# ) CHAPTER NINE

# )Further Algorithm Derivations

## ( **THE SCHORR-WAITE GRAPH MARKING ALGORITHM**

In this Chapter we give further examples of the derivation of efficient algorithms by a process of transformation starting with a specification. The first problem we will tackle is one of marking all the nodes of a connected graph; a recursive procedure to do this is developed which is simple but inefficient since it uses a variable amount of extra storage. This algorithm will be successively transformed into an iterative form which only uses a fixed amount of extra storage. The algorithm developed is known as the Schorr-Waite graph marking algorithm which was first presented (in the form of a flowchart) in [Schorr & Waite 67].

This algorithm seems to have acquired the status of a standard testbed for program verification techniques applied to complex data structures. A proof of a modified version of the algorithm using the method of invariants was given in [Gries 79], an informal development from a standard recursive algorithm is described in [Griffiths 79]. [Morris 82] proves the algorithm using the axiomatic methods of [Hoare 69]. [Topor 79] presents a proof using the method of "intermittent assertions" which are assertions of the form: "if at some time during the execution assertion **A** holds at this point then at some later time assertion **B** will hold at this point". Intermittent assertions are described in [Manna & Waldinger 78] where they are used to reason about some iterative algorithms for computing recursive functions. [Yelowitz & Duncan 77], [de Roever 78] and [Kowalski 79] also give proofs for this algorithm.

Most of these proofs start with a statement of the algorithm and hence they can give few indications as to how such an algorithm could be devised. The methods that rely on invariants give a long list of complex invariants, again with little indication of how these invariants could be developed.

Our development of the algorithm starts with the specification of a graph-marking algorithm which is transformed into a recursive procedure and thence to the iterative form. We take a different route to that of [Griffiths 79] since we follow our usual practice of doing as much simplification as possible with the recursive form of the algorithm before removing the recursion. In particular we introduce the central idea of the algorithm while it is still in the form of a recursive procedure. This gives a much clearer development than Griffith's introduction of the idea after he has removed the recursion.

## The Problem

We are given a graph stored as a set of nodes, each node has a value, a mark value and two pointers to the left and right subgraphs of the node. So the node at **x** (where **x** is an integer, the address of the node) may be represented:

ie: **x:**

where **m**[**x**] gives the value of the mark of node **x** (which as we will see later must be able to store at least four distinct values), **l**[**x**] and **r**[**x**] are the pointers to the left and right subtrees respectively with initial values **left(x)** and **right(x)** respectively. For the purposes of this algorithm we ignore the **value** fields of the nodes.

We require all nodes reachable from a root node (given in **root**) to be marked, ie the program is to terminate with **m**[**x**]=**1** for all reachable nodes **x** where initially all nodes have **m**[**x**]=**0**. To make our initial specification and subsequent development more concise we will assume that all pointers which are not used contain **0** (so that for instance all the leaves of the tree have both pointers set to **0**) and that we have a special node at position **0** with **m[0]=1, l[0]**=**0** and **r[0]**=**0** (the zero node).

We define the set of nodes "reachable" from a given node and a given array **m** as:

$$\mathbf{reachable(x,m)} =_{DF} \begin{cases} \varnothing & \text{if } \mathbf{m[x]=1} \\ \{\mathbf{x}\} \cup \mathbf{reachable(left(x),m')} \cup \mathbf{reachable(right(x),m')} & \text{if } \mathbf{m[x]=0} \end{cases}$$

where **m′** is **m** with node **x** marked, ie $\mathbf{m'[y]} = \mathbf{1}$ if $\mathbf{y=x}$
$= \mathbf{m[y]}$ otherwise.

So our specification is:

$$\mathbf{MARK(x)} \equiv \ \mathbf{m{:}=m'.}\big(\forall\mathbf{y.}\big((\mathbf{y}\in\mathbf{reachable(x,m)}\Rightarrow\mathbf{m'[y]=1}\big) \land \ \big(\mathbf{y}\notin\mathbf{reachable(x,m)}\Rightarrow\mathbf{m'[y]=m[y]}\big)\big)\big)$$

An obvious recursive procedure to implement this is to mark **x** (if it is not already marked), then mark all nodes reachable from **left(x)** and then mark all nodes reachable from **right(x)**. However, when we mark this second set of nodes we have changed the array **m** such that the nodes in the first set are already marked. So to show that this will work we need to use a little elementary graph theory to prove the following:

**Theorem:**

$\mathbf{reachable(x,m)} = \varnothing$ if $\mathbf{m[x]=1}$
$\quad\{\mathbf{x}\}\cup\ \mathbf{reachable(left(x),m')} \cup \mathbf{reachable(right(x),m'')}$
$\qquad$ if $\mathbf{m[x]=0}$

where $\mathbf{m'}$ is $\mathbf{m}$ with node $\mathbf{x}$ marked, ie $\mathbf{m'[y]} = \mathbf{1}$ if $\mathbf{y=x}$
$\qquad = \mathbf{m[y]}$ otherwise.
and $\mathbf{m''}$ is $\mathbf{m'}$ with all nodes reachable from $\mathbf{left(x)}$ marked, ie:
$\quad\mathbf{m''[y]} = \mathbf{1}$ if $\mathbf{y{\in}reachable(left(x),m')}$
$\quad = \mathbf{m'[y]}$ otherwise.

**Proof:** We use the concept of a path to a reachable node. A path is a sequence of nodes $\mathbf{p[1..\ell(p)]}$ where each node is connected to the previous one and no node is marked. ie:
$\mathbf{PATH(p,m)} =_{DF}$
$\quad \forall \mathbf{i{\in}1..\ell(p){-}1.}\big(\mathbf{p[i{+}1]{=}left(p[i])} \vee \mathbf{p[i{+}1]{=}right(p[i])}\big) \wedge \forall\mathbf{i{\in}1..ln(p).}\big(\mathbf{m[p[i]]{=}0}\big)$
We define the set of paths from a given node as:
$\mathbf{paths(x,m)} = \{\mathbf{p|PATH(p,m)} \wedge \mathbf{p[1]{=}x}\}$
note that if the graph has cycles then this set will be infinite.
As usual, the length of path $\mathbf{p}$ is $\ell(\mathbf{p})$, so the last element of a path is $\mathbf{p}[\ell(\mathbf{p})]$.

With this definition we can prove that for each node reachable from $\mathbf{x}$ there is a path to that node, and every node on a path from $\mathbf{x}$ is reachable from $\mathbf{x}$. ie:
$\mathbf{y{\in}reachable(x,m)} \iff \exists\mathbf{p{\in}paths(x,m).p[\ell(p)]{=}y.}$
The proof is by induction on path lengths and induction on the size of the set of reachable elements.
To prove the theorem we will prove:
$\quad\mathbf{y}{\in}\ \mathbf{reachable(x,m)} - \big(\{\mathbf{x}\}\cup\ \mathbf{reachable(left(x),m')}\big) \iff \mathbf{y}{\in}\ \mathbf{reachable(x,m'')}$
" $\Longrightarrow$ ": Follows from: $\mathbf{reachable(right(x),m'')} \subseteq \mathbf{reachable(right(x),m')}$
"$\Rightarrow$": If $\mathbf{y{\neq}x}$ and $\mathbf{y{\in}reachable(x,m)}$ and $\mathbf{y{\notin}reachable(left(x),m')}$ then there is a path to $\mathbf{y}$ in $\mathbf{paths(x,m)}$ which does not cross $\mathbf{x}$ or any element of $\mathbf{reachable(left(x),m')}$ except for the first element. The second element of this path must be $\mathbf{right(x)}$ since it cannot be $\mathbf{left(x)}$. Hence this path with the first element removed is a path to $\mathbf{y}$ in $\mathbf{paths(right(x),m'')}$, hence $\mathbf{y}$ is reachable from $\mathbf{right(x)}$ in $\mathbf{m''}$.

So **MARK(x)** is equivalent to:

```
mark(x) where
proc mark(x) ≡ if m[x]=0 then m[x]:=1; MARK(left(x)); MARK(right(x)) fi.
```

So applying the theorem on recursive implementation of specifications we get **MARK(x)** $\approx$

```
    mark(x) where
    proc mark(x) ≡ if m[x]=0 then m[x]:=1; mark(left(x)); mark(right(x)) fi.
```

where we have used the fact that the total number of nodes is finite and each recursive call either terminates immediately or marks at least one unmarked node.

## The Transformations

The difficulty the Schorr-Waite algorithm presents to any formal analysis of it is that it uses the same data structure for three different purposes: to store the original graph structure, to record the path from the current node to the root, and to record the current "state of play" at each node. The program is required to mark the graph without changing its structure yet works by modifying that structure as it executes, therefore any proof of correctness must also demonstrate that all the pointers are restored on termination of the program. We use our general method for deriving algorithms involving abstract data types to deal with this problem, the same method can also be used with other algorithms which use one set of storage for two purposes. The steps in the development are:

(1) Derive a version of the program which uses two data structures (for example by transforming a specification into a program). The variables representing these data structures are the abstract variables.

2 (2) Add the variables which will represent the actual data store as ghost variables (so at the moment these are assigned but not accessed). These are the concrete variables.

(3) Add assertions which show the relationship between the abstract and concrete variables.

(4) Replace references to the abstract variables by references to the concrete variables using the assertions to show that the required values are available.

(5) The abstract variables are now ghost variables (they are assigned but never referenced) so they can be removed from the program leaving the concrete variables.

To slightly simplify the development we add tests to ensure that **mark** is only called when **m**[**x**]=**0**. This gives:

```
    proc mark(x) ≡
        m[x]:=1;
        if m[left(x)]=0 then mark(left(x)) fi;
        if m[right(x)]=0 then mark(right(x)) fi.
```

Our development of the algorithm may be summerised as:

(1) Devise a simple recursive procedure (as above).

(2) Replace the parameter by a global stack (standard technique).

(3) Add **l(x)**, **r(x)** and **q** as new ghost variables (**l(x)** and **r(x)** are the pointers at node **x**, their initial and final values will be **left(x)** and **right(x)** for all **x**. **q** is a variable which will record the node at the top of the stack).

(4) Replace all references to **left(x)**, **right(x)** and the stack by references to **l**[**x**], **r**[**x**] and **q** (this will be possible because the stack will be "embedded" in the pointers).

(5) Remove the recursion using the "postponed obligations" technique.

(6) Implement the stack of "postponed obligations" by extra assignments to **m**[**x**], so that for each **x**, **m**[**x**] records what operations on **x** have been postponed.

Using global stack **S** to remove the parameter gives:

```
Prog ≡  x:=root; S:=⟨⟩; if m[x]=0 then mark fi.
mark ≡  m[x]:=1;
        if m[left(x)]=0 then S←x; x:=left(x); mark; x←S fi;
        if m[right(x)]=0 then S←x; x:=right(x); mark; x←S fi.
```

Note that **mark** preserves **S** and **x** and is only called when **m**[**x**]=**0**.

We will now add **l**[**x**], **r**[**x**] and **q** as ghost variables which have the initial values (for all nodes **x** which are reachable from the root): **l**[**x**]=**left(x)** and **r**[**x**]=**right(x)**.

The assignments will maintain these values for all nodes apart from those on the stack. We will use **q** to store the value on the top of the stack, and implement **S** as an array with an integer **i** such that:

**x←S** becomes **S**[**i**]**:=x; i:=i+1** and
**S←x** becomes **i:=i−1; x:=S**[**i**]

So we maintain the invariant: **q=S**[**i−1**].

The central idea behind the algorithm devised by Schorr and Waite is that when we return from having marked the left subtree of a node we know what the value of **left(x)** is for the current node (since we just came from there) so that while we were marking the left subtree we could use the pointer **l**[**x**] to store something else–for instance the current top of the stack (which is the node we will return to when we have finished marking this one). Similarly while we are marking the right subtree we can store this "previous node" pointer in **r**[**x**]. Since we only call **mark** when **m**[**x**]=**0** and all nodes on the stack have **m**[**x**]=**1** we know that the invariant for **l** and **r** holds for each node we encounter.

The values we want to assign to our "ghost variables" will be as follows:

First visit: **l**[**x**]=**S**[**i−1**], **r**[**x**]=**right(x)**, **q=left(x)**

Second visit: **l[x]=left(x)**, **r[x]=S[i−1]**, **q=right(x)**
Third visit: **l[x]=left(x)**, **r[x]=right(x)**, **q=S[i−1]**.

We also ensure that **q=S[i−1]** before each call of **mark**:

```
Prog ≡  x:=root; q:=0; i:=0; S[−1]:=0;
    if m[x]=0 then {m[x]=0 ∧ q=S[i−1]}; mark fi.
mark ≡  {m[x]=0}; m[x]:=1;
  if m[left(x)]=0 then  l[x]:=S[i−1]; q:=x;
                        S[i]:=x; i:=i+1; x:=left(x);
                        {q=S[i−1] ∧ m[x]=0}; mark;
                        i:=i−1; x:=S[i]; q:=left(x)
                   else l[x]:=S[i−1]; q:=left(x) fi;
  if m[right(x)]=0 then  l[x]:=left(x); r[x]:=S[i−1]; q:=x;
                         S[i]:=x; i:=i+1; x:=right(x);
                         {q=S[i−1] ∧ m[x]=0}; mark;
                         i:=i−1; x:=S[i]; q:=right(x)
                   else  l[x]:=left(x); r[x]:=S[i−1]; q:=right(x) fi;
   r[x]:=right(x); q:=S[i−1].
```

From this we can see that **mark** preserves the values of **q** and **x**, and preserves the values of the pointers of all nodes with **m[x]≠0** initially. These two facts allow us to replace all references to **left(x)**, **right(x)** and **S** by references to **l[x]**, **r[x]** and **q**: for example in the middle **if** statement we have **q=left(x)** so the assignment **l[x]:=left(x)** becomes **l[x]:=q** and so on. This gives:

```
Prog ≡  x:=root; q:=0; i:=0; S[−1]:=0; if m[x]=0 then {m[x]=0 ∧ q=S[i−1]}; mark fi.
mark ≡  {m[x]=0}; m[x]:=1;
  if m[l[x]]=0 then S[i]:=x; i:=i+1;
                    ⟨l[x],q,x⟩:=⟨q,x,l[x]⟩;
                    {q=S[i−1] ∧ m[x]=0}; mark;
                    i:=i−1;
                    ⟨x,q⟩:=⟨q,x⟩
               else ⟨l[x],q⟩:=⟨q,l[x]⟩ fi;
  if m[r[x]]=0 then S[i]:=x; i:=i+1;
                    ⟨r[x],l[x],q,x⟩:=⟨l[x],q,x,r[x]⟩;
                    {q=S[i−1] ∧ m[x]=0}; mark;
                    i:=i−1;
                    ⟨x,q⟩:=⟨q,x⟩
         else ⟨r[x],l[x],q⟩:=⟨l[x],q,r[x]⟩ fi;
  ⟨r[x],q⟩:=⟨q,r[x]⟩.
```

The next step is to remove the recursion using the "postponed obligations" technique of Chapter 6. To get the recursive procedure into the right format we create a new procedure **mark**$_1$ which is equivalent to **mark; i:=i−1; ⟨x,q⟩:=⟨q,x⟩**, we also split **mark** into three sections:

```
Prog ≡ x:=root; q:=0; i:=0; S[−1]:=0; if m[x]=0 then mark_1; ⟨x,q⟩:=⟨q,x⟩; i:=i+1 fi.
mark_1 ≡ m[x]:=1;
    if m[l[x]]=0 then S[i]:=x; i:=i+1;
                 ⟨l[x],q,x⟩:=⟨q,x,l[x]⟩; mark_1
               else ⟨l[x],q⟩:=⟨q,l[x]⟩ fi; mark_2.
mark_2 ≡ if m[r[x]]=0 then S[i]:=x; i:=i+1;
                 ⟨r[x],l[x],q,x⟩:=⟨l[x],q,x,r[x]⟩; mark_1
      else ⟨r[x],l[x],q⟩:=⟨l[x],q,r[x]⟩ fi; mark_3.
mark_3 ≡ ⟨r[x],q⟩:=⟨q,r[x]⟩; ⟨x,q⟩:=⟨q,x⟩; i:=i−1.
```

We use another stack **A** to record which section of **mark** has been postponed, this stack is also implemented as an array with **j** as the index variable:

```
Prog ≡ x:=root; q:=0; i:=0; S[−1]:=0;
    if m[x]=0
     then A[1]:=1; j:=1;
        while j≠0 do
                 m[x]:=1;
          a:=A[j]; j:=j−1;
          if a=1 → if m[l[x]]=0 then S[i]:=x; i:=i+1;
                          ⟨l[x],q,x⟩:=⟨q,x,l[x]⟩;
                          j:=j+1; A[j]:=2; j:=j+1; A[j]:=1;
                                 else ⟨l[x],q⟩:=⟨q,l[x]⟩; j:=j+1; A[j]:=2 fi
          □ a=2 → if m[r[x]]=0 then S[i]:=x; i:=i+1;
                          ⟨r[x],l[x],q,x⟩:=⟨l[x],q,x,r[x]⟩;
                          j:=j+1; A[j]:=3; j:=j+1; A[j]:=1
                    else ⟨r[x],l[x],q⟩:=⟨l[x],q,r[x]⟩; j:=j+1; A[j]:=3 fi
          □ a=3 → ⟨r[x],q⟩:=⟨q,r[x]⟩; ⟨x,q⟩:=⟨q,x⟩; i:=i−1 fi od;
        ⟨x,q⟩:=⟨q,x⟩ fi.
```

The final step is to use the extra states which can be stored in the mark part of each node to record the status of each node (ie what has been postponed for that node). We introduce another array **M** as a ghost variable which stores this inormation by maintaining the invariant: for each **k** in the stack (ie **1⩽k⩽j**) , **M[k]=A[S[k]]**. Note that the values of **i** and **j** follow each other:

```
Prog ≡ x:=root; q:=0; i:=0; S[−1]:=0;
   if m[x]=0
    then A[1]:=1; j:=1; M[x]:=1;
      while j≠0 do
                m[x]:=1;
        a:=A[j]; j:=j−1;
        if a=1 → if m[l[x]]=0 then M[x]:=2; M[l[x]]:=1
                        S[i]:=x; i:=i+1;
                        ⟨l[x],q,x⟩:=⟨q,x,l[x]⟩;
                        j:=j+1; A[j]:=2; j:=j+1; A[j]:=1;
                               else ⟨l[x],q⟩:=⟨q,l[x]⟩; M[x]:=2;
                        j:=j+1; A[j]:=2 fi
        □ a=2 → if m[r[x]]=0 then M[x]:=3; M[r[x]]:=1
                        S[i]:=x; i:=i+1;
                        ⟨r[x],l[x],q,x⟩:=⟨l[x],q,x,r[x]⟩;
                        j:=j+1; A[j]:=3; j:=j+1; A[j]:=1
                  else ⟨r[x],l[x],q⟩:=⟨l[x],q,r[x]⟩; M[x]:=3;
                        j:=j+1; A[j]:=3 fi
        □ a=3 → ⟨r[x],q⟩:=⟨q,r[x]⟩; ⟨x,q⟩:=⟨q,x⟩; i:=i−1 fi od;
      ⟨x,q⟩:=⟨q,x⟩ fi.
```

Now we can replace all references to **A[j]** by references to **M[S[j]]** which (since **i** and **j** follow each other) is the same as **M[S[i]]** which is the same as either **M[x]**, or **M[l[x]]** or **M[r[x]]**. This means we can remove **S** and then remove **i** (since it is only used in assignments to **S**). Note that **j** is only reduced in the third arm, so for **j** to be reduced to zero we must have had **j=1** and **A[j]=3**, ie

**A**[**1**]=**M**[**S**[**1**]]=**M**[**root**]=**3**. So we can replace the test **j**≠**0** by **M**[**root**]≠**3** and remove **j**. If we initialise **M** to **1** then all the assignments which set elements of **M** to **1** can be removed (since they only apply to "new" elements, ie those with **m** zero). We get:

```
Prog ≡  x:=root; q:=0;
      if m[x]=0
        then  while M[root]≠3 do
                      m[x]:=1;
               a:=M[x];
               if a=1  →  if m[l[x]]=0 then M[x]:=2; ⟨l[x],q,x⟩:=⟨q,x,l[x]⟩
                                         else ⟨l[x],q⟩:=⟨q,l[x]⟩; M[x]:=2 fi
                □  a=2  →  if m[r[x]]=0 then M[x]:=3; ⟨r[x],l[x],q,x⟩:=⟨l[x],q,x,r[x]⟩
                      else ⟨r[x],l[x],q⟩:=⟨l[x],q,r[x]⟩ fi
                □  a=3  →  ⟨r[x],q⟩:=⟨q,r[x]⟩; ⟨x,q⟩:=⟨q,x⟩ fi od;
             ⟨x,q⟩:=⟨q,x⟩ fi.
```

The usual algorithm combines **m** and **M** into a single array **m′** with the values:

**m′**[**x**]=**0** if **m**[**x**]=**0** (in this case the value in **M**[**x**] is always **1**)
**m′**[**x**]=**1** if **m**[**x**]=**1** and **M**[**x**]=**1**
**m′**[**x**]=**2** if **m**[**x**]=**1** and **M**[**x**]=**2**
**m′**[**x**]=**3** if **m**[**x**]=**1** and **M**[**x**]=**3**

This gives the following algorithm which sets **m′**[**x**] to **3** for all nodes **x** reachable from **root**:

```
Prog ≡  x:=root; q:=0;
      if m[root]=0
        then  while m′[root]≠3 do
                      m′[x]:=m′[x]+1;
               if m′[x]=1  →  if m′[l[x]]=0 then ⟨l[x],q,x⟩:=⟨q,x,l[x]⟩
                                              else ⟨l[x],q⟩:=⟨q,l[x]⟩ fi
                □  m′[x]=2  →  if m′[r[x]]=0 then ⟨r[x],l[x],q,x⟩:=⟨l[x],q,x,r[x]⟩
                          else ⟨r[x],l[x],q⟩:=⟨l[x],q,r[x]⟩ fi
                □  m′[x]=3  →  ⟨r[x],q⟩:=⟨q,r[x]⟩; ⟨x,q⟩:=⟨q,x⟩ fi od;
             ⟨x,q⟩:=⟨q,x⟩ fi.
```

This is essentially the algorithm devised by Schorr and Waite. We can get a more compact (though slightly less efficient) form of the algorithm by changing the assignments to the "ghost variables" so that the values **left**[**x**], **right**[**x**] and **S**[**i**−**1**] are rotated around the variables **l**[**x**], **r**[**x**] and **q** each time node **x** is visited.

We have:

First visit ($\mathbf{m'[x]=1}$): $\mathbf{l[x]=right[x]}$, $\mathbf{r[x]=S[i-1]}$, $\mathbf{q=left[x]}$
Second visit ($\mathbf{m'[x]=2}$): $\mathbf{l[x]=S[i-1]}$, $\mathbf{r[x]=left[x]}$, $\mathbf{q=right[x]}$
Third visit ($\mathbf{m'[x]=3}$): $\mathbf{l[x]=left(x)}$, $\mathbf{r[x]=right(x)}$, $\mathbf{q=S[i-1]}$.

This leads to the algorithm:

```
Prog ≡ x:=root; q:=0;
  while m′[root]≠3 do
        m′[x]:=m′[x]+1;
    if m[x]=1  →  if m′(l[x])=0 then ⟨l[x],r[x],q,x⟩:=⟨r[x],q,x,l[x]⟩
                                  else ⟨l[x],r[x],q⟩:=⟨r[x],q,l[x]⟩ fi
     □ m[x]=2  →  if m′(r[x])=0 then ⟨l[x],r[x],q,x⟩:=⟨r[x],q,x,l[x]⟩
              else ⟨l[x],r[x],q⟩:=⟨r[x],q,l[x]⟩ fi
     □ m[x]=3  →  ⟨l[x],r[x],q,x⟩:=⟨r[x],q,x,l[x]⟩ fi od.
```

This can now be further simplified by re-arranging the **if** statements to get the final version:

```
Prog ≡ x:=root; q:=0;
  while m[root]≠3 do
    m[x]:=m[x]+1;
    if m[x]=3 ∨ m[l[x]]=0 then ⟨l[x],r[x],q,x⟩:=⟨r[x],q,x,l[x]⟩
            else ⟨l[x],r[x],q⟩:=⟨r[x],q,l[x]⟩ fi od.
```

## An Improved Version

Although the Schorr-Waite algorithm is very efficient in the use of storage it is less efficient than the original recursive procedure in terms of the number of assignments carried out. Also if we knew that the graph structure was similar to a bushy tree (with many nodes but few long paths) then a small fixed stack would be able to deal with the majority of the nodes. For example with a binary tree in which each node had either zero or two subtrees a stack of length 40 could deal with trees containing more than 1,000,000,000,000 nodes. This suggests using a stack to deal with the short paths and using the general "pointer switching" strategy when the stack runs out. Starting with an intermediate version, we add tests so that we only assign values to the new variables **l[x]**, **r[x]** etc. when $\mathbf{i>N}$:

```
Prog ≡ x:=root; q:=0; i:=0; S[−1]:=0;
    if m[x]=0 then {m[x]=0 ∧ q=S[i−1]}; mark fi.
```

```
mark ≡ {m[x]=0}; m[x]:=1;
  if m[left(x)]=0
      then if i>N then l[x]:=S[i−1]; q:=x;
                 S[i]:=x; i:=i+1; x:=left(x); mark;
                 i:=i−1; x:=S[i]; q:=left(x)
             else S[i]:=x; i:=i+1; q:=x; x:=left(x); mark;
                 i:=i−1; x:=S[i]
      else if i>N then l[x]:=S[i−1]; q:=left(x) fi fi;
  if m[right(x)]=0
     then if i>N then l[x]:=left(x); r[x]:=S[i−1]; q:=x;
                S[i]:=x; i:=i+1; x:=right(x); mark;
                i:=i−1; x:=S[i]; q:=right(x)
             else S[i]:=x; i:=i+1; q:=x; x:=left(x); mark;
                i:=i−1; x:=S[i]
     else if i>N then l[x]:=left(x); r[x]:=S[i−1]; q:=right(x) fi fi;
    if i>N then r[x]:=right(x); q:=S[i−1]
        else x:=S[i]; q:=S[i−1] fi.
```

Here we have added assignments to **l[x]**, **r[x]** etc. only if **i>N**.
This can be developed in the same way as the original to get the following iterative version:

```
Prog ≡ x:=root; q:=0; i:=0;
  while m[root]≠3 do
        m′[x]:=m′[x]+1;
    if m[x]=1 ∧ i>N → if m′[l[x]]=0 then ⟨l[x],q,x⟩:=⟨q,x,l[x]⟩; i:=i+1
                      else ⟨l[x],q⟩:=⟨q,l[x]⟩ fi
    □ m[x]=2 ∧ i>N → if m′[r[x]]=0 then ⟨l[x],r[x],q,x⟩:=⟨r[x],q,x,l[x]⟩; i:=i+1
                      else ⟨l[x],r[x],q⟩:=⟨r[x],q,l[x]⟩ fi
    □ m[x]=3 ∧ i>N → ⟨l[x],r[x],q,x⟩:=⟨r[x],q,x,l[x]⟩; i:=i−1
    □ m[x]=1 ∧ i⩽N → if m′[l[x]]=0 then S[i]:=x; i:=i+1; q:=x; x:=l[x] fi
    □ m[x]=2 ∧ i⩽N → if m′[r[x]]=0 then S[i]:=x; i:=i+1; q:=x; x:=r[x] fi
    □ m[x]=3 ∧ i⩽N → x:=S[i]; i:=i−1; q:=S[i−1] fi od.
```

This can be made more efficient with the careful use of selective unrolling. Omitting the details, the result is expressed as a regular action system (we assume **N>0** and **m[root]≠3** initially):

```
Prog ≡ x:=root; q:=0; i:=0; A_0.
A ≡ if m[x]=0 → A_0
    □ m[x]=1 → A_1
    □ m[x]=2 → A_2 fi.
A_0 ≡ if m[l[x]]=0 then m[x]:=1; S[i]:=x; i:=i+1; if i>N then q:=x; x:=l[x]; B
                                                    else x:=l[x]; A fi
          else A_1 fi.
A_1 ≡ if m[r[x]]=0 then m[x]:=2; S[i]:=x; i:=i+1; if i>N then q:=x; x:=r[x]; B
                                                    else x:=r[x]; A fi
          else A_2 fi.
A_2 ≡ m[x]:=3; if i=0 then Z fi;
    x:=S[i]; i:=i−1; A.
B ≡ if m[x]=0 → B_0
    □ m[x]=1 → B_1
    □ m[x]=2 → B_2 fi.
B_0 ≡ if m[l[x]]=0 then m[x]:=1; ⟨l[x],q,x⟩:=⟨q,x,l[x]⟩; i:=i+1; B
          else ⟨l[x],q⟩:=⟨q,l[x]⟩; B_1 fi.
B_1 ≡ if m[r[x]]=0 then m[x]:=2; ⟨l[x],r[x],q,x⟩:=⟨r[x],q,x,l[x]⟩; i:=i+1; B
          else ⟨l[x],r[x],q⟩:=⟨r[x],q,l[x]⟩; B_2 fi.
B_2 ≡ m[x]:=3; ⟨l[x],r[x],q,x⟩:=⟨r[x],q,x,l[x]⟩; i:=i−1;
    if i⩽N then A else B fi.
```

This version minimises the number of tests and assignments.

## LARGEST STEEP SEGMENT

I am indebted to Richard Bird for suggesting the next problem as a test of my methods of program development:

**Problem:** Find the length of the largest steep segment of the array **a[1..n]** of positive integers. **a[i..j]**, **1⩽i⩽j⩽n**, is a steep segment iff $\sum_{i\leqslant k<l}$**a[k]** ⩽ **a[l]** for each **l** st **i⩽l⩽j**
So **i=j** is always a steep segment and any subsequence of a steep segment is also a steep segment.

**Solution:** Suppose we have a caterpillar crawling along the array (the precise mathematical definition of a caterpillar will be given later), and this caterpillar will only stand on a steep segment:

**a[1]** **a[n]**

**steep segment**

**i** **j**

The problem therefore is: how long can the caterpillar get?

<u>**Claim:**</u> If the caterpillar moves from left to right and only moves its tail when it can't move its head then its head will reach **a[n]** and it will have stood on the longest steep segment.
**Proof:** The head reaches the end since if the tail reaches the head then the head can move (if the head can't move on its own then the head and tail move together). Once the head has moved onto the last space of the longest steep segment the tail must be on the first space since if the tail was past the first space it must have moved while the head was still inside the segment, yet the tail only moves when the head cannot. The tail can't be behind the first space since then the caterpillar would be standing on a longer segment than the longest steep segment and the caterpillar only stands on steep segments. Hence once the head reaches **a[n]** the caterpillar must at some point have stood on the longest steep segment.
We represent a caterpillar by the pair **C**= $\langle$**i,j**$\rangle$ of positive numbers with **t(C)=i** and **h(C)=j**.
The caterpillar's movements must preserve the invariant:

$$\mathbf{i \leqslant j} \ \wedge\ \mathbf{a[i..j]} \text{ is a steep segment}$$

After each head move we update **result** which contains the length of the longest steep segment found so far.

**headmove(C)**= $\langle$**t(C),h(C)+1**$\rangle$
**tailmove(C)** = $\langle$**t(C)+1,h(C)**$\rangle$ if **t(C)<h(C)**
= $\langle$**t(C)+1,h(C)+1**$\rangle$ otherwise

The program is:

```
S_1 ≡ var C:=⟨1,1⟩,result:=1; (set up the invariant)
  while h(C)≠n do {h(C)<n};
    if SS(headmove(C)) then C:=headmove(C);
            if length(C)>result then result:=length(C) fi
          else C:=tailmove(C) fi od.
```

where **SS(C)** ≡ **C** is a steep segment ie

$$\forall \mathbf{l}.\mathbf{t(c)} \leqslant \mathbf{l} \leqslant \mathbf{h(c)} \Rightarrow \textstyle\sum_{t(C) \leqslant k < l} \mathbf{a[k]} \leqslant \mathbf{a[l]}$$

Given $\mathbf{SS(C)}$ and $\mathbf{h(C)<n}$ we have

$$\mathbf{SS(headmove(C))} \iff \textstyle\sum_{t(C)\leqslant k\leqslant h(C)}\mathbf{a[k]} \leqslant \mathbf{a[h(C)+1]}.$$

Using the technique of "finite differencing" we add an extra variable **sum** and maintain the relation: $\mathbf{sum}= \sum_{t(C)\leqslant k\leqslant h(C)}\mathbf{a[k]}$ by updating **sum** after each caterpillar move. Then:

$$\mathbf{SS(headmove(C))} \iff \mathbf{sum\leqslant a[h(C)+1]}$$

The program is now:

```
S₂ ≡ var C:=⟨1,1⟩,result:=1,sum:=a[1]; {h(C)⩽n};
  while h(C)≠n do {h(C)<n};
    if sum⩽a[h(C)+1]
     then C:=headmove(C); sum:=sum+a[h(C)];
       if length(C)>result then result:=length(C) fi
     else if h(C)=t(C)
        then C:=⟨t(C)+1,h(C)+1⟩; sum:=a[h(C)]
        else   C:=⟨t(C)+1,h(C)⟩; sum:=sum−a[t(C)−1] fi fi od.
```

Finally we replace **C** by the two variables **head** and **tail**. $\mathbf{length(C)=head-tail+1}$:

```
S₃ ≡ var head:=1,tail:=1,result:=1,sum:=a[1];
  while head≠n do
    if sum⩽a[head+1]
     then head:=head+1; sum:=sum+a[head];
       if head−tail+1>result then result:=head−tail+1 fi
     else if head=tail then head:=head+1; tail:=tail+1; sum:=a[head]
           else tail:=tail+1; sum:=sum−a[tail-1] fi fi od.
```

We can make a small improvement in efficiency by realising that once the tail has gone past $\mathbf{n-result}$ we will not be able to improve on **result** so we can change the test of the <u>**while**</u> loop from $\mathbf{head\neq n}$ to $\mathbf{tail<n-result}$.

## BACKTRACKING

In this section we consider the following form of recursive procedure:

```
    proc F ≡
      S₁; while B do S₂; F; S₃ od; S₄.
```

where $\mathbf{S_1,\ldots,S_4}$ do not call **F**.

This form of procedure frequently occurs with various searching and other recursive routines which involve backtracking. Examples are: finding all permutations of a set (where the loop picks each element in turn as the first element of the permutation and the recursive call finds all permutations of the remaining elements), listing Gray codes, finding a path (or all paths) through a maze, finding a "Knight's Tour" of a chessboard, the "Eight Queens" problem, etc.

At first sight it may appear that the following form is more general:

```
proc $F_1$ ≡
  $S_1$; while B do
      if Q then $S_a$; F; $S_b$
        else $S_c$ fi;
      $S_3$ od; $S_4$.
```

However, this can be transformed into the first form as follows:
Split the loop into two to get:

```
proc $F_1$ ≡
  $S_1$;
  while B $\wedge$ $\neg$Q do
    if Q then $S_a$; F; $S_b$ else $S_c$ fi;
    $S_3$ od;
  {$\neg$B $\vee$ Q};
  while B do
    if Q then $S_a$; F; $S_b$ else $S_c$ fi;
    $S_3$ od;
  $S_4$.
```

Simplify the body of the first loop and apply entire loop unfolding to the second loop after pushing $\mathbf{S}_3$ into the **<u>if</u>** statement:

```
proc $F_1$ ≡
  $S_1$;
  while B $\wedge$ $\neg$Q do $S_c$; $S_3$ od;
  {$\neg$B $\vee$ Q};
  while B do
    if Q then $S_a$; F; $S_b$; $S_3$; while B $\wedge$ $\neg$Q do $S_c$; $S_3$ od; {$\neg$B $\vee$ Q}
      else $S_c$; $S_3$; while B $\wedge$ $\neg$Q do $S_c$; $S_3$ od; {$\neg$B $\vee$ Q} fi od;
  $S_4$.
```

Now we can insert **{Q}** at the beginning of the second loop body which can therefore be simplified to:

```
proc $F_1$ ≡
  $S_1$;
  while B ∧ ¬Q do $S_c$; $S_3$ od;
  while B do
    $S_a$; F; $S_b$; $S_3$; while B ∧ ¬Q do $S_c$; $S_3$ od od;
  $S_4$.
```

Which is of the same form as the first version.

A general backtracking procedure to find all the solutions to a problem can be derived as follows: Suppose that the required solution is a set and all subsets of a solution are "partial solutions" and suppose we have a set of "possible additions" which may be added to a partial solution to get a bigger one. For example in finding all the paths through a maze a partial solution is a path which satisfies all the conditions of a solution (does not cross itself etc.) but does not yet extend all the way through the maze. The possible additions to such a path are all the squares adjacent to it which are within the maze. In the case of the "Eight Queens" problem the partial solutions are boards for which only the first **n** columns have a queen. Possible additions are the legal places where a queen can be added to the **n+1**th column.

A procedure to find all solutions may take the form:

```
proc Find ≡
  "Find the set of possible additions to the partial solution";
  while "More possible additions to consider" do
    if "Adding next possible addition still gives a partial solution"
     then "Add it to the partial solution";
        Find; (find all extensions to the new partial solution)
        "Remove it from the partial solution" fi;
    "Remove the next possible addition from the set" od;
```

Note that the step which finds the set of possible additions will also test if the current partial solution is in fact a full solution.

To find just one solution the only modification needed is to add an extra condition to the loop which ensures that it terminates as soon as a solution has been found. This may be is achieved by adding a global boolean variable "**success"** which is set to **false** initially and set **true** when a solution is found. The **while** loop is changed to: **while** ¬**success** ∧ **...**

To make this informal algorithm more precise we use $\Psi$ for the partial solution and suppose:
**Complete($\Psi$)** is true iff $\Psi$ is a complete solution,
**Cond($\Psi$)** is the condition that any partial solution must satisfy,
**Add($\Psi$)** is the set of possible additions to partial solution $\Psi$.
We use **Sols** for the set of solutions.

Under the condition that any subset of a solution is a partial solution, the following program sets **Sols** to the set of all solutions:

```
Prog ≡  Sols:=∅; Ψ:=∅; if Cond(∅) then Find fi.
Find ≡  var Add, a;
   if Complete(Ψ)
    then Sols:=Sols∪{Ψ}; Add:=∅
    else Add:=Add(Ψ) fi;
   if Add≠ ∅ then "Pick an element of Add and assign to a" fi;
   while Add≠ ∅ do
     if Cond(Ψ ∪ {a})
      then Ψ:=Ψ ∪ {a}; Find; Ψ:=Ψ − {a} fi;
     Add:=Add−{a};
     if Add≠ ∅ then "Pick an element of Add and assign to a" fi od.
```

Note that **Cond($\Psi$)** is true on each call of **Find**: this fact is likely to be useful when we are testing for possible additions to $\Psi$ since we may be able to define a more efficient function **Cond**$^*$**($\Psi$,a)** which tests if **Cond($\Psi \cup \{$a$\}$)** is true given that **Cond($\Psi$)** is true. Note that $\Psi$ is a global variable but **Add** and **a** are local to **Find** so they may need stacks to implement them. In many cases however, their values can be "reconstructed" after the inner call to **Find**. For example if $\Psi$ is represented by a stack then the statement **$\Psi$:=$\Psi \cup \{$a$\}$** becomes $\Psi \leftarrow$**a** and since **Find** preserves $\Psi$ we can restore the values of $\Psi$ and **a** by executing **a**$\leftarrow \Psi$. Also if the set **Add** has an order relation on it then we can refine the statement "Pick an element of **Add** and assign to **a**" to pick the smallest (or largest) element each time and then we can use the restored **a** to restore **Add** since: **Add** = $\{$**x**$\in$**Add($\Psi$)**$|$**a**$\preccurlyeq$**x**$\}$ where $\preccurlyeq$ is the order relation. Finally, the fact that $\Psi = \varnothing$ holds only for the outermost call will aid us in transforming the recursive procedure to an iterative one.

## Transformation to Iterative Form

To transform the procedure:

```
    proc F ≡
      S_1; while B do S_2; F; S_3 od; S_4.
```

to iterative form we add an extra action **L** to implement the **while** loop:

```
Prog ≡ F; Z.
F ≡ S_1; L.
L ≡ while B do S_2; F; S_3 od; S_4.
```

**L** can be transformed to a tail-recursive action:

```
L ≡ if B then S_2; F; S_3; L
      else S_4 fi.
```

To apply the standard recursion removal theorem we add an "activation counter" **c** (which we may be able to represent by some counting operation already performed within **F**: for instance the size of the partial solution $\Psi$):

```
Prog ≡ c:=0; F; {c=0}; Z.
F ≡ S_1; L.
L ≡ if B then S_2; c:=c+1; F; {c>0}; c:=c−1; S_3; L
      else S_4; /F fi.
/F ≡ skip.
```

Now we can use **c** to regularise **/F**:

```
Prog ≡ c:=0; F.
F ≡ S_1; L.
L ≡ if B then S_2; c:=c+1; F
      else S_4; /F fi.
/F ≡ if c=0 then Z
       else c:=c−1; S_3; L fi.
```

Copy **F** and **/F** into **L** and remove the recursion:

```
L ≡ do if B then S_2; c:=c+1; S_1
         else S_4; if c=0 then exit fi;
          c:=c−1; S_3 fi od; Z.
```

Copy **L** into **F** and **F** into **Prog** to get the iterative version:

```
Prog ≡ c:=0;
   do if B then S_2; c:=c+1; S_1
        else S_4; if c=0 then exit fi;
          c:=c−1; S_3 fi od.
```

## Example of backtracking: Relation Preserving Mappings

The following problem was suggested to me by Dr. H.Priestley in connection with her work on continuous lattices (see [Davey & Priestley 89]):

Given two finite sets, $\mathbf{A}$ and $\mathbf{B}$ and a finite set $\mathbf{R}$ such that each $\rho \in \mathbf{R}$ defines a relation on $\mathbf{A}$ and a relation on $\mathbf{B}$, find out how many relation-preserving functions $\mathbf{\Psi:A} \rightarrow \mathbf{B}$ exist.

**Defn:** A function $\mathbf{\Psi:A} \rightarrow \mathbf{B}$ is relation-preserving iff pairs of related elements in $\mathbf{A}$ map to pairs of related elements in $\mathbf{B}$, ie:

$$\forall\rho \in \mathbf{R}.\forall \mathbf{x,y} \in \mathbf{A}.\ \mathbf{x}\rho\mathbf{y} \Rightarrow \Psi(\mathbf{x})\ \rho\ \Psi(\mathbf{y})$$

Thus we want to find the size of the set:

$$\{\mathbf{\Psi:A} \rightarrow \mathbf{B} | \forall\rho \in \mathbf{R}.\forall \mathbf{x,y} \in \mathbf{A}.\ \mathbf{x}\rho\mathbf{y} \Rightarrow \Psi(\mathbf{x})\ \rho\ \Psi(\mathbf{y})\}$$

Note that if $\Psi$ is a map from a proper subset of $\mathbf{A}$, say from $\mathbf{A}'$ where $\mathbf{A}' \subset \mathbf{A}$, and $\mathbf{a} \in \mathbf{A} - \mathbf{A}'$ then the extension of $\Psi$ to $\mathbf{A}' \cup \{\mathbf{a}\}$ where $\Psi(\mathbf{a}) = \mathbf{b}$ is relation-preserving iff the following holds:

$$\forall\rho \in \mathbf{R}.\forall \mathbf{a}' \in \mathbf{A}_1.\ \big(\mathbf{a}\ \rho\ \mathbf{a}' \Rightarrow \mathbf{b}\ \rho\ \Psi(\mathbf{a}')\big) \wedge \big(\mathbf{a}'\ \rho\ \mathbf{a} \Rightarrow \Psi(\mathbf{a}')\ \rho\ \mathbf{b}\big) \wedge \big(\mathbf{a}\ \rho\ \mathbf{a} \Rightarrow \mathbf{b}\ \rho\ \mathbf{b}\big)$$

Thus if we know that $\Psi$ is relation-preserving on $\mathbf{A}'$ and $\mathbf{a} \notin \mathbf{A}'$ then $\Psi \cup \{\langle \mathbf{a,b} \rangle\}$ is relation-preserving on $\mathbf{A}' \cup \{\mathbf{a}\}$ iff

$$\forall\rho \in \mathbf{R}.\forall \mathbf{a}' \in \mathbf{A}_1.\ \big(\mathbf{a}\ \rho\ \mathbf{a}' \Rightarrow \mathbf{b}\ \rho\ \Psi(\mathbf{a}')\big) \wedge \big(\mathbf{a}'\ \rho\ \mathbf{a} \Rightarrow \Psi(\mathbf{a}')\ \rho\ \mathbf{b}\big) \wedge \big(\mathbf{a}\ \rho\ \mathbf{a} \Rightarrow \mathbf{b}\ \rho\ \mathbf{b}\big)$$

Also any relation-preserving function on $\mathbf{A}'$ is also relation-preserving on any subset of $\mathbf{A}'$ so the set of extensions of $\Psi$ on $\mathbf{A}'$ where $\mathbf{A}' \cup \{\mathbf{a}\} \subseteq \mathbf{A}$ is the union of all relation-preserving extensions of $\Psi \cup \{\langle \mathbf{a,b} \rangle\}$ for all values of $\mathbf{b}$. If $\Psi \cup \{\langle \mathbf{a,b} \rangle\}$ is not relation-preserving then there cannot be any relation-preserving extensions of it.

We therefore define the function $\mathbf{extend}(\Psi,\mathbf{A},\mathbf{B})$ which returns the number of relation-preserving extensions of $\Psi$ to $\mathbf{A}$ given that $\Psi$ is relation-preserving on its domain (which is a subset of $\mathbf{A}$) as follows:

```
funct extend($\Psi$,A,B) $\equiv$
⌈ D:=Dom($\Psi$); E:=A$-$D;
 if E= $\varnothing$
  then 1
  else ⌈ count:=0;
     pick any a$\in$E;
     for b$\in$B do
       if rel-pres-ext($\Psi$,a,b)
        then count:=count+extend($\Psi \cup \{\langle a,b \rangle\}$,A,B) fi od;
     r ⌋ fi ⌋.
```

Here **rel-pres-ext(Ψ,a,b)** is a Boolean function which returns true iff $\Psi \cup \{\langle \mathbf{a},\mathbf{b}\rangle\}$ is relation-preserving given that Ψ is.

This can be treated using the methods above. We represent the sets **A** and **B** by integers **SA** and **SB** where **A** contains the **SA** smallest allowed elements. Ψ can then be represented as an array of integers, and the set **R** of relations by two 3D arrays of boolean, **RA** and **RB** where **RA[i,j,k]=1** iff the **i**th relation holds between the **j**th and **k**th elements of **A**, and similarly for **B**. This leads to the iterative version:

```
Prog ≡  count:=0; D:=0;
   do if D=SA then count:=count+1; b:=Ψ[D]; D:=D−1; a:=D+1; b:=b+1
        else a:=D+1; b:=1 fi;
     do if b⩽SB then rp:=tt; ρ:=1; DO; if rp=tt then exit fi; b:=b+1
          else if D=0 then exit(2) fi;
            b:=Ψ[D]; D:=D−1; a:=D+1; b:=b+1 fi od;
     D:=D+1; Ψ[D]:=b od.
where
proc DO  ≡  while ρ ⩽R ∧ rp=tt do
      if RA[ρ,a,a] ∧ ¬RB[ρ,b,b] then rp:=ff fi;
      a′:=1;
      while a′ ⩽D ∧ rp=tt do
        if RA[ρ,a,a′] ∧ ¬RB[ρ,b,Ψ[a′]] then rp:=ff fi;
        if rp=tt ∧ RA[ρ,a′,a] ∧ ¬RB[ρ,Ψ[a′],b] then rp:=ff fi;
        a′:=a′+1 od;
      ρ:=ρ+1 od.
```

## INVERSE ENGINEERING

So far our transformations have mostly been used to transform specifications into programs and generate different versions of a program. However, they have wider applications in the field of software maintenance, particularly in the area of program analysis. Since most of the transformations are invertible they can also be used to derive the specification of a given program, starting with only the source code and possibly a vague idea of what the program is supposed to do. Our methods can be used in the verification of a “tricky” algorithm: either by attempting to transform the specification into that algorithm (as we have done for the Schorr-Waite graph-marking algorithm), or by transforming the algorithm into a more tractable (but perhaps less efficient) form, or by a combination of these techniques. Failure of an attempt to derive a specification from an incorrect algorithm can often provide valuable insights into the source of the error (as we shall see below). The process of extractinH specifications from existing code is termed "Inverse Engineering". For our final example we take a published program which was written in such a way that the structure and effect of the program are very hard to discern; we apply transformations which reveal the structure and enable its effect to be summarised as a specification.

Naturally, to show the value of these techniques for "real" programs we should give a much longer example: but space precludes such an exercise. A compromise is to choose a program which still manages to exhibit a lot of complexity due to the convoluted logic and control flow, hence the example will look rather artificial but should serve to demonstrate the applicability of the methods to real sized programs. For working with large programs a semi-automatic interactive system for applying the transformations without introducing clerical errors is desirable. The author is currently working on such a system to form part of the Alvey project "An Intelligent Knowledge-Based System for Software Maintenance" which aims to assist the maintenance programmer in understanding and modifying an initially unfamiliar system given only the source code.

The program was originally written in DataFlex and published in [Fenton 86], we have transcribed it into our notation, replacing references to files by arrays. The procedure **INHERE** was originally a label in the middle of an **if** statement in the middle of a loop! (This loop is represented by procedure **L** below).

```
PROG ≡ line:=""; m:=false; i:=1; INHERE.
L ≡ i:=i+1;
  if i=n+1 then ALLDONE fi;
  m:=true;
  if item[i]≠last then write(line); line:=""; m:=false; INHERE fi;
  MORE.
INHERE ≡ p:=number[i]; line:=item[i]; line:=line+" "+p; MORE.
MORE ≡ if m then p:=number[i]; line:=item[i]; line:=line+", "+p fi;
     last:=item[i]; L.
ALLDONE ≡ write(line); Z.
```

Applying our recursion-removal techniques to **L** we get:

```
L ≡ do i:=i+1;
    if i=n+1 then exit fi;
    m:=true;
    if item[i]≠last
     then write(line); line:=“”; m:=false;
         p:=number[i]; line:=item[i]; line:=line+“ ”+p;
         if m then p:=number[i]; line:=item[i]; line:=line+“, ”+p fi;
         last:=item[i]
     else if m then p:=number[i]; line:=item[i]; line:=line+“, ”+p fi;
         last:=item[i] fi od;
  ALLDONE.
```

Here the tests of **m** are redundant: in the first test **m** is always true and in the second test **m** is always false. We can then merge some assignments such as **line:=item[i]; line:=line+“ ”+p**. The remaining test of **m** is now redundant so the variable **m** can be completely removed. We get the following version of the program:

```
PROG ≡ i:=1; p:=number[i]; line:=item[i]; line:=line+“ ”+p; last:=item[i];
     do i:=i+1;
       if i=n+1 then exit fi;
       p:=number[i];
       if item[i]≠last
        then write(line); line:=item[i];
            line:=line+“ ”+p
        else line:=line+“, ”+p fi;
      last:=item[i] od;
     write(line).
```

The two statements **line:=line+“ ”+p** and **line:=line+“, ”+p** are almost the same, they can be made the same by adding another variable **sep** (separator) which is set to **“ ”** or **“, ”** as appropriate. Then the statement **line:=line+sep+p** can be moved out of the **if** and taken, with **last:=item[i]**, to the beginning of the loop. This leaves two copies of **sep:=“ ”**; we convert the loop to a double loop and move some statements of the outer loop to the beginning. After some further transformation (which lack of space prevents us from giving in detail) we are left with the following version of the program:

```
PROG ≡ i:=1;
       do last:=item[i]; line:=last; sep:=" ";
         do p:=number[i]; line:=line+sep+p;
            i:=i+1;
            if i=m+1 then exit fi;
            if item[i]≠last then else fi;
            sep:=", " od;
         write(line);
         if i=n+1 then exit fi od.
```

The transformations hav revealed the "true" structure of the program, which involves a double loop. The program scans through a sorted file (here represented by the arrays **item** and **number**) consisting of words and page references. The outer loop scans through distinct items and for each distinct item the inner loop steps through the page references for that item. Writing the program as a single loop whose body must distinguish the two cases of a new item and a repeated item obscured the simple basic structure which has been revealed through transformations. This kind of transformation has important applications in program maintenance: the second version is far easier to understand and modify: there is only one copy of the statement which writes to the file, for example, and the "flag" **m** which was used to direct the control flow is now not needed. The transformation from first version to second used only general transformations which have been proved to work in all cases, and so could be applied without having to understand the program first.

The transformations have also revealed a bug in the program: note that the outer loop is in the form of a **REPEAT...UNTIL** loop and so the body is executed at least once. Hence the program will not work correctly if presented with an empty file. This bug is not immediately obvious in the first version of the program. For the first version a typical "fix" would be to add a test for an empty file and **goto** a label at the end of the program. In that case this "fix" is also typical in that it further obscures the program structure, increases the program length and increases the number of identifiers used. In contrast with this, to carry out the same modification to the second version we merely change the outer loop to a **while** loop.

## Deriving a Specification

We will now perform some further transformations aimed at extracting the specification of this program. Note that the first line of the inner loop plays two roles: adding a space and the first line number to **line** and adding subsequent numbers to the line separated by commas. Expressing a loop as a single statement is easier if each execution of the loop does a "similar" job; this suggests taking out the first statement of the loop. This will enable us to transform the loop into a **while** loop:

```
PROG ≡ i:=1;
      while i≠n+1 do
         last:=item[i]; line:=last+“ ”+number[i];
         i:=i+1;
         while (i≠n+1) ∧ (item[i]=last) do
           line:=line+“, ”+number[i];
           i:=i+1 od;
         write(line) od.
```

Within the inner loop, if $\mathbf{i}_0$ and $\mathbf{line}_0$ are the initial values of **i** and **line** then we see that the condition:

$$\mathbf{line} = \mathbf{line}_0 + \sum_{i0<j\leqslant i}\langle\text{”, ”}+\mathbf{number[j]}\rangle$$

is preserved by the loop body (here $\sum$ represents concatenation of a sequence of strings). So we can assign this value to **line** after the loop and remove the assignment inside the loop. The loop becomes:

```
        while (i≠n+1) ∧ (item[i]=last)
          i:=i+1 od
```

which can be replaced by the single statement: $\mathbf{i{:=}\mu i'.\big(i' \geqslant i \ \wedge\ (i' = n+1 \ \vee\ item[i'] \neq last)\big)}$ which is read as “**i** becomes the smallest **i′** not less than **i** such that **i′ =n+1** or **item[i′]≠last**”.

So our specification of the program is:

```
PROG ≡ i:=1;
      while i≠n+1 do
        $\mathbf{i_0}$:=i; i:=$\mu$i′.(i′ >i ∧ (i′ =n+1 ∨ item[i′]≠item[i]));
        line:=item[$\mathbf{i_0}$]+“ ”+number[$\mathbf{i_0}$]+ $\sum_{i0<j\leqslant i}$⟨“, ”+number[j]⟩;
        write(line) od.
```

## “PROGRAMMING IN THE LARGE” ISSUES

In giving examples of program development we have of necessity had to restrict ourselves to fairly small programs. However we believe that the techniques developed in this thesis have application to the problems involved in writing large software systems, even though these problems can be very different to the ones encountered in writing small programs. One way of dealing with a large program is to organise it as a set of “contracts” which may be implemented completely independently of one another, as far as correctness is concerned. For efficiency reasons it may be useful to know how a contract will be used, but correctness should be the primary concern in the initial stages of a project: it doesn't matter how fast a program runs if it gives the wrong result: the "null program" **skip** can do that faster than any other program! "The preoccupation with optimisation should be removed from the early stages of programming. The ideal approach would be first, to produce a program that is demonstrably correct, and then, through a series of efficiency-improving transformations, modify this program to produce a correct and efficient one. A language is <u>optimisable</u> if it makes possible the automatic application of these transformations... it has been shown that many features that reduce optimisability also hamper readability." [Ghezzi & Jazayeri 82].

A "contract" consists of a complete and precise specification of a set of procedures, functions and data types together with their implementations. If other procedures etc. are required in the implementation then their specifications are included also. It should be possible to prove that the implementation satisfies the specification purely from the information in the contract: in this sense all contracts are independent. Once all specifications have been implemented (directly or indirectly) as programs then the contracts can be put together to form the required program which will be correct by construction.

The methods in this thesis can be used to provide the implementation part of a contract by transforming the specifications into programs, possibly generating further "subcontracts" in the process.

Each contract should be <u>small</u> in the sense that it can be read and comprehended in one sitting by someone familiar with the language and notation. This suggests that it should be no more than a few pages long. (Further research is needed to discover the "optimal length" of a contract).

Most of the contracts can be implemented in a very high-level programming language, those contracts (and only those) for which it has been demonstrated that efficiency is important can be transformed into a lower level programming language and more effort can be expended on improving their efficiency (by transforming recursion into iteration etc.).

Maintenance is probably the most important issue for all but the most trivial programs. Most professional programmers spend the major part of their time modifying existing programs to fix bugs and add features, rather than writing new ones, yet this aspect of programming has received little theoretical research. The contract approach allows the production of "bug-free" programs which are very easy to modify in order to fulfill an enhanced specification while making the most use of work already done. When the specification of the system is changed the original contract must be re-written, which is likely to require new subcontracts to be written while also re-using many of the original contracts. Re-usability of effort is also enabled by this technique. Instead of building up a collection of monolithic software packages (each imposing a maintenance burden) the programming team will build a library of general purpose specifications and implementations. Any contract may be re-used for any later system: possibly using completely different implementations of the subcontracts. This re-use of results is an important added benefit of this programming method: in particular, since most of the contracts will be written in a high-level, machine-independent language the task of porting programs onto new machines and operating systems will be made easier.

In order for this method of programming to be successful an automated system is required for carrying out the transformations, maintaining the different versions of all the contracts and the links between them and carrying out independent compilation of sets of contracts with "cross-contract" type and version number checking etc. Intelligent storage and retrieval systems for contracts will also enable the maximum use to be made of previous work.

## FURTHER WORK

### The Maintainer's Assistant

We have already given a small example of the application of this research to problems in software maintenance. The author is currently working on an Alvey project at the University of Durham which aims to produce a prototype "Maintainer's Assistant" tool. This tool employs the program transformation techniques developed here and aims to help the maintenance programmer to understand and modify an initially unfamiliar program, given only the source code. The tool consists of a structure editor, a library of proven transformations and a knowledge-based system which analyses the programs and specifications under consideration and uses heuristic knowledge to determine which transformations will achieve a given end (for example, deriving the specification of a section of code, finding the most suitable technique for recursion removal, optimising for efficiency etc.)

### Assembler Programs

The author is also currently working on a project funded by IBM which aims to develop a tool to assist in the formal transformation of assembly code into high-level language code and **Z** specifications. Assembly language programs create special problems with aliasing (common and overlapping data areas, pointers and indexes used in ad-hoc ways etc.). Our solution is to represent the whole store of the machine as a single array, with another array used to represent the registers. With other program analysis techniques this would cause insurmountable problems since the most that could be said about any operation is that "the store has changed in some way". However, we have developed special transformations for dealing with arrays which make this approach a practical one. We have been able to transform the assembler to a HLL representation, replace the "areas of store" by the data structures they implement (using transformations which change the data representation of a program), and then transform this HLL version into a specification. At the moment this has all been done by hand, but many of more tedious operations will be carried out automatically, making this a practical option for the maintenance of large assembly-language programs.