Support Algorithms for Incremental Attribute Evaluation of Asynchronous Subtree Replacements

Josephine Micallef
Bellcore
445 South Street
Morristown, NJ 07962
(201) 829-4227
Fax: (201) 455-1931
micallef@bellcore.com

Gail E. Kaiser
Columbia University
Department of Computer Science
New York, NY 10027
(212) 854-3856
Fax: (212) 666-0140
kaiser@cs.columbia.edu

13 August 1991

Abstract

We attack the problem of incremental attribute evaluation algorithm for multiple asynchronous subtree replacements applicable to arbitrary noncircular attribute grammars. Our algorithm supports multiple independent editing cursors. Concurrent evaluations processes proceed independently as long as they cover disjoint regions of the derivation tree. Evaluation processes are merged when they overlap, to prevent unnecessary attribute evaluations. The algorithm ensures that when evaluation terminates, the tree is consistently attributed. Our results fill two open problems in the original algorithm for asynchronous subtree replacements reported by Kaplan and Kaiser.

keywords: attribute grammar, computer-supported cooperative work, incremental attribute evaluation, language-based editor, programming-in-the-many.

1. Introduction

Language-based editors are well-understood tools for educating student programmers [4] and reminding casual programmers of the programming language rules. Users construct their programs by selecting from a menu of legal templates, entering by hand only identifiers and literals; thus the editor guarantees syntactic correctness. The editor aids the user in writing semantically correct programs by incrementally checking for inconsistencies after every editing command, and highlighting any errors so the user can immediately locate and understand the problem while still in context.

Even though there has been much study of language-based editors [9, 10], this technology has rarely been used by advanced students or industry programmers, due to two basic inadequacies. First, the template-based user interface is awkward for many experienced programmers. Second, the editors currently available support only an individual programmer working on a small program or a single module of a large system, and do not help at all with detecting inconsistencies among modules written by different programmers. There has been much work on solving the first problem, e.g., conventional text editing user interfaces tied to the structure editor through incremental parsing technology [32, 21] and customization of the user interface based on user modeling [22]. In this paper, we address the second problem.

Multi-user language-based editors would enable collaboration among multiple student programmers working together on a group project, and more significantly, programming-in-the-many for commercial software development. Further, multi-user language-based editors could promote collaboration on other kinds of structured documents, besides programs, and thus provide a practical basis for many areas of computer-supported cooperative work [8].

Kaplan and Kaiser [16] introduced the first algorithm for multi-user language-based editors, and later added support for distributed failure recovery [14]; we refer to this work collectively as due to KK. KK extended the basic technology commonly employed for single-user language-based editors [31, 6]: representation of the program (or other structured document) as an attributed derivation tree, modeling of edits as subtree replacements in this tree, specification of the syntactic and semantic properties of the programming language (or other application domain) with an attribute grammar, and immediate propagation of changes introduced by an edit through incremental evaluation of the attributes affected by the edit.

KK presented (1) support for multiple editing cursors on the same program (or document) to permit *asynchronous* edits by multiple users, so users do not need to ‘‘take turns’’ to make their edits; (2) *parallel* attribute evaluation to permit multiple evaluation processes initiated by different edits to proceed concurrently, so users do not need to ‘‘wait’’ for evaluations initiated by themselves or other users; and (3) *distributed* attribute evaluation to permit segmentation of the attributed derivation tree and propagation of the results of evaluation processes among multiple user segments, so users can edit modules of the same program (or components of a shared document) on their own workstations connected by a network.

However, KK left open two important problems:

- Maintaining consistency between auxiliary data structures called *characteristic graphs*, each of which summarizes the dependencies among attributes at a node in the derivation tree, and the actual dependencies implicit in the tree. Characteristic graphs are used in the evaluation algorithm to construct its scheduling graphs, which are topologically sorted to order the evaluation of the attributes. Consistency is a problem since an edit may change the (indirect) dependencies among attributes arbitrarily far away in the derivation tree, and the characteristic graphs must be updated to prevent the case where an attribute dependent on a changed attribute is inadvertently not re-computed.

- Collision detection and subsequent merging of the scheduling graphs, called *models*, which represent the connected region of the tree under consideration by an attribute evaluation process. Since all attributes are assumed to have initial values, a model is expanded only when a re-computed attributed actually changes in value, to include other attributes dependent on the changed one. When a model is expanded to include some derivation tree node already in some other model, merging the models prevents unnecessary repeated computation of the attributes of that node.

Both problems must be solved in order for the KK algorithm to work correctly. In this paper, we present solutions to these two problems for the first time. In particular, we introduce an incremental evaluation algorithm with three parts: the first updates characteristic graphs after each edit, the second detects collisions between two models, and the third merges together collided models. Our algorithm operates within each segment of the distributed derivation tree, with propagations across segment boundaries handled as described by KK. For simplicity, we explain our algorithm as if there were a single centralized derivation tree with multiple independent cursors, but this should be understood as a single segment of a global distributed tree.

A special case of our algorithm has been implemented in the MERCURY system for the special case of derivation trees consisting of a collection of monolithic segments. MERCURY takes as input an attribute grammar for the programming language, and generates a distributed multi-user language-based editor for that language. An earlier version of MERCURY [15] did not include any of the new facilities presented in this paper. Thus it was necessary for correctness that user edits within a segment ‘‘take turns’’ with propagations into a segment, restricting parallel work.

We first present background in section 2, covering both the attribute grammar formalism and incremental attribute evaluation for single subtree replacements. This section can be skipped by readers familiar with these topics. We then state the problem and give an overview of our solution in section 3. The following three sections give the details of the three parts of our algorithm: section 4 presents characteristic graph update for multiple editing cursors, section 5 gives the details of collision detection, and finally in section 6 we cover the subtleties of correctly merging two models. We discuss the complexity of all three parts in section 7.

We then describe in section 8 two distinct application scenarios for computer-supported cooperative work, one for programming-in-the-many in the Modula-2 programming language and the other a distributed calendar system for scheduling meetings among groups of people. Both of these applications have been implemented using our MERCURY system. Section 9 describes related work: most other solutions to the asynchronous subtree replacement problem are application only to subsets of the noncircular attribute grammars, while our algorithm applies to the general case. We conclude by summarizing our contributions.

2. Background and Terminology

2.1. Attribute Grammars

Attribute grammars were first introduced by Knuth to describe the context-sensitive properties (static semantics) of programming languages [19]. An attribute grammar (AG) is based on a context-free grammar (CFG) that describes the language's syntax. A context-free grammar is denoted as $G = (N, T, P, S)$, where N and T are finite sets of nonterminal and terminal symbols respectively, P is a finite set of productions, and S is the start symbol of the grammar. Productions in a context-free grammar are of the form $X \rightarrow \alpha$, where X is a nonterminal and α is a string of symbols from $(N \cup T)^*$.

An AG extends a context-free grammar G by associating a set $A(X)$ of *attributes* with each symbol X in G. Each attribute represents a specific property of the symbol, and can take on any of a specified set of values. The notation $X.a$ indicates that attribute a is an element of $A(X)$. *Semantic equations* defining these attributes are associated with productions of the grammar G. A semantic equation defines an attribute, a_0, as the value of a *semantic function* applied to other attributes of that production, $a_1, \ldots, a_k$. The attribute on the left-hand side of the equation, a_0, is *functionally dependent* on the attributes on the right-hand side, $a_1, \ldots, a_k$. Thus, semantic equations have locality of reference, involving only attributes of the derivation tree nodes whose symbols appear in a single production.

Attributes are divided into two disjoint classes: *inherited* and *synthesized*. The inherited and synthesized attributes of a symbol X are denoted by $I(X)$ and $S(X)$ respectively; $I(X) \cup S(X) = A(X)$ and $I(X) \cap S(X) = \emptyset$. A semantic equation defines a synthesized attribute of the left-hand symbol of a production, or an inherited attribute of one of the right-hand side symbols. The start symbol of the grammar, S, has no inherited attributes; that is, $I(S) = \emptyset$. In Knuth's original formulation of AGs, terminal symbols could have inherited attributes but no synthesized ones. We follow the approach of later work on AGs and make no distinction between terminal and nonterminal symbols.

The *output attributes* of a production p: $X_0 \rightarrow X_1 \cdots X_n$ are those attributes defined by semantic equations associated with p. These are the synthesized attributes of the left-hand side symbol of p (*i.e.*, $S(X_0)$) and the inherited attributes of the right-hand side symbols of p (*i.e.*, $I(X_1) \cup \cdots \cup I(X_n)$). The *input attributes* of p are those attributes which appear on the right hand side of the semantic equations of p.

An AG is in *Bochmann normal form* [2] if for any production p, the input attributes of p consist of the inherited attributes of the left-hand symbol of p (*i.e.*, $I(X_0)$), and the synthesized attributes of the right-hand side symbols of p (*i.e.*, $S(X_1) \cup \cdots \cup S(X_n)$); that is, no attribute defined in p can be used to define another attribute in p. In this paper, we assume that attribute grammars are in Bochmann normal form. Any attribute grammar can be converted to Bochmann normal form [2], so this is not a limiting assumption.[1]

Figure 2-1 gives an example of an attribute grammar fragment for declarations in a Pascal-like programming language. There are four productions in the context-free grammar, *p1* through *p4*. Each symbol in a production has associated attribute instances (declared in figure 2-1 (a)), and each production has associated semantic equations that define the values of the attribute instances (shown in figure 2-1 (b)). Occurrences of the same symbol within one production are distinguished by the use of a numerical suffix; for example, in production *p3*, there are two occurrences of *Decls*, denoted by *Decls$1* and *Decls$2* for the first and second occurrence, respectively.

The AG of figure 2-1 builds a symbol table for all declared identifiers, and also marks identifiers that are declared more than once as erroneous. The functions *Member* and *Insert*, used in the semantic equations associated with production *p4*, are defined as part of the AG specifications; the definitions of these functions are omitted from figure 2-1. AG functions are pure functions, that is, they have no side effects. Their only arguments are constants or other attribute occurrences of the production.

A *semantic tree* is a derivation tree where each tree node additionally contains fields corresponding to the attributes of its labeling grammar symbol. The *dependency graph* of a semantic tree *T*, denoted by *D(T)*, represents functional dependencies among the attribute instances of *T*, and is defined as follows: *D(T)* is a directed graph, (V, E), where

- $V = \{$ attribute instances of T $\}$, and
- $E = \{ (a,b) \mid a, b \in V$, and a is an argument of b in a semantic equation $\}$.

The value of an attribute instance is computed according to its defining semantic equation. Before an attribute can be evaluated, all other attributes that it is functionally dependent on must

---

[1]This conversion may require duplication of semantic functions. For example, if the semantic equations associated with the production p: $X_0 \rightarrow X_1\, X_2$ are $X_1.a = f(X_0.a)$, $X_2.a = X_1.a$, the equivalent normal form version would require two invocations of the semantic function *f*, resulting in the semantic equations $X_1.a = f(X_0.a)$, $X_2.a = f(X_0.a)$.

```
Decls:  { synthesized attributes:   SymTabOut;
          inherited attributes:     SymTabIn; }

Decl:   { synthesized attributes:   SymTabOut, error;
          inherited attributes:     SymTabIn; }

Id:     { synthesized attributes:   Name;
          inherited attributes:     ∅;  }

Type:   { synthesized attributes:   TpKind;
          inherited attributes:     ∅;  }
```

(a): *Context-free Symbols of the Attribute Grammar and their Attributes*

```
p1:  Program ::=  · · ·  Decls  · · ·
         { Decls.SymTabIn = NullTbl(); }

p2:  Decls ::=  /* empty rule */
         { Decls.SymTabOut = Decls.SymTabIn; }

p3:  Decls$1 ::= Decl Decls$2
         { Decl.SymTabIn = Decls$1.SymTabIn;
           Decls$2.SymTabIn = Decl.SymTabOut;
           Decls$1.SymTabOut = Decls$2.SymTabOut; }

p4:  Decl ::= Id ':' Type ';'
         { Decl.error = Member(Decl.SymTabIn, Id.Name)
                        ? "<-- Variable already declared"
                        : "";
           Decl.SymTabOut = Insert(Decl.SymTabIn, Id.Name, Type.TpKind); }
```

(b): *Productions of the Attribute Grammar Fragment and their Semantic Equations*

Figure 2-1: *An Attribute Grammar Example*

have already received values. The functional dependencies among the attributes in the tree create a partial ordering on the attribute instances in the tree. Any attribute evaluation algorithm must obey this partial order, but since the ordering is partial, there may be more than one order of evaluating the attribute instances of the tree. A simple method to evaluate all the attributes in a semantic tree would be to carry out a topological sort of the dependency graph of the tree, evaluating each attribute as it is visited.

2.2. Incremental Attribute Evaluation for Single Subtree Replacements

The use of attribute grammars in language-based editors was originated by Demers, Reps and Teitelbaum [5]. A program is represented internally by its semantic tree. The program is modified by a sequence of pruning and grafting operations on the tree; these operations are collectively called *subtree replacement* operations. At all times during editing, an *editing cursor* points to a node of the semantic tree. A subtree replacement operation can be performed at the position in the tree indicated by the editing cursor. The cursor can be moved from a node to its parent, or from a node to its child, by cursor movement operations.

After a subtree replacement, some attributes may become inconsistent. An attribute is *inconsistent* if its value is not equal to its semantic function applied to the current values of its arguments. An incremental attribute evaluator reevaluates the inconsistent attributes, thus reestablishing consistency among the attributes in the tree.

The problem of incremental attribute evaluation for single subtree replacements can be stated as follows. Starting from a consistently attributed tree T, a subtree S of T is replaced by another tree, S', which is also consistently attributed. Let T' be the tree T with S replaced by S'. Replacement of subtree S with root node r by a subtree S' with root node r' consists of pruning the subtree S from the tree, assigning the inherited attributes of r' to the inherited attributes of r, and grafting the subtree S' onto r. Nodes r and r' must be labeled with the same nonterminal symbol. Initially, the only inconsistent attributes are those associated with the root of the replaced subtree.

One approach to incrementally evaluating the attributes after such a subtree replacement would be to construct the subset of the dependency graph reachable from the attributes of r and then perform a topological sort on this subset, evaluating each attribute as it is visited. A small optimization would be to stop traversing a path when an attribute's value does not change.

This algorithm is non-optimal. In order for an incremental evaluation algorithm to be optimal, both the number of semantic function applications and the other work performed by the algorithm must be proportional to the size of the set *AFFECTED*, where *AFFECTED* is the set of attributes whose values differ in T and T'. The above algorithm is not optimal because it includes in the subset dependency graph all attributes directly or transitively dependent on the attributes of r, even those that are not in *AFFECTED*.

An optimal solution to this problem for the general class of noncircular AGs was devised by Reps [26] and elaborated by Reps, Teitelbaum and Demers [27]. This algorithm constructs a scheduling graph, different from the subset dependency graph described above, whose size is proportional to the size of *AFFECTED*. The trick is to construct this scheduling graph on the fly, avoiding including in the scheduling graph those parts of the dependency graph emanating from attributes whose values do not change. The result is as if an oracle had provided the scheduling graph a priori, and attribute evaluation had proceeded in topological order on this graph. We refer to this optimal algorithm in the sequel as the RTD algorithm.

After a subtree replacement at node r, the RTD algorithm starts by evaluating the attributes associated with r. The problem is to determine in what order the attributes of r should be evaluated. It is not sufficient to consider just the direct dependencies among the attributes of r in the two production instances where the node r appears. The reason is that although there may not be any direct dependencies among these attributes, they may be linked through a chain of dependencies arbitrarily far down the subtree rooted at r, or in the tree above the node r. (See, for example, the dependencies between the attributes *SymTabIn* and *SymTabOut* of *Decls* in figure 2-1.) For this reason, the scheduling algorithm for determining which attribute should be evaluated next must take into account transitive dependencies. For the general class of noncircular AGs, it is not possible to determine the transitive dependencies among the attributes of a nonterminal symbol from a static analysis of the AG.

For the single subtree replacement case, the RTD algorithm uses characteristic graphs to keep track of transitive dependencies among the attributes of a nonterminal occurrence in the tree. A *characteristic graph* is a directed graph, $G = (V, E)$, where V consists of the attribute instances associated with a nonterminal occurrence in the semantic tree, and an edge (v, w) is in E, where v and w are in V, and there is a path from v to w in the dependency graph of the semantic tree that does not go through any other attributes in V. A *subordinate* characteristic graph of a node r, denoted by $r.C$, only considers dependencies in the subtree rooted at r. A *superior* characteristic graph of a node r, denoted by $r.\overline{C}$, only considers dependencies in the upper tree context of r, that is, the semantic tree with the subtree rooted at r pruned.

The following graph operations, union ($\cup$), deletion ($-$), and projection ($/$), are used in the RTD algorithm:

- Given directed graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, which may or may not be disjoint, the *union* of G_1 and G_2 is defined as:

  $$G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2)$$

- The *deletion* of G_2 from G_1 is defined as:

  $$G_1 - G_2 = (V_1, E_1 - E_2)$$

- Given a directed graph $G = (V, E)$ and a set of vertices $V' \subseteq V$, the *projection* of G onto V' is defined as:

  $$G / V' = \{(v, w) \mid v, w \in V' \text{ and there exists a path from } v \text{ to } w \text{ in } G \text{ that does not contain any elements of } V'\}.$$

Figure 2-2 shows the RTD algorithm for reevaluating a semantic tree T after a subtree replacement at r has occurred (this version of the RTD algorithm was taken from Reps' thesis [28]). It makes use of two central data structures: (1) a model M, which is the scheduling

```
procedure propagate (T: semantic tree; r: nonterminal node of T at root of replaced subtree);
declare
    M: a directed graph;
    S, NeedToBeEvaluated: sets of attribute instances;
    b, c: attribute instances;
    changed: Boolean;
    OldValue, NewValue: attribute values;
begin
    M := r.C ∪ r.C̄ ;
    S := the set of vertices of M with in-degree 0 in M;
    NeedToBeEvaluated := the set of vertices of M;
    while S ≠ ∅ do
        Select and remove a vertex b from S;
        changed := false
        if b ∈ NeedToBeEvaluated then
            Remove b from NeedToBeEvaluated;
            OldValue := value of b;
            evaluate b;
            NewValue := value of b;
            if OldValue ≠ Newvalue then
                changed := true;
                if M does not contain all the successors of b in D(T) then
                    expand(M, b, S)
                fi
            fi
        fi
        while there exists c, a successor of b in M do
            Remove edge (b, c) from M;
            if in-degree of c in M is 0 then Insert c into S fi
            if changed = true then Insert c into NeedToBeEvaluated fi
        od
    od
end
```

Figure 2-2: *RTD Incremental Attribute Evaluation Algorithm*

graph described above, containing attributes that need reevaluation and direct and transitive dependency edges among them, and (2) a worklist *S*, which contains those attributes in the model that are ready to be evaluated (*i.e.*, their arguments have already been evaluated, or do not need to be reevaluated). The model *M* represents direct and transitive dependencies of a connected region of the derivation tree. Initially, *M* covers the the root of the replaced subtree, *r*, that is, it contains the attributes associated with *r* and their direct and transitive dependencies. These transitive dependencies are obtained from the characteristic graphs associated with *r*. Those attributes in *M* that have no incoming edges are placed in the worklist *S*.

Attributes are removed from S and evaluated until S is empty. When an attribute is evaluated and its value changes, other attributes that depend on it may need to be brought into the model. This is performed by the *expand* procedure, shown in figure 2-3 and explained below. The attribute that was just evaluated, as well as all its outgoing edges, are then removed from the model. This may result in some additional attributes becoming ready for evaluation, which are inserted into the worklist S.

---

procedure *expand* (M: a directed graph; b: an attribute instance; S: a set of attribute instances);
declare
  c: an attribute instance;
begin
  if there exists c, a successor of b in $D(T)$ that is not in M
      and TreeNode(c) is a child of TreeNode(b) **then**
    $M := (M -$ TreeNode$(b).C) \cup$ ExpandedSubordinate(TreeNode(b));
    Insert into S all vertices of ExpandedSubordinate(TreeNode(b) whose in-degree in M is 0
  fi
  if there exists c, a successor of b in $D(T)$ that is not in M
      and TreeNode(c) is the parent or a sibling of TreeNode(b) **then**
    $M := (M -$ TreeNode$(b).\overline{C}) \cup$ ExpandedSuperior(TreeNode(b));
    Insert into S all vertices of ExpandedSuperior(TreeNode(b) whose in-degree in M is 0
  fi
end

Figure 2-3: *Expanding a Model*

The model is expanded by procedure *expand* when an attribute b is reevaluated and changes value, and b has a successor c that is not in the model. *Expand* adds to the model all the attributes of a neighboring production of the attribute b, as well as all dependency edges among them. (Thus, the model is not the same as the subset dependency graph referred to earlier because it includes all the attributes of the expansion production, not just those directly or transitively dependent on the changed attribute. This is done to simplify the modeling of transitive dependencies, as explained in Reps' thesis [28].)

Let p denote the production $X_0 \rightarrow X_1 \cdots X_n$, and $D(p)$ the direct dependencies of the production p. If b is associated with X_0 (so c is associated with some child X_i, $1 \le i \le n$, of X_0), then the model is expanded <u>downwards</u> by production p to include all the attributes of p. If b is associated with X_i, $1 \le i \le n$ (so c is associated with the parent node X_0 or a sibling node X_j, $1 \le j \le n$, $j \ne i$), then the model is expanded <u>upwards</u> by production p. Direct and transitive dependency edges among the attributes of production p are added to the model, using the two functions *ExpandSubordinate* and *ExpandSuperior* defined below.

$$ExpandSubordinate(X_0) \equiv D(p) \cup X_1.C \cup \cdots \cup X_n.C$$

$$ExpandSuperior(X_i) \equiv D(p) \cup X_0.\overline{C} \cup X_1.C \cup \cdots \cup X_{i-1}.C \cup X_{i+1}.C \cup \cdots \cup X_n.C$$

We refer to the modeled region of the derivation tree as the *backbone* of the model. The backbone is a rooted subgraph of the tree; it includes the root node of the replaced subtree and all production instances by which the model was expanded. The attribute instances in the model are associated with the tree nodes in the backbone. We call the root node of the backbone the *apex*, and the leaves of the backbone the *frontier*. The root and leaf nodes of the backbone do not necessarily correspond to the root and leaf nodes of the derivation tree. Edges between attributes associated with the apex of the model represent superior transitive dependencies, edges between attributes of frontier nodes represent subordinate transitive dependencies, and edges anywhere else represent direct dependencies.

Since the model is expanded by an entire production instance, an attribute might be added to the model, and eventually evaluated, even if none of its arguments changes. To avoid this, a third data structure, *NeedToBeEvaluated*, is used in the incremental evaluation algorithm of figure 2-2 to keep track of the set of attributes in the model that depend on one or more changed attributes. When an attribute is selected for evaluation, it is actually evaluated only if it is in the set *NeedToBeEvaluated*.

The RTD algorithm does not require both superior and subordinate characteristic graphs to be maintained at each node in the semantic tree. After a subtree replacement at a node *r*, PROPAGATE never needs subordinate characteristic graphs for nodes on the path from *r* to the root of the tree, and it never needs superior graphs for any node that is not on the path from *r* to the root. The RTD algorithm assumes the ‘‘prepared for propagation’’ invariant on the semantic tree representing the program being edited, which is defined as follows:

- The nonterminal node where the editing cursor is placed is labeled with both its superior and subordinate characteristic graphs.
- Each node on the path from the cursor to the root of the tree is labeled with its superior characteristic graph.
- Each node that is not on the path from the cursor to the root of the tree is labeled with its subordinate characteristic graph.

The prepared for propagation invariant is reestablished after subtree replacement operations and when the cursor is moved. The functions *ComputeSubordinate* and *ComputeSuperior* compute the subordinate and superior characteristic graphs of a semantic tree node, respectively, and are defined as follows for nodes in the production $p: X_0 \to X_1 \cdots X_n$.

$$ComputeSubordinate(X_0) \equiv ExpandSubordinate(X_0) / A(X_0)$$

$$ComputeSuperior(X_i) \equiv ExpandSuperior(X_i) / A(X_i)$$

For a given grammar, the functions *ComputeSubordinate* and *ComputeSuperior* each have unit cost.

Thus the only non-unit cost is the actual evaluation algorithm. Only those attributes that depend on changed attributes are re-evaluated, and expansions are performed only when an attribute changes value. Since expansions have constant cost for a given attribute grammar, as does the number of attributes dependent on another, the total work performed by the RTD algorithm is proportional to the size of *AFFECTED*.

3. Incremental Evaluation for Multiple Asynchronous Subtree Replacements

There are m editing cursors, $m \geq 1$, pointing to nodes of the semantic tree. Each cursor can be moved independently of the other cursors from a node to the node's parent or child. Subtree replacement operations can be performed independently and asynchronously at any position in the tree indicated by an editing cursor. This can result in k concurrent evaluation processes, $k \geq 1$.

We state the problem of incremental attribute evaluation for the case when k is 2. Let T be a consistently attributed semantic tree, T' the resulting tree after subtree S in T is replaced by S', and T'' the resulting tree after subtree R in T' is replaced by R', where the subtrees S' and R' are also consistently attributed. The two modifications at S and R are *asynchronous*, that is, the second subtree replacement occurs while the evaluation initiated by the first one is still in progress. The problem is to reestablish attribute consistency in T''.

In our solution, the k evaluation processes proceed independently while they cover disjoint parts of the semantic tree, where each process uses a variation of the RTD algorithm for single subtree replacement. A separate model is maintained for each independent evaluation process to schedule inconsistent attribute instances for reevaluation. As a model changes, either when it is first initialized or when it is expanded, it may cover a semantic tree node that is already included in another model arising from a different subtree replacement that is being evaluated concurrently; this is called a *collision*. When this happens, the colliding models are *merged* into one, thereby combining the corresponding evaluation processes that had been proceeding independently prior to the collision, and evaluation continues with the merged model.

As in the single subtree replacement case, characteristic graphs are employed to efficiently initialize and expand the models used by the independent evaluation processes. Because there are multiple cursors, however, it is necessary to modify the RTD ‘‘prepared for propagation’’ invariant as follows. Let *LCA* be the lowest common ancestor of the m editing cursors. Then:

- Each node on the path from the *LCA* (inclusive) to the root of the tree is labeled with its superior characteristic graph.
- Each node on the path from an editing cursor r_i (inclusive) to the *LCA* (not inclusive) is labeled with both its superior and subordinate characteristic graphs, where $1 \leq i \leq m$.
- Each node that is not on a path from an editing cursor to the root of the tree is

labeled with its subordinate characteristic graph.

This prepared for propagation invariant, illustrated in figure 3-1, is reestablished after every cursor movement and after every subtree replacement operation.

Figure 3-1: *Characteristic Graphs Required for Multiple Editing Cursors*

Note that for a segmented semantic tree, our algorithm is defined with respect to each segment, and not for the entire semantic tree. Within a segment, there is one (real) editing cursor for the single user making changes to that segment, and $m-1$ (virtual) editing cursors for each ‘‘interface’’ node of the segment, that is, a node on the boundary with another segment. We treat an interface node of a segment as an editing cursor because an update in some other segment of the tree is propagated to the segment via an interface node, and is handled as a (virtual) subtree replacement at that interface node. The details of the propagation across segment boundaries are outside the scope of this paper (see [16, 14, 20]).

Figures 3-2, 3-3 and 3-4 give the skeleton of our algorithm following the format of the KK algorithm, but add explicit hooks for characteristic graph update, collision detection and merging, and remove their explicit hooks for propagation across segment boundaries. We allocate a data structure for each new evaluation process, consisting of its model *M*, worklist *S* and set of attributes whose arguments have changed *NeedToBeEvaluated*. The new parts of the algorithm indicated in comments in these figures are elaborated in the following three sections. Note that *replace-subtree* executes atomically, but forks the actual evaluation to run concurrently with other evaluation processes; the body of the while loop in *asynch-propagate* is also atomic, to prevent collision and merging of models while a model is in an intermediate state.

atomic procedure *replace-subtree* (*T*: semantic tree; *r*: nonterminal node at root of replaced subtree;
r': nonterminal node at root of new subtree);
begin
Check for initialization collision; /* defined in section 6 */
Prune subtree rooted at *r*, and graft subtree rooted at r' onto *r*;
Compute new subordinate characteristic graph of *r*;
if subordinate characteristic graph of *r* changes **then**
cgraph-update(r) /* defined in section 4 */
fi
startup(T, r)
end

Figure 3-2: *Replace Subtree Operation*

procedure *startup* (*T*: semantic tree; *r*: node of *T* at root of subtree replacement);
declare
ep: an evaluation process;
begin
if collision during model initialization is detected **then**
Merge initial model with existing model /* defined in section 6 */
else
Allocate new *ep*;
ep.active := **true**; /* explained in section 6 */
ep.timestamp := current time; /* explained in section 6 */
$ep.M := r.C \cup r.\overline{C}$;
ep.S := the set of vertices of *ep.M* with in-degree 0;
ep.NeedToBeEvaluated := the set of vertices of *ep.M*;
fork *asynch-propagate*(*ep*);
fi
end

Figure 3-3: *Startup Algorithm*

4. Multiple Cursors and Characteristic Graphs

In this section, we examine the two types of events that affect the prepared for propagation invariant for multiple editing cursors: cursor movement operations and subtree replacements. When an editing cursor is moved to a neighboring node in the semantic tree, characteristic graphs may have to be computed or deleted to maintain the prepared for propagation invariant; this issue is the topic of section 4.1. A subtree replacement at one of the editing cursors may invalidate characteristic graphs in other parts of the semantic tree; algorithms to recompute the necessary characteristic graphs after a subtree replacement are described in section 4.2.

```
procedure asynch-propagate ( ep: an evaluation process );
declare
    b, c: attribute instances;
    changed: Boolean;
    OldValue, NewValue: attribute values;
begin
    while ep.S ≠ ∅ and ep.active do
        atomic begin
            Select and remove a vertex b from ep.S;
            changed := false
            if b ∈ ep.NeedToBeEvaluated then
                Remove b from ep.NeedToBeEvaluated;
                OldValue := value of b;
                evaluate b;
                NewValue := value of b;
                if OldValue ≠ Newvalue then
                    changed := true;
                    if ep.M does not contain all the successors of b in D(T) then
                        asynch-expand(ep, b) /* defined in sections 5 and 6 */
                    fi
                else
                    antifreeze(ep, b) /* defined in section 6 */
                fi
            fi
            while there exists c, a successor of b in ep.M do
                Remove edge (b, c) from ep.M;
                if in-degree of c in ep.M is 0 then Insert c into ep.S fi
                if changed = true then Insert c into ep.NeedToBeEvaluated fi
            od
        atomic end
    od
    if ep.active then
        clear-model-field(apex of ep.M) /* defined in section 5 */
    fi
    Deallocate ep;
end
```

Figure 3-4: *Propagate Algorithm for Asynchronous Subtree Replacements*

4.1. Effect of Cursor Movement on Characteristic Graphs

Every cursor movement can be broken up into a sequence of two kinds of cursor movement operations: (1) *DescendToChild*(r_i, j) moves cursor r_i to the j^{th} child of r, and (2) *AscendToParent*(r_i) moves cursor r_i to its parent. We assume that the cursor movement operations, *DescendToChild* and *AscendToParent*, are atomic.

The actions that must be performed to reestablish the prepared for propagation invariant when an editing cursor r_i is moved depends on whether (1) r_i is the *LCA*, (2) r_i is not the *LCA* but is the ancestor of another editing cursor r_j, $1 \le j \le m$, $j \ne i$, or (3) r_i is not the ancestor of any other editing cursor r_j. These three cases are mutually exclusive when there are multiple editing cursors (*i.e.*, $m > 1$).

To maintain the prepared for propagation invariant, *DescendToChild*(r_i, j) performs the following actions:

- Case ($r_i = LCA$):
  - a. Compute new *LCA*.
  - b. If new *LCA* is not equal to r_i (*i.e.*, new *LCA* is equal to j^{th} child of r_i), delete subordinate characteristic graph of r_i.
- Case ($\exists\, j$, $1 \le j \le m$, $j \ne i$, r_i is ancestor of r_j):
  - a. Do nothing (superior characteristic graph of j^{th} child of r_i is already available).
- Case ($\forall\, j$, $1 \le j \le m$, $j \ne i$, r_i is not ancestor of r_j):
  - a. Compute superior characteristic graph of j^{th} child of r_i using function *ComputeSuperior* defined in section 2.2.

AscendToParent(r_i) reestablishes the prepared for propagation invariant as follows:

- Case ($r_i = LCA$):
  - a. Set new *LCA* to be parent of r_i.
  - b. Compute subordinate characteristic graph of parent of r_i using function *ComputeSubordinate* defined in section 2.2.
- Case ($\exists\, j$, $1 \le j \le m$, $j \ne i$, r_i is ancestor of r_j):
  - a. Do nothing (superior characteristic graph of r_i still needed).
- Case ($\forall\, j$, $1 \le j \le m$, $j \ne i$, r_i is not ancestor of r_j):
  - a. Delete superior characteristic graph of r_i.

The actions performed by the cursor movement operations to reestablish the prepared for propagation invariant depend on whether or not an editing cursor is an ancestor of another cursor, and may entail the recalculation of *LCA*. We now describe how this can be done efficiently so that for a given grammar, each cursor movement operation has unit cost.

We maintain an ‘‘edge-color’’ invariant on the semantic tree, which is defined as follows:

- Each semantic tree edge on the path from an editing cursor to the *LCA* is colored blue.
- Each edge that is not on a path from an editing cursor to the *LCA* is colored white.

Given the edge-color invariant, we can determine whether a semantic tree node *n* is a (proper) ancestor of an editing cursor by checking whether there is a blue edge from *n* to any child of *n*. Furthermore, we can easily recompute the *LCA* when an editing cursor r_i that is positioned at the *LCA* is moved to a child *c* — if there is a blue edge from r_i to any sibling of *c*, then *LCA* stays the same, otherwise *LCA* is set to *c*.

The edge-color invariant must be reestablished after each cursor movement operation. When an editing cursor r_i is moved to a child *c*, there are two situations when the color of the edge (r_i, c) must be changed. (1) If r_i is not an ancestor of any other editing cursor, then the (previously white) edge (r_i, c) is colored blue. (2) If r_i is positioned at the *LCA*, and moving r_i to *c* changes *LCA* to *c*, then the (previously blue) edge (r_i, c) is colored white. The part of *DescendToChild*(r_i, j) that maintains the edge-color invariant is defined as:

let $c = j^{\text{th}}$ child of r_i **in**
    if (r_i = *LCA*) **and** (there is no blue edge from r_i to a child *k* of r_i, where $k \neq j$) **then**
        LCA := *c*;
        Color edge (r_i, c) white
    else
        if there is no blue edge from r_i to any child of r_i **then**
            Color edge (r_i, c) blue
        fi fi ni

Reestablishing the edge-color invariant when an editing cursor r_i is moved to its parent *p* is similar. Again, there are two situations when the color of the edge (p, r_i) must be changed. (1) If r_i is not an ancestor of any other editing cursor, then the (previously blue) edge (p, r_i) is colored white. (2) If r_i is positioned at the *LCA*, then the (previously white) edge (p, r_i) is colored blue. The part of *AscendToParent*(r_i) that maintains the edge-color invariant is defined as:

let *p* = parent of r_i **in**
    if (r_i = *LCA*) **then**
        LCA := *p*;
        Color edge (p, r_i) blue
    else
        if there is no blue edge from r_i to any child of r_i **then**
            Color edge (p, r_i) white
        fi fi ni

Putting this all together, we obtain an efficient (unit cost) algorithm for reestablishing the prepared for propagation invariant when a cursor is moved. The resulting algorithms for the two kinds of cursor movement operations are shown in figure 4-1.

DescendToChild(r_i, j):
let c = j^{th} child of r_i **in**
if ($r_i = LCA$) **and** (there is no blue edge from r_i to a child k of r_i, where $k \neq j$) **then**
$LCA := c$;
Color edge (r_i, c) white;
$r.C :=$ **null**
else
if there is no blue edge from r_i to any child of r_i **then**
$c.\overline{C} :=$ ComputeSuperior(c);
Color edge (r_i, c) blue
fi
fi
ni

AscendToParent(r_i):
let p = parent of r_i **in**
if ($r_i = LCA$) **then**
$LCA := p$;
Color edge (p, r_i) blue;
$p.C :=$ ComputeSubordinate(p)
else
if there is no blue edge from r_i to any child of r_i **then**
Color edge (p, r_i) white;
$r_i.\overline{C} :=$ **null**
fi
fi
ni

Figure 4-1: *Reestablishing Prepared for Propagation Invariant for Multiple Cursors*

4.2. Updating Characteristic Graphs after Subtree Replacement

A subtree replacement at one of the editing cursors may invalidate characteristic graphs that may be needed for evaluations initiated by subsequent, or previously started but still in progress, evaluation processes. Consider the tree illustrated in figure 4-2 (a), with two editing cursors at the nodes labeled X and Y. The superior and subordinate characteristic graphs of all nodes in the tree contain two vertices, attributes a and b, and no edges. If the subtree whose root is Y is replaced by a new subtree in which there is a direct dependency between the attributes a and b of Y, as shown in figure 4-2 (b), the superior characteristic graphs of the nodes on the path from X to the root should now contain an edge from b to a (and from a to b for the subordinate graphs on the path from Y to the root).

(a) **(b)**

Figure 4-2: *Changing Transitive Dependencies*

Consider the possibility that the characteristic graphs are not updated after subtree replacements. If a subtree replacement at node X updated the value of X's attribute b, this change might not be reflected consistently in all attributes depending on b because the model does not have any transitive dependency edges from b to a for nodes on the path from X to the root (and from a to b for nodes on the path from Y to the root). The consequence might be that the a attributes may be evaluated before the b attributes for the left path and *vice versa* for the right path. Because the model is supposed to prevent an attribute from being assigned a value that is not its final value, once an attribute has been removed from the model and evaluated, it is never considered for evaluation again. Thus, some of the attributes (in particular, the a attributes on the left path, and the b attributes on the right path), may be incorrect when evaluation terminates.

Let $path_1, \ldots, path_m$ denote the m paths from the editing cursors $r_1, \ldots, r_m$ to *LCA*. Note that these paths are not necessarily node disjoint, and it is possible for a path to have zero length in the case when the *LCA* is itself pointed to by one of the editing cursors. In general, a subtree replacement at a cursor r_i, $1 \leq i \leq m$, may invalidate subordinate characteristic graphs associated with nodes on $path_i$, and superior characteristic graphs associated with those nodes on paths $path_j$ that are disjoint from the nodes on $path_i$, where $1 \leq j \leq m$ and $j \neq i$.

An algorithm that updates the necessary characteristic graphs after a subtree replacement is shown in figure 4-3. This procedure is invoked when a subtree replacement at a node r occurs, before the evaluation process is initiated. Note that this procedure only needs to be called if the subtree replacement changes the subordinate characteristic graph of r. *cgraph-update* executes atomically, since it is called from within the atomic procedure *replace-subtree*.

To explain procedure *cgraph-update*, we recall the definition of the functions *ComputeSubordinate* and *ComputeSuperior* from section 2.2. The subordinate characteristic graph of a node n is used in the computation of the subordinate characteristic graph of the parent of n, and in the computation of the superior characteristic graphs of siblings of n. Therefore, if

```
        procedure cgraph-update ( r: semantic tree node at root of subtree replacement );
        declare
            node, s: semantic tree nodes;
            new_C: characteristic graph;
            changed: boolean;
        begin
[1]         if (r ≠ LCA) then
[2]             node := parent(r);
[3]             changed := true;
[4]             while (node ≠ LCA) and (changed) do
[5]                 new_C := ComputeSubordinate(node);
[6]                 if (node.C = new_C) then changed := false
[7]                 else
[8]                     node.C := new_C;
[9]                     for each sibling s of node do
[10]                        superior-update(s);
[11]                    od
[12]                    node := parent(node)
[13]                fi
[14]            od
[15]            if (changed) and (LCA is pointed to by an editing cursor) then
[16]                node.C := ComputeSubordinate(node)
                fi
            fi
        end /* of cgraph-update */
```

Figure 4-3: *Algorithm to Update Incorrect Characteristic Graphs*

the subordinate characteristic graph of *n* changes, these characteristic graphs must be recomputed. The superior characteristic graph of a node *n* is used in the computation of the superior characteristic graphs of children of *n*, which must be recomputed if the superior characteristic graph of *n* changes.

Procedure *cgraph-update* is driven by a new value for the subordinate characteristic graph of *node*, computed in line [5]. The superior characteristic graphs of siblings of *node* are updated by a call to the procedure *superior-update*, shown in figure 4-4 and explained below. The subordinate characteristic graph of the parent of *node* is recomputed in the next iteration of the while loop. This process is repeated for each node on the path from the root of the subtree replacement to the *LCA*, until the subordinate characteristic graph of a node does not change (line [6]), or the *LCA* is reached.

The *LCA* is treated as a special case (lines [15]-[16]). If the *LCA* is not pointed to by an editing cursor, then it is not labeled with a subordinate characteristic graph, and therefore nothing need be done. If the *LCA* is pointed to by an editing cursor, then its subordinate characteristic graph

procedure *superior-update* (*node*: semantic tree node);
declare
    queue: FIFO list of semantic tree nodes;
    c, *p*: semantic tree nodes;
    $new\_\overline{C}$: characteristic graph;
begin
[1]   Initialize *queue* to contain *node*;
[2]   **while** (*queue* is not empty) **do**
[3]   Remove node *p* from front of *queue*;
[4] $new\_\overline{C}$:= ComputeSuperior(*p*);
[5]   **if** ($p.\overline{C} \neq new\_\overline{C}$) **then**
[6] $p.\overline{C}$:= $new\_\overline{C}$;
[7]   **for each** child *c* of *p* **do**
[8]   **if** edge (p, c) is colored blue **then** Insert *c* into *queue* **fi**
            od
        fi
    od
end

Figure 4-4: *Characteristic Graph Update (cont.)*

must be recomputed if the subordinate characteristic graph of a child of *LCA* changed. However, the siblings of *LCA* are not labeled with their superior characteristic graphs, and therefore no superior characteristic graphs need to be updated as a result of a change to the subordinate characteristic graph of *LCA*.

Procedure *superior-update*, shown in figure 4-4, recomputes the superior characteristic graph of a node *n*, and if necessary, the superior characteristic graphs of the descendents of *n*. It is necessary to recompute the superior characteristic graph of a descendent node if (1) the superior characteristic graph of *n* changes, and (2) the descendent node is labeled with its superior characteristic graph (*i.e.*, the descendent is on the path from an editing cursor to the *LCA*). The edge-color invariant is used to determine which descendent nodes are labeled with their superior characteristic graphs. The descendents of the node *n* are visited in a breadth-first manner. Traversal along a path terminates either when the superior characteristic graph of a node does not change (line [5]), or when a node that is not labeled with a superior characteristic graph is reached (line [8]).

5. Detecting Collisions

As mentioned earlier in section 3, two models are involved in a collision if they share a semantic tree node (*i.e.*, the corresponding backbones are not node disjoint). There are two cases when models collide with each other in the course of attribute evaluation:

- During initialization of one of the models.
- During expansion of one of the models.

After a subtree replacement at a node r, the initial model created for the new evaluation process contains the attribute instances associated with node r and dependencies among them. The backbone of the initial model consists of the single node r. An *initialization collision* occurs if node r is in the backbone of another model used by a different evaluation process.

Figure 5-1: *Scenario for Collisions during Model Expansion*

A model (and its associated backbone) can be expanded either downwards from a frontier node, or upwards from the apex. Consider the semantic tree illustrated in figure 5-1, where the backbones of two models M_1 and M_2 are shown. If M_1 is expanded downwards at node X, then the backbone for M_1 grows to include the production instance that applies at X. This production instance includes the node Y, which is the apex of the backbone of M_2. The result is a *downward collision* at node Y. In general, a downward expansion of the model at a node X_0, where the production that applies at X_0 is $X_0 \rightarrow X_1 \cdots X_n$, can have up to n downward collisions at the children of X_0.

A dual case arises if the model M_2 is expanded upwards from node Y, causing an *upward collision* at node X. Generally, an upward expansion of the model at a node X_i, $1 \leq i \leq n$, by production $X_0 \rightarrow X_1 \cdots X_n$, can have an upward collision at the parent X_0 of the expansion node X_i, and up to $n - 1$ downward collisions at the siblings of X_i.

To detect collisions, we add to each semantic tree node additional information indicating whether the node is at the apex, interior or frontier of the backbone of some model. This information is stored in the *in-model* field of each semantic tree node. The *in-model* field of a node *n* is equal to

- *apex/frontier* — if *n* is both the apex and a frontier node of the backbone of a model (*i.e.*, *n* is the backbone of an initial model).
- *apex* — if *n* is the apex of the backbone of a model.
- *interior* — if *n* is an interior node of the backbone of a model.
- *frontier* — if *n* is on the frontier of the backbone of a model.
- *not-in-model* — if *n* is not in the backbone of any model.

An initialization collision is detected if a subtree replacement at *r* occurs, and the *in-model* field of *r* is not equal to *not-in-model*. Initialization collisions must be detected before the old subtree is pruned from the tree. The reason is that some cleanup operations must be performed for the part of the existing model that covered the deleted subtree. (The details are explained later in section 6.)

Procedure *asynch-expand*, shown in figure 5-2, expands a model and also detects collisions during model expansion. If the model is expanding downwards at a node *n*, then each child of *n* is checked to determine whether the child node is an apex of another model, in which case a downward collision is detected. If the model is expanding upwards from *n*, then an upward collision is detected at the parent of *n* if the parent node is a frontier node of another model, and a downward collision is detected at a sibling of *n* if the sibling is an apex of another model. The elided parts of procedure *asynch-expand* merge the colliding models, and are elaborated in section 6. Recall that *asynch-expand* is called from within the critical section of *asynch-propagate* and is therefore executed atomically.

When a model is expanded by a production *p*, the *in-model* field of semantic tree nodes of *p* must be updated, as shown in procedure *asynch-expand*. The *in-model* field of nodes in the backbone of a model are cleared when the corresponding evaluation process terminates. This is accomplished by the atomic procedure *clear-model-field*, shown in figure 5-3. This procedure clears the *in-model* field of semantic tree nodes in the model's backbone during a depth-first traversal of the backbone, started from the backbone's apex.

6. Merging Models

After a collision is detected, the colliding models are ‘‘merged’’ into one. In section 6.1, we discuss the problem of merging two colliding models for the case of a downward collision caused by a downward expansion of a model. We give the full algorithm in section 6.2. Merging models subsequent to an upward collision caused by an upward expansion of a model is similar,

```
procedure asynch-expand ( ep: an evaluation process; b: an attribute instance );
declare
   c: an attribute instance;
begin
   if there exists c, a successor of b in D(T) that is not in ep.M
         and TreeNode(c) is a child of TreeNode(b) then
    /* downward expansion */
    for each child of TreeNode(b) do
         if (child.in-model = apex) or (child.in-model = apex/frontier) then
              /* downward collision: merge colliding models */
                    . . .
              child.in-model := interior
         else
              child.in-model := frontier
         fi
    od
    TreeNode(b).in-model := interior;
    /* expand ep.M */
          . . .
   fi

   if there exists c, a successor of b in D(T) that is not in ep.M
         and TreeNode(c) is the parent or a sibling of TreeNode(b) then
    /* upward expansion */
    if (parent(TreeNode(b)).in-model = frontier) or (parent(TreeNode(b)).in-model = apex/frontier) then
         /* upward collision: merge colliding models */
               . . .
         parent(TreeNode(b)).in-model := interior
    else
         parent(TreeNode(b)).in-model := apex
    fi
    for each sibling of TreeNode(b) do
         if (sibling.in-model = apex) or (sibling.in-model = apex/frontier) then
              /* downward collision: merge colliding models */
                    . . .
              child.in-model := interior
         else
              child.in-model := frontier
         fi
    od
    TreeNode(b).in-model := interior;
    /* expand ep.M */
          . . .
   fi
end
```

Figure 5-2: *Algorithm to Expand Model and Detect Collisions*

```
atomic procedure clear-model-field ( node: semantic tree node );
begin
    if (node.in-model ≠ frontier) then
        for each child c of node do
            clear-model-field(c)
        od
    fi
    node.in-model := not-in-model
end
```

Figure 5-3: *Clearing in-model Field When Evaluation Process Terminates*

and therefore omitted. Merging two models following an initialization collision is discussed in section 6.3.

6.1. Merging Problem

Figure 6-1 (a) illustrates a downward collision resulting from a downward expansion. Model M_1 expands downwards at node X_0 by production p: $X_0 \rightarrow X_1 \cdots X_n$, causing a collision with model M_2 at node X_i, where $1 \le i \le n$. If M_2 was not in the picture, the expansion would add to model M_1 the attributes of X_i, all direct edges between attributes of X_i and other attributes in p, and subordinate transitive edges among attributes of X_i (and similarly for each sibling of X_i). Let M_1^+ denote the expanded model M_1.

(a): *After expansion of M_2 at X_0 by production $X_0 \rightarrow X_1 \cdots X_n$*

(b): *Dependencies across colliding models*

Figure 6-1: *A Downward Collision due to a Downward Expansion*

Model M_2 contains those attributes of X_i that have not yet been evaluated, as well as superior transitive edges among them. To merge M_1^+ and M_2, all that may seem necessary is to delete the subordinate and superior transitive edges among attributes of X_i from M_1^+ and M_2 respectively, and then unite the two resulting graphs, where the union operation removes duplicate vertices and edges.

This simple approach to merging does not work, however, when there is an attribute *a* within the backbone of one of the colliding models, for instance, M_1, that depends on an attribute *b* in the other colliding model, M_2, and *a* has already been evaluated and removed from M_1. The attribute *a* may be affected, through *b*, by the subtree replacement corresponding to the model M_2. But with the simple merging scheme, *a* will not be evaluated again since no future expansions of the merged model will reinsert *a*. (Recall that a model is only expanded from the apex and frontier nodes.) This is clearly wrong — the semantic tree will still have inconsistent attributes after all evaluations terminate.

We solve this problem by reinserting all such attributes back into the model, together with the direct dependency edges needed to link them up to attributes still in the model. This is accomplished by the *freeze* routine, shown in figure 6-2. *Freeze* performs a search of the dependency graph within the backbone of a model, looking for dependent attributes that should be reinserted into the model. However, the search space can be reduced because of the following observation. Referring again to the case illustrated in figure 6-1, the subtree replacement corresponding to the model M_1 can only affect attributes in M_2 through the inherited attributes associated with the node X_i. Conversely, the subtree replacement corresponding to M_2 can only affect attributes in M_1 through the synthesized attributes associated with the node X_0.[2] Thus, for the example of figure 6-1, *freeze* is called for each synthesized attribute of X_0 in model M_1, and for each inherited attribute of X_i in model M_2.

Freeze takes two arguments, an evaluation process *ep* and an attribute instance *a*. *Freeze* performs a depth-first traversal of the dependency graph in the region of the semantic tree bounded by the backbone of the model *M* of *ep*, starting from *a*. Traversal along a path from *a* stops at an attribute *x* if and only if one of the following three conditions hold:

1. Attribute *x* is in the model *M* (lines [1]-[4] in figure 6-2). In this case, we have gone as far as we need to go along this path.

2. Attribute *x* has already been visited (line [8]). In this case, there is no need to continue past *x* since the rest of the path has been traversed already.

3. Attribute *x* is on the boundary of *M*. In this case, continuing along this path leads outside the backbone of *M*. (When this condition occurs, the value of *adjacent_attrs* is the empty set, for reasons which become clear in the next

[2]The proof of this observation follows directly from the definition of ‘‘synthesized’’ and ‘‘inherited’’.

```
        procedure freeze ( ep: an evaluation process; a: an attribute instance );
        declare
            adjacent_attrs: set of attribute instances;
            b: attribute instance;
        begin
[1]         if a ∈ ep.M then
[2]             if in-degree of a in ep.M is 0 then
[3]                 Remove a from ep.S
                fi
[4]             return
            fi
[5]         Mark a visited;
[6]         adjacent_attrs := find-adjacent-attrs(a);
[7]         for each attribute instance b in adjacent_attrs do
[8]             if b is not visited then
[9]                 freeze(ep, b)
                fi
[10]            Add a and red edge (a, b) to ep.M
            od
        end /* of freeze */

        function find-adjacent-attrs ( a: attribute instance ) returns set of attributes instances;
        declare
            adjacent_attrs: set of attribute instances;
        begin
[1]         if (TreeNode(a).in-model = apex/frontier) then
[2]             adjacent_attrs := transitive-transitive(a)
[3]         else if (TreeNode(a).in-model = apex) then
[4]             adjacent_attrs := transitive-direct(a)
[5]         else if (TreeNode(a).in-model = frontier) then
[6]             adjacent_attrs := direct-transitive(a)
[7]         else /* TreeNode(a) is an interior node */
[8]             adjacent_attrs := direct-direct(a)
            fi
[9]         return(adjacent_attrs);
        end /* of find-adjacent-attrs */
```

Figure 6-2: *Search for Attributes to be Reinserted into a Colliding Model* paragraph, causing the invocation of *freeze* with x to terminate.)

The procedure *find-adjacent-attrs*, also shown in figure 6-2, computes the set of attribute instances that are immediate successors of a within the region of the semantic tree bounded by the backbone of M. An *immediate successor* of a is an attribute b such that the edge (a, b) is in the model, or was in the model at some point but has been removed. Determining the immediate successors of a depends on whether a is at the apex, frontier, or interior of model M's backbone: In the case of attributes at the apex or frontier, M contains (or contained) transitive dependency edges as well as direct dependency edges, whereas for attributes at an interior node, only direct dependency edges are (or were) in the model.

The four functions that are invoked in *find-adjacent-attributes* to determine the immediate successors of an attribute a, depending on the position of a in the model's backbone, are defined as follows:

- *transitive-transitive*: Returns attributes that are successors of a because of a transitive dependency superior to the semantic tree node that a is associated with, or because of a transitive dependency subordinate to the semantic tree node of a.

- *transitive-direct*: Returns attributes that are successors of a because of a transitive dependency superior to the semantic tree node that a is associated with, or because of a direct dependency subordinate to the semantic tree node of a.

- *direct-transitive*: Returns attributes that are successors of a because of a direct dependency superior to the semantic tree node of a, or because of a transitive dependency subordinate to the semantic tree node of a.

- *direct-direct*: Returns attributes that are successors of a because of a direct dependency superior to, or subordinate to, the semantic tree node of a.

These functions use the characteristic graphs of the semantic tree node that a is associated with, and the direct dependencies of the two production instances that apply at the tree node of a.

After a recursive invocation of *freeze* with attribute instance b returns, the attribute a and the edge (a, b) are added to the model M (line [10]). As each recursive call returns, line [10] reinserts into M all attributes reachable from the attribute instance that *freeze* was initially called with that were not in M, as well as all edges between them. These edges are colored red to differentiate them from other model edges, which have no color, for reasons to be explained below.[3]

If *freeze* adds an edge (x, y) to M, where x is an attribute reinserted by *freeze* and y was in M before *freeze* was invoked, the attribute y may be in the worklist S of *ep*. In this case, y must be removed from S since it is no longer ready for evaluation (lines [2]-[3]).

---

[3]The blue and white edges of section 4.1 are derivation tree edges, not model edges as in this case.

The reason for coloring the edges that are reinserted into the model by *freeze* is to avoid reevaluating a reinserted attribute whose predecessor does not change in value. Consider again the downward expansion of M_1 causing a collision with M_2 as discussed at the beginning of this section. Suppose that a whole chain of edges and attributes were reinserted between some attribute c associated with X_0 and a successor d of c that was in M_1 at collision. Recall that this means that all attributes along the path between c and d have already been in M_1 once, reevaluated, and removed. If an attribute e along this chain is reevaluated but its value does not change, none of the attributes between e and d need to be evaluated again, and therefore should be removed from the model as soon as e is reevaluated.

This is accomplished by procedure *antifreeze*, shown in figure 6-3. It is called when an attribute a is evaluated and its value does not change. *Antifreeze* removes an attribute b (as well as all its outgoing red edges) from the model M if two conditions hold: (1) b depends only on a in the model M, and (2) b was inserted in M by a previous invocation of *freeze*. The red edge coloring identifies the dependent attributes that were reinserted by *freeze*: An attribute b that depends on a was inserted into the model by *freeze* if b has outgoing red edges.

Antifreeze may cause an attribute instance that was not inserted into the model by *freeze* (and is therefore not removed by *antifreeze*) to become ready for evaluation because of the removal of incident red edges. Line [6] of *antifreeze* inserts such attributes into the worklist S.

```
            procedure antifreeze ( ep: an evaluation process; a: an attribute instance );
            declare
                b: attribute instance;
            begin
    [1]         for each attribute b such that (a, b) is a red edge in ep.M do
    [2]             Remove edge (a, b) from ep.M;
    [3]             if red in-degree of b in ep.M is 0 then
    [4]                 if red out-degree of b in ep.M is 0 then
    [5]                     if in-degree of b in ep.M is 0 then
    [6]                         Insert b into ep.S fi
S                       else
    [7]                     antifreeze(ep, b)
                        fi
                    fi
                od
            end
```

Figure 6-3: *Antifreeze Algorithm*

6.2. Merging Models after Downward Expansion Collision

Figure 6-4 gives the complete algorithm for expanding a model by a downward expansion, detecting collisions during expansion, and merging the colliding models. When a collision at a child *ch* of the node *n* being expanded is detected, the following actions are performed. We denote the model being expanded at node *n* by M_n, and the model that covers node *ch* by M_{ch}. The evaluation process doing the expansion is denoted ep_n and the other one is ep_{ch}.

1. *Freeze* is called for each inherited attribute of *ch* to reinsert attributes in model M_{ch} that depend on attributes in M_n.

2. The worklists of the two evaluation processes are combined into a single worklist *S* associated with ep_n.

3. The evaluation process for the child node *ch* is marked for termination by setting its *active* field to **false**. This causes the loop in *asynch-propagate* to end, and the data structure to be deallocated.

4. *Freeze* is called on the synthesized attributes of *n* to reinsert attributes in model M_n that depend on attributes in M_{ch}.

5. The models M_n and M_{ch} are united by adding to M_n (a) the direct dependencies in the production that applies at *n*, (b) the entire model M_{ch} with the superior transitive dependency edges of node *ch* deleted, and (c) the subordinate transitive dependency edges of other children of *n*.

After the collision, ep_n continues with the merged model M_n. This, in effect, combines the two evaluation processes that had been executing independently prior to the collision.

There are two operations in *asynch-expand* that deserve further explanation: (1) the union of the model at the colliding child node with the expanding model, and (2) locating the evaluation process for the colliding child node.

The union operation can be performed efficiently (in unit time) if each model is represented as follows. An attribute instance in the semantic tree has a field listing the attributes adjacent to it in its model; this field is null if the attribute instance is not in a model. The data structure representing a model consists only of a pointer to the apex of the model's backbone; all the other information about the model is maintained in the semantic tree data structure. Therefore, uniting two colliding models during an expansion at node *n* involves updating the model field of attribute instances in the production that applies at *n*. For a given grammar, the cost of the union operation is bounded by a constant — the maximum number of attribute instances in a production of the grammar.

A simple solution for locating the evaluation process of a node involved in a collision consists of keeping back pointers from the apex node of each model to the corresponding evaluation process data structure. For a downward collision, this solution takes constant time since the colliding

```
procedure asynch-expand ( ep: an evaluation process; b: an attribute instance );
 declare
     c: an attribute instance;
     ExpandedSubordinate: a directed graph;
     collision: boolean;
     fp: an evaluation process;
 begin
     if there exists c, a successor of b in D(T) that is not in ep.M
             and TreeNode(c) is a child of TreeNode(b) then
         /* downward expansion */
         ExpandedSubordinate := direct dependencies of production that applies at TreeNode(b);
         for each child ch of TreeNode(b) do
             if (ch.in-model = apex) or (ch.in-model = apex/frontier) then
                 collision := true;
                 fp := lookup evaluation process of ch;
                 for each inherited attribute i of ch do
                     freeze(fp, i);
                     ep.S := ep.S ∪ fp.S;
                     fp.active := false; /* to terminate evaluation process of ch */
                 od
                 ch.in-model := interior;
                 ExpandedSubordinate := ExpandedSubordinate ∪ (fp.M − ch.C̅)
             else
                 ch.in-model := frontier;
                 ExpandedSubordinate := ExpandedSubordinate ∪ ch.C
             fi
         od
         if (collision) then
             for each synthesized attribute s of TreeNode(b) do
                 freeze(ep, s)
             od
         fi
         TreeNode(b).in-model := interior;
         /* expand ep.M */
         ep.M := (ep.M − TreeNode(b).C) ∪ ExpandedSubordinate;
         Insert into ep.S all vertices of ExpandedSubordinate whose in-degree in ep.M is 0
     fi

     /* upward expansion */
         . . .

 end
```

Figure 6-4: *Algorithm to Expand Model, Detect Collisions, and Merge Models* node is the model's apex. However, in the case of an upward collision (which can occur at any node on the frontier of the model), or an initialization collision (which can occur at any node in the model), the semantic tree may have to be traversed from the colliding node to the apex of the model. Thus, the simple solution is not satisfactory.

Our solution for locating the evaluation process of a node involved in a collision has constant expected cost (as explained in section 7) for downward and upward collisions, as well as initialization collisions. A unique timestamp is associated with each subtree replacement. The timestamp is employed by the evaluation process to label semantic tree nodes that are in the backbone of the associated model (not shown in the figures, to avoid cluttering the algorithms). The evaluation process data structures are stored in a hash table, with the timestamp used as the key.

When two models are merged, the hash table entries for the two corresponding evaluation processes must be updated. Suppose that two models, M_1 and M_2 are merged, and suppose that the apex of M_1 is an ancestor of the apex of M_2 in the semantic tree. We call M_1 the *superior model*, and M_2 the *subordinate model*.

The apex of the merged model is the apex of the superior model. Therefore, the hash table entry for the superior model is not changed by the merge operation, since the model already points to the right node. The hash table entry for the subordinate model, however, must be changed to point to the apex of the superior model. The combined worklist of the merged model is stored in both hash table entries for the two colliding models. Even though semantic tree nodes in the merged model are labeled with different timestamps, this scheme will located the correct evaluation process with its merged model and worklist for each node in the merged model's backbone.

6.3. Merging Models after Initialization Collision

After a subtree replacement at a node *r*, the initial model created for the new evaluation may overlap an existing model of a previous evaluation in progress, causing a collision. Collisions at model initialization are simpler than those at model expansion because none of the attributes in the new model have been evaluated. Therefore, it is only necessary to call *freeze* on the synthesized attributes of *r*.

There is however an additional complexity in the initialization case: the existing model involved in a collision may cover the deleted subtree. Thus, the part of the existing model in the replaced subtree must be deleted, and attribute instances in the worklist associated with nodes in the replaced subtree must be removed. These cleanup operations are performed by the routine *replace-subtree* before the replaced subtree is pruned, as shown in figure 6-5. The part of the model covering node *r* is not deleted since pruning the subtree rooted at *r* does not delete node *r* from the tree (as noted in section 2.2). The *in-model* field of *r* is set to *frontier* if an initialization collision occurred.

```
atomic procedure replace-subtree ( T: semantic tree; r: nonterminal node at root of replaced subtree;
                                   r′: nonterminal node at root of new subtree );
begin
    if (r.in-model ≠ not-in-model) then
        cleanup-model(r);
        if (r.in-model = apex) then
            r.in-model := apex/frontier
        else
            r.in-model := frontier
        fi
    fi
    Prune subtree rooted at r, and graft subtree rooted at r′ onto r;
    Compute new subordinate characteristic graph of r;
    if subordinate characteristic graph of r changes then
        cgraph-update(r) /* defined in section 4 */
    startup(T, r)
end /* of replace-subtree */

procedure cleanup-model ( node: semantic tree node );
declare
    fp: an evaluation process;
begin
    if (node.in-model ≠ frontier) or (node.in-model ≠ apex/frontier) then
        for each child c of node do
            cleanup-model(c)
        od
    fi
        fp := lookup evaluation process of r;
    for each attribute instance a of node do
        if in-degree of a in model is 0 then
            Remove a from fp.S
        fi
        a.model := null;
    od
end /* of cleanup-model */
```

Figure 6-5: *Replace Subtree Operation (revisited)*

The complete *startup* procedure, which initializes an evaluation process for a new subtree replacement, is shown in figure 6-6. An initialization collision is detected if the root r of the replaced subtree is on the frontier of an existing model. If a collision is detected, *freeze* is called for each synthesized attribute of r to reinsert attributes in the existing model that depend on attributes in the new subtree at *r*. Uniting the initial model with the existing model is performed by adding the subordinate characteristic graph of *r* to the existing model. Since the initial model has been merged with an existing model, no evaluation process has to be initiated for a subtree replacement that causes an initialization collision.

```
procedure startup ( T: semantic tree; r: node of T at root of subtree replacement );
declare
    ep, fp: evaluation processes;
begin
    if (r.in-model = apex/frontier) or (r.in-model = frontier) then
        /* initialization collision */
            fp := lookup evaluation process of r;
        for each synthesized attribute s of r do
            freeze(fp, s)
        od
        fp.M := fp.M ∪ r.C;
        Insert into fp.S vertices of r with in-degree 0;
        Insert into fp.NeedToBeEvaluated the vertices of r
    else
        Allocate new ep;
        ep.active := true;
        ep.timestamp := current time;
        ep.M := r.C ∪ r.C̄;
        ep.S := the set of vertices of ep.M with in-degree 0;
        ep.NeedToBeEvaluated := the set of vertices of ep.M;
        fork asynch-propagate(ep);
    fi
end
```

Figure 6-6: *Startup Algorithm (revisited)*

7. Analysis of Algorithm

Consider k asynchronous edits whose corresponding evaluation processes have been initiated but have not yet terminated. We define the set *ASYNC–AFFECTED* to be the minimal set of attributes that must be evaluated to restore consistency in the semantic tree modified by the k edits. The size of *ASYNC–AFFECTED* depends on the timing of the subtree replacements. When the k subtree replacements occur ‘‘almost sequentially’’, that is, the attributes affected by the i^{th} subtree replacement have almost all been evaluated before the $(i + 1)^{\text{st}}$ subtree replacement occurs, then

$$|ASYNC\text{–}AFFECTED_{SEQ}| = \sum_{i=1}^{k} |AFFECTED_i|$$

where $AFFECTED_i$ is the set of attributes affected by the i^{th} subtree replacement. In this case, attributes affected by j subtree replacements, $j \le k$, are necessarily evaluated j times.

When the k subtree replacements occur ‘‘almost simultaneously’’, *ASYNC–AFFECTED* contains the set of attributes with different values in the two trees T and T^k, where T is the semantic tree before the k subtree replacements occurred, and T^k is the updated tree after the evaluation of all k subtree replacements has completed. The size of *ASYNC–AFFECTED* for k ‘‘almost simultaneous’’ edits may be less than the cardinality of the union of the k affected sets, since an attribute’s value may have been changed as a result of one edit, only to be changed back to its original value because of a subsequent edit. That is,

$$|ASYNC\text{–}AFFECTED_{SIM}| \le |UNION\text{–}AFFECTED|, \text{ where}$$

$$UNION\text{–}AFFECTED = AFFECTED_1 \cup AFFECTED_2 \cup \cdots \cup AFFECTED_k .$$

When the timing of the k subtree replacements is in between these two boundary cases, reestablishing attribute consistency in the modified tree may require additional attribute evaluations than if the k edits were ‘‘almost simultaneous’’, but may avoid some attribute evaluations that would have been performed had the the edits been ‘‘almost sequential’’. That is, the size of the set *ASYNC–AFFECTED* is

$$|ASYNC\text{–}AFFECTED_{SIM}| \le |ASYNC\text{–}AFFECTED| \le |ASYNC\text{–}AFFECTED_{SEQ}|$$

We say that an incremental attribute evaluation algorithm for multiple asynchronous subtree replacements is *optimal* if the total work performed to update the tree after k asynchronous edits is proportional to the size of *ASYNC–AFFECTED*. However, it remains an open question whether this lower bound can be achieved.

An alternative complexity metric is to consider only the number of attribute evaluations performed to reestablish attribute consistency, separately from the bookkeeping. In this case an algorithm may be said to be optimal if the number of attribute evaluations is proportional to the size of *ASYNC–AFFECTED*. In other words, for any $k > 1$ modifications affecting the same attribute a, where the k evaluation processes are still in progress and none have yet evaluated a, the algorithm evaluates a at most once. This metric seems appropriate when actual time costs of attribute evaluation dominate, for example, as determined by experiments on the GAG compiler-compiler [33].

The number of attributes evaluated for k asynchronous subtree replacements is optimal when initialization collisions occur for each subtree replacement. If models for new evaluation processes do not collide immediately with existing models, however, attributes that are affected by more than one edit may be evaluated unnecessarily.

Our algorithm can be extended to minimize the number of attribute evaluations in all cases. To do this, we force a collision between the initial model of a new subtree replacement with the existing model, if any (in this approach there is at most one), as soon as the new subtree replacement occurs. Forcing a collision is done by expanding one or both models, depending on whether or not one model is an ancestor of the other. The model edges added by the expansions required to force a collision are colored red, for the same reason that edges added by the *freeze* routine are colored: to avoid evaluating attributes that are not affected by either subtree replacement. This extension minimizes the number of attributes evaluated, but the worst-case bookkeeping costs are proportional to the size of the semantic tree. This is because merging the two models may require expansion over parts of the tree that are not affected by either subtree replacement.

The bookkeeping costs of our algorithm (as described in the previous three sections) are analyzed in three parts: (1) the cost of maintaining characteristic graphs, (2) the cost of detecting collisions, and (3) the cost of merging the models when there is a collision. The cost of updating characteristic graphs invalidated by a subtree replacement is at most proportional to the number of semantic tree nodes on the paths from the m editing cursors to the *LCA*. In the worst case, this cost is $O(n)$. In practice, we expect that the semantic tree nodes whose characteristic graphs are changed because of a subtree replacement also have attribute instances that are affected by the subtree replacement, and therefore the bookkeeping cost incurred to update characteristic graphs after a subtree replacement is proportional to the number of attributes affected by the subtree replacement.

Collision detection is performed once when the model is initialized, and each time the model is expanded. Detecting collisions costs $O(1)$, since the algorithm simply checks if any of the semantic tree nodes added by the expansion, or during initialization of a new model, are in another model's backbone. The field of the semantic tree node indicating whether the node is within the backbone of a model can be updated in unit time at each expansion. This field is cleared when the evaluation of the k subtree replacements (all of which have collided) terminates, at a maximum cost of $O(\sum_{i=1}^{k} |AFFECTED_i|)$. (Termination is assumed only for analysis purposes. Our algorithm could be executed with a continuous stream of asynchronous edits.)

The cost of merging two models involved in a collision can be broken into the following parts:

1. *Linking the two models at the collision point node*: This operation involves the attributes associated with the collision point node, which is bounded by a constant for a given AG. Thus, this operation costs $O(1)$.

2. *Combining the worklists associated with the two models*: This is a set union operation. Since the two models before collision cover different parts of the semantic tree, the two sets are disjoint. Therefore, there is no need to check for duplicates in the sets being united, and so we treat this operation as $O(1)$.

3. *Locating the evaluation process data structure for a collision point node*: If the number of independent evaluation processes is bounded, which we anticipate in practice, then an appropriately selected hash table size and hashing function will

give an expected cost of unit time.

4. *Deleting the part of a model that covers a replaced subtree in the case of an initialization collision*: The cost of this operation is proportional to the number of attributes in the region of the replaced subtree covered by the model. The worst case is when the $(i+1)^{st}$ subtree replacement occurs when M_i has expanded to its maximum size and the root of the new subtree replacement is the apex of M_i. Then the worst-case complexity is $O(\sum_{i=1}^{k} |AFFECTED_i|)$.

5. *Reinserting attributes in one model that may be affected by attributes in the other model*: The worst case is an ‘‘almost sequential’’ sequence of k edits where the following additional conditions hold:

   - The intersection between the sets $AFFECTED_i$ and $AFFECTED_{i+1}$, $1 \le i \le k-1$, is very small.

   - Model M_i collides with model M_{i+1} at the point when both have performed all their expansions.

   - All but a few of the attributes in $AFFECTED_i$ and $AFFECTED_{i+1}$ have been evaluated at the time of the collision.

   This means that the collision will reinsert $(AFFECTED_i + AFFECTED_{i+1})$ attributes into the merged model, which will eventually be removed without ever being evaluated.

   For $k-1$ collisions between M_i and M_{i+1}, $1 \le i \le k-1$, the maximum cost of inserting evaluated attributes back into the merged model is:

   $$\begin{array}{ll} AFFECTED_1 + AFFECTED_2 + & \text{[Collision 1]} \\ AFFECTED_1 + AFFECTED_2 + AFFECTED_3 + & \text{[Collision 2]} \\ \quad \cdots & \\ AFFECTED_1 + AFFECTED_2 + \quad \ldots \quad + AFFECTED_k & \text{[Collision } k\text{-1]} \end{array}$$

   The worst-case cost of this step of the merging algorithm is $O(\sum_{i=1}^{k} |AFFECTED_i| \cdot i)$. The cost for removing the inserted attributes from the merged model if they do not need to be evaluated is the same.

Thus, the worst-case bookkeeping costs for k colliding models is $O(\sum_{i=1}^{k} |AFFECTED_i| \cdot i)$. We expect that in practice our merging algorithm will perform considerably better than the worst case. If k is large compared to the size of the semantic tree, then collisions occur soon after model initialization, and therefore the worst-case scenario does not arise. If k is very small compared to the average size of $AFFECTED_i$, then the worst-case bookkeeping costs are $O(\sum_{i=1}^{k} |AFFECTED_i|)$, which is the same as $O(|ASYNC\text{–}AFFECTED_{SEQ}|)$.

8. Application Scenarios

We describe two distinct application scenarios for multi-user language-based editors, one for programming-in-the-many in the Modula-2 programming language and the other a distributed calendar system for scheduling meetings among groups of people. Both of these applications have been implemented using MERCURY.

8.1. Smod

The Smod environment, named for Small Modula-2, supports teams of programmers writing programs in a subset of Modula-2. When one programmer modifies the interface of his or her module, the interfaces of all other modules that import this one are analyzed. If the edit has introduced new errors in the interfaces of other modules, then appropriate error messages are immediately displayed alongside the representations of those other modules. If the edit has removed previous inconsistencies among the interfaces of modules, then the previous error messages are now no longer displayed. Thus all the programmers may carry out their work simultaneously, confident that they will be automatically informed of any changes that affect them.

This is important for programming-in-the-many, where a large program is divided into modules, each assigned to a different programmer. In theory, the module interfaces are set in advance, are ‘‘correct’’ and should not change, but in practice module interface errors and their repairs account for a large portion of changes [24, 25]. In most software development efforts, programmers rely on electronic mail, meetings, and other human-directed mechanisms for informing each other about interface changes, but these are often tardy, incomplete, or misleading — so there may be long periods where programmers make out-of-date or incorrect assumptions about interfaces, resulting in much wasted effort.

MERCURY makes it possible to automate propagation of all interface changes to all affected programmers, and to only those programmers actually affected (to avoid overloading programmers with irrelevant information about interface changes that do not affect them). The goal is to increase programmer productivity and software quality by making sure that programmers always have all the information they need about the other modules that their own work depends on.

Figure 8-1 shows two windows, each representing a different programmer working on a separate module. The two windows are shown on the same screen for simplicity, but each is connected to a different user process on a different machine (liberty and freedom). At this point, the second programmer’s module `sort` is correctly using the interface exported by the first programmer’s module `input`.

Figure 8-2 shows the same two windows, but after the first programmer has modified the interface to change the `ReadPosNumber` facility exported by `input` and imported by the second programmer into `sort`. Immediately after the first programmer completes the keystroke

Figure 8-1: *Two Smod Modules Before Interface Change*

Figure 8-2: *Two Smod Modules After Interface Change* necessary to edit the interface, two error indications appear in the second programmer's window, each informing her of an inconsistency (where `ReadPosNumber` is used within `Initialize`). Thus each programmer is automatically kept up to date about all program changes that affect her.

One concern that arises is that some programmers prefer not to be informed so quickly of interface changes to other modules that affect them, particularly if there will be many changes, but instead wish to continue their work uninterrupted based on older versions of imported modules. Presumably, they plan to update their own modules later on to use the new versions after they have stabilized. We addressed this problem in the MERCURY implementation by allowing individual programmers to ‘‘turn off’’ propagation of changes *into* their modules when desired.

Another concern is that some programmers wish to experiment with interface changes in privacy, and do not want the environment to automatically propagate their changes to other programmers who would be affected if the changes were made permanent. These programmers prefer to select some command that indicates that they have ‘‘committed’’ their changes, in the sense of making them publicly available to other programmers collaborating on the program (this is not the same thing as releasing the program to the quality assurance, etc.). We also addressed this problem, by permitting individual programmers to ‘‘turn off’’ propagation of changes *out of* their modules. Turning off propagation into and out of modules can be done separately. When either is turned on again, the appropriate propagation (of only the final form, not the intermediate changes) is initiated automatically.

8.2. Calendar

The Calendar system is a non-programming computer-supported cooperative work application. Although not a full calendar system, it provides most of the automatic scheduling functionality recommended in a study of electronic calendars in office systems [18].

Each user maintains her personal calendar on-line, and enters personal appointments. When one user desires to set up a meeting involving several people, she enters a request indicating the constraints for the meeting: who should attend, the expected length of the meeting, the earliest and latest dates for the meeting, and some indication of its purpose. Calendar checks the schedules of all the users expected to attend the meeting, determines the first time period when all of the people are free between 9am and 6pm, and tentatively schedules the meeting. If there is no possible time for the meeting given the constraints, Calendar immediately informs the user who requested the meeting, who can then relax the constraints or negotiate directly with the other users to open up some time slots.

If a meeting is tentatively scheduled, it appears on all the relevant users' personal calendars in a special section that indicates it is pending. Each user must confirm the meeting by entering the meeting into her regular appointment section. If some user cannot attend, she enters her conflicting appointment into the system, and Calendar automatically withdraws the previously scheduled time and tentatively schedules the meeting again in the next available time slot. The meeting remains pending until all attendees have confirmed for the same time period. The original user can later retract (or cancel) the meeting by deleting the corresponding request from her history of commands, which are saved by Calendar for this purpose.

Figure 8-3 shows two windows, each representing a different user's personal calendar, Josephine on freedom and Gail on liberty. Figure 8-4 shows what happens after Josephine requests a meeting involving both of them. The automatically selected time appears tentatively on each user's personal calendar. Figure 8-5 shows what happens when Josephine confirms the meeting. Note that Gail has not yet confirmed, and the meeting time is still regarded as pending. Figure 8-6 shows what happens after Gail confirms.

The implementation of the Calendar functionality took only two days. The basic MERCURY facilities provide all of the support needed to propagate information among multiple users whose calendars reside on their personal workstations, and a prototype of an application can be implemented very quickly by specifying the structure of the document involved (a global calendar made up of a set of personal calendars in this case), the semantic dependencies among the document segments (*e.g.*, the constraints for scheduling meetings among busy users), and the semantic dependencies within each document segment (*e.g.*, multiple meetings involving the same user cannot be overlapping).

9. Related Work

The RTD algorithm was the first optimal algorithm for incremental evaluation of attribute grammars. This algorithm allows only a single cursor, with all cursor movements restricted to be from a child node to its parent or *vice versa*, and was intended to support single-user language-based editors. A related algorithm by Reps [30] supports incremental evaluation for a single cursor after an arbitrary movement from any point in the tree to any other, which might arise when a mouse is moved from one point on the visual display of the program to another. In both approaches, the user makes a subtree replacement, ‘‘waits’’ during the subsequent evaluation process (the delay is generally unnoticeable since the program is small), and then can move the cursor and make another change.

An algorithm for *multiple synchronous* subtree replacements was developed by Reps, Marceau and Teitelbaum [29]. Multiple synchronous changes support commands, such as program transformations, that do not map nicely to single subtree replacements. The idea is that all the subtree replacements are made first, using multiple cursors, and collected — and then a single process is employed for overall incremental evaluation. This algorithm is not effective when changes occur asynchronously, and thus is not suitable for multiple users.

A number of algorithms for handling *multiple asynchronous* subtree replacements have been reported, but most are applicable only to semantic dependencies that are expressible in some

Figure 8-3: *Two Calendar Schedules Before Meeting Request*

Figure 8-4: *Two Calendar Schedules After Meeting Request*

Figure 8-5: *Two Calendar Schedules After Original User Confirms*

Figure 8-6: *Two Calendar Schedules After Other User Confirms* restricted subclass of the noncircular attribute grammars — while our algorithm applies to the general class. The restrictions may not be a problem for language-based editors for programming, since many programming languages fit within the restrictions, but the requirements of other applications are as yet unknown.

Geitz [7] gives an algorithm that minimizes the number of attributes reevaluated by maintaining transitive dependency edges between attributes in one model that depend on attributes in another model. This algorithm is only applicable to a subset of the partitioned grammars for which the transitive dependency information required by the algorithm can be computed statically from the grammar. Since Geitz does not analyze the complexity of the bookkeeping costs of his algorithm, it is not clear how his algorithm compares with the version of our algorithm that minimizes attribute reevaluation by forcing collision immediately after model initialization.

The evaluation strategy in our algorithm and all the algorithms described above are dynamic, meaning that scheduling is determined on the fly during propagation from the actual attribute dependencies in the semantic tree. Static evaluators, in contrast, use plans generated at AG analysis time to schedule attributes for evaluation and are thus more efficient in both space and time.

Peckham [23] has devised a static incremental evaluator for multiple asynchronous subtree replacements. His algorithm, which minimizes the number of attributes evaluated, works for a subset of the partitionable AGs called the globally partitionable attribute grammars. Micallef [20] has described two static incremental evaluators, one for ordered attribute grammars [17] and one for a new subset of the ordered attribute grammars called pairwise ordered attribute grammars. Both minimize attribute evaluation, but the latter achieves lower bookkeeping costs by further restricting the subclass of attribute grammars handled.

Boehm and Zwaenepoel [3] devised a distributed non-incremental attribute evaluation algorithm, distributed for the purpose of speeding up code generation through parallel processing. Their approach combined static and dynamic approaches, with a static evaluator applied within a region, and a dynamic evaluator across regions. Alblas [1] extended this algorithm to incorporate the KK results, to support concurrent incremental attribute evaluation. Unlike Boehm and Zwaenepoel, Alblas does not actually distribute the regions across a network, but instead associates them with multiple evaluation processes; in effect Alblas's algorithm operates within a single KK segment. Thus his approach does not address propagation among multiple users residing on different workstations.

Alblas's goal is to support asynchronous subtree replacements caused by automated tree transformation systems rather than individual edits by multiple users. He describes a merging operation for overlapping models resulting from different transformations that seems similar to the one described in section 6. However, Alblas describes the merge operations as "a union operation, *i.e.*, identical arcs and vertices in two graphs become one in the resultant graphs". He does not explain the reintroduction of arcs necessary to ensure that after the evaluation terminates all attributes have consistent values, so thus we cannot determine the bookkeeping overhead of his algorithm.

Hoover [11] is primarily concerned with incremental graph evaluation, to support modifications of dependency graphs and evaluation of what he calls influenced vertices of the graph (influenced vertices correspond approximately to affected attributes). He discusses incremental attribute evaluation for general noncircular attribute grammars in this framework, which also applies to a range of other incremental computation problems. Most of his results are based on the paradigm where modification and evaluation alternate, and both are done sequentially. However, Hoover relaxes this assumption and briefly addresses the problems of allowing dependency graph updates while evaluation is in progress and of parallel evaluation on a dependency graph. The first case corresponds to the characteristic graph updates we describe in section 4. But since Hoover employs an approximate rather than exact topological order, it is possible that his algorithm will unnecessarily evaluate the same attribute multiple times for even a single dependency graph modification.

All of the above efforts are primarily concerned with language-based editors for programming. Hudson and King have applied incremental attribute evaluation to user interface updates [12] and more general database applications [13] in their Cactis system. Their two-phase evaluation algorithm first constructs the subset dependency graph described in section 2.2, marking all of these attributes as out of date, and then actually computes the new values in another pass. This works for general attributed graphs, and minimizes attribute evaluations but not overhead. Cactis supports multiple users executing serializable transactions, but the evaluation algorithm works only for one subtree replacement at a time.

10. Conclusion

Our algorithm admits a spectrum between always merging models at initialization, thus achieving optimal attribute evaluation costs at the expense of higher bookkeeping overhead, to avoiding merging entirely and reevaluating attributes affected by k subtree replacements k times. If the k edits occur almost sequentially, then the two cases are the same.

Our primary contribution is the complete set of supporting algorithms to make this *asynchronous* attribute evaluation work for the general class of noncircular attribute grammars:

- Updating characteristic graphs to maintain the prepared for propagation invariant for multiple editing cursors.
    - In response to cursor movement operations, $O(1)$.
    - In response to subtree replacements, the number of nodes on the paths from the editing cursors to their least common ancestor.
- Maintaining auxiliary information in the semantic tree in order to detect when two

models cover the same tree node.

- Detection during initialization or expansion of one of the models, $O(1)$.
- Updating this auxiliary information when the model expands, $O(1)$.
- Clearing this information after evaluation terminates, $O(|ASYNC\text{–}AFFECTED|)$.

- Merging two colliding models during model expansion or initialization.
  - Uniting the two models, $O(1)$.
  - Reinserting dependent attributes into the merged model that have previously been removed from one of the colliding models, $O(\sum_{i=1}^{k} |AFFECTED_i| \cdot i)$. If k is small, this is the same as $O(|ASYNC\text{–}AFFECTED_{SEQ}|)$. Removing these attributes from the merged model if it turns out to be unnecessary to reevaluate them has the same cost.

The complexity metrics listed above, concerned with bookkeeping costs, are analyzed on a worst-case basis. The balance between attribute evaluation and bookkeeping costs must consider the anticipated performance overheads of attribute evaluation, parallel processing and distribution. Further, minimizing attribute evaluation on a decentralized tree would incur immense synchronization penalties.

Acknowledgments

Simon Kaplan worked with one of the authors on the original KK algorithm. Travis Winfrey was responsible for much of MERCURY's distribution and user interface support. Wenwey Hseush, Yael Cycowicz and Mark Gisi also participated in the implementation effort. Tom Reps and Tim Teitelbaum developed the Synthesizer Generator system, which we modified as part of the MERCURY implementation, and also provided the initial motivation for this work.

When this work was in progress, Micallef was supported in part by an American Dissertation Fellowship from the American Association of University Women Educational Foundation. Kaiser was supported by National Science Foundation grants CDA-8920080, CCR-8858029 and CCR-8802741, by grants from AT&T, BNR, Citicorp, DEC, IBM, Siemens, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

References

[1] Henk Alblas.
Concurrent Incremental Attribute Evaluation.
In P. Deransart and M. Jourdan (editor), *International Workshop on Attribute Grammars and their Applications*, pages 343-358. Springer-Verlag, Paris, France, September, 1990.

[2] G.V. Bochmann.
Semantic Evaluation from Left to Right.
Communications of the ACM 19:55-62, 1976.

[3] Hans-Juergen Boehm and Willy Zwaenepoel.
Parallel Attribute Grammar Evaluation.
In *7th International Conference on Distributed Computing Systems*, pages 347-354. Berlin, West Germany, September, 1987.

[4] Ravinder Chandhok, David B. Garlan, Dennis Goldenson, Philip L. Miller and Mark Tucker.
Programming Environments Based on Structure Editing: The GNOME Approach.
In Anthony S. Wojcik (editor), *1985 National Computer Conference*, pages 359-370. AFIPS, Chicago, IL, July, 1985.

[5] Alan Demers, Thomas Reps and Tim Teitelbaum.
Incremental Evaluation for Attribute Grammars with Applications to Syntax-directed Editors.
In *8th Annual ACM Symposium on Principles of Programming Languages,* pages 105-116. Williamsburg VA, January, 1981.

[6] Charles N. Fischer, Gregory F. Johnson, Jon Mauney, Anil Pal, Daniel L. Stock.
The POE Language-Based Editor Project.
In *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 21-29. Pittsburgh, April, 1984.
Special issue of *SIGPLAN Notices,* 19(5), May 1984.

[7] Bob Geitz.
Asynchronous Subtree Replacement for Language-Based Editors.
1987.
Oberlin College and Cornell University.

[8] Irene Greif (editor).
Computer-Supported Cooperative Work: A Book of Readings.
Morgan Kaufman, San Mateo CA, 1988.

[9] Peter Henderson (editor).
ACM SIGSOFT/SIGPLAN Software Enginnering Symposium on Practical Software Development Environments.
ACM Press, Pittsburgh PA, 1984.
Special issue of *Software Engineering Notes*, 9(3), May 1984. Special issue of *SIGPLAN Notices*, 19(5), May 1984.

[10] Peter Henderson (editor).
ACM SIGSOFT/SIGPLAN Software Enginnering Symposium on Practical Software Development Environments.
ACM Press, Boston MA, 1987.
Special issue of *SIGPLAN Notices*, 22(1), January 1987.

[11] Roger Hoover.
Incremental Graph Evaluation.
PhD thesis, Cornell University, May, 1987.
Technical Report 87-836.

[12] Scott E. Hudson and Roger King.
Semantic Feedback in the Higgens UIMS.
IEEE Transactions on Software Engineering 14(8):1188-1206, August, 1988.

[13] Scott E. Hudson and Roger King.
Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System.
ACM Transactions on Database Systems 14(3):291-321, September, 1989.

[14] Gail E. Kaiser and Simon M. Kaplan.
Reliability in Distributed Programming Environments.
In *6th Symposium on Reliability in Distributed Software and Database Systems*, pages 45-55. Kingsmill—Williamsburg, VA, March, 1987.

[15] Gail E. Kaiser, Simon M. Kaplan and Josephine Micallef.
Multiuser, Distributed Language-Based Environments.
IEEE Software 4(6):58-67, November, 1987.

[16] Simon M. Kaplan and Gail E. Kaiser.
Incremental Attribute Evaluation in Distributed Language-Based Environments.
In *5th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 121-130. Calgary, Alberta, Canada, August, 1986.

[17] Uwe Kastens.
Ordered Attribute Grammars.
Acta Informatica 13:229-256, 1980.

[18] Christine M. Kincaid, Pierre B. Dupoint and A. Roger Kaye.
Electronic Calendars in the Office: An Assessment of User Needs and Current Technology.
ACM Transactions on Office Information Systems 3(1):89-102, January, 1985.

[19] Donald E. Knuth.
Semantics of Context-Free Languages.
Mathematical Systems Theory 2(2):127-145, June, 1968.

[20] Josephine Micallef.
Incremental Attribute Evaluation for Multi-User Semantics-Based Editors.
PhD thesis, Columbia University, May, 1991.

[21] Joseph M. Morris and Mayer D. Schwartz.
The Design of a Language-Oriented Editor for Block-Structured Languages.
In *SIGPLAN SIGOA Symposium on Text Manipulation*, pages 28-33. Portland OR, June, 1981.
Special issue of *SIGPLAN Notices*, 16(6), June 1981.

[22] Lisa Rubin Neal.
Cognition-Sensitive Design and User Modeling for Syntax-Directed Editors.
In *CHI + GI '87 Conference on Human Factors in Computing Systems and Graphics Interface*, pages 99-102. ACM, Toronto, Canada, April, 1987.

[23] Stephen B. Peckham.
Incremental Attribute Evaluation and Multiple Subtree Replacements.
PhD thesis, Cornell University, February, 1990.
TR 90-1093.

[24] Dewayne E. Perry and W. Michael Evangelist.
An Empirical Study of Software Interface Errors.
In *International Symposium on New Directions in Computing*, pages 32-38. IEEE Computer Society, Trondheim, Norway, August, 1985.

[25] Dewayne E. Perry and W. Michael Evangelist.
An Empirical Study of Software Interface Faults — An Update.
In *20th Annual Hawaii International Conference on Systems Sciences*, pages 113-126. Kona HI, January, 1987.

[26] Thomas Reps.
Optimal-time Incremental Semantic Analysis for Syntax-directed Editors.
In *9th Annual ACM Symposium on Principles of Programming Languages*, pages 169-176. Albuquerque NM, January, 1982.

[27] Thomas Reps, Tim Teitelbaum and Alan Demers.
Incremental Context-Dependent Analysis for Language-Based Editors.
ACM Transactions on Programming Languages and Systems 5(3):449-477, July, 1983.

[28] Thomas Reps.
Generating Language-Based Environments.
M.I.T. Press, Cambridge, MA, 1984.

[29] T. Reps, C. Marceau and T. Teitelbaum.
Remote Attribute Updating for Language-Based Editors.
In *13th ACM Symposium on Principles of Programming Languages*, pages 1-13. St. Petersburg Beach, FL, January, 1986.

[30] Thomas Reps.
Incremental Evaluation for Attribute Grammars with Unrestricted Movement Between Tree Modifications.
Acta Informatica 25:155-178, 1988.

[31] Thomas W. Reps and Tim Teitelbaum.
Texts and Monographs in Computer Science: *The Synthesizer Generator A System for Constructing Language-Based Editors.*
Springer-Verlag, New York, 1989.

[32] Mark N. Wegman.
Parsing for structural editors (Extended Abstract).
In *21st Annual Symposium on Foundations of Computer Science*, pages 320-327.
Syracuse NY, October, 1980.

[33] Dashing Yeh and Uwe Kastens.
Improvements of an Incremental Evaluation Algorithm for Ordered Attribute Grammars.
SIGPLAN Notices 23(12):45-50, December, 1988.

Table of Contents

List of Figures