Parallel Ordered Attribute Grammars

Eduard Klein
GMD - Research Institute at the University of Karlsruhe
email: klein@karlsruhe.gmd.de

Abstract

Ordered Attributed Grammars (OAG) are extended for use on parallel computers. A partition of the attributes of a tree leads to parallel evaluation if each partition is taken care of by a separate process. Visit sequences for parallel ordered attribute grammars (POAG) are constructed and it is shown that the POAG class is of the same power as the OAG class. The construction leads always to deadlock free attribute evaluators. The approach taken provides a uniform basis for a wide range of implementations on parallel computers, allowing fine grain or coarse grain parallelism, shared memory or distributed memory architectures, with synchronous or asynchronous communication. Several kinds of partitions are discussed and some experimental results and experiences are described.

1 Introduction

In the last 20 years attribute grammars ([14]) have proven to be an appropriate specification technique for a wide range of applications. Among them are the formal definition of the semantics of programming languages ([17]) or the description of consistency conditions in graphical editors ([16]). In addition to the possibility of specifying problems at a very high level, the advantage of attribute grammars (AGs) is that they can be automatically compiled into an executable program called attribute evaluator which checks for a given input if it satisfies the specification ([10], [4], [3], [13]). Thus from a programming language description a compiler for that language can automatically be generated.

Since the acceptance of the approach depends heavily on the efficiency of the generated programs such as run time and storage requirements, a lot of “tuning” research has been done ([9], [18]). Since parallel computers are emerging more and more, there are investigations about how the generated programs can be executed (and thus hopefully be sped up) on these machines ([12], [11], [7], [15], [19]).

In this paper an approach for implementing ordered attribute grammars (OAG in [8]) on a parallel computer is described. In particular, the generation of very efficient attribute evaluators based on visit sequences is developed for execution on several processors. This leads to several processes running in parallel, each of which is an efficient visit sequence based attribute evaluator, and each of which takes care of a different part of the input.

The division of the input is described at generation time at the AG level. For each production the attributes are divided into disjoint sets called segments. At runtime all attributes in a tree are divided into disjoint sets as follows. When attributes of different productions overlap in a given attributed tree, the corresponding segments overlap as well and are joined; this is called the melting of segments. Each set is taken care of by a distinct process. Through the notion of segments, both fine grain and coarse grain parallelism can be described. Selecting appropriate segments depends on the characteristics of the parallel hardware as well as on the attribute dependencies. In this paper various segmentation strategies are discussed.

In general, this data parallel approach ([6]) requires communication between processes if there are dependencies between attributes which are handled by different evaluators. Communication is expressed through events. Based on the partition of an input, the visit sequences are extended by calls to an event handler which guarantees correct communication points at run time. The implementation of events is in no way restricted and hence can be realized in synchronous and asynchronous environments. It can be implemented on a wide range of machines, including shared memory and distributed memory architectures.

The machine model is MIMD; computation is expressed by processes and events describing communication. The execution of processes is only synchronized by communication constraints. If the number of processes is higher than the number of available processors, a scheduler takes care of the mapping of processes to processors. Any scheduling technique may be applied.

The approach taken abstracts from specific archi-
tectures. This level of description has been chosen because it simplifies the presentation of algorithms without hiding problems occurring in practice. Moreover, since processes and events can be implemented on any computer, the theorems and properties of the approach described apply to all of them.

The attribute evaluation has static and dynamic aspects as well. Although the evaluation sequence of attributes by each process is determined at generation time, it must not be fixed if the attributes belong to distinct processes. There may only be some constraints due to communication points. Thus the approach described is more flexible than the OAG method and retains its efficiency for attribute evaluation within processes.

The ordering of attributes is guided by a criterium that prevents deadlock situations at run time and thus guarantees a correct evaluation order. When attributes of a segment are ordered, the criterium takes care of previous orderings (i.e. those which have already been done in different segments during the ordering process). With the presented approach the class of parallel ordered attribute grammars (POAG) can be handled, which are of the same power as OAGs.

The paper is structured as follows. After introducing basic concepts, the building of segments is described. The ordering of attributes for parallel evaluation and the POAG class is defined; a proof for its descriptive power is given. Then, the construction of parallel visit sequences is presented and it is proved that this leads to deadlock free attribute evaluators. Different strategies for building segments are discussed and an example is explained. Finally, a conclusion and an outlook to future work is given. The approach is compared to related research at several points in the paper.

Context free grammars, attribute grammars, graphs, relations and their graphical representation have often been (re-)defined and hence it is not repeated here. Instead, some common notations from e.g. [18] are used and explained where necessary.

2 The Construction of Segments

In the AG approach the division of the input can occur at run time (dynamic division) when the input is known or at generation time (static division) on basis of the AG. Hybrid division is an intermediate approach (see below). The advantage of the first method is that the input is completely known and an appropriate division can always be found. The disadvantage is that the overhead for computing the division diminishes the speedup of parallel evaluation. In the second method, the overhead occurs only once at generation time and does not contribute to evaluation time. The disadvantage is that there is only unprecise information available about the shape of the input. All that is known is that the input will be structured according to the productions of the AG. It turns out however that grammar analysis yields good results which can serve as a basis for an appropriate input division at run time ([15]). Hence the latter approach is followed here.

Hybrid division has static and dynamic aspects as well. Grammar analysis or other information (e.g. the shape of 'typical' programs or profiling information) is used to precompute several possibilities for input division. At run time, the final division is decided based on further information such as the size or the kind of the current input fragment ([2]). Since most computation is done at generation time, the latter approach is attractive as well and certainly needs further investigation.

In the approach described here, the attribute occurrences of every production are partitioned into static segments, which is called the horizontal melting of attributes. At run time, when symbols (and their attributes) of two productions overlap at some tree node, the corresponding segments are joined (vertical melting of segments or attributes). The vertical melting is iteratively done on the whole tree until all corresponding segments have been joined and hence no further melting is possible, thereby resulting in dynamic segments. In Fig. 1, possible horizontal (a) and vertical (b) meltings are shown for three productions $p: A \rightarrow BA$, $q: B \rightarrow BC$ and $r: A \rightarrow C$. Attributes are printed beside its associated symbols and segments are shown as regions.

In the example, the number of static segments is 6 and the number of dynamic segments is 2. In this section only the concept of segments is described. Strategies for determining appropriate segments are discussed below.

Aside from the most general possibility of building dynamic segments for an attributed tree, namely an arbitrary partition of all its attribute occurrences (which is a dynamic division), there are several restrictions to the construction of segments which have emerged in the literature. In tree oriented segments (static division), a tree is partitioned into subtrees and all attributes of a subtree belong to one segment. In nonterminal based segments (resp. production based segments) the root of a subtree is determined by each instance of a specified set of nonterminals (resp. non-

---

1 In order to facilitate the presentation, in many places the term attribute is used for either an attribute of a symbol or an attribute instance within a production or an attribute occurrence within an attributed tree.
terminals of the left hand side of certain productions). A more flexible approach allows attributes of a symbol to be in different segments for different instances of that symbol (as in Fig. 1). This high flexibility turns out to be inefficient at run time because in general the vertical melting needs more than one pass over the tree. In Fig. 2, static segments of three productions $p$, $q$ and $r$ (partially depicted) are shown. The vertical melting of the segments of $p$ depends on the situation in the tree ($p$ above $q$ (2a) or $p$ above $r$ (2b)). If the vertical melting proceeds top down, the situation is not known before the context of $q$ or $r$ is entered.\footnote{Obviously, there is also an example such that one bottom-up pass is not sufficient to do all the vertical melting.}

For similarly partitioned AGs, the vertical melting is trivial because no preprocessing of the input is necessary and there exist unique visit sequences for each segment. Hence the vertical melting can be achieved through the same efficient mechanisms as the context switches in sequential evaluation such as recursive descent or maintenance of a stack. Although similarly partitioned AGs are less flexible than arbitrary grouping of attributes it is a generalization of tree oriented segments (used in \cite{2} and \cite{15}), because with the restriction that “all attributes of a symbol be in one segment” tree oriented segments are obtained.

Visit sequences can be determined at generation time for each segment if all attributes of a symbol which are in one segment can be ordered such that for every occurrence in a tree they can be computed in that order. This criterium is equal to the ordering criterium in sequential evaluation if for each symbol all attributes of a symbol are in one segment. If the attributes of a symbol are in more than one segment, the criterium is weaker than the sequential one, because if all attributes of a symbol can be ordered then the attributes in any subset can be ordered as well but not vice versa.

3 Parallel Ordering of Attributes

Since the method presented is an extension of the OAG theory the computation of the dependency graphs (\cite{8}) is shortly summarized. For each grammar symbol $X$ a relation $IDS_{X}$ is computed containing the induced dependencies between attributes of symbol $X$. It contains at least the transitive dependencies between attributes that can occur in any tree. For each symbol $X$ these dependencies form a partial order. After that, for each $X$ the set of attributes is ordered, i.e. split into disjoint sets $A_{X,i}$.
(i = 1..mX) such that the partial order is preserved (i.e. \((a,b) \in IDS_X^i \land a \in AX_i \land b \in AX_j \Rightarrow i \leq j\)) and the sets \(AX_{mX}, AX_{mX-1}, AX_{mX-2}, \ldots\) contain alternating synthesized and inherited attributes. At evaluation time the attributes of \(X\) can then be computed in the sequence \(AX_1, AX_2, \ldots\). There is some freedom in choosing the order and Kastens gives a strategy determining the order which defines the OAG class. After that, \(IDS_X\) is extended by attribute dependencies representing the ordering, i.e. for all \(i=1..mX-1\), if \(a \in AX_i \land b \in AX_{i+1}\) then \((a,b)\) is added to \(IDS_X\). Lastly, for each production \(p\) the extended dependencies over productions \((EDP_p)\) is the relation consisting of the original dependencies between \(p\)'s attributes \((DP_p)\) united with the extended relation of \(IDS_X\) for each symbol \(X\) of \(p\). An AG is ordered iff for each production \(p\) the graph for \(EDP_p\) has no cycles. \(EDP_p\) is a partial order in \(AX_p\) (which is the set of all attributes of \(p\)). In order to determine a visit sequence for \(p\), arbitrary additional dependencies are added to \(EDP_p\) until its elements form a total order.

In parallel attribute evaluation the requirements must be refined. For similarly partitioned AGs the attributes of a symbol \(X\) can be partitioned into sets \(AX_1, AX_2, \ldots, AX_X\), where for every instance of \(X\) in the grammar each two attributes of the same set are in the same segment and each two attributes of different sets are in different segments. Since each set is taken care of by different evaluation processes, a visit order must not be determined for all attributes of \(X\) but only for those within each set.

**Deadlock Prevention**

For correct sequential evaluation (i.e. all attributes of a tree can effectively be evaluated at run time) it is sufficient to determine a total order on EDP independent for each production because for any symbol \(X\) there is a dependency between each two of its attributes. The visit sequences obtained are correct because these dependencies are the same for all instances of \(X\) in the grammar. In parallel evaluation however there are not necessarily dependencies between each two attributes of a symbol.

Assume two productions \(p: B \rightarrow C\) and \(q: C \rightarrow D\) with \(A_1^p = \{a, b\}, A_2^p = \{c\}, A_1^q = \{d\}, A_2^q = \{a', b'\}\) and \(DP_p = \{(a, c), (b, d)\}, \quad DP_q = \{(a', c), (d, b')\}\) (Fig. 3). Assume further that there is no dependency between \(c\) and \(d\), i.e. \(\exists k ((c,d) \in AC_k)\) (which implies \((c,d), (d,c) \notin IDS_C\) and that there is no dependency between \(a\) and \(b\) resp. \(a'\) and \(b'\) as well. Hence, if the attributes were ordered independently for each production, the order between \(a\) and \(b\) resp. \(a'\) and \(b'\) could arbitrarily be chosen, e.g. "\(a\) before \(b\)" and "\(b'\) before \(a'\)" as in Fig. 3. This results in a deadlock at run time because \(a\) waits for \(a'\) while \(b'\) waits for \(b\).

![Figure 3: Deadlock situation possible](image-url)

Deadlocks are prevented if auxiliary dependencies between independent attributes (e.g. "\(c\) before \(d\)" for every instance of \(C\)) are added to the dependency graph in Fig. 3. This would necessarily lead to "\(a\) before \(b'\)" because otherwise the ordering of attributes in \(q\) would result in a cycle.

In the following a sufficient and necessary condition is given for ordering the attributes of a tree (whose segments have been vertically melted) such that evaluation is deadlock free. Since at generation time no particular tree is known, a sufficient condition is given below (adding auxiliary dependencies to the dependency graph as in the above example) such that visit sequences can be determined at generation time resulting in deadlock free evaluation for all trees of the grammar at run time.

Let \(A_i\) be the set of attribute occurrences of a tree \(t\), \(a, b \in A_t\), and \(a \prec_b c \in A_t\) a relation, \((a,b) \in <t\) iff there is a (transitive) dependency from \(a\) to \(b\). For \(k \in \{1, 2, \ldots\}\) let \(WL_k = \{a \in A_t\} a\) be a list of attributes and \(\prec_{WL_k} = \prec_{WL_t} \times WL_k\) (resp. \(\leq_{WL_k}\)) the transitive (resp. transitive and reflexive) relation such that \((a,b) \in <_{WL_k}\) (resp. \(\leq_{WL_k}\)) if \(a\) precedes \(b\) in \(WL_k\) (resp. \(a = b\) or \(a\) precedes \(b\) in \(WL_k\)). Let \(\prec_{WL_t} = \prec_t \cup <_{WL_t} \cup \leq_{WL_t} \cup \ldots\).

**Theorem:** Let AG be a well defined grammar, \(t\) an attributed tree with vertically melted segments \(A_1^t, A_2^t, \ldots\) and \(WL_1, WL_2, \ldots\) the corresponding (work) lists of totally ordered attributes. A parallel attribute evaluator following the order given through \(WL_k\) \((k=1,2,\ldots)\) is deadlock free iff property \(DF\) holds:

\[
DF \equiv \forall k = 1, 2, \ldots, \forall a, b \in WL_k: a <_{WL_k} b \Rightarrow \\
\neg(b <_t a) \land (\forall k \in \{1, 2, \ldots\} \forall a', b' \in WL_k: \ \neg((a' <_{WL_k} a \land b <_{WL_k} b') \Rightarrow a' <_{WL_k} b'))
\]

Informally, \(DF\) means that if there is a constellation as in Fig. 3, i.e. \(a\) depends on \(a'\) and \(b'\) depends on \(b\) and \(a\) is before \(b\) in the visit order, then \(a'\) must...
be before $b'$ in the visit order. A formal proof of the theorem is given in [12], here it is sketched. Consider a situation where not all attributes are yet computed: 

$\Rightarrow$: If the evaluator is deadlock free, for $a, b \in \mathcal{WL}_k$ and $a \not\triangleleft_{\mathcal{WL}_k} b$, $-\langle b < a \rangle$ holds because otherwise a deadlock occurs. Let $a', b' \in \mathcal{WL}_k$ for some $k$ and $a' \not\triangleleft_{\mathcal{WL}_k} b'$. Then $a' \not\triangleleft_{\mathcal{WL}_k} b'$ holds for $k = k'$ or $k \neq k'$ because otherwise a deadlock occurs.

$\Leftarrow$: If the $\mathcal{WL}_k$-lists are constructed such that the above implication holds, it has to be shown that there is always an attribute in some WL-list which is ready for evaluation. Through contradiction it is shown that a situation where no attribute is ready for evaluation can never occur.

The POAG Class

One method to extend EDP$_P$ for parallel evaluation is as follows: For each symbol $X$ and each set $A_X^k$ ($k \in \{1, n_X\}$) consider the restriction $\mathcal{IDS}_X^k \subseteq \mathcal{IDS}_X$ on attributes of $A_X^k$, i.e. $\{a, b \in A_X^k \wedge (a, b) \in \mathcal{IDS}_X^k\}$. The attributes within each restriction are then ordered (e.g. according to Kastens' strategy), i.e. for $k = 1, n_X$ each restriction is split into disjoint sets $A_{X,1}^k, A_{X,2}^k, \ldots$ such that the partial order is preserved $\langle (a, b) \in \mathcal{IDS}_X^k \wedge a \in A_{X,i}^k \wedge b \in A_{X,j}^k \rangle \Rightarrow i \leq j$) and that the sets $A_{X,1}^k, A_{X,2}^k, \ldots$ contain alternating synthesized and inherited attributes as in the sequential case. After that, $\mathcal{IDS}_X^k$ is extended similarly to the sequential case: For all $i = 1, m_X^k - 1$, if $a \in A_{X,i}^k \wedge b \in A_{X,i+1}^k$ then $(a, b)$ is added to $\mathcal{IDS}_X^k$. The counterpart to EDP$_P$ (for some $p$) is the relation consisting of the original dependencies between $p$'s attributes united with the extended relations $\mathcal{IDS}_X^k$ for each symbol $X$ of $p$ and each $k$. The absence of cycles in the resulting graphs defines the grammar class (say OPAG, ordered parallel attribute grammars).

Since OAGs have been proven to be very powerful e.g. in describing the static semantics of Ada ([17]) at least the OAG class should be accepted by the approach. In order to express the power of the OPAG class in terms of known grammar classes for each grammar the attribute dependencies in the corresponding EDP graphs must be comparable. Unfortunately there is no easy way to relate the arcs in EDP with those in its counterpart for the OPAG class because the ordering of attributes results in non-comparable partitions for each class. However, if in the parallel approach the ordering of attributes were guided by the ordering of the OAG method, the resulting grammar class can be compared to OAGs. This leads to a different method to extend EDP for parallel evaluation:

Let $A_X$ be the set of all attributes of $X$ and $AS_X$ resp. $AI_X$ the set of all synthesized resp. inherited attributes of $X$. Assume that a similarly partitioned AG with segments is given and the sets $A_{X,i}$, $i \in \{1, m_X\}$ are determined according to the OAG method. For each $k \in \{1, n_X\}$ these sets are restricted to $A_X^k$ which is a projection of the ordered attributes of a symbol $X$ to the subset of these attributes in each segment. The projection takes care that the original dependencies are preserved. It is computed for $i = 1, 2, \ldots$ ($A_{X,i}^0$ serves as initialization): 

$A_{X,i}^k = \{b \in A_X^k \mid (\exists i < j \in [b \in A_X^{k+1}]) \wedge (\exists a \in A_X^k \wedge a < j[a \in A_{X,j}^k, b \in A_{X,i}^k]) \Rightarrow \exists i \leq i[a \in A_{X,i}^k]) \wedge

(A_{X,i}^k \subseteq AI_X \Rightarrow A_{X,i+1}^k \subseteq AS_X)

$\text{In the above formula, } b \text{ is added to } A_{X,i}^k \text{ (for some } X, i \text{ and } k) \text{ if the three conditions forming the conjunction hold: The first condition guarantees that } b \text{ is assigned exactly to one set, the second condition takes care of the original dependencies and the last condition guarantees that the computed sets contain alternating synthesized and inherited attributes.}$

Similar to the OAG method, for each symbol $X$ and $k \in \{1, n_X\}$, $m_X^k$ is the smallest index where $A_{X,k}^k = 0$ for $i > m_X^k$ and where $A_{X,k}^k$ contains no inherited attributes. Through the given construction all attributes of $X$ which are in some $A_{X,k}^k$ (and thus in one segment) can always be computed in the sequence $A_{X,1}^k, A_{X,2}^k, \ldots$. In Fig. 4 for symbol $X$, $AI_X = \{a, c, d\}$, $AS_X = \{b\}$, $A_{X,1}^k = \{a, d\}$ and $A_{X,1}^k = \{b, c\}$, possible orders and its projection on segments is given. Note, that some $A_{X,i}^k$ can be empty which results from preserving the order between all attributes of $X$.

![Diagram](image)

Figure 4: Visit orders and projection on segments

After determining $A_{X,i}^k$ (for all $X, i$ and $k$), $\mathcal{IDS}_X$ is extended similar to the sequential case by attribute dependencies representing this ordering and by auxiliary dependencies guaranteeing deadlock free evaluation.

Definition: For every symbol $X$ the parallel dependencies on symbols are 

$PDS_X := \mathcal{IDS}_X \cup VO_X \cup DF_X$

where the visit orders are 

$VO_X = \{(a, b) \exists i[a \in A_X^i, b \in A_X^{i+1}]\}$
and the dependencies guaranteeing dead lock free evaluation are
\[ DF_x = \{(a,b) \mid \exists a, b \in A_{x, i} \land \exists k \in A_{x, i+1} \land b \in A_{x, j} \} \]

Below it is proven that \( DF_x \) is sufficient to guarantee that a situation similar to Fig. 3 never occurs at run time.

For each production \( p : X_0 \rightarrow X_1 ... X_n \) the counterpart to \( EDP_p \) are the parallel dependencies over productions:
\[ PDP_p = DP_p \cup PDS_{X_1} \cup ... \cup PDS_{X_n} \]

**Definition:** Let \( AG \) be a similarly partitioned attribute grammar. \( AG \) is a parallel ordered attribute grammar (POAG) iff for each production \( p, PDP_p \) has no cycles.

The shorthand POAG resp. OPAG above stem from a functional view of the construction of visit or-

**Power of the POAG Class**

**Definition:** Let \( AG \) be an OAG which is similarly partitioned. For each production \( p : X_0 \rightarrow X_1 ... X_n \) and \( X \in \{X_0 ... X_n\} \) be \( A_{x, i}^k \) determined for all \( i \) and \( k \).

AG is in weak normal form (WNF) iff for each production \( p \) holds:
\[ \forall X \in \{X_0 ... X_n\}, \forall i = 1..m_X, \forall k = 2..n_X, \forall a, b \in A_{x, i}^k : \]
\[ a \in A_{x, i+1}^k \land b \in A_{x, j}^k \Rightarrow (b, a) \notin EDP_p^+ \]

\( \square \)

The WNF is weaker than the Bochmann normal form (NF) of AGs ([11]), which requires that an attribute which is assigned in a production \( p \) is no argument for the computation of another attribute of \( p \). Thus it is no restriction for practical grammars because every OAG can be transformed into NF.

**Corollary:** Let \( AG \) be an OAG. If \( AG \) is in NF then \( AG \) is in WNF.

**Proof:** Let production \( p : X_0 \rightarrow X_1 ... X_n \) and \( X \in \{X_0 ... X_n\} \). Let \( a, b \in A_{x, i}^k \) for some \( i \) and \( k \). Then \( (b, a) \notin EDP_p \) because AG is in NF. \( (b, a) \notin IDS_X \) because otherwise \( \exists a \in A_X [(b, c), (c, a) \in IDS_X \land \exists c \in A_X \Rightarrow a, b \in A_{i}^k] \) (due to NF) and thus \( \exists a, b \in A_{x, i}^k \) which is contradictory to the assumption.

**Theorem:** An AG in WNF is OAG iff it is POAG.

**Proof:** For each production \( p : X_0 \rightarrow X_1 ... X_n \) it must be shown that \( PDP_p \) has no cycles iff \( EDP_p \) has no cycles, i.e. a dependency occurring in \( PDP_p \) must also occur in \( EDP_p \) or introduce no cycle in \( EDP_p \) and vice versa: If \( (a, b) \in EDP_p \cup \exists \in \{X_0 ... X_n\} \) then \( (a, b) \in EDP_p \) and \( (a, b) \in PDP_p \). Let \( X \in \{X_0 ... X_n\} \):

\[ \Rightarrow \]: If \( (a, b) \in VO_X \) then \( \exists \in \{X_0 ... X_n\} \) (due to the construction of \( VO_X \)) and thus \( (a, b) \in EDP_p \). If \( (a, b) \in DF_x \), let \( a, b \in A_X, a \in A_{x, i}^k \), \( b \in A_{x, j}^k \) for some \( i, k \). Then \( (a, b) \notin EDP_p^+ \) due to NF and \( (a, b) \) introduces no cycle in \( EDP_p \).

\[ \Leftarrow \]: If \( \exists \in \{X_0 ... X_n\} \) then \( (a, b) \in VO_X \). Thus \( (a, b) \) introduces no cycle in \( PDP_p \).

**An Example**

Consider an AG (in NF) with productions \( p : S \rightarrow A, q : A \rightarrow B \) and \( r : B \rightarrow c \). The (only) attributed tree for that grammar is depicted in Fig. 5. Attribute dependencies are shown as arrows. Arrows without a source vertex mean that the attribute corresponding to the target vertex is assigned a constant value. Inherited resp. synthesized attributes have an incoming arrow from above resp. from below.

![Attribute dependencies of a POAG](image)

**Figure 5:** Attribute dependencies of a POAG

The test for the POAG class results in the following dependencies:

<table>
<thead>
<tr>
<th>IDSA</th>
<th>VOA</th>
<th>DF_A</th>
<th>PDS_A</th>
<th>DP_A</th>
<th>PDP_A</th>
</tr>
</thead>
<tbody>
<tr>
<td>a → b</td>
<td>{}</td>
<td>a → b</td>
<td>e → d</td>
<td>d → e</td>
<td>c → e</td>
</tr>
</tbody>
</table>

The grammar is POAG because \( PDP_A \) (and \( PDP_p \), \( PDP_p \), as well) have no cycles.

**4 Construction of Visit sequences**

In order to build a parallel attribution algorithm, for each production and segment the attributes have to be linearly ordered in a visit sequence containing also visits to surrounding nodes and events representing communication.

**Linear Ordering of Attributes**

For each production \( p : X_0 \rightarrow X_1 ... X_n \) and segment \( A_{x, i}^k \) those set \( A_{x, i}^k \) which don't contain attributes
that are computed in context \( p \) are represented through artificial attributes \( c_{i,j}^k \):

\[
Visits^k_{p,j} := \{c_{i,j}^k | j \in \{m_X, m_X - 1, m_X - 2, \ldots \} \}
\]

\[
(i \in \{m_X, m_X - 1, m_X - 2, \ldots \} \land i > 0)\]

The relation \( PDP_p \) is then modified by replacing each attribute which is not computed in context \( p \) (i.e. the inherited attributes of the left hand side symbol and the synthesized attributes of the right hand side’s symbols) by its corresponding \( c_{i,j}^k \) and by adding \( c_{i,j}^k \) for each empty \( A_{X,i,j}^j \). For all productions \( p \) and segments \( A^k_p \) let

\[
V_A^k_p := A^k_p \cup Visits^k_p
\]

\[
VDP_p \subseteq V_A^k_p \times V_A^k_p
\]

such that

\[
VDP_p := \{(Map_p(a), Map_p(b)) | (a, b) \in PDP_p \}
\]

\[
\cup \{(c_{i,j}^k, Map_p(b)) | \exists \{c_{i,j}^k, a \in A_{X,i,j}^j \land b \in V_A^k_p \}
\}

\[
\cup \{(c_{i,j}^k, Map_p(b)) | \exists \{c_{i,j}^k, a \in A_{X,i,j}^j \land b \in V_A^k_p \}
\}

\[
\text{where } (a \in A_{X,i,j}^j)
\]

\[
Map_p(a) = \{ a, \text{ if } a \text{ is computed in context } p \}
\]

\[
\text{and } c_{i,j}^k, \text{ if } a \text{ is not computed in context } p
\]

For each \( p \) the elements of \( VDP_p \) are topologically sorted and for each segment, the total order is restricted to the elements of the segment. For each \( p \) let \( T_p \) be the sequence representing this total order. Then for each segment \( V_A^k_p \), the subsequence \( T^k_p \) is defined for all \( a, b \in V_A^k_p \) where \(< \) defines the total order in \( T_p \) and \( T^k_p \).

The following theorem states that the described method leads to deadlock free evaluators.

**Theorem:** Let \( AG \) be a POAG. A parallel attribute evaluator is deadlock free if the visit sequences for each segment are based on restrictions of a topological sorting of \( VDP_p \) (for some \( p \)).

**Proof:** It is sufficient to show that property DF holds (the work lists \( WL_k \) correspond to visit sequences). Let \( \prec_{VDP} := \{(a, b) \in VDP_p \} \) and \( \prec_p := \prec_{VDP} \cup \prec \).

1. DF holds within each production \( p \) because for \( a', b' \in T^k_p \land [b < a \in T^k_p] \) it follows that \( [-b < a] \in T^k_p \land [a' < a \land b < b' \rightarrow a' < b' \in T^k_p] \) due to the construction of the total orders.

2. DF holds for a whole tree if it holds for each two adjacent productions: Let \( p \) and \( q \) be adjacent productions in a tree (\( p \) above \( q \)) with common symbol \( X \), let \( a < b \in T^k_p \land a' < b' \in T^k_q \) for some \( k, k' \), and \( a, d \in A_X \) with \( a < b \leq Map_p(c) \land Map_p(c) < b < Map_q(d) \). If \( a \) and \( d \) are dependent in \( VOX \) then \( DF \) can be decided within each production because the dependency occurs in every instance of \( X \). In case \( c \) and \( d \) are independent in \( VOX \), i.e. \( c, d \in A_{X,j} \) for some \( j \): If \( c \) and \( d \) are in one segment, they are inherited attributes (otherwise a cycle \( Map_p(c) < a < b < Map_p(c) \land Map_p(c) \) exists). Thus \( a' < b' \) because otherwise a cycle exists in \( q \). If \( c, d \) \( \notin \) are in different segments, then \( c \) and \( d \) are (transitive) dependent in \( DF \) and thus \( DF \) can be decided within each production.

\[ \square \]

Note that it is crucial for deadlock free evaluation that first the total order on all attributes of each production is built and then that order is restricted to the segments. Constructing the total orders independently for each segment in general leads to deadlocks at evaluation time. An example of this can be found in [12].

**Insertion of Events**

For each external dependency, which is a dependency between two attributes of different segments, events are added to the visit sequences. In an external dependency \( a \rightarrow b \), \( a \) is an external predecessor of \( b \) and \( b \) is an external successor of \( a \). If process \( P_1 \) handles \( a \) and \( P_2 \) handles \( b \), \( P_1 \) (after evaluating \( b \)) must notify \( P_2 \) that \( b \)'s argument is available and thus \( b \) can be computed. In the visit sequence for \( P_1 \) (resp. \( P_2 \)) this is represented through \( \text{OUT}(a) \) (resp. \( \text{IN}(b) \)).

For some \( p \) and \( k \) let \( a \in A^k_p \) be computed in context \( p \). If \( a \) has external predecessors and/or external successors, \( \text{IN}(a) \) and/or \( \text{OUT}(a) \) are put immediately before and/or after \( a \) in the visit sequence of \( A^k_p \). If in some production \( c_{i,j}^k \) exists for some \( i, j \), \( k \), let \( a_1, a_2, \ldots \in A^k_{X,i,j} \) be all attributes with external successors; then \( \text{OUT}(a_1), \text{OUT}(a_2), \ldots \) is put immediately after \( c_{i,j}^k \) in the visit sequence for \( A^k_p \). Obviously, with this construction all external dependencies are represented through events.

Note that if an attribute \( a \) has several external predecessors (resp. successors), \( \text{IN}(a) \) (resp. \( \text{OUT}(a) \)) means that all predecessors have to be waited for (resp. all successors are notified). Alternatively, for each external dependency events could be added to the visit sequences which results in more calls of an event handler at run time. Both techniques lead to visit sequences where communication is represented as events.

**Continuation of the Example**

The grammar in Fig. 5 yields:

\[
A^1_{X,1} = \{a\} \quad A^1_{X,2} = \{b\}
\]

\[
A^1_{Y,1} = \{c\} \quad A^1_{Y,2} = \{\}
\]

\[
A^1_{Z,1} = \{d\} \quad A^1_{Z,2} = \{e\}
\]

For each production \( p \) and segment \( A^k_p \) the visit sequences \( V^k_{DP} \) are...
Within the visit sequences of each rule, corresponding events are written one upon the other.

The Parallel Evaluation Algorithm

The visit sequences define a sequence of actions which are executed by an evaluator. There are five kinds of actions, an attribute computation, the visit of an adjacent production (visit parent, visit child) and the sending of and waiting for events. Below an interpretive evaluation algorithm is given in a parallel language with coroutines. Other implementation techniques e.g. direct coded visit sequences are possible as well.

(*initialization*)

\[ VS^1_p = a, c_1, \]
\[ VS^2_p = c_1, \]
\[ VS^3_p = c_2, O U T(a), c, c_1, b, IN(b), b \]
\[ VS^4_p = \]
\[ VS^5_p = c_2, O U T(c), b, c_2, \]
\[ VS^6_p = c_2, \]
\[ IN(e), e \]

FOR ALL \((t,p,k) \in \text{StartTriples}\) DO

\[ \text{IN PARALLEL visit}(t,p,k) \]
\[ \text{END.} \]

BEGIN

COROUTINE visit \((t; t r e e, p, k, i n t e g e r)\);

WHILE \(V S^*_p \neq \text{EmptyList}\) DO

CASE head\((V S^*_p)\) OF

Eval\((a)\): EVAL\((a)\);
Wait\((a)\): IN\((a)\);
Send\((a)\): OUT\((a)\);
VisitParent: DETACH; (*Head\((V S^*_p) = c_2, *\)*)
VisitChild\((i)\): Resume\((\text{Child}(t, i), \text{Prod}(t, i), \text{Seg}(t, k, i))\);
END;
(*Head\((V S^*_p) = c_2, i > 0^*\)*)

\[ V S^*_p := \text{Tail}(V S^*_p) \]
\[ \text{END} \]
\[ \text{END} \]

Besides the usual list operations the algorithm uses some functions. \(\text{Child}(t, i)\) returns a pointer to the \(i\) th child of the current node, \(\text{Prod}(t, i)\) determines the production applied at node \(\text{Child}(t, i)\) and \(\text{Seg}(t, k, i)\) returns the number of the segment to be entered in production \(\text{Prod}(\text{Child}(t, i))\). IN and OUT are routines of an event handling mechanism. EVAL calls an attribute evaluating function.

During initialization, for each dynamic segment one process is started (through the construct "IN PARALLEL"). Within a process, for each production and (static) segment one coroutine is called. The vertical melting is carried out through recursive calls of the coroutines.

The determination of the start triples depends on the strategy of the evaluator. Similar to the sequential case, evaluation can be done concurrently with (bottom up or top down) tree building or after the tree has been completely built. Depending on the strategy the visit sequences have to be modified. Usually evaluation proceeds top down starting as high as possible in the tree. Hence in a visit sequence \(V S^*_p\) each first visit to the parent node \((c_0)\) for some \(k\) is redundant because \(V S^*_p\) is entered via recursive call from the parent node. In order to return from the coroutine call, at the end of \(V S^*_p\) a visit to the parent node has to be inserted.

While in the sequential case the root is the single starting point, in parallel evaluation it is the highest point in the tree of each dynamic segment. Thus \(\text{StartTriples}\) can be determined while the tree is built. It consists of all segments (of all productions) in the tree which don't contain attributes of the left hand side symbol and of the segments containing attributes of the root productions's left hand side symbol.

Note that the parallel evaluation method is not restricted to the described top down strategy but other strategies can be implemented as well.

5 Appropriate Segmentations

In the method presented the construction of visit sequences is based on a given horizontal melting of attributes in each production. The shape of segments has no influence on properties of the evaluator such as deadlock avoidance but it does influence evaluation time. In the following, fine grain and coarse grain segmentations are discussed applying to a wide area of parallel machines.

Fine grain Segments

Fine grain segments lead to many processes at run time and are only appropriate for machines with many processors. Otherwise (on nowadays machines) a lot of process management is necessary which reduces the gain of parallel evaluation.

The aim of fine grain segments is to exploit all parallelism which is given through the attribute dependencies of an AG. Obviously this is the case if within a segment each two attributes must be evaluated in sequence due to the attribute dependencies. For a given AG this property holds for any tree if the horizontal melting obeys the criterium \(\forall p, a, b \in A^*_p, \forall k:\)

\[ \{a, b \in A^*_p \Rightarrow (a, b) \in ISDP^+_P \cup (b, a) \in ISDP^+_P\} \]

where for all productions \(p\) and symbols \(X\) the intersection of dependencies over productions resp. symbols is

\[ ISDP^+_P = DP^+_p \cup \{X \mid X = X_0 \Rightarrow X_0 = (a, b) \in ISD^+_X \} \]

\[ ISD^+_X = \{ (a, b) \mid \forall p: X_0 = X_1 \Rightarrow (X_o, a, X_0, b) \in ISDP^+_P \} \]

\[ \forall p: X_0 = X_1 \Rightarrow (X_o, a, X_0, b) \in ISDP^+_P \}

\[ (X_1, a, X_1, b) \in ISDP^+_P \} \]

113
ISDS resembles the construction of IDS. The difference is that IDS contains dependencies which can occur in some tree whereas ISDS contains dependencies which occur in each tree for a given AG. For practical grammars ISDS contains dependencies ranging over several productions which origin in the "threading" of attributes such as the symbol table attribute in many AGs. In a Modula-2 grammar, for all productions $p$, $ISDP_p$ contains more than twice the dependencies of $DP_p$ (3898 vs. 1771) by which the possibilities for horizontal melting with the above criterium are increased.

Following the criterium there are still many possibilities for horizontal melting each of which exploits all possible parallelism. A concrete horizontal melting strategy can obey further criteria such as minimizing the number of segments per production or the number of events. Since in most cases reducing the number of segments also reduces the number of events and thus saves communication overhead, in [12] an algorithm based on graph colouring is given computing the horizontal melting based the above criterium and approximating minimal number of segments per production.

**Coarse grain Segments**

For computers with only a few processors (less than 20) coarse grain segments are appropriate which are (rather) independent, i.e. there is only little communication between the segments. For a given tree it is easy to determine appropriate segments but since we are interested in computing segments at generation time, the shape of a tree is not known. Instead there is only imprecise information about the dependency graph at run time. Grammar analysis can be a basis for horizontal melting and for tree oriented segments [15] suggests a method for computing independent segments. For the more general similarly partitioned AGs an automatic method still is a research topic.

However, in practical AGs describing the semantics of programming languages there are often (rather) independent segments which can be described via horizontal melting. For example, the analysis of procedures or modules can be done in parallel once their environment has been computed.

**An Example**

Let AG be the description of the semantics of a Pascal like language where procedures can be used before they are declared (no one pass restriction). In Fig. 6 a dependency graph of an input according to AG is shown. Elisions are depicted with triangles. The self explaining grammar symbols are written in lower case, their attributes are prefixed with a dot. The symbol table is expressed via environment attributes: $ei$ (environment-in) contains all visible objects at a program point in a first left-to-right tree traversal, $ei_2$ contains all visible objects after analy-
sis. eo (environment-out) contains all definitions d at a program point. c contains output information such as (intermediate) code.

The similarly partitioning shown in Fig. 6 leads to 6 processes at run time where each of 5 processes takes care of a shaded area and the 6th evaluates all other attributes. This segmentation leads to highly parallel attribute evaluation since the bodies of blocks at the same nesting level can be evaluated in parallel.

A POAG processing system has been developed ([5]) which takes as input a similarly partitioned AG and generates visit sequences for each segment. A (sequential) scheduler simulating a multi-processor architecture with asynchronous communication allows to study the behaviour of parallel attribute evaluators. It returns statistics such as the number of spawned processes, its phases of activity and passivity, the mapping of processes to processors, the number (and source and target) of events and the load of the processor system. In contrast to other simulating systems for parallel attribute evaluators our simulator can be parameterized with costs for process management and communication overhead which contributes to run time in the table below. Although the simulator doesn't catch the temporal aspect of time exactly, it allows to observe its causal aspects and hence is a good basis for the evaluation of the approach.

In Fig. 7 the graphical output of the simulated evaluation on a 6 processor machine for an input of the example AG is given. In the upper part each line lists the characteristics of a process and in the lower part those of a processor. Thick process lines denote phases of activity (attribute computation), curly lines denote passivity (e.g. waiting for an event), and thin lines
that a process is ready and waits for processor allocation. Events are represented through short vertical bars. Process (de-)allocation is represented through long vertical bars. Thick processor lines denote running processes, thin lines show idle time. Process (de-) allocation is shown through long vertical bars. Event handling through short vertical bars.

As can be seen no more than 5 to 10 processes are active at the same time (independent of the input) thus burdening the scheduler not too much which process management. The parallel machine has a very high load resulting from the much which process management. The parallel processes are active at the same time (independent programs. The speedup is the ratio of sequential to parallel evaluation time. It increases with the number of processors. In most examples the load of the processor system is nearly 100% and hence the small loss to linear speedup in the above table is caused through the overhead for parallel evaluation, mainly communication and process management.

### 6 Conclusion

A method for extending ordered attributed grammars for use on a parallel computer has been developed. Following a data parallel approach, several processes evaluate the attributes of a tree in parallel. The method applies to the class of parallel ordered attribute grammars (POAG) which is of the same power as the OAG class. It has been shown that the approach always leads to deadlock free parallel evaluators based on visit sequences. Several strategies for determining appropriate divisions of the input at generation time have been discussed. An example has been given and some results of simulating parallel attribute evaluation have been discussed.

Of course a real implementation is necessary and at the moment the method is implemented on a transputer network. We also aim at an implementation on a shared memory multiprocessor. The results of first experiments are promising and it seems that parallel attribute evaluation is a good basis for speeding up programs which are automatically generated from AG specifications.

### Acknowledgements

I like to thank Doaitse Swierstra, Maarten Penning, Kai Koskimies, Andreas Hess for many hints.

### References