Graph Modification and Evaluation

6.1 Introduction

In the previous chapter, algorithms for performing various operations on the simplified copy trees are presented. We associate simplified copy trees with all the vertices in the graph, which are the heads of copy tree. The performance of the incremental attribute evaluation algorithm can be drastically improved, if all the intermediate copy vertices, used to communicate values from the heads of the copy chains to their tails, are bypassed during the evaluation phase. In this chapter, we propose algorithms to incrementally maintain the simplified copy trees embedded in the dependency graph, in the face of any dependency graph modifications.

An incremental attribute evaluation algorithm is proposed. This algorithm bypasses the copy vertices in time of the order of the number of vertices at the tails of all the copy rule chains with a common head.

6.2 Modifying the graph

The modifications on the given dependency graph can be of the following types:

- adding a vertex.
- delete a vertex.
Graph Modification and Evaluation

- adding an edge.
- delete an edge.
- adding an component.
- deleting a component.

Whenever any of the above operations takes place, we update all the copy trees which fall within the modified region of the graph. Since the dependency graph represents the objects of a particular system, any incremental modification has to be faithfully replicated in the dependency graph. The system in question is assumed to contain a substantial number of identity dependencies. These dependencies are represented as copy rule trees in the dependency graph. Further, in order to reduce the computation time for such copy vertices, we maintain simplified copy trees as self adjusting binary search trees. The advantage being that all the intermediate copy vertices can be bypassed, if structure trees are maintained in addition to the usual copy trees. The maintenance of the simplified copy trees will require some additional overheads, but the time complexity of the evaluation algorithm is substantially reduced. In fact, if the number of copy vertices in the system is large, the incremental evaluation algorithm shows substantial improvement in its time complexity. We present in the following sections algorithms needed to maintain the structure tree invariant in the face of any modification in the dependency graph. Some additional functions are needed for these routines. \(\text{indegree}(a)\) returns the indegree of the vertex \(a\), \(\text{indegree}(a) = |\text{predecessor}(a)|\). \(\text{semf}(a)\) returns the type of the semantic function associated with \(a\). The semantic function of \(a\) can be either of the type \(\text{copy}\), if it is a copy rule or it can be of the type \(\text{non_copy}\), if it is not a copy rule. The function \(\text{type}(a)\) returns the type of the vertex \(a\). It can have any of the three values: \(\text{cpy}\), \(\text{cph}\), and \(\text{ncp}\). The function \(\text{maketype}(a, x)\) changes the type of the vertex \(a\) to type \(x\). This may need changes in the structure of the vertices. The values of all the fields of the new vertex are copied from the corresponding field in the old vertex. The old vertex may contain some additional fields, which may not be needed and are redundant. The functions \(\text{add_pred}(v, S)\) and \(\text{add_succ}(v, S)\) add and delete respectively to a vertex \(v\), all the vertices in the set \(S\). Since we use self-adjusting trees to represent the copy trees in the graph, we can use the
operations *link*, *cut*, *mark*, and *unmark* to maintain the copy trees. As is apparent from the previous section, each tree modification takes an amortised \( \log(n) \) time, where \( n \) is the total number of tree vertices in the graph.

6.3 Adding an edge between any two vertices

When an edge is added between any two vertices, we have to check the vertex types, and any further action will depend on these vertex types. Since there are three possible types of vertices in the graph, we have a total of nine cases. In this section, we present routines to take care of all the nine cases.

6.3.1 Case 1:

Adding an edge from a *ncp* vertex \( v_1 \) to a *ncp* vertex \( v_2 \).

When an edge is added between two *ncp* vertices, two possibilities arise. The first, that if the edge added represents a copy rule (this is possible if the original indegree of the vertex at the directed end of the edge is 0) \( v_1 \) is made the root of a copy tree containing \( v_1 \) and \( v_2 \). The second possibility is that the edge is not a copy edge and therefore, a simple graph edge is inserted.

If an edge added between two *ncp* vertices is a copy rule and \( \text{indegree}(v_2) = 0 \) then, we construct a copy and a structure tree for these vertices.

6.3.2 Case 2:

Adding an edge from a *ncp* vertex \( v_1 \) to a *cpy* vertex \( v_2 \).

Addition of an edge directed towards *cpy* vertex results in the destruction of the copy vertex. If the vertex \( v_2 \) has any children in the copy tree, then it becomes a vertex of type
cph. Otherwise, it is transformed into a normal graph vertex of the type ncp. The link between the parent of v₂ and v₂ is severed by invoking the function cut. Therefore, if v₂ is not a leaf vertex of its copy tree, it becomes the root of the subtree rooted at v₂.

```
procedure add_edge(vertex v₁,v₂);
if type(v₁)=ncp and type(v₂)=ncp then
begin
  if indegree(v₂)=0 and semf(v₂)=copy then
    begin
      maketree(v₁); maketype(v₁,cph);
      maketree(v₂); maketype(v₂,cpy);
      link(v₁,v₂);
    end
  else
    begin
      add_pred(v₂,{v₁, parent(v₂)});
      add_succ(parent(v₂),{v₂});
      cut(parent(v₂),v₂);
      if v₂ has no children then maketype(v₂,ncp);
    end;
.......
/*the rest of the eight cases*/
end.
```

Algorithm 6.1 Inserting an edge in case 1

```
if type(v₁)=ncp and type(v₂)=cpy then
begin
  add_pred(v₂,{v₁, parent(v₂)});
  add_succ(parent(v₂),{v₂});
  cut(parent(v₂),v₂);
  if v₂ has no children then maketype(v₂,ncp);
end;
```

Algorithm 6.2 Adding an edge in case 2
6.3.3 Case 3:

Adding an edge from a ncp vertex \( v_1 \) to a cph vertex \( v_2 \).

This leads to simple graph edge addition if the edge added is not a copy rule edge. In case this is not so, \( v_1 \) becomes the new copy head and a link is established between \( v_1 \) and \( v_2 \). \( v_2 \) becomes an interior vertex of the copy tree rooted at \( v_1 \).

```
if type(v_1)=ncp and type(v_2)=cpy then
begin
  if indegree(v_2)=0 and semf(v_2)=copy then
    begin
      maketree(v_1);
      maketype(v_1,cph);
      maketype(v_2,cpy);
      link(v_1,v_2);
    end
  else
    begin
      add_pred(v_2,{v_1});
      add_succ(v_1,{v_2});
    end;
end;
```

Algorithm 6.3 Adding an edge in case 3

6.3.4 Case 4:

Adding an edge from a cpy vertex \( v_1 \) to an ncp vertex \( v_2 \).

There are two possibilities. First, if the edge added represents a copy rule, \( v_2 \) becomes a part of the copy tree containing \( v_1 \). We establish a link between \( v_1 \) and \( v_2 \) by making \( v_2 \) as a child of \( v_1 \). The second possibility being that if the edge is not a copy rule edge, we mark the vertex \( v_1 \) and make \( v_2 \) a normal graph successor of \( v_1 \).
if type(v₁)=cpy and type(v₂)=ncp then
begin
  If indegree(v₂)=0 and semf(v₂)=copy then
  begin
    maketree(v₂);
    maketype(v₂, cpy);
    link(v₁, v₂);
  end
  else
  begin
    mark(v₁);
    add_pred(v₂, {v₁});
    add_succ(parent(v₂), {v₂});
  end;
end;

Algorithm 6.4 Adding an edge in case 4

6.3.5 Case 5:
Adding an edge from a cpy vertex v₁ to a cpy vertex v₂.
The vertices v₁ and v₂ may belong to the same or different copy trees. Since an edge is
being added to v₂, originally a copy vertex, it can either become the copy head of the
subtree rooted at v₂ or, if it was a leaf, it becomes a normal ncp vertex. The link between
the parent of v₂ and v₂ is cut and v₂ becomes a normal graph successor for its former
parent as well as for v₁.

if type(v₁)=cpy and type(v₂)=cpy then
begin
  maketype(v₂, cpy);
  mark(v₁);
  add_pred(v₂, {v₁});
  add_succ(parent(v₂), {v₂});
  cut(parent(v₂), v₂);
  if v₂ has no children then maketype(v₂, ncp);
end;

Algorithm 6.5 Adding an edge in case 5
6.3.6 Case 6:
Adding an edge from a cpy vertex $v_1$ to a cph vertex $v_2$.
If the edge added is a copy rule edge then the copy tree rooted at $v_2$ becomes a part of the copy tree containing $v_1$. We link $v_1$ and $v_2$ by making $v_1$ the parent of $v_2$ followed by unmarking $v_2$ if $v_2$ does not have any normal graph successors. When the edge added does not represent a copy rule, a simple graph edge is inserted between $v_1$ and $v_2$. $v_1$ is marked as it has to communicate values to $v_2$.

\[
\begin{align*}
\text{if } & \text{type}(v_1) = \text{cpy} \text{ and type}(v_2) = \text{cph} \text{ then} \\
& \text{begin} \\
& \quad \text{if } \text{indegree}(v_2) = 0 \text{ and semf}(v_2) = \text{copy} \text{ then} \\
& \quad \quad \text{begin} \\
& \quad \quad \quad \text{maketype}(v_2, \text{cpy}) ; \\
& \quad \quad \quad \text{link}(v_1, v_2) ; \\
& \quad \quad \quad \text{if } v_2 \text{ has no successors then unmark}(v_2) ; \\
& \quad \quad \text{end} \\
& \quad \text{else} \\
& \quad \quad \text{begin} \\
& \quad \quad \quad \text{mark}(v_1) ; \\
& \quad \quad \quad \text{add_pred}(v_2, \{v_1\}) ; \\
& \quad \quad \quad \text{add_succ}(v_1, \{v_2\}) ; \\
& \quad \quad \text{end} ; \\
& \text{end} ; \\
\text{Algorithm 6.6 Adding an edge in case 6}
\end{align*}
\]

6.3.7 Case 7:
Adding an edge from a cph vertex $v_1$ to an ncp vertex $v_2$.
If the edge added is a copy rule edge, $v_2$ is included in the copy tree rooted at $v_1$ and the type of $v_2$ is changed to cpy. Otherwise, a normal graph edge is inserted between $v_1$ and $v_2$. 

Algorithm 6.7 Adding an edge in case 7

6.3.8 Case 8:

Adding an edge from a cph vertex \( v_1 \) to a cpy vertex \( v_2 \).

The vertex \( v_2 \) ceases to be of the type cpy and may either become a cph vertex, if it is the interior vertex of its copy tree or an ncp vertex if it is a leaf of its copy tree. If \( v_2 \) is an interior vertex, it becomes the copy head of its subtree in the old copy tree. The tree edge between \( v_2 \) and its parent is cut.

Algorithm 6.8 Adding an edge in case 8
6.3.9 Case 9:
Adding an edge from a \textit{cph} vertex \(v_1\) to a \textit{cph} vertex \(v_2\).
If the edge added is a copy edge, we merge the two copy trees rooted at \(v_1\) and \(v_2\) by making \(v_1\) the parent of \(v_2\). The vertex \(v_2\) is unmarked if it does not have any graph successor. If a non copy edge is added, normal graph edge addition is affected.


define for case 9
\begin{align*}
\text{if } \text{type}(v_1) = \text{cph} \text{ and } \text{type}(v_2) = \text{cph} \text{ then} \\
\text{begin} \\
\text{if } \text{indegree}(v_2) = 0 \text{ and } \text{semf}(v_2) = \text{copy} \text{ then} \\
\text{begin} \\
\text{make type}(v_2, \text{cpy}); \\
\text{link}(v_1, v_2); \\
\text{if } v_2 \text{ has no successor then } \text{unmark}(v_2); \\
\text{end} \\
\text{else} \\
\text{begin} \\
\text{add pred}(v_2, \{v_1\}); \\
\text{add succ}(v_1, \{v_2\}); \\
\text{end}; \\
\text{end};
\end{align*}

Algorithm 6.9 Adding an edge in case 9

6.4 Deleting an edge between any two vertices
Deleting an edge between any two vertices is relatively simple. Here again, we have nine cases. The function \textit{delete edge}(a,b) transforms into \textit{del succ}(a,\{b\}); \textit{del pred}(b,\{a\}). The only cases of interest are 4, 5, 6 and 8. The rest are trivial.

In case 4, an edge existing between a \textit{cpy} vertex \(v_1\) and a \textit{ncp} vertex \(v_2\) is to be deleted. A \textit{cpy} vertex can have an outgoing edge to an \textit{ncp} vertex only if it is a \textit{marked} vertex (the flag \textit{is marked} is true and it is a structured tree leaf). If \(v_1\) has no other outgoing edge to an \textit{ncp} vertex, it is \textit{unmarked} followed by deleting the edge.
Case 5 occurs when an edge between two vertices in a copy tree is to be deleted. If \( v_1 \) is the parent of \( v_2 \) and the edge connecting \( v_1 \) to \( v_2 \) is to be removed, \( v_2 \) becomes the new copy head of the subtree rooted at \( v_2 \) after cutting the edge between \( v_1 \) and \( v_2 \). If \( v_2 \) happens to be a leaf of the copy tree, i.e., it has no children, then \( v_2 \) is transformed into a normal graph vertex of type \textit{ncp}. If the vertex \( v_1 \) is not \textit{marked}, it stays \textit{unmarked} after the edge deletion.

Case 6 is simple as an edge connecting a copy vertex \( v_1 \) to a copy head vertex \( v_2 \) is to be removed. \( v_2 \) should be the copy head of a different copy tree as cycles are not allowed in the dependency graph. If \( v_2 \) is the only graph successor of \( v_1 \), \( v_1 \) is unmarked followed by the deletion of the edge.

Case 8 is interesting because an edge connecting a copy head vertex \( v_1 \) to a copy vertex \( v_2 \) is to be deleted. This implies that \( v_2 \), which is a child of \( v_1 \) the copy head of its tree, becomes the new copy head of the subtree rooted at \( v_2 \). Again, if \( v_2 \) has no children, it is changed into a normal graph vertex of the type \textit{ncp}. At the same time if \( v_1 \) has no other children left in its copy tree it is also changed to an \textit{ncp} vertex.

All the nine cases, and their associated codes, are enumerated in Table 6.1.

6.5 Changing the Vertex Function

When the function associated with the vertex is modified, it may result in addition or deletion of a number of edges. The function associated with the vertex \( b \) is of the form
\[
b.val := f(c_1, ..., c_k)
\]
where \( c_i (1 \leq i \leq k) \). Each \( c_i \) refers to a vertex in the dependency graph, and there is a directed edge from each \( c_i \) to \( b \). If the vertex function is changed so that
\[
b.val := f(c_{i_1}, ..., c_{i_m})
\]
then the edges from the vertices in the set \( \{c_1, ..., c_k\} \cap \{c_{i_1}, ..., c_{i_m}\} \) are
<table>
<thead>
<tr>
<th>v₁</th>
<th>v₂</th>
<th>code</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ncp</td>
<td>ncp</td>
<td>delete_edge(v₁,v₂);</td>
<td>normal graph edge deletion.</td>
</tr>
<tr>
<td>ncp</td>
<td>cpy</td>
<td>delete_edge(v₁,v₂);</td>
<td>cannot occur, cpy vertices have edges</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>coming from either cpy vertices or cph</td>
</tr>
<tr>
<td></td>
<td>cph</td>
<td>delete_edge(v₁,v₂);</td>
<td>normal graph edge deletion.</td>
</tr>
<tr>
<td>cpy</td>
<td>ncp</td>
<td>del_succ(v₁, {v₂});</td>
<td>v₁ may or may not be marked. If</td>
</tr>
<tr>
<td></td>
<td></td>
<td>del_pred(v₂, {v₁});</td>
<td>marked, then it is a structure tree</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>vertex as well. unmark will also</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>compress the structure tree path.</td>
</tr>
<tr>
<td>cpy</td>
<td>cpy</td>
<td>mark(v₂); cut(v₁,v₂);</td>
<td>can only occur inside a copy tree. This</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>operation gives rise to two copy trees.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A new copy tree rooted at v₂ is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>formed, if v₂ has any children, otherwise</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>v₂ is converted into an ncp vertex.</td>
</tr>
<tr>
<td>cpy</td>
<td>cph</td>
<td>del_succ(v₁, {v₂});</td>
<td>since v₂ is of the type cph, it belongs to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>del_pred(v₂, {v₁});</td>
<td>a different copy tree. If v₁ has no</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>normal graph successors, it is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>unmarked.</td>
</tr>
<tr>
<td>cph</td>
<td>ncp</td>
<td>delete_edge(v₁,v₂);</td>
<td>normal graph edge deletion.</td>
</tr>
<tr>
<td>cph</td>
<td>cpy</td>
<td>cut(v₁,v₂);</td>
<td>if v₂ is the only child of v₁, then v₂</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>becomes the new copyhead, otherwise</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cut the tree forming two new trees.</td>
</tr>
<tr>
<td>cph</td>
<td>cph</td>
<td>delete_edge(v₁,v₂);</td>
<td>normal graph edge deletion.</td>
</tr>
</tbody>
</table>

Table 6.1 Algorithm for deleting an edge

not touched. The vertices in the set \{c₁, ... ,cₖ\} - \{cᵢ,... ,cₘ\} are deleted, and vertices in the set \{cᵢ,... ,cₘ\} - \{c₁, ... ,cₖ\} ∩ \{c₁,... ,cₘ\} are added. Therefore, any modification of the vertex function can be affected using the procedures add_edge and delete_edge. In the special case when a vertex has only one incoming edge, we can streamline the modification by using the procedures change_ncp_to_cpy and change_cpy_to_ncp. When the vertex function is changed from a non copy function to the copy function, we use the routine
change\textunderscore ncp\textunderscore to\textunderscore cpy to possibly include the vertex into a copy tree. The procedure change\textunderscore cpy\textunderscore to\textunderscore ncp is invoked when the vertex function is changed from a copy rule to a non copy rule.

6.5.1 Procedure change\textunderscore ncp\textunderscore to\textunderscore cpy

If the vertex function of a vertex $v$ of type ncp is changed to copy, we need to check whether the predecessor of $v$ is a cpy or a cph vertex. If it is so, then $v$ needs to be integrated into the copy tree of its predecessors. The code for changing a non copy vertex to copy vertex is given in algorithm 6.10.

```
if indegree(v)=1 and semf(v)=copy and type(v)=ncp then
begin
    if type(predecessor(v))=cpy or type(predecessor(v))=cph then
        begin
            x:=predecessor(v);
            maketree(v);
            maketype(v,cpy);
            link(x,v);
        end;
    else
        begin
            x:=predecessor(v);
            maketree(x);
            maketype(x,cph);
            maketree(v);
            maketype(v,cpy);
            link(x,v);
        end;
end;
```

Algorithm 6.10 change\textunderscore ncp\textunderscore to\textunderscore cpy

6.5.2 Procedure change\textunderscore cpy\textunderscore to\textunderscore ncp

If a copy vertex belonging to a copy tree is changed to a non copy vertex, it becomes the copy head of the subtree rooted at the vertex. If the vertex in question has no children in the copy tree, it is changed into a normal graph vertex of the type ncp. The code for this is given in algorithm 6.11.
Graph Modification and Evaluation

Algorithm 6.11 change_cpy_to_ncp

6.6 Other Modifications

6.6.1 Adding a Vertex

Adding a vertex to a graph does not result in any change in the rest of the graph, and therefore, no action is necessary till we insert the corresponding edges.

6.6.2 Deleting a vertex

When a vertex is to be deleted, all the incoming and outgoing edges are deleted individually by invoking the delete_edge function. Once all the edges are deleted the vertex is isolated and can be removed without any further action.

6.6.3 Adding a Component

Two strategies may be followed for adding a component. If the number of vertices is small, say less than a predetermined number $c$, vertices of the component are added to the graph followed by the addition of each individual edge connected to this component by invoking add_edge.

The second technique is followed if the number of vertices is large. A Depth First Search of the component is performed and copy trees are built if they exist. This is followed by adding all the edges at the boundaries of the component till the component is fully integrated with the main graph.
6.6.4 Deleting a Component

Deleting a component is quite straightforward. The edges at the boundary of the component are removed one by one by invoking the function `delete_edge`. This is continued till the component is completely isolated from the main graph.

6.7 Incremental Graph Evaluation

In this section, we present an algorithm which restores the given dependency graph to its consistent state. This algorithm is a modified version of a similar algorithm presented in the previous chapter. In addition to the following topological order for evaluation of attributes, it also uses the structure trees to bypass all the copy rule chains presented in the dependency graph. The maintenance of chains of copy rules and their associated structure trees burden the system with some overheads. But if the number of copy vertices in the graph is large, this maintenance overhead is small in comparison to the gains obtained in the attribute evaluation phase. If \( u \) is the total number of copy vertices in the graph, then it takes an amortised \( \log(u) \) time to perform a single modification on the graph which involves addition or deletion of an edge.

We use a flag `modified` to indicate that the attribute value of a vertex has changed. In addition, we attach an attribute `colour` to each edge in the graph. An edge is coloured `red` if it is potentially inconsistent, otherwise it is coloured `white`. The algorithm `Restore_Consistency` for restoring an inconsistent dependency graph to its consistent state is given in algorithm 6.12. This algorithm uses two procedures: `traverse_and_colour` and `colour_edge` in algorithm 6.13 and 6.14 to mark all the `INFLUENCED` edges of the graph `red`. At the same time, the `modified` flag for each vertex at the directed end of the `red` edges is set to `false`.

The procedure `traverse_and_colour` is invoked when a vertex of the type `cph` is encountered. It uses a function `successor_{ct}(u)` to return all the copy tree successors of the
copy tree rooted at $u$. These are the vertices for which the \textit{is\_marked} flag has been set to true by the graph modification routines.

\textbf{Lemma 6.1:} The set $\text{successor}_{\text{CT}}(u)$ for a simplified copy tree $C_p(T(u), N)$ can be computed in time $O(|N|)$.

Proof: Since we maintain structure trees, in addition to the normal copy trees, in the form of simplified copy trees $C_p$. Since the number of vertices in $ST(T(u), N)$ is $O(|N|)$, where $N$ is the set of marked vertices of the copy tree. Therefore, a simple traversal of the structure tree $ST$ gives all the marked vertices in the copy tree. Since the vertices in $ST$ are of the order of $N$, the set $\text{successor}_{\text{CT}}$ can be computed in $O(|N|)$ time.

The procedure $\text{traverse\_and\_colour}$, therefore, colours red all the direct and transitive successor edges of the marked vertices belonging to a particular copy tree. The procedure $\text{colour\_edge}$ is invoked to actually colour the edges red. The procedure $\text{colour\_edge}$ actually colours an edge red if it is not already so. In this manner all direct and transitive edges are traversed and coloured red if not already so. If a copy head vertex is encountered, $\text{traverse\_and\_colour}$ is invoked to take care of the copy tree successors.

The set $\text{Directly\_Modified}$ contains all the vertices which are modified as a result of some transaction performed by the client process. A work set $W$ is used to temporarily include all the vertices which may be potentially inconsistent. A vertex is ready for evaluation if all its incoming edges are coloured white. The value of a vertex is actually computed if at least one of its predecessors (graph and copy tree) has changed values. This is indicated by the \textit{modified} flag of its predecessors. If the \textit{modified} flag of a predecessor is \textit{true}, then the predecessor has indeed changed values. If the attribute value of a vertex is \textit{modified} then, all its graph and copy tree successors, if any, are included in the work set $W$ and the vertex marked \textit{modified}.
Procedure Restore Consistency(set of vertices Directly_Modified);
set of vertices W;
vertex a,b,x,y;
begin
S1: for each a ∈ Directly_Modified do
    begin
        a.modified:=false;
        if type(a)=cph then traverse_and_colour(a);
        for each b ∈ successor(a) do
            begin
                if type(b)=cph then traverse_and_colour(b);
                colour_edge(a,b),
            end;
    end;
S2: W:=Φ;
    for each a ∈ Directly_Modified do
        W:=W ∪ {a};
S3: while W ≠ Φ do
    begin
        S31: Pick a node a from W with no incoming edges coloured red;
        S41: if at least one predecessor of a is marked modified then
            begin
                old_val:=a.val;
                a.val:=eval(a);
                if old_val ≠ a.val then a.modified:=true;
            end;
        S51: if type(a)=cph then
            begin
                for each x ∈ successor(a) do
                    begin
                        if a.modified then
                            begin
                                x.modified:=true;
                                x.val:=a.val;
                                W:=W ∪ {x};
                            end;
                    end;
                for each y ∈ successor(x) do
                    if a.modified then
                        begin
                            (x,y).color := white;
                        end;
            end;
        S61: for each b ∈ successor(a) do
            begin
                (a,b).color :=white;
                if a.modified then W:=W ∪ {all successors of a};
            end;
Algorithm 6.12 Restore Consistency
procedure traverse_and_colour(vertex a);
vertex x,y;
begin
if all the incoming edges of a are coloured white then
begin
for each x \in \text{successor}_{c}(a) do
begin
  x.modified:=false;
end;
end;
for each y \in \text{successor}(x) do
begin
  if type(y)=cph then traverse_and_colour(b);
  colour_edge(x,y);
end;
end;

Algorithm 6.13 traverse_and_colour

procedure colour_edge(vertex a,b);
vertex c;
begin
if edge(a,b).colour \neq \text{red} then
begin
  edge(a,b).colour:=\text{red};
b.modified:=false;
  for each \: c \: \in \text{successor}(b) do
  begin
    if type(c)=cph then traverse_and_colour(c);
    end;
end;
end;

Algorithm 6.14 colour_edge

Further, all the direct and copy tree successor edges of the vertex are coloured white. Since the algorithm may be invoked after every single transaction, a number of vertices might be marked modified and edges coloured red from previous invocations of the algorithm. These markings do not affect the current evaluation process, as it carries out its own set of fresh markings before picking up any vertex for evaluation. Vertices which lie
outside the red region of the graph are assumed to be consistent. If there are any other red regions in the graph which are not connected through red edges to the currently marked red region, all the vertices in those regions are assumed to be consistent.

Any vertex, once it is picked up for evaluation, is removed from the work set \( W \). Further, a vertex is not evaluated unless all its predecessors have attained their correct consistent values. Therefore, the algorithm follows a topological order with respect to the given dependency graph, to evaluate the solution of the graph. The evaluation presented here allows recomputation of attributes after unconstrained dependency graph modifications.

### 6.8 Correctness

We present a simple proof of correctness for the algorithm *Restore Consistency*. The input to the algorithm is *Directly Modified*, the set of nodes which have been modified as a result of edit consequences. Any edge addition or deletion is reflected in this set through the inclusion of the node at the directed end of the edge. Node additions or deletions or changes in the vertex functions are included by inserting the concerned nodes into the *Directly Modified* set. We need to make some assumptions before proving the correctness of the algorithm. First, all vertex functions are evaluated correctly with the output solely dependent on the parameters of the function. This implies that one set of input parameters for a function will always result in the same output under all conditions. The second assumption is that Side effects as a result of computation are not allowed. However, some benign side effect may be allowed which might be useful in other parts of the system. These should not be detected by the attribute evaluation system and should not influence it in any manner. A series of edits (addition, deletion, etc., of vertices and edges) is termed a transaction. The algorithm *Restore Consistency* is invoked immediately after a transaction. Since the proof is for an incremental evaluation system, we also assume that
before the first transaction, the dependency graph is consistent. In other words, all the attribute values satisfy their defining vertex functions.

The correctness of Restore\_Consistency can be proved by the following assertion:

After the execution of Restore\_Consistency, following any transaction, all attributes in \( D(T) \) have correct consistent values.

The proof of the above statement follows from a stronger assertion:

After the execution of Restore\_Consistency following any transaction
1. all the attributes in the set AFFECTED are consistent
2. all the attributes are consistent and marked modified or not modified.

We prove this assertion inductively over a series of transactions. The base case is that the assertion holds true before the first transaction. All the attributes are consistent and no attribute is marked modified before the first transaction, and hence the assertion.

For the inductive step, we assume that the assertion holds true before some transaction. We need to prove that it also holds true after execution of Restore\_Consistency, following the transaction.

We assume that the dependency graph is consistent before the \( k^{th} \) transaction, when the state of the dependency graph is represented by \( \tilde{D}_{k-1}^0(T_{k-1}) \). During the \( k^{th} \) transaction, an arbitrary set of graph modifications take place, transforming the dependency graph into its possibly inconsistent state \( \tilde{D}_{k-1}^i(T_{k-1}) \). On application of Restore\_Consistency it is transformed into its consistent state \( \tilde{D}_k(T_k) \). All the graph modifications during the \( k^{th} \) transaction are reflected in the set Directly\_Modified. For simplicity we ignore the subscripts, assuming that all the sets pertain to the \( k^{th} \) transaction.
We define $INFLUENCED_c$ as a subset of $INFLUENCED$, such that it excludes all the unmarked copy vertices in $INFLUENCED$.

$$INFLUENCED_c = INFLUENCED - \{ \text{all the unmarked copy vertices in } INFLUENCED \}$$

The unmarked copy vertices in $INFLUENCED$ =

$$C_p(T(u_1), N_1) \cup \ldots \cup C_p(T(u_n), N_n) - \{ u_1, \ldots, u_n \} \cup N_1 \cup \ldots \cup N_n$$

Also, $AFFECTED_c = AFFECTED - \{ \text{all the unmarked copy vertices in } AFFECTED \}$

We state the following lemmas to prove the assertion.

**Lemma 2:** After execution of $S_1$, the red region of the dependency graph indicates the $INFLUENCED_c$ area. All the vertices in this area are marked not modified.

This is obvious as $S_1$ traverses all the direct and transitive successors of the modified area of the graph.

**Lemma 3:** No Vertex in $W$ is evaluated more than once.

**Proof:** A vertex $a$ with no incoming edge coloured red is picked up for evaluation only in $S_{31}$. $S_{41}$ and $S_{51}$ insert the copy tree successors and graph successors of $a$ into $W$. Since the dependency graph is assumed to be acyclic, there cannot exist a path from $a$ to itself. Therefore, $a$ cannot be inserted into $W$ more than once. This further implies that $a$ is picked up for evaluation only once.

**Lemma 4:** The algorithm Restore Consistency terminates when $W$ becomes empty.

No proof need be given as it is quite obvious from the algorithm.

**Lemma 5:** A Vertex in $W$ is evaluated only after all its predecessors have correct consistent values.

**Proof:** To prove this Lemma, we define a set $Dependent(a)$ as the set of vertices on which $a$ depends directly or transitively. $Dependent(a) = \{ \exists x : (x, a) \text{ is a path in } D(T) \}$

It may also be called the reverse transitive closure of $a$. We define another set $Dependent_c(a)$ such that $Dependent_c(a) \subseteq Dependent(a)$. 

$771-6695$
Graph Modification and Evaluation

\[ \text{Dependent}_c(a) = \{ x \mid x \in \text{Dependent}(a) \cap \text{INFLUENCED} \} \]

In order to take care of copy vertices which are bypassed, we define yet another set \( \text{Dependent}_c'(a) \) which is defined as:

\[ \text{Dependent}_c'(a) = \text{Dependent}_c(a) - \{ \text{all unmarked copy vertices in Dependent}(a) \} \]

\( \text{Dependent}_c'(a) \) is the set of vertices which need to be evaluated and consistent before \( a \) can be taken up for evaluation. The vertices not in \( \text{INFLUENCED} \) need not be evaluated as they are already consistent. We need to prove that all the vertices in \( \text{Dependent}_c'(a) \) have to be evaluated before \( a \) is picked up for evaluation. In \( S_3 \), a vertex is picked up for evaluation only when its all incoming edges are coloured \text{white}. It is known that after evaluation of a vertex, all its outgoing edges are coloured \text{white}. We prove the lemma by negation. Assume a vertex \( x \in \text{Dependent}_c'(a) \) is not consistent. This implies that either it has not been picked up for evaluation or at least one of its predecessors is inconsistent. It must have been picked up for evaluation as all its outgoing edges are \text{white}. Therefore, at least one of its predecessors, say \( y \), must be inconsistent. We apply the same argument to prove that \( y \) has at least one inconsistent predecessor and so on. This implies that there is a path from a vertex in \( \text{Directly Modified} \) to \( a \) such that all the vertices on this path are inconsistent. Therefore, there exists an inconsistent vertex in \( \text{Directly Modified} \) where the computation was initiated. This is a fallacy because the computation is always initiated at a consistent vertex in \( \text{Directly Modified} \). Therefore, a vertex is evaluated only when all its predecessors have correct consistent values.

**Lemma 6:** After a vertex in \( W \) is picked up and evaluated, it assumes its correct consistent value, if not already so.

**Proof:** By Lemma 5, a vertex in \( W \) is evaluated only when all its predecessors have consistent values. If all the predecessors of \( a \) in the dependency graph have consistent values, then the vertex \( a \) also achieves its correct consistent value after evaluation.
Lemma 7: A vertex $a$ which is inserted with $W$ is eventually picked up for computation.

Proof: The proof here is simple. If a vertex $a$ is inserted in $W$, one of its direct graph or copy tree predecessors must have been evaluated. The insertion of $a$ into $W$ must have been effected by sections $S_{51}$ or $S_{61}$ or both. If all the predecessors of $a$ have been evaluated, then all their outgoing edges must be colored white. Suppose $a$ is not picked for evaluation implying that at least one predecessor of $a$ has not been evaluated and its outgoing edge is red. Applying this logic, we have a chain of vertices going back to the Directly Modified set leading to a fallacy. This implies that $a$ is ready for evaluation as all its incoming edges must be colored white, and eventually be picked up for evaluation. –

Lemma 8: In section $S3$ only vertices in $AFFECTED_c'$ are inserted into $W$.

Proof: To prove this we need to prove the following Lemmas.

Lemma 9: Only vertices which are direct and transitive successors of vertices in Directly Modified are inserted into $W$.

Proof: This is clearly so because section $S_2$ inserts all vertices in Directly Modified into $W$, whereas section $S_3$ inserts all the direct and transitive successors of vertices in Directly Modified –.

Lemma 10: Successors of vertices which do not change values are not inserted into $W$.

Proof: Sections $S_{51}$ and $S_{61}$ inserts vertices into $W$. $S_{51}$ inserts all the copy tree successors and $S_{61}$ inserts all the graph successors of a modified vertex into $W$. If a vertex is not modified, its successors cannot be inserted into $W$ and hence the Lemma –.

By definition, the set of vertices which change values is $AFFECTED_c$. If there exists a vertex $x \in INFLUENCED_c$, By Lemma 9, it is possible that $x$ may get inserted into $W$. Further, if there is an edge from $a \in AFFECTED_c$ to $x$, then $x$ does not change values and hence $x$ is not modified. Since $x$ is not modified by Lemma 10, its successors are not inserted into $W$. Therefore, the only vertices which are inserted into $W$ are $AFFECTED_c \cup \{a | \exists x \in AFFECTED_c \land (a, x) \text{ is an edge in the graph} \}$.
This is the set \( \text{AFFECTED}_c' \). Hence the proof.

By Lemma 2, after the execution of \( S_1 \), all the vertices in the set \( \text{INFLUENCED}_c \) are marked \textit{not modified}. These vertices are potentially inconsistent. In addition, at least one of the incoming edges to each of these vertices is coloured \textit{red}. Since the set \( \text{INFLUENCED}_c \) includes all the direct and transitive successors of vertices in \( \text{Directly Modified} \), therefore, the \textit{red} region of the graph represents \( \text{INFLUENCED}_c \). There might be other vertices for which some incoming edges might be coloured red from some previous application of algorithm \textit{Restore Consistency}. Such vertices will not affect the outcome of the algorithm, as there exists no direct path to them from vertices in \( \text{Directly Modified} \). All the edges on the paths from vertices in \( \text{Directly Modified} \) to vertices in \( \text{INFLUENCED}_c \) are coloured \textit{red} with the exception of copy nodes.

By Lemma 8, only vertices in \( \text{AFFECTED}_c' \cup \text{Directly Modified} \) are inserted into set \( W \).

By Lemma 7, a vertex which is inserted into \( W \) is eventually picked up for evaluation. By Lemma 6, if a vertex is picked up for evaluation, it assumes its correct consistent value, if not already so. By Lemma 9, the algorithm \textit{Restore Consistency} eventually terminates when \( W \) becomes empty. From definition, \( \text{AFFECTED}_c' \subseteq \text{AFFECTED} \) since all the vertices in \( \text{AFFECTED}_c' \) assume their correct consistent values, therefore, all the vertices in \( \text{AFFECTED}_c \) also assume their consistent values. From the definition, only vertices in \( \text{AFFECTED} \) are inconsistent before the \( k \text{th} \) application of \textit{Restore Consistency}. The remaining vertices are consistent. Therefore, after the execution of \textit{Restore Consistency}, all the attributes in \( \bar{D}_k(T_k) \) are consistent. And hence, the result.

\textbf{Theorem 1: The time complexity of section S1 is } O(|\text{INFLUENCED}_c|)

\textbf{Proof:} The proof of the theorem is quite simple. By Lemma 2, all the vertices which are directly or transitively dependent on \( \text{Directly Modified} \) are marked \textit{not modified} by \( S_1 \). The exceptions are the copy vertices for which the flag \textit{is marked} is \textit{false}. This by
definition is the set $\text{INFLUENCED}_c$. Since a vertex is marked not modified, at the most once, the time taken by $S1$ is $O(|\text{INFLUENCED}_c|)$.

**Theorem 2:** The time complexity of phases $S2$ and $S3$ is $O(\text{AFFECTED}_c)$.

**Proof:** By Lemma 8, only vertices in $\text{AFFECTED}_c'$ are inserted into $W$. By Lemma 3, each vertex in $\text{AFFECTED}_c'$ is evaluated at the most once. We assume here that the dependency graph has vertices with bounded indegree and outdegree, as is normal, with dependency graphs based on $AG$s. Therefore, the number of edges in the dependency graph is proportional to the number of vertices. Hence, the total amount of work done in phases $S2$ and $S3$ is clearly bounded by $O(\text{AFFECTED}_c')$, which is of the order of $O(\text{AFFECTED}_c)$.

### 6.9 Comparisons with other Algorithms

In this section, we review some of the existing techniques and compare them with proposed algorithms.

The various techniques for incremental evaluation of attributes can be broadly categorised into those based on $AG$'s and those based on any other generalised scheme. The $AG$ based techniques are further classified into static and dynamic evaluation schemes. The static schemes presume a lot of information about the various dependencies that arise out of the defining equations of the attributes in an $AG$. The evaluation is generally faster in this scheme, but a thorough analysis of the $AG$ is required in the evaluated construction time. It is almost impossible to construct evaluators for a generalised $AG$, as this is an $NP$ complete problem. Therefore, most of the attempts at constructing static evaluators have been for sub-classes of $AG$s where, it is sometimes guaranteed that the evaluators can be constructed in a polynomial time. The attribute evaluation through this class of evaluators is generally faster, but with the major constraint of being applicable to sub-classes of $AG$. 
The dynamic evaluators on the other hand, have no such restrictions and most of them can work for any class of AGs. The only restriction under which perhaps some of them have to work is that AG must be non-cyclic. But the usual algorithms for incremental dynamic evaluation of attributes may be extended to support cyclic AGs [Jon90]. Here, we separate the strongly connected components in the underlying dependency graph, evaluate them separately through fixed point techniques finally restoring consistency in the usual manner.

The genesis of most of the techniques for dynamic incremental evaluation lies in Knuth's Topological Sort. Since all the vertex functions are to be evaluated in addition to the book keeping work, the work done is proportional to $O(EVAL(V) + |E| + |V|)$, $E$ and $V$ denoting edges and vertices in $D(T)$. The earliest results were reported by Reps [Rep83], who prefers to call his technique Optimal Time Change Propagation. His algorithm achieves sub-optimal reevaluation set, that of $O(INFLUENCED)$ and his book keeping time is proportional to $O(IAFFECTEDI)$. However, his system imposes constraints like single edit cursor movements, and large data structures called Superior and Subordinate characteristic graphs need to be maintained at all points of time. Jaleli [Jal85], introduced the concept of lazy attribute evaluation, where the computation starts from the demand vertices and evaluates only those vertices which are actually needed. The book keeping in this case is $O(|V| + |E|)$.

Reps [Rep86], extends his algorithm for unrestricted movements between tree modification points. The time complexity of this modified algorithm is also dependent on the size of the tree and is given by $O(|AFFECTED|) \cdot \sqrt{n} + EVAL(|AFFECTED|)$ where $n$ is the size of the attributed tree. Hoover [Hoo87], proposes to maintain the vertices incrementally in a true topological order. The vertices are maintained in a generalised linked list with $O(INFLUENCED) \cdot \log(INFLUENCED) + EVAL(AFFECTED)$ time complexity.
Another algorithm proposed by Hoover [Hoo87], called Approximate Topological Ordering, uses a heuristic to approximate the topological order of the possibly inconsistent vertices. The complexity is \( O(g(AFFECTED) + EVAL(AFFECTED)) \) where \( g \) is an exponential function. Its worst case performance is of exponential order but for small incremental changes, the performance might be quite reasonable.

The algorithm proposed by Alpern [Alp87], uses a priority driven method where each attribute is assigned a priority number. The time spent in record keeping by the complete algorithm is \( O(u + (INFLUENCED) \log(INFLUENCED)) \) where \( u \) is the number of nodes that either receive updated priority numbers or depend upon such nodes. Hudson's [Hud91], evaluator identifies goals and computes only those attributes that are actually needed, thus minimising evaluation time. It does not maintain any extra data structures. An overall amortised complexity of \( O(INFLUENCED) + EVAL(AFFECTED) \) is achieved over a series of transactions, otherwise, for a single transaction the algorithm takes \( O(INFLUENCED) + |mayneed| \) overheads, where \( mayneed \) is the reverse dependency set of the demanded attributes for the \( i \)th transaction. The Algorithm Evaluate2, even though being data driven, achieves the optimal reevaluation set, and the overhead complexity for any transaction is \( O(INFLUENCED) \). Thus the total time complexity of the algorithm is \( O(INFLUENCED) + EVAL(AFFECTED) \). It is better than Reps algorithm as it can be used for unconstrained dependency graph modification.

The time complexity of algorithm Restore_Consistency is \( O(INFLUENCED_c) + EVAL(AFFECTED_c) \). It can be used after unconstrained dependency graph modifications and does not use any extra data structures. Further, any graph modification can be carried out in time \( O(\log n) \) where \( n \) is the total number of copy vertices in the dependency graph. The algorithm is better than other existing algorithms for its space complexity, as it does not use any extra data structures except structure trees which are
superimposed on copy trees. In addition it allows unconstrained dependency graph modifications to be carried out, and can be used for generalised AG based systems.