Incremental Attribute Evaluation with Applications to Multi-User Language-Based Environments

Josephine Micallef

Columbia University
Department of Computer Science
New York, NY 10027

Thesis Proposal

27 April 1989

CUCS-444-89

Abstract

The proposed research addresses three problems associated with performing incremental evaluation of attribute grammars: (1) multiple asynchronous subtree replacements in the parse tree that are initiated by external agents, (2) segmentation of the parse tree according to granularity of access rights with respect to these agents, and (3) distribution of the segments across a reliable network. The research focuses on one exemplary application, distributed multi-user language-based environments, where the parse tree represents a program being developed, the subtree replacements are changes to the program, the external agents are programmers, the granularity of segments corresponds to the modularization of the program, and the nodes of the network are the programmers' workstations.

Copyright © Josephine Micallef

Table of Contents

1. Motivation 1
2. Background 5
    2.1. Attribute Grammars 5
    2.2. Incremental Attribute Evaluation 8
3. Proposed Research 13
    3.1. Multiple Subtree Replacements 13
        3.1.1. Problem Formulation 13
        3.1.2. Solution Approaches 14
    3.2. Segmented Parse Tree 18
        3.2.1. Optimality Considerations 20
    3.3. Distribution of Segments 21
4. Conclusion 23
    4.1. Preliminary Implementation 23
    4.2. Research Plan 23
    4.3. Contributions 24

1. Motivation

Development and maintenance of large software systems by many cooperating programmers, an activity which is called programming-in-the-many, suffers from a well-known problem — the *communication* problem [Tichy 79, Brooks 82]. Lack of communication of changes among the project group members results in a breakdown of the software engineering process, resulting in defective systems whose components do not functionally fit together. Although the communication problem can be simplified by sound design principles, such as modular decomposition, structured programming and information hiding, it cannot be totally eliminated. The reason is that it is unlikely that interfaces are designed perfectly and completely from the beginning. Even if the design is correct, implementors often change their assumptions about the inputs and outputs of their modules.[1] In addition, system evolution is inevitable because of bug fixes, performance tuning, ports to new environments and enhancement, many of which necessitate changes to the modules' interfaces.

Solutions to the communication problem have, up to now, relied on the goodwill of the programmer to "do the right thing". In the early days of programming, teams maintained a project workbook where changes were recorded. (This was done in the development of IBM's OS/360 [Brooks 82].) The changes were distributed on a regular and frequent basis to all members of the team, making it possible, in theory though not in practice, for everyone to be aware of changes that affected them. Though the job was made easier by the development of tools such as electronic mail and cross-reference capabilities, the programmer was still ultimately responsible for communicating changes. Even more advanced programming environments have not done much better.

This thesis investigates a class of programming environments for programming-in-the-many where the environment assumes the responsibility for communicating changes. This assures that changes are communicated (1) in a timely manner, and (2) to exactly the set of programmers that are affected by the change. Timely communication is of critical importance as it prevents work from having to be undone or redone later. The reason for not overwhelming the programmer

[1] Throughout this proposal, the word *module* is used in a generic sense to denote the program unit that is developed by one programmer, such as a Modula-2 *module*, an Ada *package*, or a CLU *cluster*. Note that the word *module* may be used to refer to two program units that are at different levels in the program structure hierarchy, for example, it may refer to an Ada package, or to an Ada subprogram within that package, since these two units may be developed by two different programmers.

with information on changes that do not affect him, aside from saving him from doing unnecessary work, is to prevent the programmer from becoming indifferent to future change notifications that may be meant for him. The goal of the proposed research is to devise solutions to the communication problem which can be automated, not managerial or manual ones.

Furthermore, these solutions must accommodate distributed hardware configurations, where each programmer has a personal workstation connected by a local area network to the workstations used by others in the programming team. The reason for this constraint is that in the last few years there has been a pronounced change in the hardware base from large timesharing systems supporting the entire project team to computing environments consisting of workstations connected by local area networks. This trend stems from the advantages offered by personal workstations, including more predictable response time, increased reliability of the system as a whole, and incremental expandability.

In order for an environment to effectively solve the communication problem, it must be able to:

- Detect a change made to a module of the program being developed as soon as the change is made.
- Determine the extent and implication of the change, that is, what other modules are affected by the change, and what further changes are required in these modules in order to reestablish consistency in the program. This is called *change analysis*.
- Use the results of change analysis to propagate the change to affected modules. This is called *change propagation*.

These requirements underlie our motivations for basing our solution to the communication problem on language-based editing environments. Each environment is centered around an editor, which makes it possible to detect changes at the keystroke[2] level. The second requirement is a much more difficult one to achieve as it requires the environment to understand what makes a program "consistent". We currently restrict consistency analysis to include only syntactic and static semantic consistency analysis of the program. Such checking depends on the programming language in which the program is written. Therefore, the editors are language-based: they understand the syntax and semantics of the particular programming language for which they are tailored.

---

[2]Our editors are structure-oriented, and therefore a keystroke is at the granularity of a syntactic program entity, for example, a variable name.

The end-product of the proposed research is the development of a system, MERCURY, which generates distributed language-based environments from a formal specification of the desired programming language, namely the attribute grammar (AG) formalism. Using attribute grammars as the basis of the research offers many advantages. The most obvious of these advantages is that the generation of new environments from AG specifications is much more cost-effective than handcoding each environment. In addition, there is a large body of theory on the AG formalism, including optimal algorithms for incremental change analysis in single-user environments.

The central algorithm of an AG-based environment is the incremental attribute evaluator. When the program is modified, this algorithm reevaluates those attributes that are affected by the change, thereby performing change analysis and propagation. Previous incremental attribute evaluation algorithms assume that the program, a single monolithic text, is being developed by one programmer[3]. The proposed thesis will address three problems associated with performing incremental evaluation of attribute grammars in multi-user environments: (1) asynchronous modification of the program, (2) segmentation of the program into modules, and (3) distribution of the program across a network.

The algorithms to be developed in this thesis are applicable to a wide variety of applications that are based on attribute grammars, not just language-based environments. Potential applications have the following characteristics:

- The representation of the central data objects of the application is a tree.
- The tree can be modified by multiple asynchronous subtree replacements initiated by external agents.
- The tree is partitioned into segments according to the granularity of access rights with respect to these agents.
- The segments are distributed across a reliable network.

We foresee straightforward application of these algorithms to the distributed database managers for object-oriented data bases [Hudson 88], and interpreters of distributed dataflow languages [Kaiser 89]. The discussion of the problem and approaches for its solution in his proposal (and the thesis), as well as the implementation of the algorithms, will focus on the application of multi-user language-based environments.

---

[3]Exceptions will be described and contrasted with our work in Section 3.1.

The rest of this proposal is organized as follows. Section 2 gives background work on which the proposed research is based: first an overview of attribute grammars, and then an introduction to incremental attribute evaluation. Section 3 describes the proposed research — AG algorithms for multiple program updates, segmentation of the program into modules, and distribution of the modules across the network. Section 4 concludes with a description of an initial implementation of the research, a plan for completing the work proposed, and a summary of the primary contributions expected.

2. Background

2.1. Attribute Grammars

Attribute grammars were first introduced by Knuth [Knuth 68] to describe the context-sensitive semantics of a programming language, complementing the way a context-free grammar describes the language's syntax. An AG extends a context-free grammar by attaching *attributes* to the nonterminal symbols in the grammar, and *semantic equations* defining these attributes to the productions of the grammar. Each attribute represents a specific property of the nonterminal, and can take on any of a specified set of values. A semantic equation defines an attribute (LHS of equation) as the value of a *semantic function* applied to other attributes of that production (RHS of equation). The attribute on the LHS is *functionally dependent* on the attributes in the RHS of the equation. Note that semantic functions are pure functions; that is, they have no side effects. Attributes are divided into two disjoint classes: *synthesized* and *inherited*. 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.

Figure 2-1 gives an example of an attribute grammar fragment for declarations in a Pascal-like programming language. There are four productions, *p1* through *p4*. Each nonterminal occurrence in a production has associated attribute instances, and each production has associated semantic equations that define the value of the attribute instances. An attribute *a* associated with a nonterminal symbol *X* is denoted by *X.a*. Occurrences of the same nonterminal instance within one production are distinguished by the use of a numerical suffix (*e.g.* 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 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 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.

These concepts are best illustrated operationally by an example. Consider the following string

```
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:    ∅;   }
```

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); }
```

Productions of the attribute grammar fragment and their semantic equations

Figure 2-1: *An attribute grammar example*

derived by the grammar of figure 2-1:

```
a: integer;
b: boolean;
```

Figure 2-2 (a) shows the parse tree for this string; the nodes in the tree are labelled with nonterminal symbols of the grammar. Figure 2-2 (b) is the semantic tree for the same declarations.[4] A *semantic tree* is a parse tree where each tree node additionally contains fields

[4]The *error* attribute is not shown in the figure for clarity.

corresponding to the attributes of its labelling 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 $\}$.

The dependency graph for our running example is shown in figure 2-2 (c).

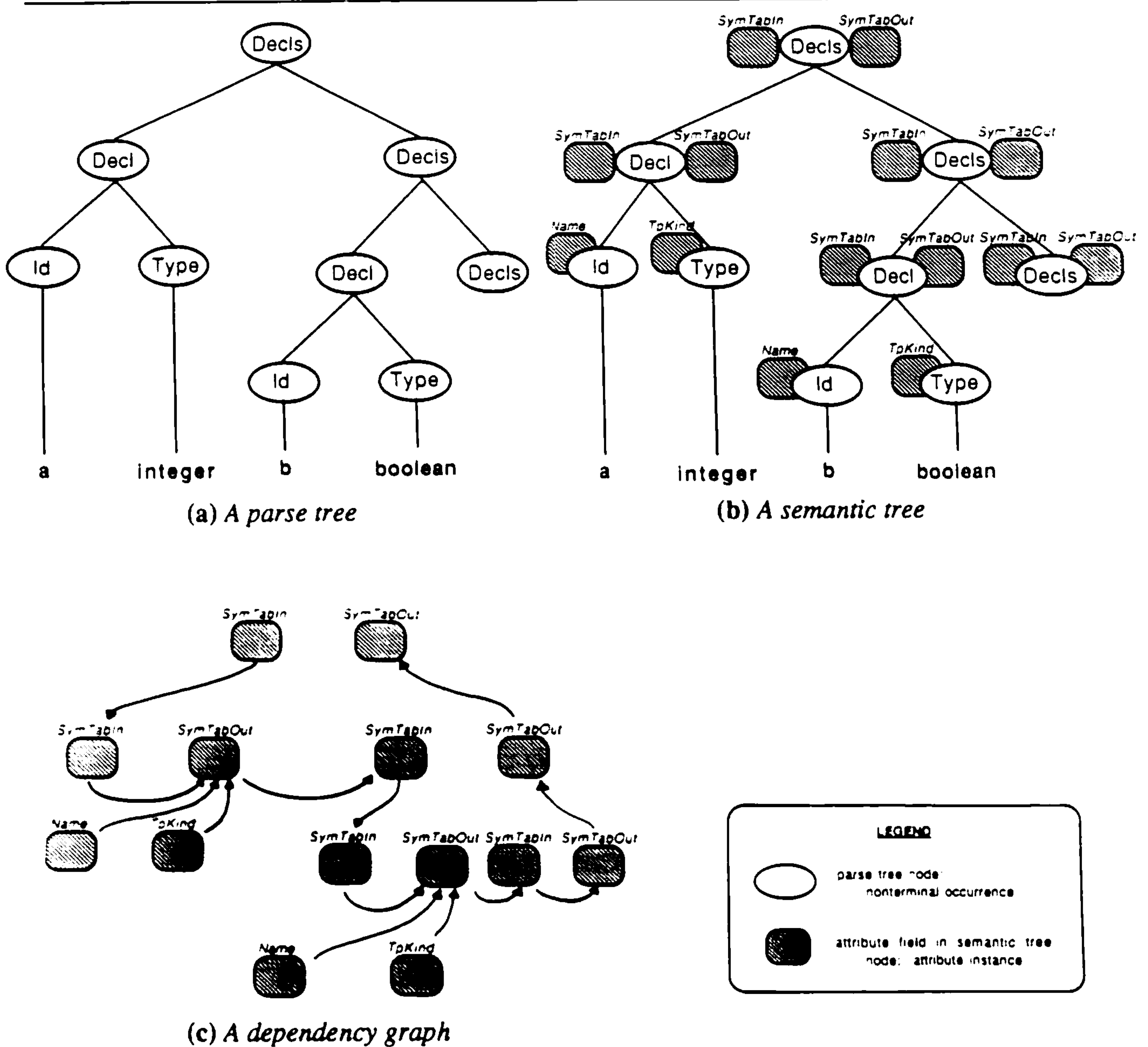

Figure 2-2: *A string derived from the grammar of figure 2-1*

Knuth [Knuth 68] describes a simple algorithm for evaluating all the attributes in a semantic tree that makes use of the dependency graph. The vertices of the dependency graph are first topologically sorted, and then the attributes are evaluated according to their topological order. This algorithm only works if the dependency graph is acyclic. Many attribute evaluators assume that the AG is *well-defined*, that is, that the dependency graph for any string derived by the grammar is acyclic. However, it is hard to verify this assumption since to do so requires exponential time [Jazayeri 75]. The proposed research will only consider well-defined AGs.

Although the algorithm described above is simple and works for any well-defined AG, its performance is poor for both time and space: all decisions are made at run-time, and the dependency graph must be kept around. More efficient algorithms for performing attribute evaluation have been developed for use in practical compiler-compilers [Farrow 84, Kastens 82, Ganzinger 77]. These algorithms perform most of the work at grammar-analysis time, once for each AG, thus improving the performance of the evaluator. The disadvantage of these evaluators, which are called *static* or *semi-static* evaluators depending on how much work is done at run-time, is that they do not work for all well-defined AGs but only for subclasses of them.

2.2. Incremental Attribute Evaluation

The use of attribute grammars in language-based programming environments was originated by Reps [Reps 82]. A program is represented internally by its semantic tree. The program is modified by a sequence of pruning, grafting, or derivation operations on the tree; these operations are collectively called *subtree replacement* 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.

Continuing with the example from the previous subsection, suppose that a programmer, Joe, was editing a program with the two declarations of *a* and *b*. If he were to add the following line to his program:

```
a: character;
```

then, after attribute reevaluation, the value of the *error* attribute associated with this declaration

would be "<-- Variable already declared". Such attributes can be displayed by the programming environment as part of the program text to notify the programmer of inconsistencies in the program. So Joe would see the following on his display after making the change:

```
a: character; <-- Variable already declared
```

This illustrates how change analysis and change propagation are accomplished by means of attribute evaluation. It is desirable that the evaluation strategy be incremental, that is, it does not evaluate all the attributes in the semantic tree from scratch, but only those that are affected by the change.

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. The root node of the two subtrees, S and S', must be labelled with the same nonterminal grammar symbol. Let T' be the tree T with S replaced by S'. The problem is to evaluate the minimum number of attributes in T' so that attribute consistency is reestablished. An optimal solution to this problem for the general class of well-defined AGs was devised by Reps [Reps 83].

Before describing Reps' algorithm, we define some terminology and state the assumptions used. The *output attributes* of a production, p, are those attributes defined by semantic equations associated with p. 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 *normal form* if for any production p, the output attributes of p consist of the synthesized attributes of the LHS symbol of p and the inherited attributes of the RHS symbols of p, and the input attributes of p consist of the inherited attributes of the LHS symbol of p and the synthesized attributes of the RHS symbols of p. It is assumed that the AG is in *normal form*.[5]

When a subtree S with root node r is replaced by a subtree S' with root node r',[6] which of the two sets of attribute instances associated with the roots of the subtrees S and S' should be used? In Reps' algorithm, the synthesized attributes of r and the inherited attributes of r' are used. This decision, together with the fact that the AG is in normal form implies that initially, the only

[5]This assumption is made only to simplify the exposition. It is not a restriction because any grammar can be put in normal form, and the algorithm to be described can be easily extended to deal with non-normal form grammars.

[6]Recall that nodes r and r' must be labelled with the same nonterminal symbol.

inconsistent attributes are those associated with the root of the replaced subtree. Therefore, after a subtree replacement at node r, the algorithm starts by evaluating the attributes associated with r. The problem is: in what order should they 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 2-2 (c).) For this reason, the scheduling algorithm for determining which attribute should be evaluated next must take into account transitive dependencies.

For the class of well-defined AGs, the transitive dependencies among the attributes of a nonterminal symbol may be different for different occurrences of the symbol in a semantic tree. The *upper tree context* of a nonterminal instance r in a semantic tree is the resulting tree after the subtree rooted at r is pruned . There can be several different subtrees derived from r, and several upper tree contexts. Therefore, it is in general not possible to determine the transitive dependencies from a static analysis of the AG.

For the single subtree replacement case, Reps 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.

Reps' incremental evaluation algorithm, which is described below, is optimal because of the fact that these characteristic graphs are maintained cheaply. Each editing operation maintains the following invariants on the semantic tree representing the program being edited:

- Both superior and subordinate characteristic graphs are kept for the nonterminal occurrence where the editing cursor is placed.
- All nodes on the path from the cursor to the root of the tree have superior characteristic graphs.

- All other nodes in the tree have subordinate characteristic graphs

Figure 2-3 shows the incremental algorithm for reevaluating a semantic tree T after a subtree replacement at r has occurred [Reps 84a]. It makes use of two data structures: (1) a model M, which is a graph 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). M initially contains the attributes of the root of the replaced subtree, r. The (direct and transitive) dependencies among these attributes 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 done by the EXPAND procedure, which is not shown here. EXPAND brings into the model all the attributes of a neighboring production of the attribute that caused the expansion, as well as all dependency edges among them. The attribute that was just evaluated, as well as all its outgoing edges, are then removed from the model. This may result in some attributes becoming ready for evaluation; such attributes are inserted into the worklist S.

```
procedure EVALUATE(T: semantic tree; r: nonterminal occurrence at root of replaced subtree)
declare
    S: set of attribute instances
    M: directed graph
    b, c: attribute instances
    OldValue, NewValue: attribute values
begin
    M := r.C ∪ r.C̄
    S := the set of vertices of M with in-degree 0 in M
    while S ≠ ∅ do
        Select and remove a vertex b from S
        OldValue := value of b
        evaluate b
        NewValue := value of b
        if OldValue ≠ Newvalue and M does not contain all the successors of b in D(T)
            then EXPAND(M, b, S)
        Remove b and all outgoing edges from M
        Insert any attributes whose in-degree is now 0 in S
    od
end
```

Figure 2-3: *An incremental attribute evaluation algorithm for single subtree replacements*

Reps' incremental evaluation algorithm for a single subtree replacement is time-optimal. This means that both the number of semantic function applications, as well as the bookkeeping costs of the algorithm, are proportional to the size of the set *AFFECTED*, where *AFFECTED* is the set of attributes whose values differ in T and T'.

3. Proposed Research

The proposed research addresses three problems associated with performing incremental evaluation of attribute grammars in multi-user distributed environments: (1) multiple asynchronous subtree replacements in the parse tree that are initiated by external agents, (2) segmentation of the parse tree according to granularity of access rights with respect to these agents, and (3) distribution of the segments across a network. The proposed research assumes that the network is reliable, and therefore issues of fault-tolerance and recovery will not be discussed.

3.1. Multiple Subtree Replacements

3.1.1. Problem Formulation

Let T be a semantic tree of some attribute grammar G, 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'. The two modifications at S and R are *asynchronous*, that is, the second one may occur while the evaluation of the first one is still in progress. The problem is to design an incremental evaluation algorithm that can handle this scenario, generalized to k asynchronous changes, in an optimal way. This problem arises naturally in multi-user environments as multiple programmers make changes simultaneously in different modules of the program.

An incremental evaluator for asynchronous subtree replacements is optimal if it meets the following requirements:

1. For any one modification, the algorithm will evaluate only those attribute instances affected by the modification.
2. For any $k > 1$ modifications affecting the same attribute a, where the k evaluations are still in progress and none have yet evaluated a, the algorithm will evaluate a only once.
3. The bookkeeping costs of the evaluation algorithm are proportional to the number of attributes evaluated.

This is an ideal definition of optimality and it is as yet an open question whether an algorithm that achieves all these requirements can be designed. As will be seen in section 3.1.2, there seems to be a tradeoff between the bookkeeping costs of the algorithm and the number of attributes reevaluated. That is, a particular evaluation algorithm may evaluate the minimum number of attributes but only by being sub-optimal in its bookkeeping costs. Such an algorithm would be useful for an application where attribute evaluation is expensive compared to the

bookkeeping costs, such as the proof checker described in [Reps 84b].

The second requirement is the more important one for the purposes of this section of the proposal, so we shall state it a little more formally for the case when $k = 2$. Suppose that subtree S was replaced at time t_1, and subtree R at time t_2, where $t_1 < t_2$. Let $AFFECTED_S$ be the set of attributes that were affected (and therefore must be reevaluated) because of the subtree replacement at S, and similarly, $AFFECTED_R$ the set of attributes affected by the subtree replacement at R. Furthermore, suppose that the evaluations from the two modifications overlap, that is,

$$AFFECTED_S \cap AFFECTED_R \neq \emptyset$$

If the evaluation due to the subtree replacement at S is still in progress at the time of the second modification, t_2, then $AFFECTED_S$ can be divided into two subsets: (1) *EVAL*, containing those affected attributes that have already been evaluated at the time of the second replacement, and (2) *UNEVAL*, containing the attributes still needing evaluation.

$$AFFECTED_{S,t_2} = EVAL_{S,t_2} \cup UNEVAL_{S,t_2}$$

Note that all these sets are not known *a priori* but are determined as the evaluation is proceeding. The second optimality requirement states that every attribute a, such that

$$a \in UNEVAL_{S,t_2} \cap AFFECTED_R,$$

is evaluated only once.

3.1.2. Solution Approaches

A naive approach to the multiple replacement problem is to perform k sequential applications of Reps' algorithm for single subtree replacements. Thus, a subtree replacement operation is blocked until an evaluation of a previous one has completed. This algorithm is optimal if the sets $AFFECTED_i$, $1 \leq i \leq k$, are disjoint. If they are not disjoint, however, attributes in the intersection of these sets would always be evaluated more than once. Thus, the naive algorithm does not satisfy the second optimality requirement.

Before examining the asynchronous subtree replacement problem, we first consider a simplified problem — k synchronous subtree replacements. This problem has been studied before in the context of single user environments in order to allow the editing model to include other commands besides subtree replacements [Reps 86]. Such a command would be viewed as multiple subtree replacements occurring at different nodes in the tree, but since all subtree replacements happen by this one command, they are synchronous. Hence, the k subtree

replacement tree operations are done first, and then evaluation of all of them proceeds.

The design of an optimal algorithm for multiple subtree replacements is once again non-trivial because of transitive dependencies among attributes, although in a different way from the single subtree replacement case. Suppose there are two subtree replacements, one at node r_1 in the semantic tree and another at node r_2. There are two attributes associated with r_1, a and b, and two attributes associated with r_2, c and d. The characteristic graphs described in section 2.2 give us enough information to determine that a should be scheduled for evaluation before b, and c before d, because of transitive dependencies from a to b and c to d respectively. However, there is no information to allow us to determine whether a should be evaluated before c. The answer depends on whether there is a dependency from b to c (in which case a should go first), or from d to a (in which case c should go first).

For multiple subtree replacements, two kinds of characteristic graphs are required:

- TDS_X — This graph represents direct and transitive dependencies between attributes of a nonterminal symbol, X. This is the same as Reps' characteristic graphs described above.
- $TDPS_{X,Y}$ — This graph represents direct and transitive dependencies between attributes of a pair of symbols, X and Y.

The complexity of an algorithm for evaluating multiple synchronous subtree replacements depends on (1) the cost of maintaining these two kinds of characteristic graphs for cursor movement and subtree replacement operations, and (2) the cost of using the characteristic graphs in the attribute evaluation algorithm.

A problem that will be addressed in the thesis is determining how efficiently TDS and TDPS graphs can be maintained and used for different classes of AGs. By restricting the class of grammars that the evaluator can handle, it may be possible to compute the transitive information statically, thus improving the performance of the evaluation algorithm. Our goal is twofold: (1) to understand the restrictions that must be placed on an AG in order to compute the transitive information as efficiently as possible, and (2) to find efficient evaluation algorithms for the major known classes of AGs, such as *partitioned* grammars, *absolutely non-circular* grammars,

and *ordered* attribute grammars.[7]

Reps *et al* give an incremental attribute evaluation algorithm for k synchronous subtree changes [Reps 86]. This algorithm is not applicable to the general class of well-defined AGs, but only to a subset of them.[8] The amortized cost of the algorithm after a sequence of update operations is $O((|AFFECTED| + k) \times \log n)$, where k is the number of synchronous subtree replacements and n is the number of nodes in the semantic tree, so this is a known upper bound on the problem.

For asynchronous subtree replacements, a subtree replacement can be interleaved with ongoing evaluations of previous subtree replacements. For the k asynchronous subtree replacement problem, we assume the existence of k editing cursors (*i.e.*, nodes) in the parse tree at which subtree replacements can occur. Unlike the synchronous case, knowing the transitive dependencies among the roots of the replaced subtrees is not sufficient. For example, in figure 3-1, if a subtree replacement at r_1 occurs while evaluation of r_2 is in progress, attributes that are arbitrarily far from r_2 may have to be stopped from being evaluated.

There is a tradeoff between (1) minimizing the number of attributes reevaluated, and (2) the cost of maintaining necessary transitive dependencies to do so. Let us illustrate this tradeoff by describing three algorithms that compromise between these costs in different ways.

Algorithm 1: The first algorithm is prepared to handle a subtree replacement at r_1 at any time. Initially, $TDPS_{r_1,r_2}$ is known. As evaluation of r_2 proceeds, transitive dependencies between attributes of r_1 and attributes that become ready for evaluation in r_2 are computed. This means that as evaluation of r_2 proceeds, the $TDPS_{r_1,r_2}$ graph changes to be $TDPS_{r_1,model(r_2)}$. The transitive information among neighboring nodes can be used to compute this information incrementally as evaluation at r_2 proceeds. The advantage of this algorithm is that if a subtree replacement occurs at r_1 all the necessary information is there to immediately stop those

---

[7]The best result that could come out of the thesis is the determination of lower bounds for the problem of multiple synchronous subtree replacements, and the design of the "best" algorithm for different classes of grammars. If this result is not achieved, though, the design of evaluation algorithms for different classes of AGs that are more efficient than currently known, is still significant.

[8]The class of AGs for which the algorithm described in [Reps 86] works is a subset of the class of grammars that pass step 2 of Kastens' construction for ordered attribute grammars [Kastens 80]. This subclass includes AGs for which the relations TDS_X, for all nonterminals X, and $TDPS_{X,Y}$, for all nonterminals X and Y such that $X \stackrel{+}{\rightarrow} Y$, can be determined statically to be acyclic.

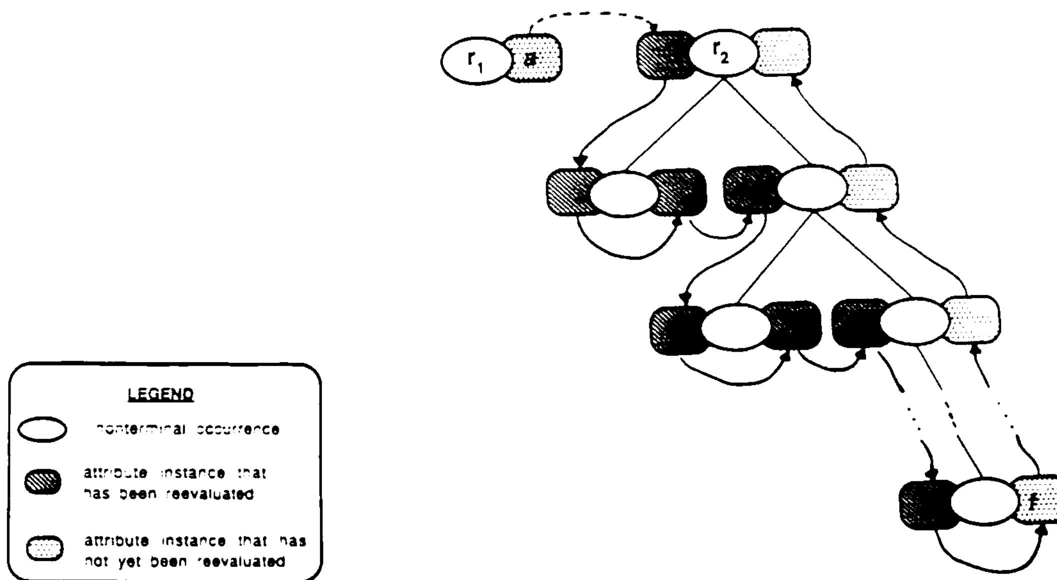

Figure 3-1: *Transitive dependencies for asynchronous changes*

attributes in r_2 that are descendents of attributes of r_1 from being evaluated prematurely, and vice versa. The disadvantage is that if no change is made at r_1 before the evaluation of r_2 completes, all this work is done in vain.

Algorithm 2: The second algorithm only computes the transitive dependencies between attributes of r_1 and attributes in the model of r_2 when a subtree replacement at r_1 occurs. The disadvantage is that a lot of transitive information may have to be computed before any evaluation can proceed, possibly resulting in an unacceptably slow response time after certain editing operations. The advantage is that the cost is only incurred if it is actually needed.

Algorithm 3: The third approach reduces the cost of maintaining transitive information at the risk of recomputing some attributes unnecessarily. In this version, *TDPS* graphs are not maintained at all. Two (or more) evaluations for asynchronous subtree replacements proceed independently until one reaches an attribute evaluated by the previous one. This indicates that a dependency between attributes in the two subtrees may exist. At this point necessary transitive dependencies are computed from scratch to determine which attribute evaluations to suspend. This algorithm is being implemented in our prototype implementation of MERCURY. More details can be found in [Micallef 89].

In the thesis I will design and analyze evaluation algorithms that balance the number of attributes reevaluated and the cost of maintaining transitive dependencies in different ways for well-known classes of attribute grammars. I already have some initial results for this problem [Micallef 88a]. I have designed algorithms for evaluating asynchronous changes for two classes of AGs: ordered attribute grammars (OAGs), and a subclass of OAGs that I call pairwise-ordered attribute grammars (POAGs). For OAGs, the algorithm minimizes the number of attribute evaluations but the bookkeeping costs are not optimal. The complexity of the algorithm is $O(|AFFECTED| \times h)$, where h is the height of the semantic tree. The algorithm for POAGs also minimizes attribute evaluation, but because it can compute more information statically, the bookkeeping costs are proportional to $O((|AFFECTED| + k) \times \log n)$, where k is the number of subtree replacements and n is the size of the semantic tree.

The problem of incremental evaluation for asynchronous subtree replacements has been addressed before. Kaplan and Kaiser were the first to describe an algorithm to solve the problem for the general class of well-defined AGs [Kaplan 86]. Their algorithm is similar to Algorithm 3 described above, but the details of determining the transitive dependencies when the models collided were not included. The algorithm for k synchronous edits of Reps *et al* [Reps 86] was extended to handle asynchronous edits by Geitz [Geitz 87]; the asynchronous version handles the same class of grammars as the synchronous version. Geitz' algorithm minimizes the number of attributes evaluated; however, the bookkeeping costs of the algorithm are not optimal by our definition of optimality. We expect the thesis research to lead to improvements on these results as well as achievement of new results for classes of AGs that were not considered before.

3.2. Segmented Parse Tree

In the previous subsection dealing with multiple subtree replacements, there was a single attribute evaluator that had access to the entire semantic tree in which the changes were being made. This section will consider the issues that arise when the semantic tree is divided into segments, where each segment corresponds to a subtree in the original monolothic semantic tree. For the application of multi-user programming environments, the segment corresponds to the unit of the program being developed by one programmer, such as an Ada package, a Clu cluster, or a Modula-2 module.

Clearly, an evaluator is required for each segment. The evaluator is invoked whenever a change

in that segment occurs. The evaluator for one segment interacts with evaluators of other segments since attribute dependencies may cross segment boundaries resulting in attributes flowing among segments. The collection of cooperating evaluator processes is called the *global evaluator.*

A *segment evaluator* is similar to an evaluator for a monolithic semantic tree except for its actions at interface nodes. An *interface node* is a node that is on the boundary between two segments. Interface nodes are duplicated in the semantic tree of the two segments which they bound. An interface node is either at the root or a leaf of a segment. The *input attributes of a segment* are the inherited attributes of the root interface node of the segment and the synthesized attributes of leaf interface nodes of the segment. The *output attributes of a segment* are the reverse, that is, the synthesized attributes of the root interface node and the inherited attributes of leaf interface nodes. When an input attribute of a segment S changes value because of an edit in an adjoining segment, the evaluator for S treats this as a subtree replacement at the corresponding interface node. When an output attribute's value changes, the evaluator sends a message with the new value to the adjoining evaluator, which will in turn fire up a new evaluation.

How are segments defined? There are two possibilities: statically or dynamically. A segment is statically defined by specifying in the AG which nonterminal symbols can be interface nodes. This can be restricted further to only allow a flat segment structure; that is, two nonterminal symbols X and Y, where $X \xrightarrow{+} Y$, cannot both be interface nodes. A segment is dynamically defined if the programmer can create a new segment from any node in the parse tree at any time during program development.

A thesis problem is to design segmented evaluators for both statically defined segments and dynamically defined segments, and compare them with respect to ease of construction and efficiency of the evaluator. Boehm and Zwaenepoel describe a parallel compiler based on AG technology which is related to the segmented evaluators described here [Boehm 86]. The parse tree of the source program that is being translated is divided into subtrees, which are evaluated in parallel by evaluators executing on different machines. Their algorithm uses a combined static and dynamic evaluation strategy: attributes that depend on other attributes associated with nodes in a different subtree are computed dynamically, whereas those whose arguments are in the same subtree are computed according to a statically precomputed plan. The main difference from the

problem described above is that the evaluator for each segment is not incremental — it evaluates all the attributes in the segment.

3.2.1. Optimality Considerations

A segment evaluator is *locally optimal* if evaluation of multiple subtree replacements within the segment, arising from local changes within that segment as well as remote changes in other segments that have propagated to this one, is optimal according to the definition given in section 3.1.1. *A thesis problem is to determine whether restricting segments to be defined statically makes it possible to reduce the cost of keeping track of transitive dependencies, thus improving the complexity of the evaluation algorithm.*

What about optimality among segment evaluators? A subtree replacement may cause several visits to an adjoining evaluator. *Heuristics for improving performance of the cooperating evaluators by (1) scheduling as few visits as possible, and (2) scheduling a visit a soon as possible will be investigated in the thesis.* Yellin has shown that a problem related to (1) is NP-complete [Yellin 84]. Heuristics have been developed for non-incremental evaluators used in compiler-compilers; I will determine whether these heuristics can be adapted for our problem, or whether new ones are needed.

An issue that arises in the construction of the segment evaluators concerns the organization of segment interconnection information. There are two possible ways of keeping track of this information:

- Centralized — the interconnection structure between all the segments is managed by a separate process, and communication between different segment evaluators is done via this process.
- Distributed — each evaluator has built into it which other evaluators it can communicate with.

What are the tradeoffs for these two options? Keeping the segment interconnection information distributed, with each evaluator keeping track of the evaluators it communicates with, is attractive because it avoids a heavy concentration of messages at a central server, and information is localized where it is used. However, when the segments reside on different machines (the topic of the next subsection), keeping the segment interconnection information as a separate layer is advantageous for segment location independence.

3.3. Distribution of Segments

When segments are distributed on different machines, replication of segment interconnection information is desirable for high availability. For example, consider a program *P* that consists of a main block containing some global declarations and three modules *A*, *B*, and *C*. Suppose that *P* is divided into four segments, *Main* and the three modules, where *Main* and module *A* reside on workstation *Frodo*, and *B* and *C* reside on workstation *Gollum*. Furthermore, suppose that there is a chain of attribute dependencies from module *B* to module *C* which go through *Main*. If a change occurs in *B*, *B*'s evaluator sends a message to the evaluator for *Main* on *Gollum*, which eventually sends a message to the evaluator for *C* back on *Frodo*. The propagation of the change to *C*, therefore, depends on the message transmission time between the two machines and the load on *Gollum*. By replicating *Main* on both machines, *C* will be informed of the new information immediately, irrespective of the load or communication delay on *Gollum*.[9]

Replication, however, does not come without cost. There needs to be additional mechanisms for keeping the replicated entities consistent. Keeping replicated information consistent is usually done pessimistically, by synchronizing updates to avoid inconsistencies. In certain situations, an optimistic approach may be possible; that is, updates are allowed to happen without synchronization, and compensatory actions taken if a conflict arises. The optimistic approach is preferable because it avoids synchronization among the segment evaluators, and synchronization necessarily means waiting. *A thesis problem is to find ways to minimize synchronization costs in our application, multi-user environments. I will also evaluate the tradeoffs involved with replication to determine under what conditions replication pays off.*

An example of when synchronization may be necessary is the creation of new segments. A segment can be created in two ways:

- By splitting off a subtree of an existing segment and making it a new segment.
- By creating an initially independent segment and then specifying its position in the program structure.

---

[9]Replication is even more crucial when dealing with unreliable networks as it is not acceptable for the evaluation algorithm to come to a stall because some machine is inaccessable, due to either machine or network failure. As mentioned previously, the scope of the thesis is limited to reliable networks, and therefore we will only consider the issues that arise with replication when the network is reliable. In particular, we are not concerned with algorithms for restoring consistency among the replicated entities after recovery from a machine or network fault. Such algorithms have already been developed, both for this application of multi-user distributed environments [Kaiser 87a], and for more general database applications [Garcia-Molina 88].

The first situation poses no problems if only one programmer has write access to a segment, which is a reasonable assumption to make. However, in the second case, it may be possible for two programmers to create two segments independently that when added to the program as specified cause a conflict. For example, if Ada is the implementation language, adding two packages with the same name to a program is illegal since packages must be uniquely identified. How should the environment handle such a situation? It is inappropriate to flag one of the packages as erroneous as is done if the collection of packages were compiled since it is impossible to determine which of the packages was added first. (We do not assume that there is a global clock in the distributed environment.). A more appropriate action would be to flag both of them as erroneous and leave it to the two programmers to resolve. Another approach would be to synchronize the operation of attaching new segments to the program, that is, guarantee that only one programmer can be doing this at any one time. This effectively places a total order on the creation of new segments so that the environment can flag just the duplicate module that was added last as the erroneous one.

When segments are distributed on a network, an additional factor must be considered in the performance of the attribute evaluation algorithms — network traffic. This can be measured either in the number of messages transmitted due to one evaluation, or in the number of bytes transmitted. *Another thesis issue is to find algorithms that minimizes the number of messages/bytes sent among the segment evaluators.*

Some initial work has been done along this line already concerning aggregate attributes at interface nodes. An *aggregate attribute* consists of many components; for instance, a symbol table is usually defined by an aggregate attribute where each component corresponds to an entry for one symbol. A well-known problem with aggregate attributes is that a change to one component of the aggregate results in the reevaluation of all attributes that depend on <u>any</u> component of the aggregate. For instance, a new variable declaration results in reevaluation of all variable references in the scope of the changed declaration. In the distributed environments, a change to an exported facility in one segment is propagated to <u>all</u> segments, including those that do not import the facility. I have designed a new attribute evaluation algorithm called *selective propagation* which propagates a change in one program segment to a second segment only if the latter actually <u>uses</u> the changed information [Micallef 88b]. This is done by maintaining *use-lists* for each component of the aggregate interface attributes.

4. Conclusion

4.1. Preliminary Implementation

A preliminary implementation of the research described here has been done in the MERCURY prototype [Kaiser 87b], a generator of distributed editing environments. MERCURY is implemented on top of the Cornell Synthesizer Generator, which generates editing environments for single users developing monolothic programs. The generated editors run on Suns or Vaxen connected by an ethernet. The incremental attribute evaluation algorithm currently implemented handles asynchronous edits within a segment sequentially. The segment interconnection organization is currently limited to a flat organization, and new segments are added to the program with no synchronization. The segment interconnection information is encapsulated within the *attribute propagation layer* (APL), which handles inter-editor attribute propagations. The APL is implemented as a continuously running process on each machine in the environment, and replicates the segment interconnection information on each machine.

4.2. Research Plan

Below we summarize the tasks to be achieved in the thesis. The entire thesis should be completed by May, 1990.

- Multiple asynchronous subtree replacements:
    - Find lower bounds on cost of maintaining transitive dependencies for different classes of attribute grammars.
    - Design attribute evaluation algorithms that achieve these lower bounds.
    - Explore balance between bookkeeping costs and number of attribute reevaluations for a given AG.
- Segmented Evaluators:
    - Design and compare segmented evaluators for statically and dynamically defined segments.
    - Explore whether statically defined segments can be exploited in order to optimize bookkeeping costs required to handle multiple changes within a segment.
    - Find and compare heuristics for scheduling interactions among evaluators to improved global performance of the cooperating segment evaluators.
- Distributed Segments:
    - Evaluate tradeoffs involved with replication.

- Explore alternatives for minimizing synchronization costs in multi-user environments.
- Design incremental attribute evaluation algorithms that minimize network traffic.

The most important part of the thesis, and also where most of the difficulty lies, is the algorithms and lower bound complexity analysis for multiple subtree replacements. I intend to tackle the three subproblems listed above for this milestone in parallel, that is, design the algorithms while simultaneously try and understand the inherent difficulty of the problem. I expect the research for this part of the thesis to be completed by December 1989. The construction of segmented evaluators depends to a certain extent on the evaluation algorithms. Therefore, some of the research in this subtopic will have to occur after the first milestone is completed. I expect to have this section of the thesis done by February 1990. The issues that arise when the segments are distributed are loosely coupled from the other two problem characteristics. I have made some initial headway in this area, so I expect this part to be finished first, by October 1989.

4.3. Contributions

The practical contribution of this research is the development of a programming environment generator — MERCURY — that supports teams of programmers collaborating on the development and maintenance of large software systems. The environment supports incremental checking of interdependencies among modules, whether residing on the same machine or distributed across multiple machines connected by a network. Each module is edited using a language-based programming environment, previously suited only to programming-in-the-small. By supporting change propagation among many such environments, we achieve programming-in-the-many. The technical contribution of the research lies in the development of new attribute grammar algorithms for the kernel of the distributed multi-user environments, which expands the body of knowledge about AGs. These algorithms can be used in other applications besides language-based editors, such as support for distributed data bases and implementation of distributed dataflow languages.

References

[Boehm 86] Hans-Juergen Boehm and Willy Zwaenepoel.
Parallel Attribute Grammar Evaluation.
1986.
Rice University.

[Brooks 82] Frederick P. Brooks, Jr.
The Mythical Man Month.
Addison Wesley, Reading, MA, 1982.

[Farrow 84] Rodney Farrow.
Generating a Production Compiler from an Attribute Grammar.
IEEE Software 1(4), October, 1984.

[Ganzinger 77] Harald Ganzinger, Knut Ripken and Reinhard Wilhelm.
Automatic Generation of Optimizing Multipass Compilers.
In *Information Processing 77*, pages 535-540. North-Holland Pub. Co., New York, 1977.

[Garcia-Molina 88]
Hector Garcia-Molina and Boris Kogan.
Achieving High Availability in Distributed Databases.
IEEE Transactions on Software Engineering :886-896, July, 1988.

[Geitz 87] Bob Geitz.
Asynchronous Subtree Replacement for Language-Based Editors.
1987.
Oberlin College and Cornell University.

[Hudson 88] Scott E. Hudson and Roger King.
Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System.
Technical Report, The University of Arizona, Department of Computer Science, March, 1988.

[Jazayeri 75] M. Jazayeri, W.F. Ogden and W.C. Rounds.
The Intrinsically Exponential Complexity of the Circularity Problem for Attribute Grammars.
Communications of the ACM 18(12):697-706, December, 1975.

[Kaiser 87a] Gail E. Kaiser and Simon M. Kaplan.
Reliability in Distributed Programming Environments.
In *Sixth Symposium on Reliability in Distributed Software and Database Systems*, pages 45-55. Kingsmill—Williamsburg, VA, March, 1987.

[Kaiser 87b] Gail E. Kaiser, Simon M. Kaplan and Josephine Micallef.
Multiple-User Distributed Language-Based Environments.
IEEE Software :58-67, November, 1987.

[Kaiser 89] Gail E. Kaiser, Steven S. Popovich, Wenwey Hseush and Shyhtsun Felix Wu.
Melding Multiple Granularities of Parallelism.
In *European Conference on Object-Oriented Programming*. Nottingham, UK, July, 1989.
In press.

[Kaplan 86] 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.

[Kastens 80] Uwe Kastens.
Ordered Attribute Grammars.
Acta Informatica 13:229-256, 1980.

[Kastens 82] U. Kastens, B. Hutt and E. Zimmermann.
Lecture Notes in Computer Science. Volume 141: *GAG: A Practical Compiler Generator.*
Springer-Verlag, Heidelberg, 1982.

[Knuth 68] Donald E. Knuth.
Semantics of Context-Free Languages.
Mathematical Systems Theory 2(2):127-145, June, 1968.

[Micallef 88a] Josephine Micallef.
Incremental Evaluation of Ordered Attribute Grammars for Asynchronous Subtree Replacements.
Technical Report, Columbia University, Computer Science Dept., September, 1988.

[Micallef 88b] Josephine Micallef and Gail E. Kaiser.
Version and Configuration Control in Distributed Language-Based Environments.
In *International Workshop on Software Version and Configuration Control.* Grassau, West Germany, January, 1988.

[Micallef 89] Josephine Micallef and Yael Cycowicz.
Merging Scheduling Graphs for Asynchronous Subtree Replacements.
1989.
In progress.

[Reps 82] Thomas Reps.
Optimal-time Incremental Semantic Analysis for Syntax-directed Editors.
In *Ninth Annual ACM Symposium on Principles of Programming Languages.* January, 1982.

[Reps 83] 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.

[Reps 84a] Thomas Reps.
Generating Language-Based Environments.
M.I.T. Press, Cambridge, MA, 1984.

[Reps 84b] Thomas Reps and B. Alpern.
Interactive Proof Checking.
In *11th ACM Symposium on Principles of Programming Languages.* January, 1984.

[Reps 86] T. Reps, C. Marceau and T. Teitelbaum.
Remote Attribute Updating for Language-Based Editors.
In *Thirteenth ACM Symposium on Principles of Programming Languages,* pages 1-13. St. Petersburg Beach, FL, January, 1986.

[Tichy 79] Walter F. Tichy.
Software Development Control Based on Module Interconnection.
In *4th International Conference on Software Engineering.* September, 1979.

[Yellin 84] Daniel Yellin.
A Survey of Tree-Walk Evaluation Strategies for Attribute Grammars.
Technical Report, Columbia University, Computer Science Dept., September, 1984.