* attribute grammars, and prove that they coincide with the l-ordered AG.

Clearly, the simple multi-visit AG generalize the (simple) multi-pass AG of [2, 8], for which the above tree-walking evaluator is restricted to perform left-to-right (or right-to-left) passes over the tree. The adjective *simple* was added by Alblas [1] to stress the fact that each attribute of a nonterminal has a fixed pass-number. This provided us with another motivation to introduce the simple multi-visit AG, in which each attribute has a fixed visit-number. Dropping the restriction of a fixed pass- (or visit-) number for each attribute, but keeping the bounds on the number of passes (or visits), the larger classes of *pure* multi-pass AG [1] and *pure* multi-visit AG are obtained. Concerning the complexity of deciding these properties of AG the following can be said: The pure multi-pass property is exponential-time complete [5]. The simple multi-pass property is decidable in polynomial time by the algorithm in [2]. The pure multi-visit property is exponential-time complete, because it coincides with the property of noncircularity [7]; in fact, for every noncircular AG there is a tree-walking evaluator which makes a bounded number of visits to any node of a derivation tree (cf. [13, 14]). Finally, as previously said, we shall prove in this paper that the simple multi-visit property is *NP*-complete, even if the number of visits is restricted to two (the one-visit property can be decided in polynomial time, cf. [4]).

The paper is organized as follows: Section 1 contains preliminary definitions. In Section 2 we shall define the notion of simple multi-visit AG and show that it can be decided in nondeterministic polynomial time. The way this is shown is as follows: We first prove that there is an algorithm which, given an AG G and a partition $\langle A_1(F), A_2(F),..., A_k(F)\rangle$ of the attributes of nonterminal F (for every F; where k may depend on F), decides in deterministic polynomial time whether a tree-walking evaluator exists which always computes the attributes of $A_j(F)$ at the jth visit of a node labeled F (and if so, the evaluator can also be constructed in polynomial time and has properties (i)–(iii) mentioned above). The algorithm simply consists of checking whether there are no cycles in certain graphs, easily constructed from the semantic rules of G and the partitions $A_j(F)$. Clearly, the simple multi-visit property can then be decided by guessing a partition for each nonterminal and checking whether it is correct in the above sense. We also discuss how to simplify a given (correct) set of partitions. We end the section by showing that every (noncircular) AG can be transformed into an equivalent simple multi-visit AG (using the same semantic rules).

In Section 3 we shall define l-ordered AG and prove that an AG is l-ordered if and only if it is simple multi-visit. Finally in Section 4 we shall show that the simple

multi-visit (and the simple 2-visit) property is *NP*-hard by reducing the satisfiability problem for Boolean formulas to it.

1. PRELIMINARIES

We denote by $[n, m]$ the set of integers $\{i \mid n \leqslant i \leqslant m\}$ and, if B is a finite set, we indicate with $\#B$ the number of its elements.

We now recall the definition of attribute grammar (AG) [2, 12] and discuss some related concepts which will be useful in the following parts.

DEFINITION 1.1. An *attribute grammar* G consists of (1)–(4) as follows:

(1) G has a context-free grammar $G_0 = (T, N, P, Z)$, called the *underlying context-free grammar* of G, consisting of terminals, nonterminals, productions, and initial nonterminal, respectively. We shall always denote production $p \in P$ of G_0 (or of G) as

$$p: F_0 \to w_0 F_1 w_1 \cdots w_{n_{p-1}} F_{n_p} w_{n_p},$$

where $F_i \in N$ and $w_i \in T^*$, for $i \in [0, n_p]$ and $n_p \geqslant 0$. When considering a derivation tree of G_0, we assume its leaves to be labeled by terminals (or the empty string); a derivation tree is said to be *complete* if its root is labeled by Z. We assume the underlying context-free grammar to be reduced in the usual sense.

(2) Each nonterminal F of G has two associated disjoint, finite sets, denoted $S(F)$ and $I(F)$, of synthesized and inherited *attributes*, respectively (shortly *s*- and *i*-attributes). The initial nonterminal Z does not have any *i*-attribute and one of its *s*-attributes is designated to hold the translation of any complete derivation tree in G. We indicate the set of all attributes of nonterminal F of G with $A(F) = I(F) \cup S(F)$ and, to avoid trivial cases, we assume that $A(F) \neq \varnothing$ for all F. An attribute $b \in A(F)$ is also denoted with $b(F)$. Note that terminals do not have attributes.

(3) With each attribute a of G a set of possible *values* of a is associated, indicated with $V(a)$.

(4) With each production $p \in P$ of G is associated a set r_p of *semantic rules* which have the following form: $a_0(F_{i_0}) = f(a_1(F_{i_1}),..., a_m(F_{i_m}))$, where $i_j \in [0, n_p]$ and f is a mapping from $V_1 \times V_2 \times \cdots \times V_m$ to V_0, where $V_j = V(a_j)$, for all $j \in [1, m]$. We say that $a_0(F_{i_0})$ *depends on* $a_1(F_{i_1}),..., a_m(F_{i_m})$ *in* p. When the identity of the nonterminals is not important we indicate a semantic rule simply by $a_0 = f(a_1,..., a_m)$. We assume that the semantic rules in r_p define *all and only* the attributes in $S(F_0)$ and $I(F_j)$ using as arguments *only* attributes in $I(F_0)$ and $S(F_j)$, $j \in [1, n_p]$ (see [2], we shall refer to this assumption as Bochmann normal form). ∎

When considering a derivation tree t of an AG, we shall often identify a node n of t with the nonterminal labeling it. Therefore, if F is this nonterminal, we shall say

that n has a set of attributes and indicate this set with $A(n) = A(F)$ and, similarly, for the i- and s-attributes $I(n) = I(F)$ and $S(n) = S(F)$.

Working with AG, it is useful to visualize the dependencies among attributes by means of graphs.

DEFINITION 1.2. Given an AG G, let production p of G be of the form $F_0 \to w_0 F_1 w_1 \cdots w_{n_p-1} F_{n_p} w_{n_p}$. The *production graph* of p (denoted by $\mathrm{pg}(p)$) is the graph having as nodes the attributes of all nonterminals F_j of p, $j \in [0, n_p]$, and in which there is an edge running from attribute a_1 to attribute a_2 iff a_2 depends on a_1 in p (see also [11, 12]). ∎

Because of the obvious fact that derivation trees consist of productions pasted together, for each derivation tree t of a given AG G, we obtain, as usual, a dependency graph, called *derivation tree graph* of t (denoted by $\mathrm{dtg}(t)$), by pasting together the $\mathrm{pg}(p)$'s of all the productions p used in t (see also [11, 12]).

An AG G is said to be *noncircular* (well formed in [12]) if there is no derivation tree t in G such that $\mathrm{dtg}(t)$ contains an oriented cycle.

2. THE SIMPLE MULTI-VISIT PROPERTY

We start this section by giving an intuitive description of the concepts of attribute evaluation strategy and visit for AG, and, on this basis, we introduce the class of simple multi-visit AG as a natural extension of already known classes of AG. Then we give formal definitions of these concepts and we prove some results on the problem of testing whether an AG is simple multi-visit and its complexity. Finally, we describe an attribute evaluation algorithm for this class of AG and we show that any (noncircular) AG can be transformed into a simple multi-visit one defining the same translation.

When talking about attribute evaluation strategy on a derivation tree t of an AG (cf. [4]), we have an intuitive idea of it which can be described as follows: starting from the root of t and going from node to node, semantic rules are evaluated and the values of the computed attributes are appended to the corresponding nodes, until all attributes have been computed and the root is reached again. Being a little more specific on the way nodes of t are traversed, we say that from a node n of t it is possible to go next to either a son of n, a brother of n, the father of n or stay in n itself. In our study of AG we want to have a way of precisely describing these actions of walking through a derivation tree computing attributes. There may be many ways to do so; we have chosen a particular one based on the following observations.

In our definition of AG we have assumed to consider AG in Bochmann normal form. This assumption has an important consequence on the way information flows through a derivation tree during attribute evaluation.

Consider Fig. 1. The i-attributes of F are defined in production p_1 and (possibly) used in p_2, whereas for the s-attributes of F it is exactly the opposite. In other words,

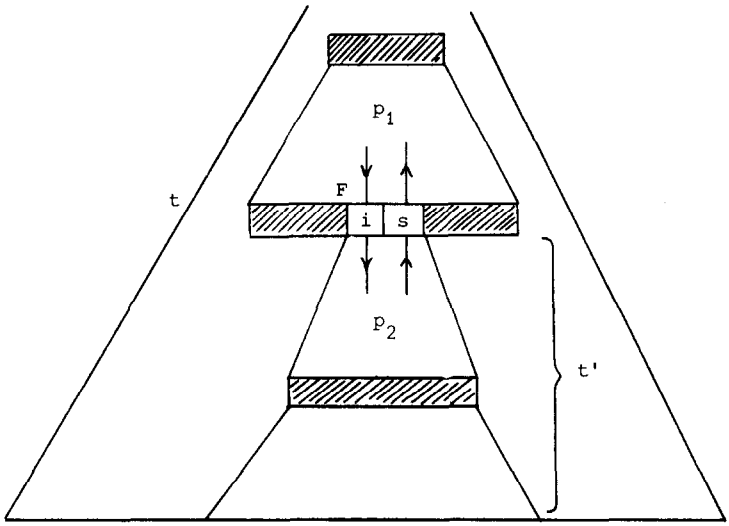

FIG. 1. Flow of information through node F of a derivation tree t.

we can say that the i-attributes of F transmit information coming from the parts of t surrounding t' (the context of t') into t', whereas the s-attributes of F take information concerning t' into its context. Therefore, computing some i-attributes of F is sensible only if immediately afterwards their values are used, i.e., their information is passed down into t' (t' is entered). In the same way, computing some s-attributes of F is sensible only if their values are immediately passed up in the context of t' (t' is exited). By *it is sensible* we actually mean that, if this does not happen, then the computation of the attributes could be postponed. We capture this fact saying roughly that:

> F (and so t') is *entered* if it is reached from *above* (context) and, in this case, we allow only i-attributes of F to be computed.
>
> F (and so t') is *exited* if it is reached from *below* and, in this case, we allow only s-attributes of F to be computed.

Being more precise, in the previous statements, *above* can be either the father of F or a brother of F (in p_1) or F itself (when F is just exited), and *below* can be either a son of F (in p_2) or F itself (when F is just entered).

The idea is then to consider any attribute evaluation strategy as a sequence of actions of either entering or exiting a node. A sequence of actions corresponding to entering F, walking through t' and then exiting F is called a *visit* to t'. Recursively, we say that a visit to t' consists of entering F, a sequence of visits to the subtrees rooted in some of the sons of F and finally exiting F. This intuitive concept of visit is sufficient to introduce the classes of AG in which we are interested.

In [2] some classes of AG are defined which allow particularly simple attribute evaluation strategies. The first class is that of *one-pass, left-to-right evaluatable* AG

(L-AG). We can say shortly that an AG G is L iff for each derivation tree t of G there is an attribute evaluation strategy computing all attributes and such that it traverses t from left to right visiting each subtree of t exactly once. Such a strategy is called a (left-to-right) pass.

Also in [2] the concept of one-pass AG is extended to that of multi-pass AG and in [1] it is shown that this extension gives rise to two types of multipass AG, the pure and the simple multi-pass AG, defined as follows:

> An AG G is pure m-pass, $m > 0$, iff for each complete derivation tree t of G m (left-to-right) passes over t are sufficient to evaluate all its attributes.
>
> An AG G is simple m-pass, $m > 0$, iff there exists, for each nonterminal F of G, a partition $A_1(F),\ldots, A_m(F)$ of the set of attributes $A(F)$ such that for any complete derivation tree t of G all the attributes of t can be evaluated in m (left-to-right) passes over t and in the jth pass all and only the attributes in $A_j(F)$ are computed, $j \in [1, m]$.

Clearly an AG G is pure (simple) multi-pass iff it is pure (simple) m-pass for some $m > 0$. Note that the class of simple multi-pass AG is that actually considered in [2].

In [4] the concept of L-AG is extended to that of one-visit AG simply dropping from the above definition of L-AG the requirement that the traversal of the derivation tree must be from left-to-right.

An AG G is one-visit iff, for every complete derivation tree t of G, there exists an attribute evaluation strategy computing all attributes of t and visiting each subtree of t at most once.

In exactly the same way it is possible to extend to visits the simple and pure multi-pass properties. It is straightforward, in fact, to define the classes of pure multi-visit AG [14] and of simple multi-visit AG as follows:

> An AG G is pure m-visit, $m > 0$, iff for every complete derivation tree t of G, there is an attribute evaluation strategy which computes all attributes of t visiting each subtree of t at most m times.
>
> An AG G is simple m-visit, $m > 0$, iff for each nonterminal F of G, there exists a partition $A_1(F),\ldots, A_k(F)$ of the set of attributes $A(F)$, where $0 < k \leqslant m$ and k may depend on F, i.e., $k = k(F)$, such that, for any complete derivation tree t of G, there exists an attribute evaluation strategy for t which, for every occurrence of F in t, computes all the attributes in $A_j(F)$ during the jth visit to the subtree rooted in F, for all $j \in [1, k(F)]$.

Also in this case then an AG G is pure (simple) multi-visit iff it is pure (simple) m-visit for some $m > 0$. Whereas it has been shown in [14] that every noncircular AG is pure multi-visit, the same is not true for simple multi-visit AG. In Section 4 we shall give examples of noncircular AG which are not simple multi-visit.

We now start giving the formal definitions, based on the intuitive description given

above. As we have already said, we shall view an attribute evaluation strategy as a sequence of actions of either entering or exiting a node. From this, it is clear that the computation performed by an attribute evaluation strategy on a derivation tree t can be described by means of a sequence of pairs of the form: (n, IW) or (n, SW), where n is a node of t, $IW \subseteq I(n)$, and $SW \subseteq S(n)$. The pair (n, IW) (or (n, SW)) corresponds to the action of entering (or exiting) node n computing the attributes in IW (or SW, respectively). We call such action of the attribute evaluation strategy a *basic action* and the corresponding pair a *basic action symbol* (ba-symbol). In general, we indicate a ba-symbol with (n, XW) where X stands for either I or S. We now give a formal way of describing the computation of any attribute evaluation strategy (it is equivalent to the corresponding definition in [14]).

DEFINITION 2.1. Given an AG G and a complete derivation tree t of G, a *computation sequence* for t is a string h of ba-symbols (n, XW), where n is a nonterminal node of t and, for X equal to I or S, $XW \subseteq X(n)$, such that the following four conditions hold:

(1) *Start–end condition.* The first and the last ba-symbols of h correspond respectively to the basic actions of entering the root of t (i.e., $(Z, \varnothing)$ because $I(Z) = \varnothing$) and of exiting it (i.e., (Z, SW), where $SW \subseteq S(Z)$).

(2) *Sequentiality condition.* For any two contiguous ba-symbols $(n_1, X_1 W_1)(n_2, X_2 W_2)$ in h, n_2 is either the father of n_1, or a son of n_1, or a brother of n_1, or n_1 itself. Moreover, the following relationships hold among n_1, n_2, X_1 and X_2:

if	n_2	is the father of	n_1,	then	$X_1 = S$	and	$X_2 = S$,
if	n_2	is a son of	n_1,	then	$X_1 = I$	and	$X_2 = I$,
if	n_2	is a brother of	n_1,	then	$X_1 = S$	and	$X_2 = I$,
if	n_2	is equal to	n_1,	then:			

either, $X_1 = I$ and $X_2 = S$ (entering n and exiting it immediately), or, $X_1 = S$ and $X_2 = I$ (exiting n and reentering it immediately).

Conditions (1) and (2) imply that, if we consider a particular node n of t and all the ba-symbols of h concerning n, then h can be written as $h = u_1(n, IW_1)\, v_1(n, SW_1)\, u_2 \cdots u_k(n, IW_k)\, v_k(n, SW_k)\, u_{k+1}$, where, if t' is the subtree of t rooted in n, each u_j is composed of ba-symbols concerning nodes of t outside t' and each v_j contains ba-symbols concerning only nodes of t' (apart from its root n).

(3) *Feasibility condition.* If $h = (n_1, X_1 W_1)(n_2, X_2 W_2) \cdots (n_r, X_r W_r)$, then no attribute in $X_j W_j$ depends in some production p, used in t, on an attribute in $X_i W_i$, for $i \geqslant j$ and $i, j \in [1, r]$.

(4) *Completeness condition.* For each node n of t, if (n, IW_1), (n, SW_1),..., (n, IW_k), (n, SW_k) is the sequence of ba-symbols of h concerning node n, then $\langle IW_1 \cup SW_1, ..., IW_k \cup SW_k \rangle$ is a partition of $A(n)$. ∎

The first two conditions of Definition 2.1 reflect very closely the intuitive description of an attribute evaluation strategy outlined initially. Condition (3) checks that the computation sequence represents an attribute evaluation strategy which can actually be executed. Finally, condition (4) guarantees that all the attributes of t are computed and only once.

A computation sequence h for a complete derivation tree t represents an attribute evaluation strategy for t. Therefore, a visit of the attribute evaluation strategy to a subtree t' of t, rooted in node n, is represented by a substring v of h which starts and ends with ba-symbols corresponding respectively to the basic actions of entering and exiting node n and in which all other ba-symbols concern nodes of t' below its root n. We call such a string v a *visit-trace* of h for t'. If h contains m distinct visit-traces for t', we say that *it visits t' m times*. In the same way we say that h is an *m-visit computation sequence* for t if it visits each subtree of t at most m times.

Note that the class of pure multi-visit AG [14], which we mentioned in the first part of this section, is the class of those AG for which, for some $m > 0$, there exists an m-visit computation sequence for every complete derivation tree. It is, instead, more complicated to define the class of simple multi-visit AG. We have already seen, informally, that an AG belongs to this class iff there exists, for every complete derivation tree, a computation sequence (attribute evaluation strategy) satisfying certain conditions on which attributes are computed, for a node n, during each visit to the subtree rooted in n, viz. the partition in the completeness condition, Definition 2.1(4), should only depend on the nonterminal labeling node n.

We shall now make this definition precise. We need first to introduce the concept of a set Π of partitions of the attributes for an AG. We shall use the concept of partition of a set A in a slightly different way than usual: it is a *sequence* $\langle A_1,\ldots, A_k\rangle$ of mutually disjoint subsets of A whose union is A (thus an order is added to the usual concept).

For an AG G we denote by Π a set containing for each nonterminal F of G a partition $\Pi(F)$ of $A(F)$. The partition $\Pi(F)$ is indicated by $\langle A_1(F),\ldots, A_k(F)\rangle$, $k \geqslant 1$, and for any $j \in [1, k]$, we denote by $IA_j(F)$ and $SA_j(F)$ the sets of the i- and s-attributes of $A_j(F)$, respectively. It will always be understood that k may depend on F, i.e., $k = k(F)$. We identify by $\max(\Pi)$ the maximum of all $k(F)$. Note that $A_j(F)$ may be empty and then $IA_j(F) = SA_j(F) = \varnothing$, or it may contain only i- or only s-attributes in which case $SA_j(F)$ or $IA_j(F)$, respectively, would be empty.

The concept of computation sequence and that of set of partitions are now combined as follows:

DEFINITION 2.2. Given an AG G, a set Π of partitions for it, and a complete derivation tree t of G, a computation sequence h for t *respects* Π if the following condition holds for h: Let n be a node of t, t' the subtree rooted in n and F the label of n. If $\Pi(F) = \langle A_1(F),\ldots, A_k(F)\rangle$, then h visits t' $k(F)$ times and in such a way that the jth visit-trace of h for t' begins with the ba-symbol $(n, IA_j(F))$ and ends with the ba-symbol $(n, SA_j(F))$, for all $j \in [1, k(F)]$. ∎

Note that the condition we require for a computation sequence to respect Π implies that it satisfies the completeness condition (Definition 2.1(4)). We are finally able to define formally the class of simple multi-visit AG.

DEFINITION 2.3. (i) A set Π of partitions for AG G is *simple multi-visit* (smv) if for each complete derivation tree of G there is a computation sequence for it respecting Π.

(ii) An AG G is *simple multi-visit* (smv) iff there exists an smv-set of partitions Π for G. ∎

Clearly, an AG G is simple m-visit iff there is an smv-set of partitions Π for G such that $\max(\Pi) \leqslant m$.

We shall now show that there is an easy way to test whether a given set of partitions is smv. The test is based on the noncircularity of graphs which represent the precedence-relations (among attribute sets) that are created by the partitions and by the actual dependencies in the semantic rules of G. These graphs are defined as follows:

DEFINITION 2.4. Let G be an AG and Π a set of partitions for G with $\Pi(F) = \langle A_1(F),..., A_k(F)\rangle$ for any nonterminal F of G.

(i) The *partition graph for nonterminal* F ($\Pi(F)$-graph) is the graph which has as nodes the sets $IA_j(F)$ and $SA_j(F)$, for $j \in [1, k(F)]$, and in which edges run, for every $j \in [1, k(F)]$, from $IA_j(F)$ to $SA_j(F)$ and, for every $i \in [1, k(F) - 1]$, from $SA_i(F)$ to $IA_{i+1}(F)$.

(ii) Let production p in G be $F_0 \to w_0 F_1 w_1 \cdots w_{n_{p-1}} F_{n_p} w_{n_p}$; the *partition graph for* p ($\Pi(p)$-graph) is the graph constructed by combining the $\Pi(F_i)$-graphs, for $i \in [0, n_p]$, and adding to it edges as follows: an edge runs from node $XA_j(F_n)$ to node $X'A_i(F_m)$ iff (at least) one attribute of the second set depends on an attribute of the first in production p, where n and $m \in [0, n_p]$, $j \in [1, k(F_n)]$, $i \in [1, k(F_m)]$, and X, $X' \in \{I, S\}$. ∎

We point out that the concept of brother graph, used in [4] to characterize the class of one-visit AG, can be viewed as a particular case of $\Pi(p)$-graph: the case in which the trivial partition is chosen for all nonterminals, i.e., $\Pi(F) = \langle A(F)\rangle$. This clearly corresponds to the fact that one-visit AG are a particular case of smv-AG.

In order to make the reader more comfortable with the last concepts introduced we give:

EXAMPLE 2.1. An AG G consists of: nonterminals Z, A, and B; terminals a and b. The sets of attributes of each nonterminal are: $I(Z) = \emptyset$, $S(Z) = \{s\}$; $I(A) = \{i_1, i_2\}$, $S(A) = \{s_1, s_2\}$, $I(B) = \{i_1, i_2\}$, $S(B) = \{s_1, s_2\}$. The productions of G with their corresponding sets of semantic rules are as follows:

Production 1. $Z \to AB$, $r_1 = \{s(Z) = s_2(A)\, s_1(B)$, $i_2(B) = s_1(A)$, $i_1(A) = s_2(B)$, $i_2(A) = s_1(B)$, $i_1(B) = 1\}$.

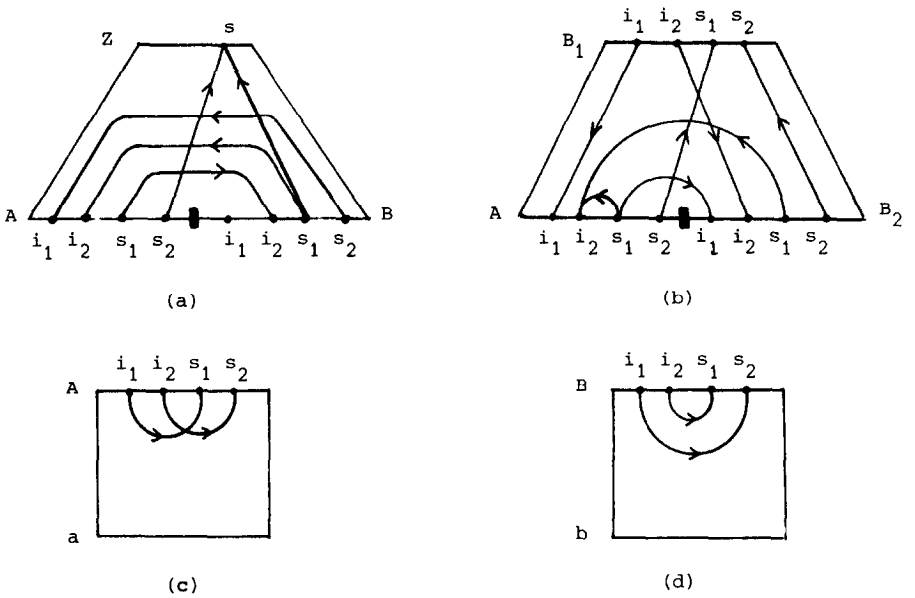

FIG. 2.. In (a), (b), (c) and (d) are represented, respectively, the production graphs of Productions 1, 2, 3, and 4 of the AG G of Example 2.1.

Production 2. $B_1 \to AB_2$ (the subscripts are added to distinguish the two occurrences of B), $r_2 = \{i_1(A) = i_1(B_1),\ i_2(A) = s_1(B_2)\ s_1(A),\ i_1(B_2) = s_1(A),\ i_2(B_2) = i_2(B_1),\ s_1(B_1) = s_2(A),\ s_2(B_1) = s_2(B_2)\}$.

Production 3. $A \to a$, $r_3 = \{s_1(A) = i_1(A), s_2(A) = i_2(A)\}$.

Production 4. $B \to a$, $r_4 = \{s_1(B) = i_2(B), s_2(B) = i_1(B)\}$.

In Fig. 2 the production graphs for G are represented.

Let us now consider the set Π of partitions for G whose elements are as follows: $\Pi(Z) = \langle\{s\}\rangle$; $\Pi(A) = \langle\{i_1, s_1\}, \{i_2, s_2\}\rangle$; $\Pi(B) = \langle\{i_1, s_1\}, \{i_2, s_2\}\rangle$. The corresponding partition graph of Production 1 of G is represented in Fig. 3 in which, because each node is labeled with a singleton, for simplicity we dropped the parentheses. ∎

Using the concept of partition graph for a production we show the following important result:

THEOREM 2.1. *Given an* AG G *and a set* Π *of partitions for* G, Π *is simple multi-visit iff none of the corresponding partition graphs for the productions of* G *contains an oriented cycle.*

Proof. (a) (Π is smv $\Rightarrow$ no cycles in the $\Pi(p)$-graphs). Consider the partition graph for production p of G and any complete derivation tree t of G in which there is an occurrence of production p. Let h be a computation sequence for t which respects

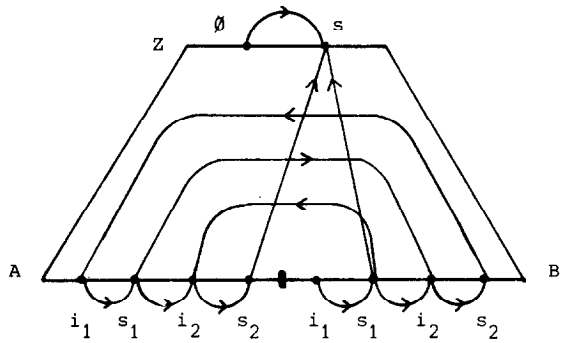

FIG. 3. The partition graph of Production 1 of the AG G of Example 2.1.

Π; we know that such an h exists because Π is an smv-set of partitions for G. There is clearly a correspondence between nodes of the partition graph of p and ba-symbols of h. In fact, by Definition 2.2, for any $A_j(F_i) \in \Pi(F_i)$, h contains the two ba-symbols $(F_i, IA_j(F_i))$ and $(F_i, SA_j(F_i))$ whereas, by Definition 2.4, $IA_j(F_i)$ and $SA_j(F_i)$ are nodes of the partition graph of p, $i \in [0, n_p]$ and $j \in [1, k(F_i)]$.

At this point it is sufficient to observe that, if in the partition graph for p there is an oriented path running from node $XA_j(F_i)$ to node $X'A_m(F_r)$, where i and $r \in [0, n_p]$, $j \in [1, k(F_i)]$, $m \in [1, k(F_r)]$, X and $X' \in \{I, S\}$, then, the ba-symbol of h corresponding to the first node must precede in h that corresponding to the second one. This is true because an oriented path in the partition graph for p may consist of:

(i) edges which are part of the partition graph of some F_i ($\Pi(F_i)$-graph, see Definition 2.4(i)), $i \in [0, n_p]$, and then the precedence relation in h is guaranteed by the fact that h respects Π;

(ii) edges which represent dependencies between attributes in p. In this case it is clear that if we have one such edge from $XA_j(F_i)$ to $X'A_m(F_r)$, then, by the feasibility condition (Definition 2.1), the ba-symbol corresponding to the first node must precede in h that corresponding to the second.

This obviously shows that there cannot be an oriented cycle in the partition graph of p.

(b) (no cycles in $\Pi(p)$-graphs $\Rightarrow$ Π is smv). We want to show that if there are no cycles in the partition graphs for Π of the productions of G, then, for each complete derivation tree of G, there is a computation sequence respecting Π. We want to prove this using induction on the height of derivation trees and, therefore, we have to consider incomplete derivation trees and to define for them a concept analogous to that of computation sequence for complete derivation trees. In order to do this we first observe that a computation sequence h, respecting Π, for a complete derivation tree t can be represented as $h = h_1 \cdots h_{k(Z)}$, where $\Pi(Z) = \langle A_1(Z), \ldots, A_k(Z) \rangle$ and h_i is the ith visit-trace of h for the whole tree t, $i \in [1, k(Z)]$. Second, we observe that in Definition 2.1 of computation sequence, the fact that only complete derivation trees are considered, is of importance only in the start–end and in the feasibility condition. If we extend the definition of computation sequence to incomplete derivation trees,

then, clearly, the start–end condition must be changed allowing any nonterminal, and not only Z, as root of the tree. Again, since the root of an incomplete derivation tree t' may be any nonterminal F, it may possess i-attributes which, because we assume AG to be in Bochmann normal form, would depend on attributes outside t'. Despite this fact, the feasibility condition of Definition 2.1 can still be used for incomplete derivation trees as it is. The only thing to notice is that for those ba-symbols which contain i-attributes of the root of t' the feasibility condition is vacuous. Immediately after Definition 2.1 we observed that the feasibility condition guarantees that the attribute evaluation strategy, represented by a computation sequence, can actually be executed. In the same way, the feasibility condition of a computation sequence for an incomplete derivation tree t' guarantees that the attribute evaluation strategy, represented by it, can actually be executed, once values for the i-attributes of the root are available.

It is easy to see that also in Definition 2.2, of computation sequence respecting Π, the fact that only complete derivation trees are considered is of no importance.

For these reasons we extend the concept of computation sequence respecting Π to incomplete derivation trees still using for it Definition 2.1 (with the change of the start–end condition mentioned above) and Definition 2.2 (see also the notion of an inside (n) computation sequence, Definition 4.2 of [14]).

In what follows a computation sequence respecting Π for a possibly incomplete derivation tree t' will be called a *visit-trace tuple* for t' and will be indicated, inserting, for notational convenience, markers $\#$ to separate different visit-traces, by $v(t') = v_1(t') \# v_2(t') \# \cdots \# v_k(t')$, where $k = k(F)$, F is the root of t' and $v_i(t')$ is the ith visit-trace for t' of the computation sequence. Notice that each $v_i(t')$ starts and ends with the ba-symbols $(F, IA_i(F))$ and $(F, SA_i(F))$, respectively, $i \in [1, k(F)]$.

Using induction we shall prove this part of the theorem by showing that for any derivation tree t of G there exists a visit-trace tuple $v(t)$. This clearly proves what we want because complete derivation trees and computation sequences respecting Π are special cases of derivation trees and visit-trace tuples, respectively.

The proof is divided into two parts. First we construct for each production p of G a sequence O_p of the nodes of the partition graph for p with some markers inserted, called *visit-sequence* for p, and then we use these visit-sequences in the actual induction proof to construct visit-trace tuples. The construction of the visit-sequence O_p is analogous to the corresponding construction in [10].

Construction of the Visit-Sequence O_p for p

Let production p of G be as usual $F_0 \to w_0 F_1 w_1 \cdots w_{n_p-1} F_{n_p} w_{n_p}$. From the way the partition graph of p ($\Pi(p)$-graph) is defined and the fact that it is acyclic, we can construct a visit sequence $O_p = u_1 \# u_2 \# \cdots \# u_k$ by linearly ordering the nodes of the partition graph of p in such a way that the following conditions hold:

(1) If in the $\Pi(p)$-graph there is an edge from n to n', then in O_p n precedes n'.

(2) The first symbol of O_p is $IA_1(F_0)$ and the last is $SA_k(F_0)$, where $k = k(F_0) \geqslant 1$.

(3) For all $j \in [1, n_p]$ and $i \in [1, k(F_j)]$, the string $IA_i(F_j)\, SA_i(F_j)$ is in O_p.

(4) For all $i \in [1, k(F_0) - 1]$, the string $SA_i(F_0) \# IA_{i+1}(F_0)$ is in O_p.

All these conditions on O_p are possible for the following reasons, respectively:

(1) the $\Pi(p)$-graph is acyclic,

(2) in the $\Pi(p)$-graph, node $IA_1(F_0)$ has no incoming edges,

(3) in the $\Pi(p)$-graph, there is only one edge entering any node $SA_i(F_j)$ and this edge comes from node $IA_i(F_j)$, $j \in [1, n_p]$, and $i \in [1, k(F_j)]$,

(4) in the $\Pi(p)$-graph there is only one edge entering any node $IA_{i+1}(F_0)$ and it comes from node $SA_i(F_0)$, $i \in [1, k(F_0) - 1]$.

From this it should be clear that O_p is actually a string of the form $u_1 \# u_2 \# \cdots \# u_k$, where each u_i starts and ends with $IA_i(F_0)$ and $SA_i(F_0)$, respectively, and internally, it consists of couples $IA_m(F_j)\, SA_m(F_j)$, where $i \in [1, k(F_0)]$, $j \in [1, n_p]$, and $m \in [1, k(F_j)]$. Notice that, in general, for a given production p, there may be many possible visit-sequences; we just choose one of them. We can view O_p as representing a visit-trace tuple for *any derivation tree* t with top production p. In fact, in the first place, the initial and final symbols of each u_i clearly correspond to the initial and final ba-symbols of the ith component (visit-trace) of any visit-trace tuple for t. In the second place, an internal couple such as $IA_m(F_j)\, SA_m(F_j)$ (of u_i) indicates that the mth component of a visit-trace tuple for the subtree rooted in F_j must be substituted for it in O_p. So, what we shall do in the induction proof, which follows, is to construct a visit-trace tuple for a derivation tree with top production p by substituting components of the visit-trace tuples for the subtrees rooted in the F_j's, in the appropriate positions of O_p, $j \in [1, n_p]$. This process should be easy to understand from our previous observation that a visit to a derivation tree t, with root n, is nothing else than entering n, visiting its sons in some order and exiting n.

Let us now do the induction proof.

HYPOTHESIS. *For any derivation tree t in G there is a visit-trace tuple, denoted $v(t)$, for t.*

Base. Derivation trees of height 1. Consider a derivation tree t which consists of a terminal production $p: F \to w$ of G. For this production the visit-sequence O_p is particularly simple, in fact, each substring u_i of O_p, $i \in [1, k(F)]$ is just the string $IA_i(F)\, SA_i(F)$. Therefore, from each u_i we construct immediately the ith visit-trace $v_i(t)$ of $v(t)$ as follows: $v_i(t) = (F, IA_i(F))(F, SA_i(F))$, and then $v(t) = v_1(t) \# \cdots \# v_k(t)$. From its form it is easy to see that $v(t)$ satisfies the start–end, the sequentiality and the completeness condition. The fact that Π is respected is also obvious. Finally, point (1) of the construction of O_p guarantees that the feasibility is also respected. Hence $v(t)$ is a visit-trace tuple for t.

Step. Assume that the induction hypothesis holds for all derivation trees of G of height $n - 1$. Consider a derivation tree t of height n. Let the top production of t be

$p: F_0 \to w_0 F_1 w_1 \cdots w_{n_p-1} F_{n_p} w_{n_p}$, and let t_j be the subtree rooted in F_j, $j \in [1, n_p]$. From the induction hypothesis we know that for each t_j there is a visit-trace tuple $v(t_j) = v_1(t_j) \# \cdots \# v_k(t_j)$, $k = k(F_j)$. Let O_p be $u_1 \# u_2 \# \cdots \# u_{k(F_0)}$; $v(t)$ is constructed from O_p by performing in each u_i, $i \in [1, k(F_0)]$, the following substitutions:

> for the initial and final symbols of u_i, $IA_i(F_0)$ and $SA_i(F_0)$, substitute the corresponding ba-symbols $(F_0, IA_i(F_0))$ and $(F_0, SA_i(F_0))$, respectively;
>
> for each pair $IA_m(F_j)\, SA_m(F_j)$ present in u_i, substitute the mth component, $v_m(t_j)$, of $v(t_j)$, where $j \in [1, n_p]$ and $m \in [1, k(F_j)]$.

In order to prove that the $v(t)$, constructed in this way, is actually a visit-trace tuple for t, we go through the required conditions and show that $v(t)$ satisfies each of them, but before doing this, it is useful to make the following observations.

OBSERVATION 1. A component $v_i(t)$ of $v(t)$, $i \in [1, k(F_0)]$, looks as follows:

$$(F_0, IA_i(F_0)) \cdots \underbrace{(F_j, IA_m(F_j)) \cdots (F_j, SA_m(F_j))}_{m\text{th visit-trace for } t_j} \cdots (F_0, SA_i(F_0))$$

$j \in [1, n_p]$ and $m \in [1, k(F_j)]$.

OBSERVATION 2. By point (1) of the construction of O_p and the way $v(t)$ is constructed from O_p, the following is true: for $j \in [1, n_p]$, if m and $n \in [1, k(F_j)]$, and $m < n$, then, due to the edges of the partition graph of F_j, in O_p the couple $IA_m(F_j)\, SA_m(F_j)$ precedes the couple $IA_n(F_j)\, SA_n(F_j)$ and, therefore, in $v(t)$ the mth visit-trace of $v(t_j)$, $v_m(t_j)$, precedes the nth one, $v_n(t_j)$.

Finally we show that $v(t)$ meets the required conditions. The fact that $v(t)$ satisfies the start–end condition is clear from Observation 1. Also from Observation 1, we know that a component $v_i(t)$ of $v(t)$, $i \in [1, k(F_0)]$, represents the basic actions of entering F_0, visiting the subtrees rooted in some of the sons of F_0, and finally exiting F_0. This fact plus the induction hypothesis implies that $v(t)$ satisfies the sequentiality condition.

In order to prove that the feasibility condition is satisfied, it is sufficient to prove the following two points:

(a) Every ba-symbol $(F_j, IA_m(F_j))$—beginning of the mth visit-trace for t_j contained in $v(t)$—satisfies the feasibility condition, $j \in [1, n_p]$ and $m \in [1, k(F_j)]$. It is easy to see that, if this is true, the induction hypothesis and Observation 2 guarantee that also the remainder of the mth visit-trace for t_j satisfies feasibility in $v(t)$.

(b) Every ba-symbol $(F_0, SA_i(F_0))$ of $v(t)$—end of the ith visit-trace $v_i(t)$ of $v(t)$—satisfies the feasibility condition, $i \in [1, k(F_0)]$.

Clearly, if both points hold, then $v(t)$ is feasible because they guarantee that all the ba-symbols of $v(t)$ are feasible, apart from the ba-symbol $(F_0, IA_i(F_0))$ of each $v_i(t)$, $i \in [1, k(F_0)]$, for which feasibility is vacuous (cf. earlier remark). For understanding that both, points (a) and (b), are satisfied, it is sufficient to recall point (1) of the construction of O_p: it shows that all the dependency relations among the nodes of the partition graph for p are respected in O_p. This immediately implies that points (a) and (b) are true. Finally we note that the fact that Π is respected by $v(t)$ can easily be shown from Observations 1 and 2 and the induction hypothesis (and this also implies the completeness condition).

This ends our induction proof and thereby the proof of the theorem. ∎

Using Theorem 2.1 it is easy to see that the set of partitions Π we have chosen in Example 2.1 of the AG G is not an smv-set of partitions: the partition graph of Production 1 of G, represented in Fig. 3, contains an oriented cycle (concerning the nodes $i_1(A)$, $s_1(A)$, $i_2(B)$, $s_2(B)$). However, G is an smv-AG. In what follows we shall give an smv-set of partitions for G with the corresponding (acyclic) partition graphs.

EXAMPLE 2.2. Consider the AG G of Example 2.1. Represented in Fig. 4 are the partition graphs of the four productions of G for the set of partitions Π defined as

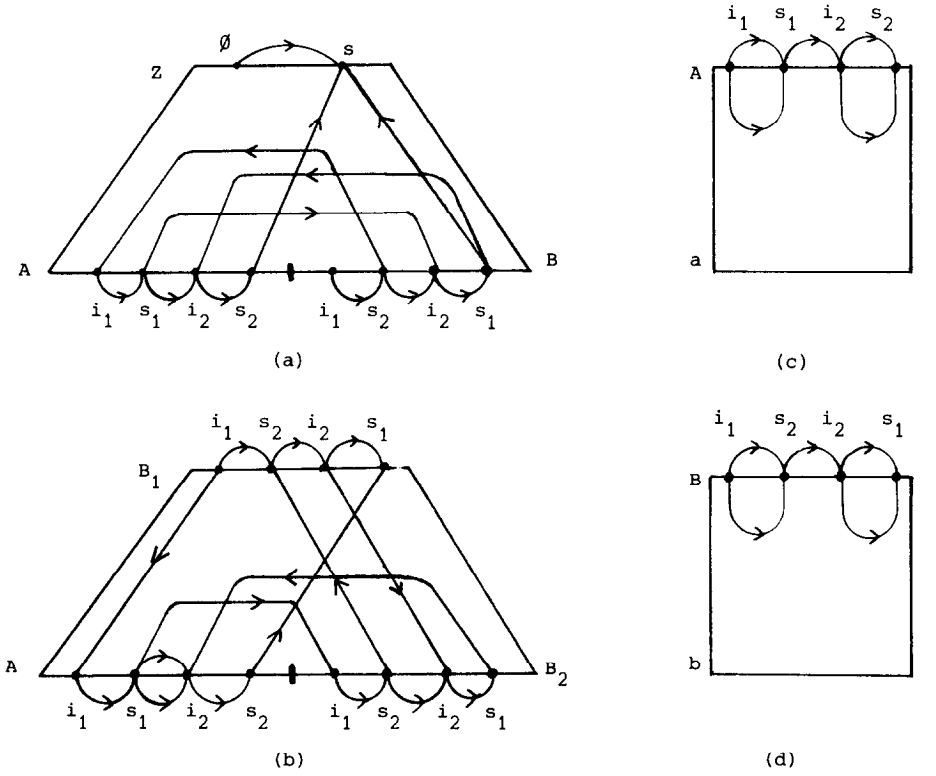

FIG. 4. The production graphs of G for the set of partitions Π of Example 2.2.

follows: $\Pi(Z) = \langle\{s\}\rangle$; $\Pi(A) = \langle\{i_1, s_1\}, \{i_2, s_2\}\rangle$; $\Pi(B) = \langle\{i_1, s_2\}, \{i_2, s_1\}\rangle$. The fact that all these partition graphs are acyclic shows (by the preceding theorem) that G is an smv-AG (precisely, simple 2-visit). ∎

The fact that testing whether a given set of partitions Π is smv is based on the noncircularity of the $\Pi(p)$-graphs (Theorem 2.1), has an important consequence which allows us to restrict our attention to sets of partitions, which we call *reduced*, in which none of the partitions contains an empty set (cf. reduced computation sequence of [14]).

LEMMA 2.1. *An* AG G *is* smv *iff there exists a reduced* smv*-set of partitions for* G.

Proof. We shall prove that if G has an smv-set of partitions, then it also has a reduced smv-set of partitions. Assume Π is an smv-set of partitions for G and that it is not reduced, i.e., for some nonterminal F of G, $A_i(F) \in \Pi(F)$ is empty for some $i \in [1, k(F)]$. We want to show that the set of partitions Π', obtained from Π by eliminating $A_i(F)$, i.e., $\Pi'(F) = \langle A_1(F),..., A_{i-1}(F), A_{i+1}(F),..., A_k(F)\rangle$, and $\Pi'(F') = \Pi(F')$ for all the other nonterminals F' of G, is an smv-set of partitions for G. In order to do this it is sufficient to construct the $\Pi(p)$-graph of any production p of G which contains F. In this graph the situation of the two nodes $IA_i(F)$ and $SA_i(F)$, corresponding to $A_i(F)$ is represented in Fig. 5. It is important to notice that because the two nodes do not represent any attribute of F, those of Fig. 5 are the only edges entering and exiting them. From this, it is clear that the two nodes can be eliminated from the $\Pi(p)$-graph and substituted with a direct connection from the preceding node (if any) $SA_{i-1}(F)$ to the successive one (if any) $IA_{i+1}(F)$, without creating any cycles in the obtained graph. By Theorem 2.1, this proves that if Π is smv, then the set of partitions Π' obtained from Π eliminating $A_i(F)$, is also smv. Clearly, by repeated applications of this process, one obtains a reduced smv-set of partitions for G. ∎

We now use the preceding results in order to show that the problem whether a given AG is smv (the smv-problem) can be answered in nondeterministic polynomial time (in the size of the AG).

COROLLARY 2.1. The smv-problem is in *NP*.

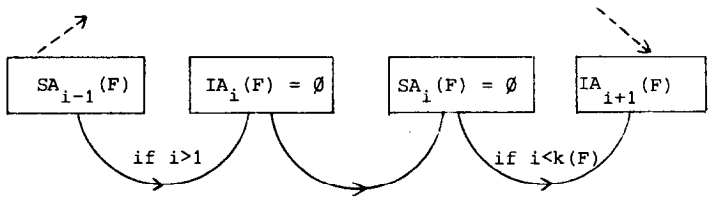

FIG. 5. Situation of two nodes, corresponding to empty sets, in the partition graph of p.

Proof. We know that an AG G is smv iff there exists an smv-set of partitions for it. The nondeterministic algorithm for deciding whether G is smv is as follows:

(i) Guess a reduced set Π of partitions for G. By Lemma 2.1 we can restrict our attention to such sets of partitions and it is clear that one of them can be guessed in polynomial time.

(ii) Test whether Π is smv. By Theorem 2.1 we know how to perform this test and it is easy to see that it takes only polynomial time in the size of G. ∎

In the rest of the section we do three things. First, we exploit the technique used in the induction proof of part (b) of Theorem 2.1 in order to describe an attribute evaluation algorithm for smv-AG. Second, we refine the result of Lemma 2.1 showing that an AG G is smv iff there is an smv-set of partitions for G such that in each partition $\Pi(F) = \langle A_1(F),..., A_k(F)\rangle$ of Π, when F is a nonterminal of G and $k(F) \geqslant 2$, $A_1(F)$ contains at least one s-attribute, $A_k(F)$ at least one i-attribute and for every $j \in [2, k(F)-1]$, $A_j(F)$ contains both i- and s-attributes. We shall call such a Π a *good set of partitions*. Finally, we show that every noncircular AG (pure multi-visit) can be transformed into an smv-AG without changing its semantic rules and the translation it defines.

In part (b) of the proof of Theorem 2.1 we constructed, given an AG G and a set Π of partitions for G, for each production p of G a visit-sequence O_p which we used for constructing a visit-trace tuple respecting Π *for any derivation tree* with top production p. Here, we exploit these O_p's in order to construct an attribute evaluation algorithm for G (cf. [10]) such that its computation on any complete derivation tree t of G is described by a computation sequence for t which respects Π. We call this algorithm the smv-algorithm.

Smv-Algorithm

Let G be an smv-AG and Π an smv-set of partitions for G. The smv-algorithm consists of a routine, VISIT TREE, which has two arguments: the node which is the root of the subtree to be visited and the number of the visit to be performed (bounded by $\max(\Pi)$). In order to describe the routine in an understandable way we first fix the following notation:

(i) production p of G, is, as usual, of the form $F_0 \to w_0 F_1 w_1 \cdots w_{n_{p-1}} F_{n_p} w_{n_p}$,

(ii) the visit-sequence O_p for p is $O_p = u_1 \# u_2 \# \cdots \# u_{k(F_0)}$, where u_i, $i \in [1, k(F_0)]$, is as follows:

$$IA_i(F_0)\, IA_{m_1}(F_{j_1})\, SA_{m_1}(F_{j_1})\, IA_{m_2}(F_{j_2})\, SA_{m_2}(F_{j_2}) \cdots IA_{m_r}(F_{j_r})\, SA_{m_r}(F_{j_r})\, SA_i(F_0),$$

where $r \geqslant 0$ and, for $s \in [1, r]$, $j_s \in [1, n_p]$ and $m_s \in [1, k(F_{j_s})]$.

The fact that u_i is as above has the important consequence that the smv-algorithm, when visiting any derivation tree with top production p for the ith time, $i \in [1, k(F_0)]$, always performs the following sequence of actions:

enter F_0 computing $IA_i(F_0)$,
pay r visits to the subtrees rooted in $F_{j_1}, \ldots, F_{j_r}$, (*)
exit F_0 computing $SA_i(F_0)$.

Let us see, then, how the routine VISIT TREE performs these actions.

procedure VISIT TREE (F_0, i); **node** F_0; **integer** i;
⟨we assume that production p is applied to node F_0; therefore, because the visit number is i, we shall use the component u_i of O_p of point (ii)⟩
begin
compute all attributes in $IA_i(F_0)$; ⟨ba-symbol: $(F_0, IA_i(F_0))$⟩
for $k = 1$ **to** r **do**
VISIT TREE (F_{j_k}, m_k)
od;
compute all attributes in $SA_i(F_0)$ ⟨ba-symbol: $(F_0, SA_i(F_0))$⟩
end.

Attribute evaluation of a complete derivation tree t is then obtained by means of the following program:

for $j = 1$ **to** $k(Z)$ **do** VISIT TREE (Z, j) **od**, where Z is the root of t.

From what we have said in Theorem 2.1(b) it should be easy for our readers to convince themselves that the smv-algorithm actually performs attribute evaluation for any complete derivation tree of G respecting the given smv-set of partitions Π.

It should be clear that this routine VISIT TREE depends on two things: if node n is the first parameter, it depends on the number of the visit to be paid to the subtree rooted in n (2nd parameter) and the production used to expand n. In order to write a routine which would take care of all possibilities two case statements would be sufficient. The outer case statement would take care of which production is applied to n and the inner one would take care of which visit must be performed and would, therefore, consist of pieces of code like the one we have given above. A complete smv-algorithm for the AG of Examples 2.1 and 2.2 will be given after some further remarks on the algorithm in general.

It is interesting to note that remark (*), we made above, implies that smv-AG actually satisfy an apparently more restricted definition than the one we gave (cf. the introduction, point (iii)). In fact, in Definition 2.3 we required only the existence of a computation sequence respecting Π for every complete derivation tree, allowing, therefore, the freedom of choosing, for a given visit number $i \in [1, k(F_0)]$, different sequences of visits to the subtrees rooted in the F_j's, $j \in [1, n_p]$, for different occurrences of production p. Thus, the smv-algorithm shows that the freedom allowed in Definition 2.3 does not actually add any power. An analogous situation can be found in [4], where a *dynamic* definition of one-visit AG (similar to Definition 2.3) was proved to be equivalent to an apparently more restricted one (called *static* there) and which implied the existence of an attribute evaluation algorithm for one-visit AG

of which the smv-algorithm can be viewed as an extension (as smv-AG are an extension of one-visit AG). We finally note that, from an smv-set of partitions Π for a given AG G an smv-algorithm can be constructed for G in deterministic polynomial time: construct the visit-sequences O_p from the partition graphs of the productions of G. Such a process is illustrated in

EXAMPLE 2.3. Consider again the AG G of Example 2.1. We construct an smv-algorithm for G using the smv-set of partitions Π for it, given in Example 2.2, which is as follows: $\Pi(Z) = \langle \{s\} \rangle$, $\Pi(A) = \langle \{i_1, s_1\}, \{i_2, s_2\} \rangle$, $\Pi(B) = \langle \{i_1, s_2\}, \{i_2, s_1\} \rangle$. First we must find for each production of G a corresponding visit-sequence. It is easy to see (looking at the partition graphs of the productions of G represented in Fig. 4) that the following visit-sequences are correct:

$$O_1 = \varnothing \underline{i_1(B)\, s_2(B)}\, \underline{i_1(A)\, s_1(A)}\, \underline{i_2(B)\, s_1(B)}\, \underline{i_2(A)\, s_2(A)}\, s(Z).$$

$$O_2 = i_1(B_1)\, \underline{i_1(A)\, s_1(A)}\, \underline{i_1(B_2)\, s_2(B_2)}\, s_2(B_1)$$

$$\# \, i_2(B_1)\, \underline{i_2(B_2)\, s_1(B_2)}\, \underline{i_2(A)\, s_2(A)}\, s_1(B_1).$$

$$O_3 = i_1(A)\, s_1(A) \# i_2(A)\, s_2(A).$$

$$O_4 = i_1(B)\, s_2(B) \# i_2(B)\, s_1(B).$$

The routine VISIT TREE for these visit-sequences is as follows:

```
procedure VISIT TREE (n, i); node n; integer i;
case production applied to n of
    prod. 1: skip;
             VISIT TREE (B, 1); VISIT TREE (A, 1); VISIT TREE (B, 2);
             VISIT TREE (A, 2);
             compute s(Z);

    prod. 2: case i of
             1: compute i1(B1);
                VISIT TREE (A, 1); VISIT TREE (B2, 1);
                compute s2(B1);
             2: compute i2(B1);
                VISIT TREE (B2, 2); VISIT TREE (A, 2);
                compute s1(B1);
             esac;

    prod. 3: case i of
             1: compute i1(A); compute s1(A);
             2: compute i2(A); compute s2(A);
             esac;
```

```
        prod. 4: case i of
                   1: compute i_1(B); compute s_2(B);
                   2: compute i_2(B); compute s_1(B);
                   esac;
    esac.
```

The main program is simply VISIT TREE $(Z, 1)$. ∎

We turn now to improving the result of Lemma 2.1 as follows:

THEOREM 2.2. *An* AG G *is* smv *iff there is a good* smv*-set of partitions for* G.

Proof. We shall prove that, given a reduced smv-set of partitions Π for G, we can construct a good one which is also smv. First we need some notation. Consider a production p of G and its corresponding partition graph. We distinguish two types of edges in this graph: edges belonging to the partition graph of some nonterminal of p, which we call precedence edges, and edges which represent actual dependencies among attributes in p, which we call dependency edges. In the same way we call a dependency path a path, in the partition graph of p, which contains at least one dependency edge.

Assume that Π is not a good set of partitions. We distinguish two cases.

Assume first that for some nonterminal F of G, $\Pi(F)$ has an element $A_j(F)$ which contains only s-attributes, for $j \in [2, k(F)]$. Consider, then, any production p of G in which F occurs and construct the partition graph for it, corresponding to Π. In Fig. 6 we represent the situation of the two nodes corresponding to $A_j(F)$ in the partition graph for p. It is easy to see that eliminating the empty node $IA_j(F)$ and merging the two nodes $SA_{j-1}(F)$ and $SA_j(F)$, would give rise to a cycle in the obtained graph only if there were, in the partition graph of p, a dependency path from $SA_{j-1}(F)$ to $SA_j(F)$. To see that such a path does not exist, observe that it would have to pass through the empty node $IA_j(F)$ (by the assumption that G is in Bochmann normal form), but no dependency edge enters this node because it has no attributes. Therefore, the graph obtained by merging the two nodes is still acyclic. By Theorem 2.1, this proves that the set of partitions obtained from Π by eliminating from $\Pi(F)$ the set $A_j(F)$ and changing $A_{j-1}(F)$ into $A_{j-1}(F) \cup A_j(F)$ is still an smv-set of partitions for G.

The second case, in which $A_j(F)$ has only i-attributes, $j \in [1, k(F) - 1]$, is handled in the same way. Therefore, we say only that the transformation on Π for eliminating

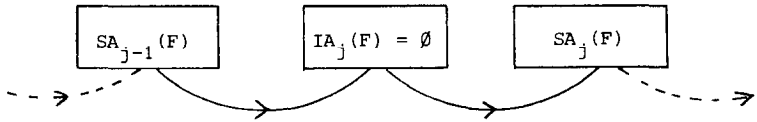

FIG. 6. Situation of two nodes, one of which corresponds to the empty set, in the partition graph of p.

$A_j(F)$ would be in this case: delete $A_j(F)$ and change $A_{j+1}(F)$ into $A_{j+1}(F) \cup A_j(F)$. It is clear that, applying a finite number of such transformations to the reduced smv-set of partitions Π, we shall end up with a good smv-set of partitions for G. ∎

As we already said in this section, an AG is simple m-visit iff there is an smv-set of partitions Π for G such that $\max(\Pi) \leqslant m$. The result of Theorem 2.2 can be used to determine the least m such that any AG G is simple multi-visit iff it is simple m-visit, i.e., the least upper bound on the number of visits needed by an smv-AG. It is easy to see that in any good smv-set of partitions Π for G the partition $\Pi(F)$ can have at most $\min(\#I(F), \#S(F)) + 1$ elements. Thus, G can be at most simple m-visit, where $m = \max_F(\min(\#I(F), \#S(F)) + 1)$. Note that there are smv-AG which actually need this number of visits.

In what follows we shall show that any noncircular AG G can be transformed into an smv-AG G' which defines the same string-to-value translation as G and has the same semantic rules as G (apart from different names of the attributes occurring in the rules). This means that the transformation is independent of the semantics (i.e., it is *schematic* as in the theory of program schemes). The string-to-value translation defined by an AG is the mapping which maps each string which is the yield of a complete derivation tree of G to the corresponding value of the designated s-attribute of its root (cf. Definition 1.1).

The reasoning behind the construction is based on the following facts: In [14] it has been proved that every noncircular AG G is pure k-visit for some k which is $0 < k \leqslant \max_F(\#A(F))$, i.e., for every complete derivation tree t of G there is a k-visit computation sequence for t. The fact that a k-visit computation sequence for a complete derivation tree t determines, for the attributes of each node of t, a partition of at most k elements (i.e., the one mentioned in the completeness condition of Definition 2.1), and the fact that the number of such partitions, for any nonterminal F of G, is obviously finite, shows that the difference between simple and pure multi-visit AG can be viewed as follows: in the first, to each nonterminal is associated a unique partition of its attributes which must always be respected, whereas in the second, a finite set of possible partitions is associated to each nonterminal.

It is also important to notice that if we consider the partitions of the attributes of every node of a complete derivation tree t, determined by a computation sequence h for t, and, using these partitions, we construct for any production p of t the corresponding partition graph, then this graph is acyclic. The intuitive reason for this is that h respects these partitions (cf. Definition 2.2). Formally, a proof of this can be given which is entirely analogous to part (a) of the proof of Theorem 2.1.

The construction which, given a noncircular AG G, builds an smv-AG G' which defines the same string-to-value translation of G and uses the same semantic rules, is as follows:

(1) For each nonterminal F of G, G' has as many nonterminals of the form $(F, \Pi(F))$ as there are partitions $\Pi(F)$ of the attributes of F with at most $\#A(F)$ elements. Each nonterminal $(F, \Pi(F))$ is given, in G', the attribute set $A(F)$.

(2) For each production p of G of the form $F_0 \to w_0 F_1 w_1 \cdots w_{n_p-1} F_{n_p} w_{n_p}$, G' has as many productions p': $(F_0, \Pi(F_0)) \to w_0(F_1, \Pi(F_1))\, w_1 \cdots w_{n_p-1}(F_{n_p}, \Pi(F_{n_p}))\, w_{n_p}$, as there are combinations of the partitions $\Pi(F_i)$, $i \in [0, n_p]$, such that the following condition holds: the partition graph of p, constructed using the partitions $\Pi(F_i)$, is acyclic. Each such production p' of G' is given the same set of semantic rules as that of p in G, where, of course, the names of the nonterminals are appropriately changed. This ends the construction.

It is easy to see that the AG G', produced in this way, is smv: the set of partitions for G' defined by taking for each nonterminal $(F, \Pi(F))$ the partition $\Pi(F)$ is an smv-set of partitions (immediate from Theorem 2.1 and point (2) of the construction of G').

To prove that G' defines the same string-to-value translation as G, we say that t and t' are *corresponding* derivation trees (of G and G', respectively) if t is obtained from t' by replacing each nonterminal $(F, \Pi(F))$ by nonterminal F. Clearly, every derivation tree t' of G has a corresponding derivation tree t in G (by the construction of G'). Also, vice versa, every derivation tree t of G has a corresponding one in G' for every k-visit computation sequence h for t, where $0 < k \leqslant \max_F(\#A(F))$. This can be shown as follows: Let t' be the result of replacing the label of every node of t, assume it to be F, by $(F, \Pi(F))$, where $\Pi(F)$ is the partition of the attributes of node F determined by the computation sequence h (for t). As we observed before all the partition graphs of the productions in t, constructed by using the partitions determined for each node of t by h, are acyclic. Therefore by point (2) of the construction, t' is a derivation tree of G'.

Since corresponding (complete) derivation trees translate the same string into the same value (because the semantic rules of G and G' are the same), this shows that G and G' define the same translation. This proves

THEOREM 2.3. *For every noncircular* AG G *there exists a simple multi-visit* AG G' *such that* G' *defines the same string-to-value translation as* G *and uses the same semantic rules as* G *(apart from attribute names).* ∎

3. THE l-ORDERED PROPERTY

As we mentioned in the introduction, the class of smv-AG is equal to the class of l-ordered AG [9, 10], where an AG G is l-ordered if, for each nonterminal F of G, there exists a linear order of its attributes such that, for every occurrence of F in any derivation tree of G, its attributes can be evaluated in that order.

In this section we give a formal definition of l-ordered AG and prove their equivalence with smv-AG. (For dtg(t), recall Section 1.)

DEFINITION 3.1. An AG G is *l-ordered* iff there exists a set O_G containing, for each nonterminal F of G, a linear order O_F of the attributes of F, such that, for every

complete derivation tree t of G, there exists a linear order O_t of the nodes of $\mathrm{dtg}(t)$ such that the following two conditions hold:

(i) if a and b are two attributes of a production p of G, which occurs in t, and b depends on a in p, then $a < b$ in O_t (feasibility);

(ii) if a and b are attributes of nonterminal F, which occurs in t, and $a < b$ in O_F, then $a < b$ also in O_t (O_G is respected). ∎

In what follows we call a set O_G which satisfies Definition 3.1 a *correct* set of l-orders for G.

In order to intuitively see why the class of l-ordered and smv-AG coincide we observe that it is possible to define smv-AG by means of an apparently less restrictive concept than that of computation sequence respecting Π for every complete derivation tree. This concept is that of an (ordered) partition, respecting Π, of the attributes of a complete derivation tree and is defined as follows: Given an AG G, a set Π of partitions for G and a complete derivation tree t of G, if we view the attributes of each node F of t as partitioned in the subsets $IA_1(F)$, $SA_1(F)$,..., $IA_k(F)$, $SA_k(F)$ as specified by $\Pi(F)$ of Π, then a partition $\Pi(t)$ of all the attributes of t is a linear order of the subsets of attributes of all the nodes of t such that the following conditions hold:

(i) $\Pi(t)$ is feasible (as in Definition 2.1),

(ii) for each node F of t, if i and $j \in [1, k(F)]$ and $i < j$, then in $\Pi(t)$, $IA_i(F) < SA_i(F) < IA_j(F) < SA_j(F)$.

Using this concept, we define the class of smv-AG as follows: an AG G is smv iff there exists a set of partitions Π for G such that for each complete derivation tree t of G there is a $\Pi(t)$ respecting Π.

The fact that even though this definition seems more free than the one we have given, the class of AG is still the same can be understood by means of the following two observations:

(1) The concept of partition of the attributes of a derivation tree corresponds to that of computation sequence in which the sequentiality and the start–end conditions have been dropped, or, in other words, a computation sequence in which jumping from a node to another in the derivation tree is allowed.

(2) From the fact that any jump can be simulated by sequentially walking through the tree, it is clear that from any partition of the attributes of a derivation tree t which respects Π we can construct a computation sequence h for t such that the partition of the attributes of any node F of t, defined by h, contains all the elements of $\Pi(F)$ (in the same order) plus some empty sets. Thus, using the same technique as in Lemma 2.1, these empty sets can be eliminated obtaining a computation sequence for t which respects Π.

Observe now that this definition of smv-AG, we have just given, is very close to Definition 3.1 of l-ordered AG: the partition $\Pi(t)$ of the attributes of t is an extension of the linear order O_t of the nodes of $\mathrm{dtg}(t)$ we used there.

This argument should be sufficient as an intuitive explanation of the fact that l-ordered and smv-AG are equivalent (which is proved in Theorem 3.1) and it also shows that, although the concept of l-ordered AG is probably easier to grasp and to define formally than that of smv-AG, this last is more strictly connected with the existence of a natural attribute evaluation algorithm (the smv-algorithm) which walks through the derivation tree. Note that, while now we have loosened the definition of smv-AG to get closer to that of l-ordered AG, it would also be possible to further restrict it so as to get even closer to the smv-algorithm (cf. the discussion after the smv-algorithm).

THEOREM 3.1. *An* AG *is l-ordered iff it is simple multi-visit.*

Proof. (a) (smv $\Rightarrow$ l-ordered). This direction is easy. Let G be an smv-AG and Π an smv-set of partitions for it. We want to show that from Π we can construct a correct set of l-orders O_G for G. For each nonterminal F of G, $O_F \in O_G$ is constructed from $\Pi(F) \in \Pi$ as follows:

(1) for each $i \in [1, k(F)]$, fix any linear orders $O(IA_i(F))$ of the i-attributes in $A_i(F)$ and $O(SA_i(F))$ of the s-attributes in $A_i(F)$,

(2) O_F is obtained from these linear orders by *concatenating* them as follows: $O_F = O(IA_1(F))\, O(SA_1(F)) \cdots O(IA_k(F))\, O(SA_k(F))$.

The fact that the set O_G, constructed in this way, is a correct set of l-orders for G can be easily understood. Let t be a complete derivation tree of G and h a computation sequence for t respecting Π. From h we can build a linear order O_t of the nodes of $\mathrm{dtg}(t)$, which satisfies the two conditions of Definition 3.1, simply substituting in h for each ba-symbol $(F, XA_i(F))$ the linear order $O(XA_i(F))$ of point (1), $i \in [1, k(F)]$ and $X \in \{I, S\}$.

(b) (l-ordered $\Rightarrow$ smv). Let G be an l-ordered AG and O_G a correct set of l-orders for G. From O_G we construct a set of partitions Π for G. For each nonterminal F of G, let $O_F \in O_G$ be $a_1 < a_2 < \cdots < a_n$, then $\Pi(F) = \langle \{a_1\}, \{a_2\}, \ldots, \{a_n\} \rangle$ is in Π.

We want to show that this set of partitions Π is smv for G, i.e., by Theorem 2.1, no cycles are contained in the partition graphs of the productions of G, constructed from Π. This is shown by using a very similar argument to that of Theorem 2.1(a).

First we notice that, with such a Π, the partition graphs of the productions of G have nodes corresponding to either one or no attribute. We call nodes of the first type attribute nodes. Consider now a complete derivation tree t of G containing production p and a linear order O_t of the nodes of $\mathrm{dtg}(t)$ which satisfies Definition 3.1. It is easy to see that, if in the partition graph of p there is an oriented path from attribute node $\{a\}$ to attribute node $\{b\}$, then $a < b$ in O_t. Thus, from the obvious fact that any cycle in the partition graph of p contains at least one attribute node, it is clear that no cycle can be present in it and, therefore, Π is an smv-set of partitions for G. ∎

Observe that in Theorem 3.1(b) we could use Theorem 2.2 in order to transform the smv-set of partitions Π into a good set of partitions. For a given l-order O_F: $a_1 < a_2 < \cdots < a_n$, this would produce the partition $\Pi(F)$, obtained by cutting O_F into maximal pieces of i-attributes followed by s-attributes. As an example, if O_F is $i_1 < i_2 < s_1 < s_2 < s_3 < i_3 < s_4 < i_4$, then $\Pi(F) = \langle \{i_1, i_2, s_1, s_2, s_3\}, \{i_3, s_4\}, \{i_4\} \rangle$.

4. *NP*-Completeness

In this section we show that it is *NP*-hard to decide whether an attribute grammar is simple multi-visit (and hence *NP*-complete by Corollary 2.1). To prove this the equivalent concept of l-ordered AG (Section 3) is easier to handle. Before giving the formal proof, let us first try to see intuitively why the l-ordered property is hard to decide (see [9, 10]).

In any AG G some (partial) order between the attributes of a nonterminal is forced by the dependencies as given by the semantic rules of G. If, e.g., there is a path from a to b (attributes of a node) in some derivation tree graph (see Section 1), then a is before b in any correct set of l-orders for G; moreover, this order between a and b may be viewed as an additional dependency and hence force an order between other attributes, and so on. More precisely, cf. [10], one can (recursively) define *a is forced before b* (where $a, b \in A(F)$ for some nonterminal F) if there is a production p and a sequence $a_1, a_2, \ldots, a_k$ of attributes in p such that $k \geqslant 2$, $a_1 = a$, $a_k = b$ and for all i $(1 \leqslant i \leqslant k-1)$ either a_{i+1} depends on a_i in p or a_i is forced before a_{i+1} (or both). It is not difficult to see that the relation *forced before* can be computed in polynomial time (the computation is similar to the computation of i/o graphs in [11], where only forced order between i- and s-attributes is considered). Clearly, if a is forced before b, then a is before b in any correct set of l-orders for G. Hence if a cycle is detected (i.e., b is forced before b, for some attribute b), then the grammar is not l-ordered. In Example 2.1, $i_1 < s_1 < i_2 < s_2$ is forced for A and $i_1 < s_2 < i_2 < s_1$ is forced for B; no cycle exists and precisely one correct set of l-orders is forced. In general, the absence of cycles in the relation *forced before* is unfortunately not sufficient to ensure the existence of a correct set of l-orders; the reason is that not every linear order which is compatible with *forced before* is correct: the attribute dependencies may also influence the possible orderings of *different* nonterminals, i.e., the linear orders associated to different nonterminals are not independent. The easiest example of this situation is given in Fig. 7, where the production graph (Definition 1.2) of a

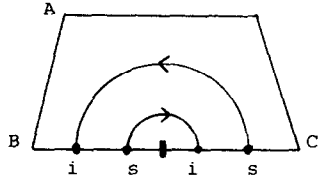

FIG. 7. Cycle with holes.

production $A \to BC$ is shown with the semantic rules $i(B) = s(C)$ and $i(C) = s(B)$; this production graph might be called a *cycle with two holes*. Clearly, the dependencies in Fig. 7 exclude linear orders in which $i(B)$ is before $s(B)$ and simultaneously $i(C)$ is before $s(C)$, i.e., for which *the holes of the cycle are closed*, but allows all others. Thus (assuming that, for B and C, no order is forced between i and s by the other productions) B and C each have two possible orders, but together only three combinations are possible: those which satisfy the Boolean formula *B or C*, where $s(B)$ *is before* $i(B)$ is interpreted as $B = true$ and $i(B)$ *is before* $s(B)$ as $B = false$, and similarly for C. In the proof of Theorem 4.1 we shall show how we can simulate in this way the satisfiability problem for Boolean formulas. For all unexplained notions see, e.g., [6].

THEOREM 4.1. *The following problems are NP-complete*:

(i) *whether an arbitrary* AG *is simple multi-visit* (*i.e., l-ordered*),

(ii) *whether an arbitrary* AG *is simple* 2*-visit.*

Proof. By Corollary 2.1 these problems are in *NP* (note that in the 2-visit case it suffices to guess a set Π of partitions with $\max(\Pi) \leqslant 2$). To prove *NP*-hardness we shall show that the satisfiability problem for Boolean formulas is reducible (in polynomial time) to the l-ordering problem for attribute grammars (and the simple 2-visit problem) by constructing for every Boolean formula F_0 an AG $G(F_0)$ such that F_0 is satisfiable iff $G(F_0)$ is l-ordered iff $G(F_0)$ is simple 2-visit.

Thus, let F_0 be a Boolean formula, built from literals by the operations *and* and *or*, where a literal is either P or *not* P for some Boolean variable P (we may even assume that F_0 is in conjunctive normal form). The AG $G(F_0)$ is constructed as follows: For every Boolean variable P occurring in F_0, $G(F_0)$ has a corresponding nonterminal P with $I(P) = \{i, i_1\}$, $S(P) = \{s, s_1\}$ and a production $P \to \lambda$ (where λ is the empty string) with semantic rules $s_1(P) = i(P)$ and $s(P) = i_1(P)$, i.e., the production graph of Fig. 8a. Furthermore, for every subformula F_j of F_0, $G(F_0)$ has a corresponding nonterminal F_j with $I(F_j) = \{i\}$ and $S(F_j) = \{s\}$, and the following productions: if $F_j = P$ for some Boolean variable P, then $F_j \to P$; if $F_j = not\ P$, then $F_j \to not\ P$; if $F_j = F_k\ and\ F_m$, then $F_j \to F_k\ and\ F_m$; and if $F_j = F_k\ or\ F_m$, then $F_j \to F_k\ or\ F_m$; the production graphs of these productions are given in Figs. 8c–f, respectively (the dotted lines in Figs. 8c,d are only added to help the intuition: they are the dependencies of Fig. 8a). Finally, $G(F_0)$ has the initial nonterminal Z without attributes, and the production $Z \to F_0$ with the semantic rule $i(F_0) = s(F_0)$, see Fig. 8b, where F_0 is, of course, the nonterminal corresponding to the whole formula. This ends the construction of $G(F_0)$. Note that there is exactly one production for each nonterminal, and so $G(F_0)$ has only one complete derivation tree (of which the structure corresponds precisely to the Boolean formula F_0).

As an example, the AG corresponding to $F_0 = (P\ or\ (not\ Q))\ and\ ((not\ P)\ or\ Q)$ has productions $Z \to F_0$, $F_0 \to F_1\ and\ F_2$, $F_1 \to F_3\ or\ F_4$, $F_2 \to F_5\ or\ F_6$, $F_3 \to P$, $F_4 \to$

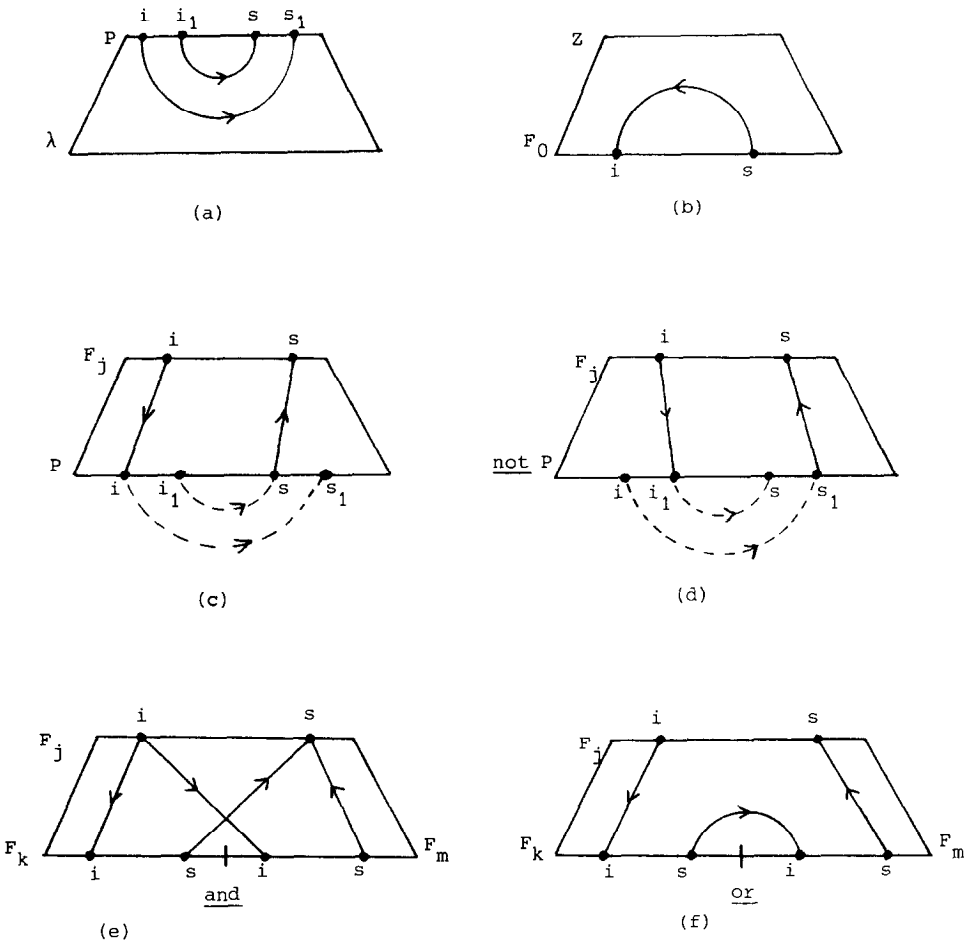

FIG. 8. Production graphs of $G(F_0)$.

not Q, $F_5 \to$ *not* P, $F_6 \to Q$, $P \to \lambda$ and $Q \to \lambda$. The complete derivation tree of $G(F_0)$ is given in Fig. 9 together with its derivation tree graph. This ends the example.

It should be clear that $G(F_0)$ can be constructed from F_0 in deterministic polynomial time. We now have to prove that $G(F_0)$ is l-ordered (simple 2-visit) iff F_0 is satisfiable.

Intuitively, we shall take *false* to correspond to l-orders with $i < s$ and *true* to l-orders with $s < i$ (cf. the example of Fig. 7). The fact that the order has to be correct for all occurrences of the same nonterminal ensures that all occurrences of the same Boolean variable will receive the same truth value. Note that Fig. 9 may be viewed as a combination of two *cycles with holes*, i.e., two times Fig. 7, one cycle corresponding to F_1 and one to F_2, with the edge of $Z \to F_0$ in common; if P is false and Q is true then the F_1-cycle will be closed, and similarly, if P is true and Q is false, the F_2-cycle;

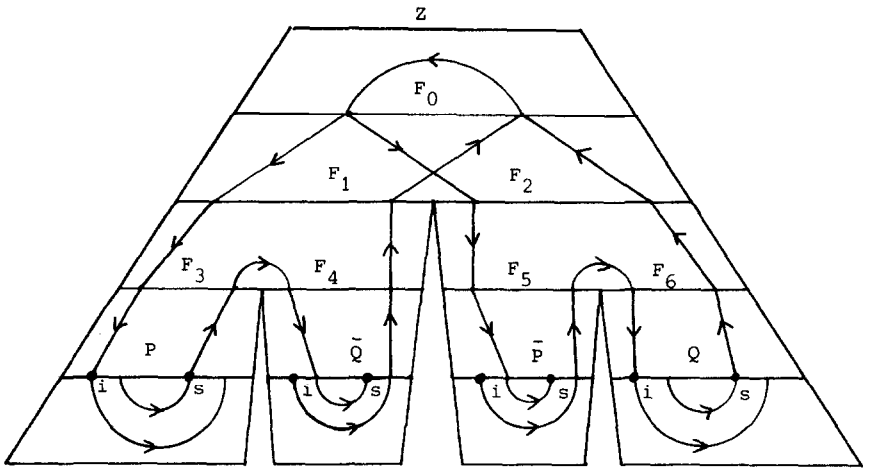

FIG. 9. Derivation tree graph for (*P or* (*not Q*)) *and* ((*not P*) *or Q*).

thus only l-orders for which P and Q have the same truth value will be allowed. In general, if F_0 is in conjunctive normal form, then each *or*-clause corresponds to such a cycle with holes (called *indirect cycle* in [9]).

We first show that if $G(F_0)$ is l-ordered, then F_0 is satisfiable. To do this we need some terminology and a few facts. For a truth assignment α of the Boolean variables and for a Boolean formula F, we denote by $\alpha(F)$ the truth value of F under the assignment α. For a derivation tree graph g of $G(F_0)$ and a truth assignment α, we define $\alpha(g)$ to be the graph obtained from g by adding edges as follows: for every Boolean variable P, if $\alpha(P) = \text{true}$, then an edge from $s(P)$ to $i(P)$ is added for each occurrence of P in g, and similarly an edge from $i(P)$ to $s(P)$ is added in case $\alpha(P) = \text{false}$. The following two statements now hold for any truth assignment α:

(1) For every subformula F_j of F_0, $\alpha(F_j) = \text{false}$ iff there is a path from $i(F_j)$ to $s(F_j)$ in $\alpha(g_j)$, where g_j is the derivation tree graph of the (unique) derivation tree with root F_j.

(2) $\alpha(F_0) = \text{false}$ iff there is a cycle in $\alpha(g)$, where g is the derivation tree graph of the (unique) complete derivation tree of $G(F_0)$.

Statement (1) can easily be shown by induction on the structure of F_j using the production graphs of Fig. 8c–f. Note in particular that, in Fig. 8c, if $i(P)$ is connected to $s(P)$, then so is $i(F_j)$ to $s(F_j)$, whereas in Fig. 8d, if $s(P)$ is connected to $i(P)$, then so is $i(F_j)$ to $s(F_j)$, using the dotted lines; in the other cases, there is no path from $i(F_j)$ to $s(F_j)$. From statement (1), statement (2) follows because every cycle in $\alpha(g)$ includes the edge in the production graph of $Z \to F_0$ (Fig. 8b).

Assume now that $G(F_0)$ is l-ordered. Let α be the truth assignment such that $\alpha(P) = \text{true}$ iff $s(P)$ is before $i(P)$ in the linear order for P. By Definition 3.1 there exists an order O_t of the nodes of g (where t is the complete derivation tree) which

respects the linear order and all dependencies and, therefore, respects all edges in $\alpha(g)$ in the sense that if there is an edge from a to b in $\alpha(g)$, then a precedes b in O_l. Hence $\alpha(g)$ has to be acyclic and so, by statement (2), $\alpha(F_0) = \text{true}$. This proves that F_0 is satisfiable.

We now prove that if F_0 is satisfiable, then $G(F_0)$ is simple 2-visit (and hence l-ordered by Theorem 3.1). Assume therefore that F_0 is satisfiable and let α be the truth assignment such that $\alpha(F_0) = \text{true}$. Using the criterion of Theorem 2.1 it can be checked that the set Π of partitions defined as follows is simple multi-visit, and hence $G(F_0)$ is simple 2-visit ($\max(\Pi) = 2$).

$$\alpha(F_j) = \text{false} \rightarrow \Pi(F_j) = \langle \{i, s\} \rangle,$$

$$\alpha(F_j) = \text{true} \rightarrow \Pi(F_j) = \langle \{s\}, \{i\} \rangle,$$

$$\alpha(P) = \text{false} \rightarrow \Pi(P) = \langle \{i, s_1\}, \{i_1, s\} \rangle,$$

$$\alpha(P) = \text{true} \rightarrow \Pi(P) = \langle \{i_1, s\}, \{i, s_1\} \rangle.$$

As an example, if $F_j = F_k$ *or* F_m, $\alpha(F_j) = \alpha(F_k) = \text{true}$ and $\alpha(F_m) = \text{false}$, then the corresponding partition graph is given in Fig. 10 and the corresponding part of the smv-algorithm (see Section 2) looks as follows:

```
procedure VISIT TREE (F_j, x); node F_j; integer x;
    case x of
    1: begin skip
            VISIT TREE (F_k, 1); VISIT TREE (F_m, 1);
            compute s(F_j)
       end;
    2: begin compute i(F_j);
            VISIT TREE (F_k, 2);
            skip
       end
    esac.
```

This proves the theorem. ∎

Thus the *only* way to decide the smv-property is to guess a set of partitions and check whether it is smv. Note that in the simple multi-pass case one need not guess a

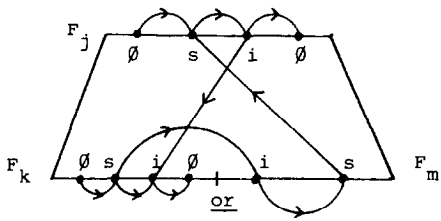

FIG. 10. A partition graph for $G(F_0)$.

partition due to the existence of a best partition in which each attribute has its earliest possible pass number ([2, 1]).

Note that for every Boolean formula F_0 which is not satisfiable, the construction in the proof of Theorem 4.1 produces an AG $G(F_0)$ which is not smv, but which is pure multi-pass (because it has only one derivation tree) and also absolutely noncircular (see [11]; it is easy to see that the i/o graph of P looks like Fig. 8a, whereas all other i/o graphs are empty).

We end this section with a few remarks on the OAG (ordered attribute grammars) of [10]. In our terminology, an AG G is an OAG if (i) for no attribute b, b is forced before b (cf. the introduction of this section), and (ii) Π_G is an smv-set of partitions, where Π_G is a particular set of partitions compatible with the relation *forced before* (see [10, Definition 4] for the definition of Π_G, denoted A there). Since the relation *forced before* (called IDS in [10]) is computable in polynomial time and it can also be checked in polynomial time whether Π_G is smv (cf. Theorem 2.1), it takes polynomial time to decide whether G is an OAG, as shown in [10] (the partition graphs for Π_G correspond to EDP in that paper). Clearly, the OAG are a (proper) subclass of the l-ordered AG. An AG is l-ordered if (dummy) dependencies can be added to it such that the resulting AG is an OAG, i.e., in the terminology of [10], G is arranged orderly by the augmenting dependencies. Note that all (simple) multi-pass AG are l-ordered, but not necessarily OAG.

Conclusion

We conclude this paper by taking together a number of decidability results on properties of AG concerning visits and passes (cf. the introduction). Fig. 11 shows an inclusion diagram for seven classes of AG, where V = visit, Pa = pass, M = multi, 1 = one, S = simple, P = pure, and ANC = absolutely noncircular. The distinction between pure and simple is taken from [1]. As indicated in Fig. 11, PMV is equal to the class of all noncircular AG [14], SMV equals the class of l-ordered AG (proved in this paper), and the l-ordered AG are included in ANC, cf. [10]. Deciding whether

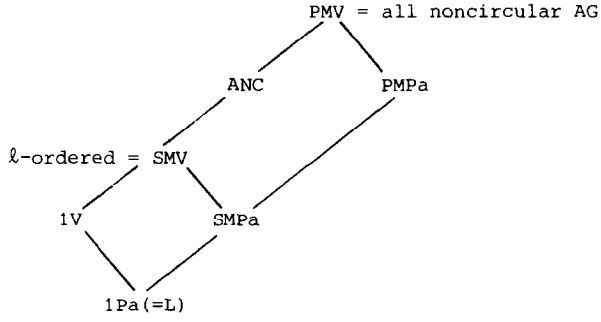

FIG. 11. Classes of multi-visit and multi-pass AG.

an arbitrary, i.e., possibly circular, AG is in these classes takes time, as follows (we also discuss the k-pass and k-visit cases for fixed k):

1Pa (usually called L): polynomial [2].
SMPa: polynomial [2], see [1] for an alternative algorithm; the simple k-pass property is also polynomial.
PMPa: exponential (inherently); the pure k-pass property is polynomial [5].
1V: polynomial [4].
SMV: *NP*-complete, even for fixed $k \geqslant 2$ (shown in this paper).
PMV: exponential (inherently) [7]; the pure k-visit property is decidable [14], but the precise complexity is not known (the algorithm in [14] is at least double exponential).
ANC: polynomial [11].

In case all AG are assumed to be noncircular, the PMV-property becomes trivial (always true), whereas all other results stay the same.

It is not clear whether a natural characterization of the ANC attribute grammars is possible in terms of restrictions on the attribute evaluation strategy (i.e., on computation sequences), and whether a similar concept would apply to passes, giving rise to a class of AG between SMPa and PMPa.

The translations and output sets of the above types of AG have been investigated formally in, e.g., [3, 4, 13, 14], but more research is needed to characterize their structure and complexity, and to link them to more classical types of tree manipulating systems.

Acknowledgments

We thank the members of the "attribute group" at our department for many stimulating discussions. We thank Sven Skyum and Hanne Riis for useful conversations.

References

1. H. Alblas, A characterization of attribute evaluation in passes, *Acta Inform.* **16** (1981), 427–464.
2. G. V. Bochmann, Semantic evaluation from left to right, *Comm. ACM* **19** (1976), 55–62.
3. J. Duske, R. Parchmann, M. Sedello, and J. Specht, IO-macrolanguages and attributed translations, *Inform. and Control* **35** (1977), 87–105.
4. J. Engelfriet and G. Filè, The formal power of one-visit attribute grammars, *Acta Inform.* **16** (1981), 275–302.
5. J. Engelfriet and G. Filè, Formal properties of one-visit and multi-pass attribute grammars, *in* "Proceedings of the 7th ICALP at Noordwijkerhout," pp. 182–194, Lecture Notes in Computer Science No. 85, Springer-Verlag, Berlin, 1980.
6. M. R. Garey and D. S. Johnson, "Computers and Intractability: A Guide to the Theory of *NP*-Completeness," Freeman, San Francisco, 1979.
7. M. Jazayeri, W. F. Ogden, and W. C. Rounds, The intrinsically exponential complexity of the circularity problem for attribute grammars, *Comm. ACM* **18** (1975), 697–706.

8. M. JAZAYERI AND K. G. WALTER, Alternating semantic evaluator, *in* "Proc. of ACM Ann. Conf." pp. 230–234, 1975.
9. U. KASTENS, Ordered attributed grammars, Bericht No. 7/78, Universität Karlsruhe, 1978.
10. U. KASTENS, Ordered attributed grammars, *Acta Inform.* **13** (1980), 229–256.
11. K. KENNEDY AND S. K. WARREN, Automatic generation of efficient evaluators for attribute grammars, *in* "Conf. Record of 3rd Symp. on Principles of Programming Languages," pp. 32–49, 1976.
12. D. E. KNUTH, Semantics of context-free languages; *Math. Systems Theory* **2** (1968), 127–145. Correction: *Math. Systems Theory* **5** (1971), 95–96.
13. H. RIIS, Subclasses of attribute grammars, DAIMI PB-114, Aarhus University, 1980.
14. H. RIIS AND S. SKYUM, *K*-visit attribute grammars, *Math. Systems Theory* **15** (1981), 17–28.