4.1 Introduction

Dynamic incremental evaluators for SOEs are normally based on the Attribute Grammar formalism. In this chapter we present a brief description of AGs. Further, since dynamic evaluators operate on dependency graphs arising out of attributes and their defining equations, algorithms for evaluating attributes are proposed. We approach the problem with the technique of step-wise refinement. A naive non incremental evaluation algorithm is presented and we refine the algorithm, pointing out the lacunae and at the same time defining bounds. Finally, a data driven algorithm is proposed which forms the basis of the incremental attribute evaluation algorithm presented later in this thesis.

4.2 Attribute Grammars

Attribute Grammars(AG)[KNU68,71] are a powerful mechanism for the specification of context sensitive properties of a language. Formally an AG is a five tuple $AG = (G, SD, AD, R, C)$, where

1. $G = (V_T, V_N, P, Z)$ is a context free grammar. $V_T$ are $V_N$ a finite set of terminal and nonterminal symbols. $P$ is a finite set of productions and $Z \in V_N$ is the distinguished symbol. A production $p \in P$ is $p : X_{p_0} \rightarrow X_{p_1}, \ldots, X_{p_n}$ where $n_p \geq 0, X_{p_0} \in V_N, V = V_T \cup V_N$ and $X_{p_k} \in V$ for $1 \leq k \leq n_p$. 

2. $SD$ is the semantic domain denoting a finite set of types and a finite set of total functions. $SD = (T, F)$ where $T$ is a finite set of sets and $F$ is a finite set of total functions of type $t_i \times \cdots \times t_n \rightarrow t_0$, where $n \geq 0$ and $t_i \in T$ for $0 \leq i \leq n$.

3. $AD = (A, I, S, T_y)$ is a set of attribute descriptions. Each symbol $X \in V$ has an associated set of attributes $A(X)$ which can be partitioned into two disjoint subsets $I(Y)$ and $S(Y)$, the inherited and, the synthesized set of attributes respectively. The set of all attributes is $A = \bigcup_{X \in V} A(X)$. For $a \in A, T_y(a) \in T$ is the set of possible values of attribute $a$.

4. $R(p)$ is a set of attribute evaluation rules (semantic rules) associated with $p \in P$.

A production $p$ is said to have an attribute occurrence $(a, p, k)$ if $a \in A(X_p)$ for a production. The set of all attribute occurrences of a production $p$ will be denoted by $AO(p)$. This set may be partitioned into two disjoint subsets of defined occurrences $DO(p)$ and used occurrences $UO(p)$, respectively.

$$DO(p) = \{(s, p, o) | s \in S(Y_p) \} \bigcup \{(i, p, k) | i \in I(Y_p) \land 1 \leq k \leq n_p \}$$

$$UO(p) = \{(i, p, o) | i \in I(Y_p) \} \bigcup \{(s, p, k) | s \in S(Y_p) \land 1 \leq k \leq n_p \}$$

The semantic rules in $R(p)$ compute the values of attribute occurrences in $DO(p)$ as a function of other attribute occurrences in $AO(p)$. A semantic rule is of the form $a := f(a_1, \ldots, a_m)$, where $a \in DO(p), a_i \in AO(p)$ for $1 \leq i \leq m$.

$$f : T_y(a_1) \times \cdots \times T_y(a_m) \rightarrow T_y(a), f \in F$$

An AG is in normal form if $a_i \in UO(p)$ for $1 \leq i \leq m$ holds for every semantic rule.

5. $C(p)$ is a finite set of semantic conditions associated with production $p$. These conditions are predicates of the form $f(a_1, \ldots, a_m)$ where

$$f : T_y(a_1) \times \cdots \times T_y(a_m) \rightarrow \{true, false\}, f \in F \text{ and } a_i \in AO(p) \text{ for } 1 \leq i \leq m.$$
sentence generated by $G$ is a sentence of $AG$ only if the semantic conditions hold true.

Each sentence generated by a $CFG$ can be assigned a single derivation tree for a production $p:X_{P_0} \to X_{P_1},...,X_{P_n}$, the derivation tree is such that the node is labelled $X_{P_0}$ and its children are labelled $X_{P_1},...,X_{P_n}$ respectively.

Given a derivation tree, instances of attributes when attached to it, convert it into a semantic tree. If a node $r$ is labeled with the syntactic symbol $X$, then for each attribute $a \in A(X)$, an instance of $a$ is attached to node $r$.

**Definition 4.1:** A semantic tree, together with an assignment of a value to each attribute instance of the tree is an attributed tree. It may or may not be consistent.

**Definition 4.2:** A consistently attributed derivation tree is a derivation tree in which all the attribute instances are defined according to their associated attribute evaluation rules.

**Definition 4.3:** An attribute grammar is well defined if for each structure tree corresponding to a sentence of $L(AG)$, the language defined by $AG$, all attribute instances are effectively computable.

**Definition 4.4:** A dependency graph $D(T)$ for an attributed derivation tree $T$ may be formed as follows: The vertices of $D(T)$ are all the attribute instances of $T$ while directed edges $(a,b) \in D(T)$ iff attribute $b$ depends on attribute $a$.

Let $i$ be the indegree of $v$.

If $v.val = f_i(pred(0).val,pred(1).val,...,pred(i-1).val)$ where $pred(k), 0 \leq k \leq i$, are the predecessors of node $v$, then $v$ is consistent else inconsistent. If all $v \in V$ are consistent, then $D(T)$ is a consistent dependency graph otherwise inconsistent.

We also assume that there is a set of vertices needed by the user, termed the demand vertices.
Definition 4.5: An attribute evaluator computes the values of all attribute instances attached to a derivation tree by evaluating the semantic functions attached to these attribute instances.

Definition 4.6: An attribute grammar is well formed or non circular iff the graph of $D(t)$ is acyclic for each structure tree $t$ corresponding to a sentence of $L(G)$.

For a structure tree $t$ corresponding to a sentence of $L(AG)$, there exists an evaluation order iff the $AG$ is well formed.

An attribute instance is available if its value is defined, otherwise it is unavailable. At the beginning of the evaluation process, all attributes with the exception of the inherited attributes of the root and the synthesised attributes of the leaves of the parse tree are unavailable. We may also define a third class of attributes -- the intrinsic attributes. Attributes such as the value of a constant or the symbol of an identifier which arise in conduction with the structure tree construction are the intrinsic attributes.

For a well formed $AG$, a non-deterministic algorithm may be used to compute the values of all attributes in any structure tree belonging to the language $L(AG)$. Intrinsic attributes become available as soon as the parse tree is built. Computation starts with the intrinsic attributes, and proceeds with other attribute instances.
4.2.1 Example

\[ P_1 : \quad Z ::= N_1 N_2 \]
\[ v^Z = v^{N_1} + v^{N_2} \]
\[ r^{N_1} = 0 \]
\[ r^{N_2} = -1^{N_2} \]

\[ P_2 : \quad N_0 ::= N_1 B \]
\[ v^{N_0} = v^{N_1} + v^B \]
\[ l^{N_0} = l^{N_1} + 1 \]
\[ r^B = r^{N_0} \]
\[ r^{N_1} = r^{N_0} + 1 \]

\[ P_3 : \quad N ::= \varepsilon \]
\[ v^N = 0 \]
\[ l^N = 0 \]

\[ P_4 : \quad B ::= 1 \]
\[ v^B = 2^r \]

\[ P_5 : \quad B ::= 0 \]
\[ v^B = 0 \]

Attribute Grammar AG1

The derivation tree and the associated attribute dependency graph for a string 101.101 is given in Figure 4.1. This is perhaps one of the examples most used to explain the various intricacies of Attribute Grammars. AG1 specifies the syntax of a binary floating point number. The semantic rules associated with the symbols, compute the decimal equivalent of the binary number. Attributes \( r, v, l \) are attached to non terminal \( N \), and attributes \( r \) and \( v \) to nonterminal \( B \). Attribute \( v \) attached to \( Z \), contains the decimal equivalent after the final evaluation.
4.2.2 One visit AGs

Single or one visit AG's have such property that in a single tree walk, all the attribute instances can be computed without visiting any subtree a second time. [Eng81] introduced the concept of a brother graph, which for a production $X_{i_1} \rightarrow X_{r_1}, \ldots, X_{i_r}$ of AG, is a graph whose nodes are the non-terminal occurrences on the RHS of $p$ such that there is
an arc from \( X_p \) to \( X_{P_i} \) iff some attribute occurrence \((b, p, k) \in DO(p)\) depends on some attribute occurrence \((a, p, j) \in UO(p)\).

An attribute grammar is one visit iff for every production \( p \) of \( AG \), the brother graph \( BG_p \) has no oriented cycles.

An \( AG \) is \( l \)-\textit{attributed} if there are no right to left dependencies between the attribute occurrences of its productions.

\textbf{Definition 4.7:} An \( AG \) is \( l \)-\textit{attributed} (one pass left to right) if for each production \( X_{P_0} \rightarrow X_{R_1}, \ldots, X_{P_n} \), with attribute occurrences \((a, p, j)\) and \((b, p, k)\), such that \((b, p, k)\) depends on \((a, p, j)\); the following condition is satisfied: if \( 1 \leq k \leq n_p \) then \( j < k \).

[Lew74] proves that for an \( LL(k) \) context free grammar which forms the basis of an \( l \)-\textit{attributed} \( AG \), attribute instances in any derivation tree can be evaluated in one pass during the top down parsing. [Lew74] also state that any translation by an \( l \)-\textit{attributed} grammar with an \( LL(k) \) input grammar can be performed by a deterministic attributed pushdown machine as well by a recursive descent parser.

During the evaluation pass over a derivation tree, each tree node is visited twice. Inherited attribute instances of a node are evaluated during the pre-order visit and synthesised attribute instances during the post-order visit. [Gie78] define an \( AG \) as \( LL \)-\textit{attributed} if the grammar is \( l \)-\textit{attributed} and the underlying \( CFG \) is \( LL(k) \).

\subsection*{4.2.3 S-attributed AGs}

An attribute grammar is \( S \)-\textit{attributed} if it has only synthesised attributes and no inherited attributes. \( Y_{x \in S} I(X) = \varnothing \)
Attribute evaluation in any derivation tree of an $S$-attributed $AG$ whose underlying context free grammar is $LR(k)$ can be evaluated in one pass during bottom up parsing. Synthesized attribute instances of a tree node are evaluated during a post-order visit of that node.

One visit $AG$s are not as powerful as the other classes of $AG$s, but they offer themselves to very simple evaluation strategies. Quite a few systems have been developed taking them as their theoretical foundation. [Fil83] show that arbitrary $AG$s are more powerful than $I$-attributed $AG$s, and that $I$-AGs are only slightly more powerful than $S$-AGs. More often than not, $AG$s have been classified by means of their evaluators rather than their inherent properties. According to [Fil83], $AG$s can be defined by means of their static properties of their attribute dependencies, and along with these, an attribute evaluation scheme is defined for each such class.

### 4.3 Dependency Graph

A dependency graph is normally denoted by $D(T) = (V, E)$, where

$V$ is a finite set of vertices and $E$ is a finite set of directed edges $(v_1, v_2)$ where $v_1, v_2 \in V$. The formal definition of a dependency graph is given in definition 4.2.

We shall slightly modify the definition of a dependency graph by including an additional set of vertices, namely, the result vertices.

The result vertices $R \subseteq V$, may be the same as demand vertices, the set of vertices as demanded by the user.

Therefore, a dependency graph will be denoted by $D_R(T) = (V, E)$, if the result vertices are specified.
Definition 4.8: If all the vertices in \( D(T) \) are consistent, then \( D \) is a consistent dependency graph. The set \( \{ \text{val}_v | v \in V \} \) for a consistent dependency graph \( D \) is known as a solution of \( D \). The set \( \{ \text{val}_v | v \in R \} \) is called the result of \( D \).

4.3.1 Example of a Dependency Graph

The dependency graph of Fig 4.11 is consistent. The defining equations of the vertices are given below.

\[ v_1.\text{val}=5 \]
\[ v_2.\text{val}=\text{if } v_1.\text{val}<3 \text{ then true else false} \]
\[ v_3.\text{val}=\text{if } v_2.\text{val} \text{ then } v_1.\text{val}^2 \text{ else } v_2.\text{val}^3 \]
\[ v_4.\text{val}= v_3.\text{val}+ v_{14}.\text{val} \]
\[ v_5.\text{val}= v_3.\text{val} \]
4.3.2 Graph Evaluation

The process of obtaining a result of a given dependency graph is called graph evaluation. The algorithm which performs this process is known as a graph evaluator.

There are a number of approaches to graph evaluation, but, the core idea behind each remains the same. The dependency graph is traversed in a topological order, and, the vertex functions are evaluated to compute the vertex values corresponding to the result vertices. This gives the result of the algorithm.

If $D(T)$ is an acyclic graph then, the evaluation may be affected using Knuth's topological sort.

4.3.3 Knuth's Topological Sort

The value of an attribute instance is computed according to its defining equations. Here, the attributes are evaluated under such a constraint that a vertex function is available only when all its argument attributes are available. The functional dependencies among the attributes in the graph create a partial ordering on the attribute instances.

Any evaluation algorithm must obey this partial order. The attribute instances are therefore, sorted in a topological order, preserving the partial order among them. Since, all
the vertex functions have to be evaluated, in addition to the bookkeeping work, the cost is proportional to $\Theta(EVAL(V) + |E| + |V|)$, where $EVAL(V)$ is a measure of the time taken to compute all the vertex functions. Normally, the computation time for vertex functions is taken as $O(1)$. But, since this might not always be the case, we use the undefined measure $EVAL$ to take care of situations where the exact nature of the functions are not known. The genesis of most of the techniques presented hereafter lie in Knuth's topological sort.

If the dependency graph contains unnecessary computations and the vertex functions are not strict, then the evaluation time can be reduced by performing lazy computations starting from the result vertices.

### 4.3.4 Demand Driven Evaluator

Computations are initiated at the result vertices and the values of arguments needed are demanded. This demand triggers off the computation of yet other vertices and the process continues till a vertex is reached, whose function uses no arguments. The computations are stored at their respective vertices, so that they may be re-used if required. The bookkeeping time [Jal85] is here again $\Theta(|E| + |V|)$, with the difference that only those vertex functions are computed which are actually needed for computing the result vertices.

### 4.3.5 Incremental Attribute Evaluation

The abstract syntax tree is a basic data structure around which most of the operations of an SOE work. They allow the user to directly manipulate the underlying AST. An arbitrary program edit may result in a sequence of pruning and grafting operations on the tree. These may be collectively termed as the subtree replacement operations. There are two distinct strategies that are followed. First, if the tree is taken as the basis of operation, tree walking algorithms restore the tree to its consistent state. A host of such tree walking
algorithms have been proposed. Second, the dependency graph with vertices representing
the attribute instances and the directed edges representing the dependencies, may be taken
as the structure for operations. Any subtree replacement operation may, therefore, result
in addition, deletion or modification of the dependency graph. Consequently, the problem
essentially is reduced to evaluating the graph.

4.3.6 Dependency Graph Modifications

Given a dependency graph \( D(T) \), let \( \tilde{D}(T) \) be a dependency graph obtained by an
operation like deleting or adding an edge or a component, or by changing a vertex
function. The process of changing \( D(T) \) into \( \tilde{D}(T) \) is a dependency graph modification.
Later in this chapter, we give an enhanced definition of dependency graph modifications.

4.3.7 Incremental Graph Evaluation

We use dependency graphs to represent computations that we wish to make incremental.
Given an initial dependency graph \( D_0(T) \), which is consistent, we perform a sequence of
dependency graph modifications \( O_1, O_2, \ldots, O_n \), where \( O_k \) alters \( D_{k-1}(T) \), forming \( D_k(T) \). The
process of determining the result for each \( D_k(T) \) is called incremental graph evaluation. It
is performed by an algorithm termed the incremental graph evaluator.

Initially, a non-incremental evaluator may be invoked to restore a dependency graph to its
consistent state. The other way is to interleave graph building and modification steps with
calls to the incremental graph evaluator.

Hoover[Hoo87] defines a graph evaluable scheme as follows: let \( B \) be a set of objects. A
system that translates a computation on any \( b \in B \) into a dependency graph and also
translates modifications to \( B \) into dependency graph modifications is known as a graph
evaluable scheme for the objects in \( B \).
A graph evaluable scheme $S$ is said to be bounded if there is a constant $c$ such that for every object $b \in B$, the indegree and the outdegree of each vertex in the dependency graph is less than $c$. Attribute Grammar based subtree replacement operations are examples of bounded graph evaluation schemes.

4.4 Attribute Grammar Based SOEs

AGs have traditionally been used as specification tools for compiler writing systems like FOLDS[Fan72], Delta[Lor77], HLP[Rai78], GAG[Kas82] and LINGUIST 86[Far82]. They have also been used as a specification language for systems generating code generators and code optimisers. Here we shall present a brief description of how AGs can be used in systems for generating SOEs.

AGs are powerful because they are capable of formally specifying the context sensitive properties of a particular language, which normal context free grammars are not able to represent. Generally, when AG based SOEs are used for development of programs, the various attribute instances and the interrelation between them is used for performing the static semantic analysis of the program under development. We shall describe a very simple model of editing which was used in one of the earliest SOEs, the Cornell Program Synthesiser.

The editor represents the file as an attributed tree of the specifying AG. When additions, deletions or modifications are made in the program, corresponding changes are reflected in the attributed program tree. Consequently, the tree may become inconsistent. Semantic analysis is carried out by re-evaluating the values of attribute instances and reestablishing consistency throughout the tree. Certain attribute values may be used to display error information, if a violation of any of the context sensitive rules is detected. Other types of
attribute values may be used to produce prettyprinting functions or format the display for the user's convenience.

4.4.1 A Simple Model of Editing

In this model, a program is developed by growing a semantic derivation tree. During the process of development, a partial derivative tree is formed because of unexpanded nonterminals. This can be a source of problem, because, an unexpanded non-terminal cannot provide the synthesised attribute values, which may be demanded by its sibling or parent non-terminals in the derivation tree.

To overcome this problem, we generally include in the grammar a completing production rule $X \rightarrow \bot$, for each non-terminal symbol $X$ of the grammar. The symbol $\bot$ denotes an unexpanded non-terminal. By using this, every production can be considered to be complete. With the assignment of some attribute values to the $\bot$ symbol, the derivation tree becomes fully attributed as well.

Modifying a program results in restructuring of the derivation tree by pruning and grafting subtrees on to the derivation tree. Let $T$ be a semantic tree and $U$ be a subtree of $T$ with root node $r$ labelled $X$. $U$ is pruned from $T$ by removing the subtree rooted at $r$. Let $\overline{T}$ be a semantic tree with root $\overline{r}$, also labelled $X$. $\overline{T}$ is grafted on to $T$ at leaf $r$ labelled $X$, by assigning the synthesised attribute values of $r$ to the synthesised attribute instances of $\overline{r}$and then replacing $r$ by $\overline{T}$ in $T$. The subtree replacement of $U$ by $\overline{T}$ is defined as pruning of $U$ followed by grafting of $\overline{T}$ in its place.

At every stage during the editing, the editing cursor is placed at an interior node of the semantic tree. An editing session may be viewed as a sucession of replacement operations
and cursor motions, starting from the complete fully attributed semantic tree of Figure 4.III.

With the cursor positioned at the root, each insertion at an unexpanded non-terminal labelled $X$ is viewed as a replacement of an instance of the completing production of $X$ by a free standing tree $\overline{t_3}$ with root labelled $X$.

Suppose the derivation is made according to the production $X \rightarrow ABC$, where $A$ and $C$ are non-terminals. $\overline{t_3}$ is as in the Figure 4.IV.

Each deletion is viewed as the replacement of the subtree $U$ with root $X$ by an instance of the completing production of $X$. 
During the development of the program, modifications are apt to take place at any location in the program, and not necessarily at the node derived from the root symbol. A subtree which may be deleted from the main program tree becomes a free standing tree with root $X$. Such trees are retained and may be inserted into the program elsewhere.

An incremental evaluator is required to produce a consistent, fully attributed tree after each such subtree replacement.

### 4.5 Incremental Evaluators

We use graph evaluation to implement a graph evaluable scheme, so that after each derivation step is performed, the result of the new dependency graph is readily available. Evaluating the whole graph after a minor modification in the object can be very expensive and may not be feasible in practice, except in very specific situations. A small change in some of the objects represented in the graph may lead to only a minor perturbation in the overall graph, with very limited impact and reach. It is therefore, necessary to go in for an incremental evaluation scheme, so that the cost of computation is minimised, and only those vertices which change values are computed. We define some of the parameters which will be used to characterise these evaluators.

**Definition 4.9:** $AFFECTED$: Let $D(T)$ be a possibly inconsistent dependency graph. The set $AFFECTED$ of $D(T)$ is the set of vertices in $D(T)$ whose value is different from the solution of the $D(T)$.

**Definition 4.10:** $EVAL$: Let $D(T)$ be a dependency graph with a vertex subset $S$. $EVAL$ ($S$) is defined by the total time required to evaluate the vertex function of each element of $S$ using values of solutions of $D(T)$ as the function argument.
An optimal incremental evaluator will compute only the set $AFFECTED$ with a time complexity of $\Theta(EVAL(AFFECTED))$. Any evaluator whose time complexity is greater than this will lead to sub-optimal evaluation.

The most important and widely used variety of incremental evaluators follow the concept of graph propagation.

### 4.5.1 Transactions

Let $\bar{D}_0(T_0)$ be the initial dependency graph. We assume a series of dependency graph modifications:

$$
\bar{D}_0(T_0) \xrightarrow{\alpha} D_0^1(T_0^1) \xrightarrow{\alpha} \ldots \xrightarrow{\alpha} D_0^i(T_0^i)
$$

Let $\bar{D}_0(T_0)$ be the initial dependency graph which is consistent. We perform a series of dependency graph operations $O_n, 1 \leq n \leq i$, on the dependency graph and obtain an inconsistent dependency graph $D_0^i(T_0^i)$. The various operations $O_n$, on the dependency graph can be of the following forms:

1. Adding a vertex,
2. Deleting a vertex,
3. Adding an edge,
4. Deleting an edge,
5. Adding a component,
6. Deleting a component, or
7. Modifying a vertex function.

In a consistently attributed derivation tree $T_0$, $\bar{D}_0(T_0)$ represents the underlying dependency graph reflecting the relationship between the attribute instances at the various nodes. Any modification in the derivation tree as a result of a sub-tree replacement
(deletion and subsequent addition of another sub-tree) or any other change will transform it into another tree \( T'_0 \) with \( D'_0(T'_0) \) as the corresponding dependency graph. After a series of such changes we have a derivation tree \( T'_0 \) whose underlying graph is \( D'_0(T'_0) \). Graph propagation is applied to this graph to return it to its consistent state \( \bar{D}_i(T_i) \) again. A series of such modifications is called a transaction. After each transaction, the graph propagator is invoked and the dependency graph restored to its consistent state. Before the \( k^{th} \) invocation of the of the graph propagator, the state of the dependency graph is represented by \( D_{k-1}(T_{k-1}) \), assuming \( i \) modifications were involved in the \( k-1^{th} \) transaction. The graph propagator restores this dependency graph to its consistent state \( \bar{D}_k(T_k) \).

### 4.5.2 Graph Propagation

As we keep on making modifications in the dependency graph, we keep a set of vertices which may have possibly changed values or may have been added to the graph. This is the set of possibly inconsistent attributes. In order to find a solution of the graph or to re-establish consistency in the graph, we use a graph propagation algorithm. This algorithm takes the dependency graph along with the list of possibly inconsistent vertices and outputs a fully attributed consistent graph.

A host of graph propagation algorithms have been proposed and used in incremental systems. Very few can work absolutely arbitrary dependency graphs but a majority works on sub-classes and restricted classes of dependency graphs. In the following sections, we discuss in detail some such algorithms and further propose certain variants which are useful in specific situations.
4.5.3 Bounds For Graph Propagation

To understand how the graph propagation works, we need to define a few more sets over the dependency graph. \( D(T) = (V, E) \).

**Definition 4.11:** \( MODIFIED \) is the initial set of possibly inconsistent vertices which the graph propagator takes as an input.

If \( \tilde{D}_{k-1}(T_{k-1}) \) undergoes a series of \( i \) changes to give \( \tilde{D}_k(T_k) \) and if for each change, \( U_{i-1}^n, 1 \leq n \leq i \) is a set of possibly inconsistent set of vertices produced as a result of operation \( O_n \). Then \( MODIFIED = \bigcup_{1 \leq m \leq i} U_{i-1}^m \). It is also possible that \( U_{i-1}^1 \cap U_{m}^1 \neq \emptyset ; 1 \leq m, m \leq i \).

This implies that an operation \( O_m \) can further modify some of the vertices which operation \( O_i \) might have modified earlier. Therefore, over a series of operations, a number of vertices are liable to be modified repeatedly. When the graph propagation algorithm is invoked after a series of such edit modifications, the overall work that it does may be far cheaper than if consistency is restored after each individual edit. Further, the time when the graph propagation is called is a prerogative of the client process.

**Definition 4.12:** \( INFLUENCED \): We define the set \( INFLUENCED \) as a set of vertices that are directly or indirectly dependent or \( MODIFIED \). In other words, \( INFLUENCED \) is the transitive closure of the \( MODIFIED \).

\[ INFLUENCED = \{ u | \exists a \in MODIFIED \land (a, u) \text{ is a path in } D(T) \} \]

Finding the solution of \( D(T) \) is complicated. First, we compute the set \( AFFECTED \). This is difficult because \( AFFECTED \subseteq INFLUENCED \), and since the extent of change is the whole of \( INFLUENCED \) set, \( AFFECTED \) can only be computed by actually evaluating all the vertex functions and stopping evaluation at the boundaries of the \( AFFECTED \) when the new evaluated value comes out to be equal to the old evaluated value. The
computation of the vertex value have to follow the topological order of $D(T)$. The first of the vertices $v$ to be computed should not have a predecessor in the set $INFLUENCED$. This is guaranteed by the definition of $INFLUENCED$. We also assume that $D(T)$ does not contain any cycles, as that can complicate the process of finding $v$.

### 4.6 A Naive Graph Evaluation Algorithm

Here we present a naive graph evaluation algorithm whose genesis lies in Knuth's Topological Sort.

![Algorithm 4.1 Evaluate](image)

When evaluate is applied to $D(T)$, we maintain a work list of all vertices that have no predecessors. Next, after removing a vertex from $W$ for computation, we insert all its
successors which are ready for computation into $W$. The process is continued till $W$ is empty. The readiness condition for a vertex is easily determined. If all the predecessors of a vertex have been computed, then the vertex may be considered to be ready for evaluation. This ensures that a topological order of $D(T)$ is followed. The initial members of $W$ can be found in $O(D(T))$ time by traversing $T$. Set insertions and deletions can be done in unit time following standard techniques. The running time of the algorithm is therefore, $O(D(T) + EVAL(V))$.

4.7 Algorithms For Graph Propagation

We now present an incremental version of the above algorithm, where, in a initially consistent dependency graph, some modifications have been affected. The algorithm follows graph propagation.

```
Graph_propagate1 (D(T), W)
    D(T): Dependency graph;
    W: Set of initially modified vertices ;
    f: Vertex function of a vertex $b$ with arguments $c_i, 1 \leq i \leq k$;

begin
    while $W \neq \phi$ do
        begin
            Select and remove a vertex $b$ from $W$;
            $b:=f(c_1, \ldots, c_k)$;
            $W:=W \cup \{\text{All the successors of } b\}$
        end;
end.
```

Algorithm 4. 2 Graph_propagate1

The algorithm, Graph_propagate1 when invoked with an inconsistent dependency graph and a set of modified vertices, restores the consistency. This algorithm has several shortcomings.
First, if the new computed value of a vertex $b$ turns out to be the same as its old value, we need not follow that chain of dependencies. This is because all the successors of $b$ are not bound to change as it continues to retain its old value. Therefore, the successors of $b$ need not be added to the work list $W$. The next algorithm is a modified version which takes care of this shortcoming.

Algorithm 4.3 Graph_propagate2

Graph_propagate1 is the same as Graph_propagate2 except that the successors of a vertex are added to the work list only if the new computed value of the vertex is different from the old value.

4.7.1 Analysis

Initially, the work list $W$ contains all the values in the set MODIFIED. The process of selecting and inserting a vertex into $W$ ensures that all the vertices which are directly or
transitively dependent on MODIFIED, get inserted and removed from \( W \) at some point of time. Therefore, for Graph_propogate1 all the vertices in INFLUENCED are in \( W \) at some point of time and get evaluated. The algorithm Graph_propogate1 therefore, evaluates the graph with an overhead \( O(g(INFLUENCED)) \). On the other hand, Graph_propogate2 has less overhead than Graph_propogate1. A vertex gets inserted into \( W \) only if the value of any of its predecessors has changed. If a vertex retains its old value even after fresh computation, then its successors do not get included in \( W \). Clearly, all the vertices in AFFECTED, find a place in \( W \). The overhaed complexity of Graph_propogate2 is, therefore, \( O(h(AFFECTED)) \). Both \( g \) and \( h \) may be exponential in nature.

In both Graph_propogate1 and Graph_propogate2, we do not necessarily follow the topological order with respect to the dependency graph. This can in turn lead to a situation where a vertex \( v \) is inserted into \( W \) and evaluated a number of times before it finally achieves its consistent value. This happens each time a vertex is evaluated with some inconsistent argument values. If a vertex \( b \) has \( n \) predecessors out of which the first \( m \) are in the set AFFECTED (for Graph_propogate2), and INFLUENCED (for Graph_propogate1)

\[
b := f(C_1, \ldots, C_m, C_{m+1}, \ldots, C_n)
\]

Then, \( b \) can be evaluated at best \( m \) times in each evaluation phase. Therefore, the same vertex may be inserted repeatedly and evaluated leading to a sub-optimal performance. The efficiency can also drastically vary, depending upon the order in which vertices are picked up for evaluation from the worklist \( W \).

4.8 Topological Evaluation

A graph propagation algorithm therefore, re-establishes the consistencies in a dependency graph. The input to such algorithm is a dependency graph and a list of possibly inconsistent vertices. The algorithm terminates with a consistently attributed dependency
Incremental Attribute Evaluation

graph. Numerous propagation algorithms that have been proposed for incremental systems cannot work for arbitrary dependency graph. The basic presumption followed by all these algorithms is that all attribute values must be up to date at all times or whenever the client process requires them to be so. However, this criteria is slightly relaxed in the case of lazy evaluators. A number of observable facts may be enunciated here.

1. If a node is temporarily assigned an incorrect value, then these spurious changes are apt to propagate, resulting in wasted computation.

2. If a recomputed value at a node turns out to be the same as its old value, there is no need to compute further along that chain as no change need be propagated.

3. There might be certain attributes, whose values may not be directly useful. Their computation may be delayed till the need arises. Lazy evaluators follow this strategy.

4. For non-strict rules, all the parameters need not be computed, if the client process does not require the result.

Next we present an algorithm which respects the topological order of $D(T)$.

```
Topo_evaluate(D(T), W)
D(T) :Dependency graph;
W: Vertices with an indegree zero;
begin
    G:=D(T);
    while W≠∅ do
        begin
            Select and remove a vertex $b$ from W;
            $b:=f(c_1, ..., c_k)$;
            for each $a$ that is a successor of $b$ in G do
                begin
                    Remove edge(b,a) from G;
                    If indegree$_G$(a) = 0 then insert $a$ into W;
                end;
        end
end.
```

Algorithm 4.4 Topo_evaluate
Algorithm *Topo_evaluate* strictly follows a topological order of $D(T)$. This algorithm is a non-incremental version and initially starts with all the vertices with an indegree zero in $D(T)$. All the vertices in $D(T)$ get inserted into $W$ at some point of time.

Because of the condition that a vertex can only be inserted if its indegree is reduced to zero, a vertex gets inserted into $W$ only once during the life-time of algorithm *Topo_evaluate*. This further implies that a vertex is computed only once and hence our first criteria listed below is satisfied. A vertex cannot therefore, be assigned an incorrect value, as it is assigned a value only after all its predecessors have been computed. This is a direct outcome of Knuths topological sort.

The problem is to devise an algorithm which is an incremental version of algorithm *Topo_evaluate* and follows the listed criteria.

1. Reevaluating only those vertices which need to be evaluated.
2. Reevaluating a vertex at the most once.
3. Organising the record-keeping, so that any changes in the sub-state lead to minimal changes in the number of vertices.

Two major targets, therefore, to be achieved are:

1. The optimal re-evaluation set
2. The optimal overhead complexity.

The optimal re-evaluation set is clearly the set *AFFECTED*, as it is a set of vertices which actually change value after graph propagation restores the consistency. The algorithm would have optimal overheads and its time complexity is $O(AFFECTED)$. 
4.9 The Data Driven Algorithms

The algorithms which do not use specific priority ordering on the dependency graphs may be broadly classified into two types: data driven, which starts at the point (s) points of change and propagate outwards and, demand driven, which starts at the point where the data are ultimately used and works towards the point of change. These may be highly recursive in nature. The objectives here being to minimize the amount of computation needed to maintain the dependency graph in the face of graph modifications. A naive data driven algorithm is presented here. The algorithm follow topological ordering, but neither the evaluation set nor the overhead complexity is optimum.

Procedure Evaluate1;
W: work set of vertices;
a: vertex;

begin
1. Set all the edges transitively dependent on directly_modified as marked;
2. W := ∅; Pick a vertex from directly_modified with no incoming edges marked and let W := {a}
3. while W ≠ ∅ do
   begin
      Pick an element a from W with no incoming edges marked;
      Evaluate a;
      Unmark all the outgoing edges;
      W := W U {all successor nodes of a}
   end;
end.

Algorithm 4.5 Evaluate1

The overhead complexity as well the reevaluation set are O(INFLUENCED). We present an algorithm which gives an optimal reevaluation set, but the overhead complexity still remain sub-optimal. The difference in the reevaluation set comes from the fact that we evaluate a vertex only if at least one of its predecessors have changed values. For this purpose, an additional tag is attached to each vertex to indicate if it has changed values or not.
Procedure Evaluate 2;
W: work set of vertices;
a: vertex;
begin
1. Set all the edges transitively dependent on directly_modified as marked;
2. W := φ; Pick a vertex a from directly_modified with no incoming edges marked and let W := {a};
3. while W ≠ φ do
   begin
      Pick an element a from W with no incoming edges marked;
      If at least one predecessor of a is changed then
      begin
         evaluate a;
         if old(a) = new(a) then mark a as changed;
      end;
      Unmark all the outgoing edges of a;
      W := W ∪ (all successor node of a);
   end;
end.

Algorithm 4.6 Evaluate2

The algorithm Evaluate2 computes all the attribute values, while following the topological order with respect to the dependency graph. An attribute value is picked up for computation only once and an attribute is actually evaluated if all its predecessors are consistent. We present an enhanced version of this algorithm in the following chapter and analyse its complexity in detail.

The evaluation set for algorithm Evaluate2 is \(O(AFFECTED)\) while the Overhead complexity is \(O(INFLUENCED)\).