Transforming a Program into a Specification

M.Ward
Centre for Software Maintenance
School of Engineering and Applied Science
University of Durham

July 16, 1993

Abstract

There has been much research in recent years on the problems of program and system development but very little work has been done on the problems of maintaining developed programs. This is despite the fact that for many years it has been well-known that maintenance consumes the largest percentage of the programming budget. We apply the techniques of program transformation to a published program which was written in such a way that the structure and effect of the program are very hard to discern. This will reveal the true structure of the program and enable its effect to be summarised as a specification. The method is language-independent and so can be used with a wide variety of programming languages, the same method can be used to derive a program from a specification, transform a program from one language to another, and (as illustrated here), to transform a program into a specification.

Introduction

In this paper we examine a published program which was written in such a way that the structure and effect of the program are very hard to discern. Various transformations to the program to reveal its structure and enable its effect to be summarised as a specification. The method is essentially language-independent which is very important, for example, in transforming from one programming language to another. Also in transforming a specification into a program one needs a language in which *any* specification can be expressed.

Naturally, to show the value of these techniques as applied to "real" programs they should be applied to a much longer example than this one—but that would be far too big to publish in a paper. A compromise is to choose a small program which still manages to exhibit a lot of complexity due to the convoluted logic and control flow. Hence the example will look slightly artificial but should show that the methods will indeed "scale up" to real sized problems (with the aid of a suitable semi-automatic system for applying the transformations without introducing clerical errors). Note that it is not necessary to follow exactly how the program works in order to follow the transformations since they have been rigorously proved and apply to any program. Indeed, one application of these techniques is for a maintenance programmer to apply them to a program he does not understand to discover what the program does.

All the transformations made use of in this paper are proved in the author's DPhil thesis **[Ward 87]**. The proof method is based on "weakest preconditions", invented by Dijkstra in **[Dijkstra 76]**. For a given program S and condition on the final state R the weakest precondition $\mathsf{WP}(\mathsf{S},\mathsf{R})$ is the weakest condition on the initial state such that the program will terminate in a state satisfying condition R. It is possible to express the weakest precondition of any program or specification as a single formula in infinitary logic (an extension of first order logic which allows infinitely long formulae). It was shown in **[Ward 87]** that two programs S_1 and S_2 are equivalent if and only if the corresponding weakest preconditions $\mathsf{WP}(\mathsf{S}_1,\mathsf{R})$ and $\mathsf{WP}(\mathsf{S}_2,\mathsf{R})$ are equivalent formulas for any formula R. This reduces the problem of proving two programs to be equivalent to a problem in logic to which all the standard methods and techniques of theorem proving can be applied. The success of this technique may be judged by the large number of useful transformations which have been proved and the wide range of problems to which they can be applied—of which the problem in this paper is a small example.

Program transformations are used in program development in **[Griffiths 76]**, **[Griffiths 79]**, **[Bauer 76]** and **[Bauer 79]**. However their methods cannot cope with general specifications or with transforming programs into specifications.

The next section gives some examples of transformations which will be used later in the paper; then the program is presented and various transformations are appled to reveal its structure. The effect of a typical program modification is discussed with respect to the two versions of the program. Finally these ideas are related to the practical problems of program maintenance and the current work at the Centre for Software Maintenance at Durham.

Examples

This section describes some examples of the kinds of transformations used in the rest of the paper.

Assignment merging

Suppose we have the sequence: x:=x*2; x:=x+1 where we want to merge the two assignments. The weakest precondition is:

$$\begin{aligned} \mathsf{WP}(\mathsf{x{:=}x{*}2;\ x{:=}x{+}1,\ R}) &\Leftrightarrow \mathrm{WP}(x{:=}x{*}2,\ \mathrm{WP}(x{:=}x{+}1,\ R)) \\ &\Leftrightarrow \mathrm{WP}(x{:=}x{*}2,\ R[x{+}1/x]) \end{aligned}$$

where R[x+1/x] is R with x replaced by x+1.

$$\begin{aligned} &\Leftrightarrow R[x{+}1/x][x{*}2/x] \\ &\Leftrightarrow R[(x{*}2){+}1/x] \\ &\Leftrightarrow \mathrm{WP}(x{:=}(x{*}2){+}1,\ R) \end{aligned}$$

This has shown that x:=x*2; x:=x+1 is equivalent to x:=x*2+1.

In addition to the usual <u>while</u> loops we will be using loops of the form: <u>do</u> S <u>od</u> where S is the body of the loop. When a statement <u>exit</u>(n) is encountered (where n is an integer constant) then the nth <u>do</u>-loop enclosing the statement will be terminated with execution continuing after the loop. <u>exit</u>(1) is written <u>exit</u> and <u>exit</u>(0) is equivalent to skip, as might be expected. Such loops have been used in program transformations in **[Arsac 79]** and **[Arsac 82]**. **[Peterson et al 73]** showed how these statements can be used to remove <u>g</u>oto statements from a program without increasing the size of the text or introducing flag variables.

Loop Inversion

We define a "proper sequence" to be a statement which contains no <u>exit</u> statements which would leave a loop enclosing the sequence. Suppose S_1 is a proper sequence and S_2 is any statement. Then the following programs are equivalent:

<u>do</u> S_1;
    S_2 <u>od</u>

S_1;
<u>do</u> S_2;
    S_1 <u>od</u>

This is often described as "moving the statement S_1 to the end/beginning of the loop": it is a special case of a transformation proved in **[Ward 87]**. It can be useful in the reverse direction for removing duplicate copies of a statement and in the forward direction for converting a loop of the form:

```
do S1;
   if B then exit fi;
   S2 od
```

where S_1 and S_2 are proper sequences to the while loop:

```
S1;
while ~B do
    S2;
    S1 od
```

Absorption

By "absorption" we mean moving a statement inside a compound statement which preceds it, for example:

```
if B then S1 else S2 fi;
S3
```

is equivalent to:

```
if B then S1; S3 else S2; S3 fi
```

If this results in an exit statement followed by another statement then the subsequent statement can be removed as it cannot be reached, for example:

```
if B then exit fi;
S1
```

is equivalent to:

```
if B then exit else S1 fi
```

Statements can also be absorbed into a loop, the absorbed statement must be incremented according to the number of nested loops it is moved into. By "incrementation" we mean that exit is replaced by exit(2) etc. For example:

```
do S1;
   if B then exit fi;
   S2 od;
if Q then S3 fi
```

is equivalent to:

```
do S1;
   if B
     then if Q then S3; exit
                else exit fi fi;
   S2 od;
```

False Loop

Any statement can be converted to a loop if all the statements which cause termination of that statement are incremented and the statement is enclosed in do..od brackets. This can be combined with the inverse of absorption to "factor out" multiple copies of a statement. For example:

```
if B then if B1 then S1
               else S fi
   else if B2 then S2
              else S fi fi
```

becomes:

```
do if B then if B1 then S1; exit fi
       else if B2 then S2; exit fi fi;
   S; exit od
```

where the two copies of S have been combined into one. To see that these are equivalent, absorb the single S into the preceding if statement and remove the false loop.

The Program

The program was originally written in DataFlex but has been transcribed into C which is a more familiar language to many programmers. The program was published in **[Fenton 86]**. Note that the variable `morenum` is used to control the execution flow; note also that the statement `goto inhere` jumps into the middle of an `if` statement that appears in the middle of a loop. (See Figure 1).

The program reads the file "DIDB", a sorted text file which consists of a number of lines each containing a page number followed by a word. It produces an output file listing each word followed by a list of the corresponding page numbers. Examining the operations applied to the text file suggests that we can represent the "file" data structure as a sorted array item[1..n] and a corresponding array page[1..n] (where n>0 is the number of records in the file) together with an index variable i which represents the current position in the file (ie the position of the *next* record to be read). With this representation:

`fp_in = fopen("DIDB","r")`	becomes	i:=1
`filestat = fscanf(fp_in,"%s%s",page,item)`	becomes	i:=i+1

The variable `filestat` is set to `EOF` when we attempt to read past the end of the file, ie when i reaches the value n+1, thus we can replace the test `(filestat == EOF)` by (i=n+1). Replacing `theline` by line, `lastitem` by last, `page` by p, `morenum` by the Boolean variable m (for brevity), the program becomes: (where procedure L represents the `for(;;)` loop and PROG the whole program):

```
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:=line+", "+p fi;
       last:=item[i]; L.

ALLDONE ≡ write(line); Z.
```

Here the procedure **Z** causes termination of the whole program.

```
#include <stdio.h>

main()          /* DataFlex transcription */
{
int morenum, filestat;
char page[6], theline[51], item[31], lastitem[31];
FILE *fopen(), *fp_in, *fp_out;
    fp_in = fopen("DIDB","r");
    fp_out = fopen("DID2INX.TXT","w");
    theline[0] = '\0';
    morenum = 0;
    filestat = fscanf(fp_in,"%s%s",page,item);
    goto inhere;
    for(;;) {
filestat = fscanf(fp_in,"%s%s",page,item);
if (filestat == EOF) goto alldone;
morenum = 1;
if (strcmp(item,lastitem)) {
    fprintf(fp_out,"%s\n",theline);
    theline[0] = '\0';
    morenum = 0;
  inhere:
    strcpy(theline,item);
    strcat(theline," ");
    strcat(theline,page);
}
if (morenum) {
    strcat(theline,", ");
    strcat(theline,page);
}
strcpy(lastitem,item);
    }
  alldone:
    fprintf(fp_out,"%s\n",theline);
    close(fp_in);
    close(fp_out);
    }
```

Figure 1: The Original C Program

Applying the Transformations

This shows the true structure of the original program (which at first glance may have looked quite structured!). We can now apply our transformations to simplify this structure without changing the effect of the program. First copy INHERE into L and then copy MORE into L (twice) to obtain:

```
L  ≡  i:=i+1;
      if i=n+1 then ALLDONE 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:=line+", "+p fi;
             last:=item[i]; L fi;
      if m then p:=number[i]; line:=line+", "+p fi;
      last:=item[i]; L.
```

The first test of m is redundant as we know that m has just been assigned the value true. Note that every procedure call results in a call to another procedure and the program terminates immediately when the procedure Z is called. Thus the "procedure calls" are really goto statements as the execution will never return from a procedure to the procedure which called it. This means that the second test of m is also redundant as it can only be reached when item[i]=last. Removing these tests gives:

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

Absorbing the last two lines into the preceding if statement makes the procedure L tail-recursive; this is because the sequence L; p:=number[i];... can be replaced by L because we know that the procedure L can only be terminated by a call to Z which causes immediate termination of the program so any statements after L will never be executed so can be ignored. The tail-recursive procedure L can now be transformed into a loop:

```
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;
                last:=item[i]
           else p:=number[i]; line:=line+", "+p;
                last:=item[i] fi od;
      ALLDONE.
```

We can merge some of the assignments (eg. line:="" followed by line:=item[i]) and take common statements out of the two arms of the if to obtain:

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

Copy ALLDONE into L, L into MORE, MORE into INHERE and INHERE into PROG to get the (iterative) version of the whole program:

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

The remaining test of m is now redundant so can be removed. We then find that m is only assigned and never accessed so it can be removed altogether (it is a "dead variable"). The call Z at the end can be removed since the procedure is not recursive so will terminate at the end of the first call anyway. Now merge assignments as above:

```
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 (common) statement line:=line+sep+p can be taken out of the if statement and then moved with the statement last:=item[i] to the beginning of the loop:

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

We want to take the second occurrence of sep:=" " to the end of the loop (so we can move it to the beginning of the loop and remove the first occurrence). We can do this by converting the loop to a double loop and then taking some statements out of the inner loop:

```
PROG ≡ i:=1; p:=number[i]; line:=item[i];
       sep:=" ";
       do do line:=line+sep+p; last:=item[i];
             i:=i+1;
             if i=n+1 then exit(2) fi;
             p:=number[i];
             if item[i]≠last
               then exit
               else sep:=", " fi od;
          write(line);
          line:=item[i];
          sep:=" " od;
       write(line).
```

Now move the last two statements of the outer loop to the beginning:

```
PROG ≡ i:=1; p:=number[i];
       do line:=item[i];
          sep:=" ";
          do line:=line+sep+p; last:=item[i];
             i:=i+1;
             if i=n+1 then exit(2) fi;
             p:=number[i];
             if item[i]≠last
               then exit
               else sep:=", " fi od;
          write(line) od;
       write(line).
```

Now we want to remove one of the copies of p:=number[i] so we move the second copy to the end of the inner loop, the statement sep:=", " can also be moved out of the if statement:

```
PROG ≡ i:=1; p:=number[i];
       do line:=item[i];
          sep:=" ";
          do line:=line+sep+p; last:=item[i];
             i:=i+1;
             if i=n+1 then exit(2) fi;
             if item[i]≠last then p:=number[i]; exit fi;
             sep:=", ";
             p:=number[i] od;
          write(line) od;
       write(line).
```

Take the copy of p:=number[i] before the exit out of the inner loop. We can then remove the outer copy and move the statements to the beginning of the outer loop:

```
PROG ≡ i:=1;
       do p:=number[i];
          line:=item[i]; sep:=" ";
          do line:=line+sep+p; last:=item[i];
             i:=i+1;
             if i=n+1 then exit(2) fi;
             if item[i]≠last then exit fi;
             sep:=", ";
             p:=number[i] od;
          write(line) od;
       write(line).
```

Insert last:=item[i] before the inner loop and move it to the end of the inner loop, then use the fact that last=item[i] is true at the end of the inner loop to remove that statement. Also absorb the two write(line)s into their loops:

```
PROG ≡ i:=1;
       do last:=item[i]; line:=item[i]; sep:=" ";
          p:=number[i];
          do line:=line+sep+p;
             i:=i+1;
             if i=n+1 then write(line); exit(2) fi;
             if item[i]≠last then write(line); exit fi;
             sep:=", ";
             p:=number[i] od od.
```

Now the statement p:=number[i] can be moved to the beginning of the inner loop. Also both exit and exit(2) can be replaced by:

if i=n+1 then exit(2) else exit fi

This is because we know i=n+1 is true when exit(2) is executed and false when exit is executed. This statement and write(line) can be taken out of the inner loop, note that the if statement must be decremented to:

if i=n+1 then exit else exit(0) fi

which is the same as:

if i=n+1 then exit fi

The program becomes:

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

```
#include <stdio.h>

main()          /* DataFlex transcription */
{
    int filestat;
    char page[6], theline[51], item[31], lastitem[31], sep[3];
    FILE *fopen(), *fp_in, *fp_out;
    fp_in = fopen("DIDB","r");
    fp_out = fopen("DID2INX.TXT","w");
    filestat = fscanf(fp_in,"%s%s",page,item);
    do {
strcpy(lastitem,item);
strcpy(theline,lastitem);
strcpy(sep," ");
for(;;) {
    strcat(theline,sep);
    strcat(theline,page);
    filestat = fscanf(fp_in,"%s%s",page,item);
    if (filestat == EOF) break;
    if (strcmp(item,lastitem)) break;
    strcpy(sep,", ");
}
fprintf(fp_out,"%s%s",theline,"\n");
    } while (filestat != EOF);
}
```

Figure 2: The Transformed C Program

Finally line:=item[i] can be replaced by line:=last since we have just assigned item[i] to last. This gives the final 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=n+1 then exit fi;
              if item[i]≠last then exit fi;
              sep:=", " od;
           write(line);
           if i=n+1 then exit fi od.
```

Transcribing this back into C we obtain Figure 2.

Discussion

As can be seen, the transformations have revealed the "true" structure of the program—which involves a double loop. The program scans through the sorted file `DIDB` each line of which contains an item and a page reference. The outer loop scans through the distinct items and for each distinct item the inner loop steps through the page references for each item. Writing the program as a single loop, whose body must distinguish the two cases of a new item and a repeated item, obscures the simple basic structure. The transformations have revealed this structure, and because they have all been rigorously proved (see **[Ward 87]** for proofs) they can be applied without having to understand the program first.

This kind of transformation has important applications in program maintenance. The second version of the program is far easier to understand and modify—there is only one copy of the statement which writes to the output file for example. However, it is obtained from the first by applying a set of general rules which indicate which transformations to apply, without having to understand the program first.

```
#include <stdio.h>

main()          /* DataFlex transcription */
{
    int filestat;
    char page[6], theline[51], item[31], lastitem[31], sep[3];
    FILE *fopen(), *fp_in, *fp_out;
    {
fp_in = fopen("DIDB","r");
fp_out = fopen("DID2INX.TXT","w");
filestat = fscanf(fp_in,"%s%s",page,item);
while(filestat != EOF) {
    strcpy(lastitem,item);
    strcpy(theline,lastitem);
    strcpy(sep," ");
    for(;;) {
strcat(theline,sep);
strcat(theline,page);
filestat = fscanf(fp_in,"%s%s",page,item);
if (filestat == EOF) break;
if (strcmp(item,lastitem) != 0) break;
strcpy(sep,", ");
}
    fprintf(fp_out,"%s%s",theline,"\n");
    }
}
    }
```

Figure 3: The modified Program

The transformations have also revealed a bug in the program—the outer loop is in the form of a repeat...until loop and hence will always be executed at least once. Hence the program will not work correctly when presented with an empty file. This bug is not immediately obvious in the first version of the program. For the first version a typical solution is to add the line:

```
if (filestat == EOF) goto finish;
```

after the first `fscanf` and add another label `finish` just before the `close` statements. This is typical in that it makes a large change in the program structure which is only reflected in a small change in the program text (which incidentally gets longer). To carry out the same modification to the second version of the program we merely change the outer loop to a while loop:

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

This then transcribes to Figure 3.

Future Work

At Durham we are currently researching into a semi-automatic program transforming system which could carry out the transformation from the first version to the second with very little human assistance. We regard this as an essential step that should be performed before modifications which change the program effect are carried out. The system will also assist in the process of further transforming a program to extract its specification. This will generally involve the use of operations which, although precisely defined in mathematical terms, are not generally available as primitive operations in programming languages.

Deriving a Specification

We will now carry out some further transformations in order to demonstrate how to produce the specification of this program. The first stage is to collapse the inner loop into a single statement. The first line of the inner loop plays two roles—adding a space and the first number to line and adding subsequent numbers to 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 step of the loop and moving it to the end of the loop.

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

Now we can remove the variable sep and if item[n+1] is accessible with some arbitrary value we can write the inner loop as a while loop as follows:

```
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.
```

Consider the inner loop. Add the assignments $\mathsf{i_0}$:=i and $\mathsf{line_0}$:=line just before the loop. Then we see that the condition:

$\mathsf{line} = \mathsf{line}_0 + \sum_{i_0<j\leq i}\langle", "+\mathsf{number[j]}\rangle$

is preserved by the loop body. Here the $\sum$ indicates concatenation of the sequence of strings. Hence the condition will be true after the loop and we can add the assignment $\mathsf{line}:=\mathsf{line}_0 + \sum_{i_0\leq j\leq i}\langle", " + \mathsf{number[j]}\rangle$ after the loop. Then the assignments to line *inside* the loop are redundant and can be removed. The loop becomes:

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

This loop can be replaced by the assignment $\mathsf{i}:=\mu\mathsf{i}'.(\mathsf{i}'\geq\mathsf{i} \wedge (\mathsf{i}'=\mathsf{n}+1 \vee \mathsf{item[i']}\neq\mathsf{last}))$

which is read as "i becomes the smallest i′ greater than or equal to i such that i′=n+1 or item[i′]≠last.

The program becomes:

```
PROG ≡ i:=1;
       while i≠n+1 do
           last:=item[i]; line:=last+" "+number[i];
           i:=i+1;
           i_0:=i;
           line_0:=line;
           i:=μi'.(i'≥i ∧ (i'=n+1 ∨ item[i']≠last));
           line:=line_0+Σ_{i_0<j≤i}⟨", "+number[j]⟩
           write(line) od.
```

This can be simplified to:

```
PROG ≡ i:=1;
       while i≠n+1 do
           i_0:=i;
           i:=μi'.(i'>i ∧ (i'=n+1 ∨ item[i']≠item[i]));
           line:=item[i_0]+" "+number[i_0]+Σ_{i_0<j≤i}⟨", "+number[j]⟩
           write(line) od.
```

After the assignment to i we have either i=n+1 or $\mathsf{item[i]}\neq\mathsf{item[i_0]}$. We also have:

$\forall \mathsf{j}.(\mathsf{i_0} \leq \mathsf{j}<\mathsf{i} \Rightarrow \mathsf{item[j]}=\mathsf{item[i_0]})$.

So the outer loop steps through the distinct items in the item array and for each distinct item writes a line consisting of the item followed by a comma-separated list of the numbers in the number array which correspond to that item.

References

[Arsac 79]	J.Arsac	Syntactic Source to Source Program Transformations and Program Manipulation. Comm. A.C.M. 1979 P.43-54.
[Arsac 82]	J.Arsac	Transformation of Recursive Procedures. in **[Neel 82]** P.211-265 1982.
[Bauer 76]	F.L.Bauer	Programming as an Evolutionary Process. in **[LNCS v46]** P.153-182 1976.
[Bauer 79]	F.L.Bauer	Program Development By Stepwise Transformations - the Project CIP. in **[LNCS v69]** P.237-266 1979.
[Dijkstra 76]	E.Dijkstra	A Discipline of Programming. Prentice–Hall 1976.
[Fenton 86]	M.Fenton	Developing in DataFlex, Book 2, Reports and other outputs. B.E.M. Microsystems 1986.
[Griffiths 76]	M.Griffiths	Program Production by Successive Transformation. in **[LNCS v.46]** P.125-152 1979.
[Griffiths 79]	M.Griffiths	Development of the Schorr-Waite Algorithm. in **[LNCS v.69]** P.464-471 1979.
[Neel 82]	D.Neel (Ed)	Tools and Notations for Program Construction. Cambridge University Press 1982.
[Peterson et al 73]	W.W.Peterson, T.Kasami, N.Tokura	On the Capabilities of While, Repeat and Exit Statements. Comm. A.C.M. 16,8, Aug 1973, P.503-512.
[LNCS v.46]	F.L.Bauer K.Samelson (Eds)	Language Hierarchies and Interfaces. Lecture Notes in Computer Science. Volume 46. Springer Verlag 1976.
[LNCS v.69]	G.Goos H.Hartmanis (Eds)	Program Construction. Lecture Notes in Computer Science. Volume 69. Springer Verlag 1979.
[Ward 87]	M.Ward	Proving Program Refinements and Transformations D.Phil Thesis, Oxford University, 1987.