1983

Program development by inductive step wise refinement

R. Geoff Dromey
University of Wollongong

Program Development by Inductive Stepwise Refinement

R. Geoff Dromey

Department of Computing Science
University of Wollongong

Preprint No. 83-11 **December 16, 1983**

P.O. Box 1144, WOLLONGONG N.S.W. 2500, AUSTRALIA
tel (042)-270-859
telex AA29022

Program Development by Inductive Stepwise Refinement

R. Geoff Dromey

Department of Computing Science,
University of Wollongong,
P.O. Box 1144,
Wollongong N.S.W. 2500
Australia.

ABSTRACT

A constructive method of program development is presented. It seeks to unify two important ideas about program development. Namely that programming is a goal-oriented activity and that there should be a correspondence between data and program structures. The latter concept is seen to be extensible beyond the data processing context in which it was originally proposed. Induction provides the vehicle for program development by stepwise refinement, with the final program being constructed by application of a sequence of progressively more powerful generalizations. The design process employed guarantees the correctness of the final program provided each of the refinement steps have been correctly taken. The method is illustrated by a number of examples.

Key Words:

program development, inductive stepwise refinement, structure clashes, induction, orienting mechanism, postcondition, syntax diagrams, programming methodology.

Program Development by Inductive Stepwise Refinement

R. Geoff Dromey

Department of Computing Science,
University of Wollongong,
P.O. Box 1144,
Wollongong N.S.W. 2500
Australia.

ABSTRACT

A constructive method of program development is presented. It seeks to unify two important ideas about program development. Namely that programming is a *goal-oriented activity* and that there should be a correspondence between data and program structures. The latter concept is seen to be extensible beyond the data processing context in which it was originally proposed. Induction provides the vehicle for program development by stepwise refinement, with the final program being constructed by application of a sequence of progressively more powerful generalizations. The design process employed guarantees the correctness of the final program provided each of the refinement steps have been correctly taken. The method is illustrated by a number of examples.

1. INTRODUCTION

There have been two far-reaching contributions to our understanding of program development over the past decade, one concerning the nature of programming, and the other concerning the *structure* of programs. Dijkstra [1], and more recently Gries [2] have demonstrated the importance of treating program development as a *goal-oriented activity*. On the other hand, Jackson [3] has shown the advantages of matching the program structure with the structure of the data. We shall present a method of program development that seeks to unify these two important ideas.

Inductive stepwise refinement is a constructive method of program development. It is based on a partitioning rule which requires that the simplest aspects of the final solution are always dealt with first. The final program is constructed in a stepwise fashion by applying a sequence of progressively more powerful generalizations - a systematic application of induction. This strategy for development has two advantages. Firstly the correctness of a given refinement step can be judged without reference to parts of the mechanism that have yet to be developed. Secondly the correctness of the final program can be guaranteed provided each of the refinement steps has been carried out correctly.

Programs developed using inductive stepwise refinement possess a structure that conforms to the structure of the data. Where the structure of the data is unimportant in influencing the structure of the program it is found that programs assume a structure consistent with the properties of the transformations that are applied to the data. Common modularity is identified and exploited in both the development and structure of programs produced by this method.

2. INDUCTION AND PROGRAM DEVELOPMENT

Induction is variously defined as "a beginning, commencement, the process of reasoning or drawing a conclusion from particular facts or individual cases" [4].

In the deductive sciences the usefulness of induction is recognized. For example Polya in his classic works on mathematical problem-solving gives an excellent account of the importance and constructive role of induction (as distinct from mathematical induction) in the solution of problems [5, 6, 7].

In program development, induction has usually been equated with bottom-up design. This has meant that it has gained the reputation of being useful in identifying specific components of a solution to a problem but of lacking a systematic procedure for controlling how these components should be organized [8]. Clearly, if induction is to have any credibility in program development, it will need to be applied in a way that transcends the difficulties associated with traditional bottom-up design - a systematic procedure is essential.

There are two central concerns in program development, what must be accomplished, and how it is to be accomplished. The postcondition gives a specification for what is to be accomplished while iteration provides a primary means for establishing the goal. **The role that induction can play in program development is to provide a vehicle for systematically linking iteration (or recursion) to the postcondition.** This idea forms the basis of program development by inductive stepwise refinement.

3. INDUCTIVE STEPWISE REFINEMENT

The application of induction in program development proceeds in two stages. Firstly an appropriate starting point must be chosen then a succession of generalizing steps are applied. Guidelines are needed to assist both stages of development.

3.1. The Orienting Mechanism

An appropriate starting point for the development of a program is the identification and implementation of the most **elementary** iterative (or recursive) mechanism that can establish the postcondition under the most restrictive conditions. This mechanism is the **orienting mechanism.** Its role is to provide a suitable reference point for the subsequent development of the program. The orienting mechanism establishes the postcondition by changing the least number of variables with the least computational effort.

There are several reasons for choosing a starting point with these properties:

- it can help to ensure that the degree of complexity that must be dealt with initially and subsequently is kept as small as possible

- it is necessary to ensure that data-structure and problem-structure dependencies are recognized and exploited in the design

- it gives the opportunity for subsequent detection and exploitation of common modularity.

To identify the orienting mechanism for a particular problem both the data structure specifications and the postcondition need to be carefully examined. What must be sought in these specifications is a component that requires the simplest repetitive action to establish the postcondition under the most restrictive conditions - this usually means changing only a single variable.

For example, if the postcondition specifies that a set of data is to be sorted, then the simplest iterative mechanism that can establish the postcondition is one which merely confirms that the data is already sorted. Obviously this mechanism can only establish the postcondition in a very restricted case. It does however provide the basis for the subsequent development of an insertion sort.

Some criteria that can assist in the identification of the orienting mechanism are:

- establishment of the postcondition by an iterative mechanism without the need to produce output has precedence over a mechanism that must produce output (e.g. text formatting a file of blanks)
- establishment of the postcondition by an iterative mechanism which produces output for only a single class has precedence over a mechanism that produces output for more than one class.
- establishment of the postcondition by an iterative mechanism without conditional testing has precedence over a mechanism that applies conditional testing (e.g. in a sequential file update under special conditions (an empty transactions file) it is possible to establish the postcondition by simply copying from the old master to the new master file. This avoids conditional testing whereas processing a set of transactions involves conditional testing to identify the category of the transaction).

These criteria are fairly widely applicable. However, in dealing with a particular problem the specifications and the model for the problem will play the most significant role in identifying what should be the orienting step.

It is useful to identify some of the roles that the orienting mechanism may assume in the overall development of the program. These roles can provide a basis for the formulation of a set of composition rules that may be used in development of the programs. Some of the more prominent roles of the orienting mechanism are:

- it may describe an iterative mechanism which can **confirm** the postcondition (e.g. sorting)
- it may describe an iterative mechanism which in turn can be iteratively applied to establish the postcondition.
- it may describe an initializing or finalizing step for a more general iterative mechanism.
- it may set up a configuration which can either subsequently be confirmed to establish the postcondition or which must be changed to establish the postcondition.
- it may provide one of a number of independent alternative means for establishing the postcondition.

Having identified the orienting mechanism in terms of the postcondition R_0 that it establishes there are two other tasks that must be undertaken. Firstly it is necessary to implement the orienting mechanism according to the specifications required by R_0. In taking this step it frequently found that the subgoal R_0 itself consists of a composite (compound) iterative mechanism. Should this be the case then development proceeds by first identifying the more primitive orienting mechanism R_{00} needed as part of the mechanism to establish the sub-goal R_0. This strategy for decomposition may be thought of as resembling the **in-order** traversal of a tree-structure.

The actual development to establish R_0 can proceed in a formal manner by first weakening the predicate R_0 to obtain a loop invariant and then constructing the loop accordingly. This implementation strategy should be used if a rigorous constructive proof is sought.

3.2. Generalization by Inductive Stepwise Refinement

With respect to the postcondition R the orienting mechanism can have two possible states of termination

(i) one where R is established

(ii) a second where R_0 is established but the more general R is not satisfied.

If the possibility for this second state of termination exists then a more general mechanism needs to be built into the program structure to accommodate it. To proceed with the development of this additional mechanism an analysis must first be made of what R_0 accomplishes when it terminates without having established R. This analysis of R_0 must be made in relation to what is required to establish R. The development step which must follow this analysis will represent a generalization of the existing mechanism. **It should be the most elementary mechanism (usually iterative) that can be added to the existing mechanism to establish the postcondition when the existing mechanism fails to establish the**

general postcondition R. To identify the additional mechanism the same criteria can be applied as were used to identify the orienting mechanism.

When this second generalizing step has been completed, it too, like the orienting mechanism, may have two possible states of termination:

(i) one where R is established

(ii) a second where R_1 is established (R_1 being more general than R_0) but where the more general R is not satisfied.

If the possibility for the second possible state of termination exists then a further generalization of the program will be needed to accommodate it. The strategy for the development of this generalization, and any other subsequent generalizations needed to establish the general postcondition R, is the same as that for the first generalization.

This suggests a way of systematically using the postcondition to guide the stepwise development of the program.

4. COMPOSITION RULES FOR GENERALIZATION

The commonly employed composition rules needed for inductive stepwise refinement follow directly from the roles that have been ascribed to the orienting mechanism. It can also happen that a mechanism existing at a later stage development fulfills a role that the orienting mechanism may assume. It follows that the composition rules (with respect to L1) that are outlined below may be applied not only to the orienting mechanism but also to constructs existing at other stages in the development.

In the rule definitions to be given, **I** will be used to denote an *initialization mechanism*, **F** a finalization mechanism, **B** a guard, and **S** a block of one or more statements. A loop **L** has the following definition:

$$L \overset{\Delta}{=} I;\ \text{do } B \rightarrow S \text{ od};\ F$$

Numerical suffixes are added to distinguish different components. With these conventions the most common composition rules are:

C1. Straight Sequential Composition

C1: L1; S2

C2. Iterative Sequential Composition

C2: L1; I2; do B2 → S2 od; F2

C3. Hierarchical Composition

C3: I2; do B2 → L1 od; F2

C4. Initialized Iterative Sequential Composition

C4: L1; I2; do B2 → S2; L1 od; F2

In applying these rules for generalization the conventions of structured programming are followed. The steps in program development by inductive stepwise refinement can be summarized as follows:

(i) Data analysis, precondition Q, and postcondition R specification

(ii) Identification of the orienting mechanism and its associated postcondition R_0 by use of the information in (i)

(iii) Development of the orienting mechanism to establish R_0. For more complex problems this may involve a recursive application of the method. When the point is reached where a recursive strategy is not required, the most efficacious way to proceed is by weakening the predicate R_0 to obtain a loop invariant P_0 and then developing the iterative mechanism accordingly using the strategy originally proposed by Dijkstra [1] and advocated by Gries [2]).

(iv) Application of inductive stepwise refinement. This involves analysis of the postcondition R_0 in relation to R in order to identify the most elementary mechanism (usually iterative) that can be added to the mechanism which is capable of establishing the postcondition R under restricted conditions. In general this mechanism will guarantee only the more restrictive postcondition R_1. Again the development of the mechanism to satisfy R_1 may involve a recursive application of the method.

(v) Inductive stepwise refinement is repeated by reapplication of step (iv) for the most recently established postcondition R_i until a mechanism has been developed that will establish the general postcondition R for the given precondition.

Sufficient background has now been presented to consider application of the method to a number of examples.

5. PROGRAM DEVELOPMENT EXAMPLES

To illustrate the development of programs by inductive stepwise refinement examples have been chosen which have been discussed elsewhere in the literature.

5.1. Text Formatting

The first problem we will consider was mentioned by Floyd in his 1978 Turing award lecture [9]. The problem may be stated as follows:

Read lines of text until either a completely blank line is found or an end-of-file is encountered. Eliminate redundant blanks between words. Print the text with a maximum of **limit** characters to a line without breaking words between lines.

The description can be clarified by adding that there should be no leading or trailing blanks on a line. The end-of-file requirement has been added to Floyd's original problem description. This in no way simplifies the problem.

Floyd makes the remark that "the problem is surprisingly hard to program in most programming languages". It should therefore provide something of a test for our proposed method of program development.

Data Analysis:

There are four definitions relevant to this problem, a raw text input line and a formatted text output line, and the corresponding text files. Syntax diagrams have been used for these definitions after a suggestion made by Barter [10]. The inverted **defined** symbol "$\overset{=}{\nabla}$" has been included to indicate definition. (see fig. 1)

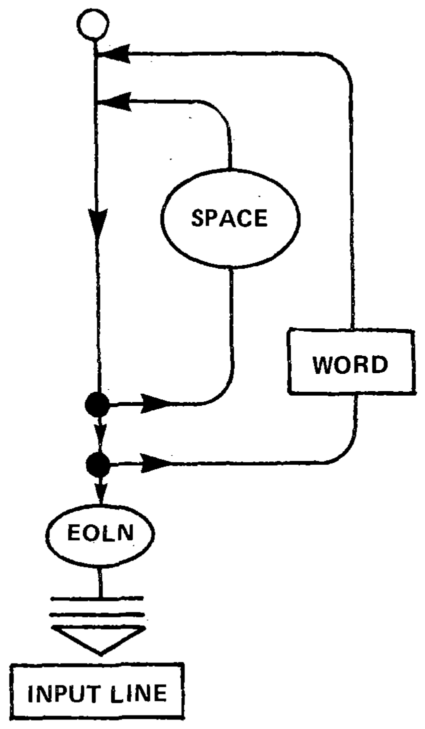
A **WORD** is defined as a consecutive sequence of one or more non-blank characters on a single line.

Precondition:

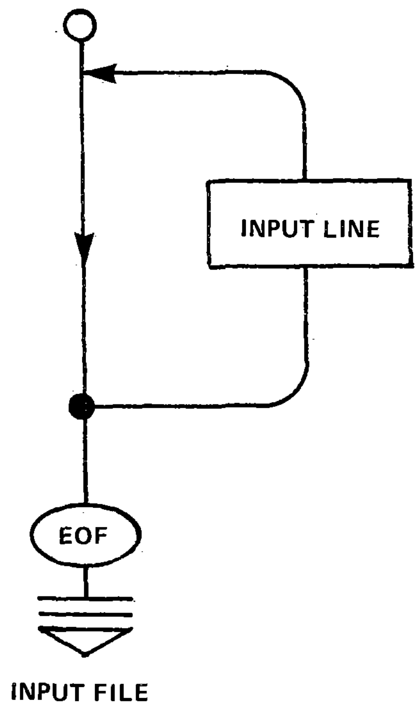
No single word is more than **limit** characters long where **limit** is the maximum length of the formatted line.

FIGURE 1

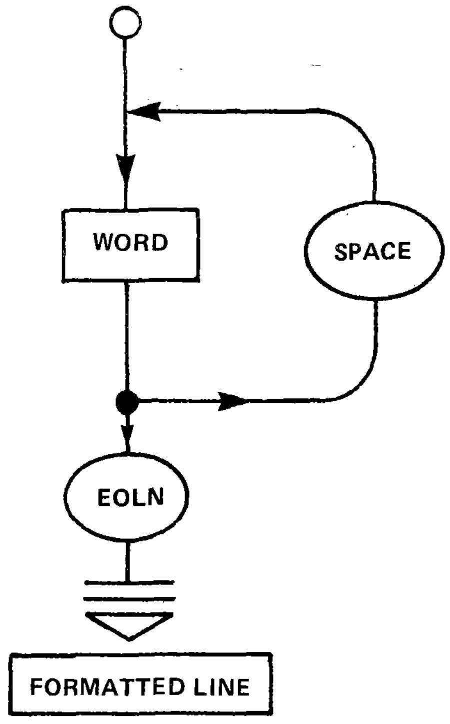
(a) INPUT LINE:



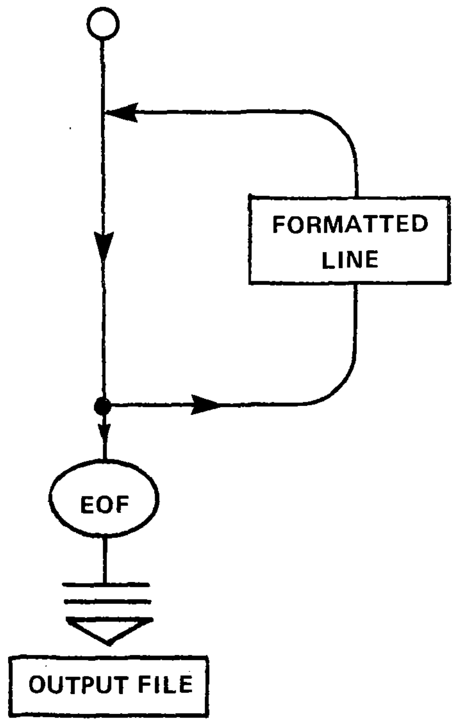
(b) INPUT FILE:



(c) FORMATTED LINE:



(d) OUTPUT FILE:



Postcondition:

An informal description of the postcondition is:

R: All lines of text have been read and formatted with only single blanks separating words and a maximum of **limit** characters on a line with no words broken and the maximum number of words on each formatted line and either a blank line or an end-of-file has been encountered.

R_0. Zero Words - Orienting Step:

The first step in the development is to identify the most elementary iterative mechanism that can establish the postcondition. The simplest way to establish the postcondition for a non-empty file would be to encounter and process a line of blanks. It requires no formatted output. In constructing a mechanism to establish the postcondition under these conditions it is necessary to take into account that this mechanism may not always be able to establish the postcondition. To read multiple spaces the procedure **getspaces** can be used (essentially Dijkstra's guarded commands augmented with Pascal I/O have been used throughout for implementations):

```
procedure getspaces (data:textfile; var j:boolean; N:boolean);
const SPACE = ' ';
var j, n:boolean;
     n := N;
     do j ≠ n →
          if data↑ = SPACE → get(data); j := eoln(data)
          [] data↑ ≠ SPACE → n := j
          fi
     od
end
```

This mechanism has two possible states of termination:

(i) End-of-line is true (i.e. j = N) and *R* is established

(ii) End-of-line is not true (i.e. j ≠ N) and *R* is not established.

The second possible state of termination will need to be accommodated in any general solution to the problem. Therefore we have:

ZERO WORDS:

```
const EOLN = true;
var data:textfile; j, N:boolean;
     N := EOLN; j := eoln(data);
     getspaces (data, j, N);
     if j ≠ N → "further processing"
     [] j = N → skip {R established}
     fi
     {R₀}
```

This mechanism will establish the postcondition *R* for all configurations of zero words. In the more general case it terminates with R_0 specifying that all leading blanks (if any) on the first line have been read.

R_1. One Word at Most - First Generalization:

Termination of the orienting mechanism with j ≠ N implies that there must be at least one word to be formatted and printed. The most elementary generalization that can establish *R* will therefore be one that handles the case of a single word. The associated postcondition is R_1. As part of this mechanism a procedure **getword** may be defined:

```
procedure getword (data: textfile; var word:charray; var w:integer; var j, N:boolean);
const SPACE = ' ';
var n:boolean;
    n := N; w := 0;
    do j ≠ n →
        if data↑ ≠ SPACE → word[w+1] := data↑; w := w+1; get(data); j := eoln(data)
        [] data↑ = SPACE → n := j
        fi
    od
end
```

To complete the task a procedure **writeword**(word, l) is needed. Its implementation is omitted.

The procedure **getword** has two possible states of termination:

(i) End-of-line true (i.e. j = N)

(ii) End-of-line not true (i.e. j ≠ N)

The second possible termination state signals that there are still characters on the line to be processed, the first of which must be a space. The procedure **getspaces** may therefore be applied. From the INPUT LINE specification it is apparent that the inclusion of **getspaces** handles the most general case of a single input word. Therefore we have:

ONE WORD AT MOST:

```
const EOLN = true; LIMIT = 60;
var data:textfile; j, N:boolean; s:integer; word:array[1..LIMIT] of char;
  N := EOLN; j := eoln(data);
  getspaces(data, j, N);
  if j ≠ N →
     getword(data, word, s, j, N);
     writeword (word, s);
     getspaces(data, j, N);
     if j ≠ N → "further processing"
     [] j = N → skip {R established if EOF after readln}
     fi;
     writeln
  [] j = N → skip
  fi
  {R₁}
```

This mechanism will establish the postcondition R for all configurations of at most a single input word. In the more general case it terminates with R_1 specifying that at most a single input word (possibly with leading and trailing blanks) has been read and formatted. The part of the mechanism in **bold** indicates what has been added to accommodate a single input word. Throughout bold type has been used to identify the most recent development step at each stage of development.

R_2. One Input Line at Most - Second Generalization:

If the "one-word" mechanism terminates without having reached the end of the input line (i.e. j ≠ N) it implies that there is **more than one** input word to be processed on the first input line. In providing a mechanism to establish R, this case will need to be accommodated.

The first question that may be asked at this point is whether the mechanism developed for a single word can be applied iteratively to handle more than one word on the first input line? The answer is no because it is necessary to ensure that the output line length LIMIT is not exceeded. Consideration of the FORMATTED LINE specifications indicates that all words on the formatted line, apart from the first, must be preceded by a single space. Taking these conditions into account and using a line of reasoning similar to that employed for the single-word case the **getline** procedure is proposed.

```
procedure getline (data:textfile; var j,N:boolean; var s:integer);
const LIMIT = 60; SPACE = ' ';
var n:boolean; w:integer;
    n := N;
    do j ≠ n →
        getword(data, word, w, j, N);
        if s + w + 1 ≤ LIMIT → write(SPACE); writeword(word, w); s := s + w + 1
        [] s + w + 1 > LIMIT → writeln; writeword(word, w); s := w
        fi;
        getspaces(data, j, N)
    od
end
```

After a single input line has been read and processed it is appropriate to do a **readln** to re-establish the end-of-line and end-of-file status. The general mechanism to handle at most one input line therefore becomes:

ONE INPUT LINE AT MOST:

```
const EOLN = true; EOF = true; LIMIT = 60;
var data:textfile; k,j,M,N:boolean; s:integer; word:array[1..LIMIT] of char;
    N := EOLN; M := EOF; j := eoln(data);
    getspaces(data, j, N);
    if j ≠ N →
        getword(data, word, s, j, N);
        writeword(word, s);
        getspaces(data, j, N);
        getline(data, j, N, s);
        readln(data); j := eoln(data); i := eof(data);
        if i ≠ M → "further processing"
        [] i = M → skip {R established by a writeln}
        fi;
        writeln
    [] j = N → skip
    fi
    {R2}
```

This mechanism will establish the postcondition for all configurations of at most a single input line {i.e. R_0, R_1, R_2}. In the more general case it terminates with R_2 specifying that at most a single input line has been read and one or more lines satisfying the FORMATTED LINE specifications have been produced. Notice that the **writeln** is still needed for this more general case.

R. One or More Input Lines - Third Generalization:

A further generalization of the mechanism described in the preceding section is needed to accommodate the case where more than one input line has to be processed.

A mechanism already exists for processing the first line of the input text. To handle subsequent input lines it may be expected that either this mechanism, or a slightly modified version could be applied iteratively.

Analyzing R_2 indicates that the most elementary iterative mechanism that can establish R after having read a single input line is:

```
getspaces (data, j, N)
```

This mechanism can terminate in one of two states:

(i) either with j = N and R established

(ii) or with j ≠ N and R not established.

In the second case the **getline** procedure can be applied. Therefore to process the second line the following mechanism may be used.

```
getspaces (data, j, N);
if j ≠ N → getline(data, j, N, s); readln(data); i := eof(data); j := eoln(data)
[] j = N → skip {R established by a writeln}
fi
```

This mechanism can be applied iteratively until either a blank line is read or until an end-of-file is reached. The complete mechanism therefore takes the form:

ONE OR MORE INPUT LINES:

```
const EOLN = true; EOF = true; LIMIT = 60;
var data:textfile; i, j, M, N:boolean; s:integer; word:array[1..LIMIT] of char;
    N := EOLN; M := EOF; j := eoln(data);
    getspaces(data, j, N);
    if j ≠ N →
        getword(data, word, s, j, N);
        writeword (word, s);
        getspaces(data, j, N);
        getline(data, j, N, s);
        readln(data); j := eoln(data); i := eof(data);
        m := M;
        do i ≠ m →
            getspaces(data, j, N);
            if j ≠ N → getline(data, j, N, s); readln(data); j := eoln(data); i := eof(data)
            [] j = N → m := i
            fi
        od;
        writeln
    [] j = N → skip
    fi
{ R }
```

The most recent generalization leads to a mechanism that will establish the postcondition *R* in the general case for one or more input lines. The logical structure for dealing with the first line and the loop body for dealing with subsequent lines is essentially the same. The only differences are that the first word on subsequent lines is no longer special (hence the steps for it are deleted from the loop body implementation) and there must be provision to force termination of the loop (by m := i) if a blank line is encountered(a discipline for forced termination is described elsewhere [11]).

Recapping the development, the orienting mechanism and three stages of refinement are:

(i) Zero words **AND** EOLN ⇒ R

(ii) One input word **AND** EOLN **AND** EOF ⇒ R

(iii) One input line **AND** EOF ⇒ R

(iv) Many input lines **AND** EOF **OR** BLANKLINE ⇒ R

These steps represent the special conditions under which successively more general mechanisms are able to establish the postcondition. The logical structure reflected in the development is also clearly visible in the final program implementation. More will be said about the structure of this algorithm after the next problem has been considered.

5.2. Telegrams Analysis Problem

The telegrams analysis problem has been used by several authors [3,10,12,13] in the discussion of program development methods. For this problem a program is required to process a stream of telegrams each terminated by the word "ZZZZ". The data is stored in blocks of size M. Words are separated by one or more spaces and they do not extend across block boundaries. It is required to count the words in each telegram (excluding "ZZZZ") and print each telegram with a single space between words and no leading or trailing blanks.†

Data Analysis:

There are seven main input and output data definitions relevant for the specification of this problem. They are included below:

Precondition:

At least the NULL TELEGRAM is present in the INPUTFILE.

Postcondition:

An informal description of the postcondition is

> R: The INPUT FILE has been read including the terminating NULL TELEGRAM and a corresponding OUTPUT TELEGRAM FILE has been produced according to the data specification supplied.

R_0. **Null Telegram - Orienting Step:**

As with the previous problem the first step in the development is to *identify the most elementary* iterative mechanism that can establish the postcondition. Reference to the data specifications suggests that the simplest way to establish the postcondition would be to encounter and process the NULL TELEGRAM. No output is required to establish the postcondition R in these circumstances. A procedure **fillbuffer** that reads **data** from a file and fills a buffer **buff** with M characters, and a second procedure **getspaces** that consumes the leading spaces in the buffer are assumed. The consecutive application of **fillbuffer** and **getspaces** cannot guarantee to consume **all** leading spaces preceding the first word (postcondition R_{00}). It is therefore necessary to first complete a mechanism to establish R_{00} before considering in detail the mechanism for R_0. The consecutive application of **fillbuffer** and **getspaces** has two possible states of termination as indicated below:

```
const M = buffsize;
var data: textfile; buff: array[1..M] of char; j : integer;
   fillbuffer(data, buff, M); j := 0;
   getspaces(buff, j, M);
   if j = M → "further processing"
   [] j ≠ M → skip {$R_{00}$ established}
   fi
```

To guarantee R_{00}, **fillbuffer** and **getspaces** need to be iteratively applied. With this refinement the procedure **prefirstword** can be defined:

```
procedure prefirstword(var data: textfile; var buff: array[1..M] of char; var j:integer);
   repeat
      fillbuffer(data, buff, M); j := 0;
      getspaces(buff, j, M)
   until j ≠ M
end
{$R_{00}$}
```

The procedure **prefirstword** terminates with j ≠ M implying that a word in **buff** is available for access. Assuming a procedure **getword** that moves the next available word from **buff** to the array **word**, the mechanism that can establish R for the NULL TELEGRAM may be defined.

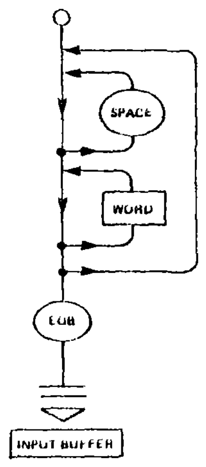
NULL TELEGRAM †

† In Henderson's original specification [12] there were also STOP words which were not to be counted and oversize words which were to be specially noted. These considerations do not influence the structure of the problem and so have been omitted from the present discussion.

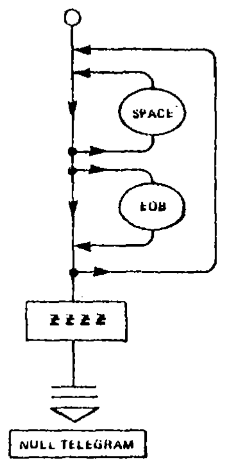
† The variable "j" is passed to **getword** as a value parameter and therefore is not changed by **getword**.

FIGURE 2

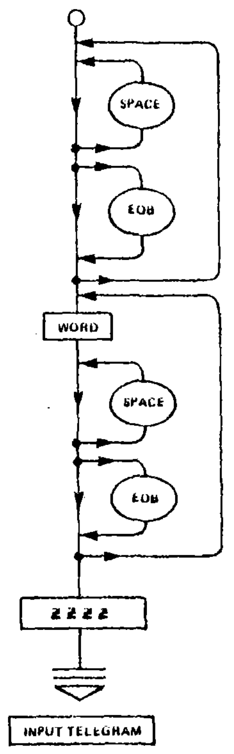
(a) INPUT BUFFER:



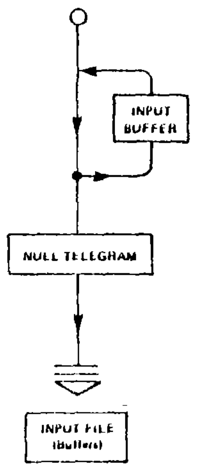
(b) NULL TELEGRAM:



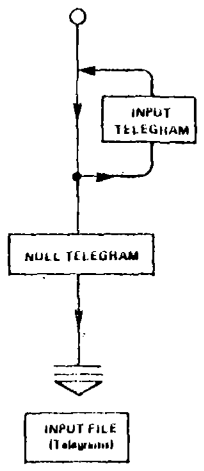
(c) INPUT TELEGRAM (Real).



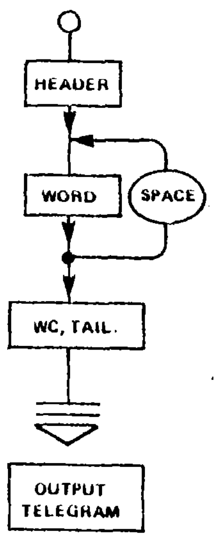
(d) INPUT FILE (Buffers);



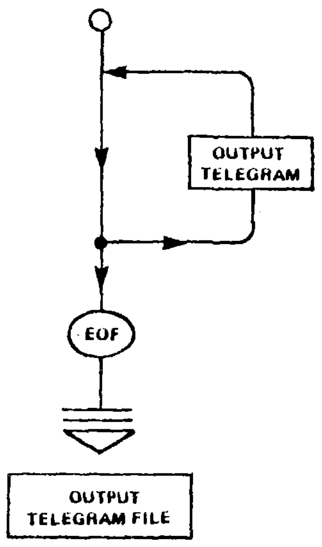
(e) INPUT FILE (Telegrams):



(f) OUTPUT TELEGRAM:



(g) OUTPUT TELEGRAM FILE:



```
const M = buffsize;
var data:textfile; var buff, word: array [1..M] of char; j, w:integer
    prefirstword(data, buff, j, M);
    getword(buff, word, w, j, M);
    if word ≠ 'ZZZZ' → "further processing"
    [] word = 'ZZZZ' → skip {R established}
    fi
    {R_0}
```

where **w** is the length of the word most recently consumed. It is assumed that the language can string-match the **word** array with 'ZZZZ'. This mechanism will establish R for all configurations of the NULL TELEGRAM. In the more general case it terminates with R_0 specifying only that the first word in the first telegram has been read and stored in **word**.

R_1. One Telegram with One Word - First Generalization:

There are two possible states of termination for the NULL TELEGRAM mechanism. Termination with **word ≠ 'ZZZZ'** implies that there is at least one telegram with one countable word to be processed. It also implies that at least one more word will need to be processed to complete the current telegram.

The word that has just been read can be written by the following mechanism:

```
writeheader; writeword (word, w); wc := 1; j := j+w;
if j ≠ M → "further processing - current buffer"
[] j = M → "further processing - next buffer"
fi
```

Development can continue by adding an iterative mechanism to find the next word (if any) in the current buffer or beyond (postcondition R_1). The stages in this phase of the development follow closely those employed in dealing with the NULL TELEGRAM. The outcome of this phase of development is the definition of a procedure **preword**:

```
procedure preword(var data: textfile; var buff: array[1..M] of char; var j, M: integer);
    getspaces(buff, j, M);
    do j = M →
        filbuffer(data, buff, M); j := 0;
        getspaces(buff, j, M)
    od
end
```

The mechanism for one telegram with one countable word can then be defined:

ONE TELEGRAM WITH ONE WORD:

```
const M = buffsize; SPACE = ' ';
var data: textfile; buff, word: array[1..M] of char; j, w,wc: integer;
    prefirstword(data, buff, j, M);
    getword(buff, word, w, j, M);
    if word ≠ 'ZZZZ' →
        writeheader; writeword(word, w); wc := 1; j := j+w;
        if j ≠ M →
            preword(data, buff, j, M);
            getword(buff, word, w, j, M);
            if word ≠ 'ZZZZ' → "further processing - current telegram"
            [] word = 'ZZZZ' → skip {"finished-current telegram"}
            fi
        [] j = M → "further processing-next buffer"
        fi;
        write (wc); writetail
    [] word = 'ZZZZ' → skip
    fi
    { $R_1$}
```

where **wc** is the word count for the current telegram.

R_2. One Telegram with One or More Words - Second Generalization

The most elementary generalization that can be applied to the mechanism in the preceding section is one that allows it to handle a single telegram with one or more words. For this purpose the mechanism in the segment guarded by **if j ≠ M → ...** can be iteratively applied until either the current buffer is exhausted (i.e. j = M) or an end-of-telegram word "ZZZZ" is encountered (postcondition R_{20}).

ONE TELEGRAM, ONE BUFFER:

```
const M = buffsize; SPACE = ' ';
    var data:textfile; buff,word:array[1..M] of char; j,w,wc,m,M:integer;
        prefirstword(data,buff,j,M);
        getword(buff,word,w,j,M);
        if word ≠ 'ZZZZ' →
            writeheader; writeword(word,w); wc := 1, j := j+w;
            m := M;
            do j ≠ m →
                preword(data,buff,j,M);
                getword(buff,word,w,j,M);
                if word ≠ 'ZZZZ' → write(SPACE);writeword(word,w);
                        wc := wc+1, j := j+w
                [] word = 'ZZZZ' → m := j
                fi
            od;
            if j = M → "further processing-current telegram"
            [] j ≠ M → skip {$R_{20}$ established}
            fi;
            write(wc); writetail;
            "further processing - to establish $R_2$"
        [] word = 'ZZZZ' → skip
        fi
```

Termination of the most recently added loop with the condition "j = M" implies that the single telegram extends across more than one buffer. The next most elementary generalization will therefore be one that accommodates this case (postcondition R_{21}). Again this is a subgoal of R_2.

The most recent refinement has provided a mechanism to handle all the words of a single telegram in a single buffer. This mechanism can be iteratively applied to handle more than one buffer. Making this refinement gives:

ONE TELEGRAM, MORE THAN ONE BUFFER:

```
const M = buffsize; SPACE = ' ';
var data:textfile; buff,word:array[1..M] of char; j,w,wc,m,M:integer;
    prefirstword(data,word,w,j,M);
    if word ≠ 'ZZZZ' →
        writeheader;writeword(word,w); wc := 1; j := j + w;
        repeat
            m := M;
            do j ≠ m →
                preword(data,buff,j,M);
                getword(buff,word,w,M);
                if word ≠ 'ZZZZ' → write(SPACE);writeword(word,w);
                        wc := wc+1; j := j+w
                [] word = 'ZZZZ' → m := j
                fi
            od;
            if j = M → fillbuffer(data,buff,M); j := 0
            [] j ≠ M → skip
            fi
        until j = m;
        write(wc); writetail; j := j+w;
        {$R_{21}$ established}
        "further processing to establish $R_2$"
    [] word = 'ZZZZ'
    fi
```

A mechanism has now been established which will read, reformat, and write out a single telegram in the general case. However, to establish the postcondition for a single real telegram it is necessary to subsequently detect a NULL TELEGRAM (see specification for INPUT FILE (Telegrams)).

The most elementary iterative mechanism that can be applied after termination of the mechanism for processing a single telegram is **getspaces**. We will not pursue this refinement further other than to comment that it follows closely the development for R_1. With this stage of development complete, the following mechanism is obtained.

ONE TELEGRAM WITH ONE OR MORE WORDS:

```
const M = buffsize; SPACE = ' ';
var data:textfile; buff,word:array[1..M] of char; j,w,wc,m,M:integer;
    prefirstword(data,buff,j,M);
    getword(buff,word,w,j,M);
    if word ≠ 'ZZZZ' →
        writeheader;writeword(word,w); wc := 1; j := j + w;
        repeat
            m := M;
            do j ≠ m →
                preword(data,buff,j,M);
                getword(buff,word,w,M);
                if word ≠ 'ZZZZ' → write(SPACE); writeword(word,w);
                        wc := wc + 1; j := j + w
                [] word = 'ZZZZ' → m := j
                fi
            od;
            if j = M → fillbuffer(data,buff,M; j := 0
            [] j ≠ M → skip
            fi
        until j = m;
        write(wc); writetail; j := j + w;
        preword(data,buff,j,M);
        getword(buff,word,w,M);
        if word ≠ 'ZZZZ' → "further processing-next telegram"
        [] word = 'ZZZZ' → skip {R established}
        fi
    [] word = 'ZZZZ' → skip
    fi
    { $R_2$}
```

The most recent generalization leads to a mechanism that will establish the postcondition in the general case for at most a single real telegram.

R. More than One Telegram - Third Generalization

If the mechanism developed for a single real telegram terminates with "word ≠ 'ZZZZ' " it implies that there is at least one more telegram to be processed. A mechanism has already been developed for a single telegram and so it is appropriate to consider whether a segment of this mechanism can be iteratively applied to handle more than one telegram. In pursuing this next generalization it should be noted that the steps

```
.
.
.
getword(buff,word,w,M);
if word ≠ 'ZZZZ' → "further processing - next telegram"
[] word = 'ZZZZ' → skip
fi
```

at the end of the mechanism duplicate the same structure at the beginning of the program text. Taking this into account the mechanism to handle more than one telegram takes the form:

MORE THAN ONE TELEGRAM:

```
const M = buffsize; SPACE = ' ';
var data:textfile; buff,word:Array[1..M] of char; j,w,wc,m,M:integer;
   prefirstword(data,buff,j,M);
   repeat
      getword(buff,word,w,j,M);
      if word ≠ 'ZZZZ' →
         writeheader;writeword(word,w); wc := 1; j := j+w;
         repeat
            m := M;
            do j ≠ m →
               preword(data,buff,j,M);
               getword(buff,word,w,M);
               if word ≠ 'ZZZZ' → write(SPACE); writeword(word,w); wc := wc+1; j := j+w
               [] word = 'ZZZZ' → m := j
               fi
            od;
            if j = M → fillbuffer(data,buff,M); j := 0;
            [] j ≠ M → skip
            fi
         until j = m;
         write(wc); writetail; j := j+w;
         preword(data,buff,j,M)
      [] word = 'ZZZZ' → m := j
      fi
   until j = m
   {R}
```

The most recent generalization yields a mechanism that will establish the postcondition *R* in the general case for a file of one or more telegrams. The development strategy and the resulting control structure employed for this example differs from that of the previous example. In the previous example the loops employed in the development are over defined input structures with no development steps being over defined output structures. Using Jackson's metaphor "inversion" has been applied with respect to the output. In contrast to this, in the telegrams' problem, loops employed in the development have been over **both** defined input and output structures (i.e. there are loops over the input buffer and over telegrams). Several consequences follow from adopting the development strategy employed for the telegrams' analysis problem. Such programs usually contain additional loops. This apparent increase in complexity is offset by there being no bias against making either input or output-dependent program changes (a problem that can arise with programs that have been inverted either with respect to their input or their output). In short this development strategy represents a flexible and robust alternative to program inversion [3] for the resolution of boundary structure clashes.

5.3. Sequential File Update

The variation of the sequential file update problem which allows for the possibility of more than one transaction on a **single** key is often regarded as a non-trivial problem [1]. The problem involves a specialized file merge.

Data Analysis:

Three files are involved, a file of transactions records **trans** which specifies amendments to be made to an old master file **old** to produce a new master file **new**. Successive records in the **old** and **new** files have monotonically increasing values of their keys. Each transaction record includes a sub-field which identifies whether the transaction for that particular key is to be an insert, update or delete. Records in the old master file not involved in transactions should be copied to the new master file in a way that maintains the order of the new file.

Keys in the new master file should be unique. Relevant specifications are given in fig. 3:

Precondition:

There may be zero or more transaction records and zero or more old master file records.

Postcondition:

The postcondition for this problem can be stated informally as follows:

> R: All transactions and all records in the old master
> file have been read and processed to produce the new
> ordered master file.

R_0. **Zero Transactions - Orienting Step:**

The first development step involves the identification and implementation of the most elementary iterative mechanism that can establish the postcondition. The precondition suggests that either of the files might be empty. To decide upon the orienting mechanism a choice must be made between a mechanism that establishes the postcondition for *zero transactions and one* that establishes the postcondition for zero old master records. The data analysis indicates that the zero-transactions case is simpler as it requires no identification of the type of the transaction. The corresponding postcondition R_0 characterizes the situation where the general postcondition R is not established for the precondition of an empty transaction file. To establish the postcondition in the case of zero transactions in general will require an iterative mechanism to copy records from the old master file to the new master file. For this purpose the procedure **copyold** may be used:

```
procedure copyold(var old,new:master; j,N:boolean);
   do j ≠ N →
      new↑ := old↑; put(new);
      get(old); j := eof(old)
   od
end;
```

Using this procedure the orienting mechanism may take the form

ZERO TRANSACTIONS:

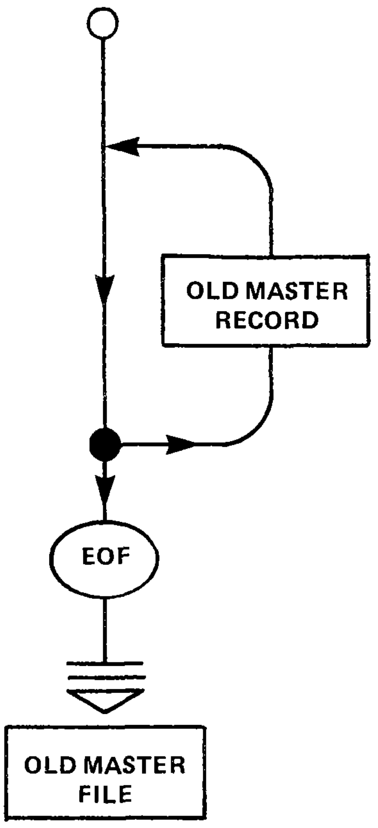
```
const EOF = true;
var k,j,M,N:boolean;
M := EOF; N := EOF; i := eof(trans); j := eof(old);
if i ≠ M → "further processing"
[] i = M → skip {R established by copy below}
fi;
copyold(old,new,j,N)
{ R0}
```

R_1. **One Transaction at Most - First Generalization:**

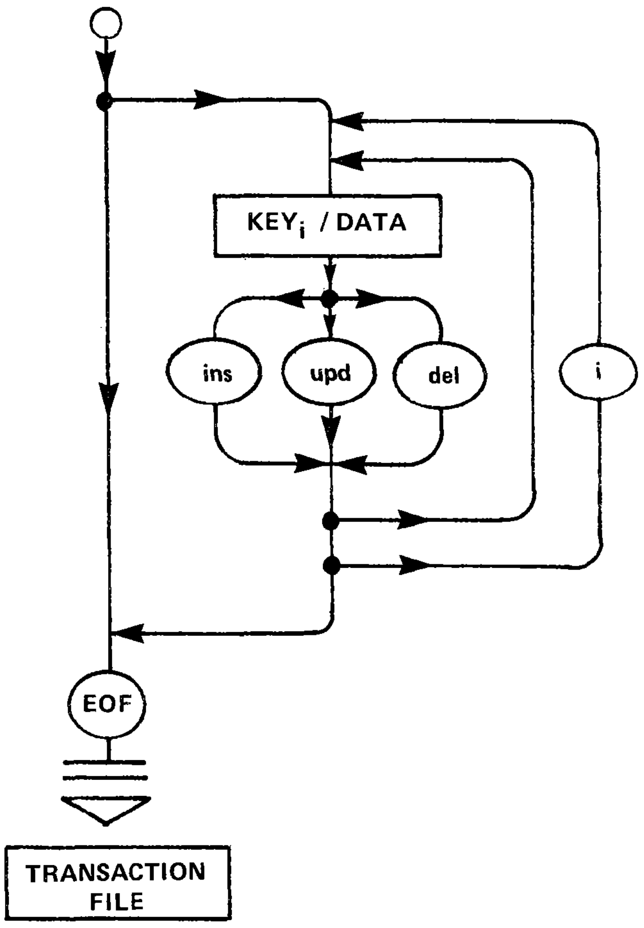
It is necessary, in the general case, to handle the precondition where the transaction file is not empty (i.e. i ≠ M). This condition implies that one or more transactions must be processed. Consider first a single transaction. The corresponding postcondition R_1 characterizes the situation where the general postcondition R is not established for a single transaction. To *establish the postcondition R for a* **single** *transaction an additional iterative mechanism is* required. It must search the **old** master file to locate the"transaction position". In the process records in the old file need to be copied to the new master file. Once the transaction position has been located the transaction may be performed. Treating these two components separately we get:

FIGURE 3

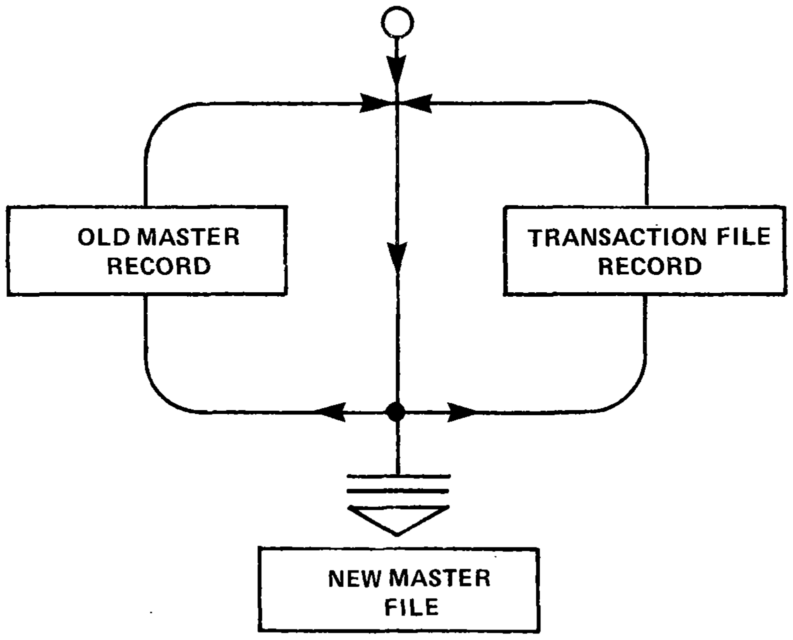
(a) OLD MASTER FILE:



(b) TRANSACTION FILE:



(c) NEW MASTER FILE:



Search and Copy:

```
procedure searchcopy(old,new:master; trans:transaction; var j, N:boolean);
var n:boolean;
   n := N;
   do j ≠ n →
      if old↑.key < trans↑.key → new↑ := old↑; put(new); get(old); j := eof(old)
      [] old↑.key ≥ trans↑.key → n := j
      fi
   od
end
```

Transaction:

The transaction mechanism can be appended to the search mechanism as a finalization mechanism. Three possible actions insert(ins), delete(del), and update(upd) must be accommodated. The transaction can therefore take the following general structure:

```
if trans↑.action = ins → firstinsert(...,valid)
[] trans↑.action = upd → firstupdate(...,valid)
[] trans↑.action = del → firstdelete(...,valid)
fi;
get(trans); i := eof(trans);
if valid → put(new)
[] ¬valid → skip
fi
```

Details are needed for each of the three possible transactions.

firstinsert:

The sequential search terminates in one of two states.

(i) with eof(old) true (i.e. j = N)

(ii) with eof(old) false (i.e. j ≠ N)

If the old master file is not exhausted then the condition **trans↑.key** < **old↑.key** must also hold for it to be possible to make a valid insertion. With both these conditions true the following actions are appropriate.

```
new↑.key := trans↑.key; new↑.data := trans↑.data; valid := true;
```

The same steps are appropriate if the old master file is exhausted. However, if the condition trans↑.key ≥ old↑.key applies an invalid transaction is being attempted. To ensure that the corresponding old master record is not lost due to an invalid transaction the following steps are needed.

```
new↑ := old↑; get(old); j := eof(old); valid := true;
writeln('Insertion error', trans↑.key)
```

Details of **firstinsert** are therefore:

```
procedure firstinsert(old,new:master; trans:transaction; var j,N, valid:boolean);
   if j ≠ N →
      if trans↑.key < old↑.key → new↑.key := trans↑.key; new↑.data := trans↑.data
      [] trans↑.key ≥ old↑.key → new↑ := old↑; get(old); j := eof(old);
               writeln ('Insertion error', trans↑.key)
      fi
   [] j = N → new↑.key := trans↑.key; new↑.data := trans↑.data
   fi;
   valid := true
end
```

The structure of **firstupdate** and **firstdelete** follow a similar pattern, except that **valid** will be false after a deletion. A mechanism that will establish the postcondition R in the general case for a *single* transaction may take the form:

ONE TRANSACTION:

```
var i,j,M,N:boolean;
    M := true; N := true; i := eof(trans); j := eof(old);
    if i ≠ M →
        searchcopy(old,new,trans,j,N);
        if trans↑.action = ins → firstinsert(old,new,trans,j,N,valid)
        [] trans↑.action = upd → firstupdate(old,new,trans,j,N,valid)
        [] trans↑.action = del → firstdelete(old,new,trans,j,N,valid)
        fi;
        get(trans); i := eof(trans);
        if i ≠ M → "further processing - same key"
        [] i = M → skip {R established by put, and copy}
        fi;
        if valid → put(new)
        [] ¬valid → skip
        fi
    [] i = M → skip
    fi;
    copyold(old,new,j,N)
    {R1}
```

R_2. **More than One transaction for single key - Second Generalization**

To accommodate more than one transaction there are two possible generalizations to consider:

(i) a number of transactions for a single key

(ii) transactions for more than one key

The first of these cases is more elementary because it does not involve a change in the key variable. It is therefore the next generalization step.

In the first generalization a mechanism was developed to handle the first transaction for a given key. It is therefore necessary to decide whether

(a) it can be applied iteratively to handle many transactions for a single key

(b) or whether it may serve as the initializing step for handling many transactions for a single key.

Investigation reveals that situation (b) applies because no sequential search of the old master file is required. The validity of the next transaction for the same key will depend only on the most recent valid transaction and not on whether the old master file is exhausted.

The following rules summarize valid(✓) and invalid(X) transactions.

Transaction Currently	Possible transaction		
in Buffer	Insert	Delete	Update
valid	×	✓	✓
not valid	✓	×	×

Subsequent insertions for a particular key after the first transaction for that key may be handled by the following procedure:

```
procedure insert(new:master; trans:transaction; var valid:boolean);
    if valid → writeln ('Insertion error', trans↑.key)
    [] ¬valid → new↑.key := trans↑.key; new↑.data := trans↑.data; valid := true
    fi
end
```

The **delete** and **update** procedures are similar to **insert**. They can be derived from the table of rules. The complete mechanism to establish R for a single key may therefore take the form:

MORE THAN ONE TRANSACTION FOR A SINGLE KEY:

```
var i,j,M,N:boolean; ckey:name;
M := true; N := true; i := eof(trans); j := eof(old);
if i ≠ M →
    searchcopy(old,new,trans,j,N);
    if trans↑.action = ins → firstinsert(old,new,trans,j,N,valid)
    [] trans↑.action = upd → first update(old,new,trans,j,N,valid)
    [] trans↑.action = del → firstdelete(old,new,trans,j,N,valid)
    fi;
    get(trans); i := eof(true);
    m := M; ckey := new↑.key;
    do i ≠ m →
        if trans↑.key = ckey →
            if trans↑.action = ins → insert(new,trans,valid)
            [] trans↑.action = upd → update(new,trans,valid)
            [] trans↑.action = del → delete(new,trans,valid)
            fi;
            get(trans); i := eof(trans)
        [] trans↑.key ≠ ckey → m := i
        fi
    od;
    if valid → put(new)
    [] ¬valid → skip
    fi;
    if i ≠ M → "further processing-other keys"
    [] i = M → skip {R established by copyold}
    fi
[] i = M → skip
fi;
copyold(old,new,j,N)
{R₂}
```

R_3. Many Transactions for One or More keys - Third Generalization

The mechanism for more than one transaction for a single key can terminate in a state where R has not been established (i.e. i ≠ M). This implies that more than one key must be accommodated in the general case. In the preceding section a mechanism was developed which would handle a *single* key in the general case. Examination of this mechanism reveals that it can be applied iteratively to handle more than one key. Making this refinement gives:

MANY TRANSACTIONS FOR ONE OR MORE KEYS

```
var i,j,M,N:boolean; ckey:name;
M := true; N := true; i := eof(trans); j := eof(old);
do i ≠ M →
    searchcopy(old,new,trans,j,N);
    if trans↑.action = ins → firstinsert(old,new,trans,j,N,valid)
    [] trans↑.action = upd → firstupdate(old,new,trans,j,N,valid)
    [] trans↑.action = del → firstdelete(old,new,trans,j,N,valid)
    fi;
    get(trans); i := eof(trans);
    m := M; ckey := new↑.key;
    do i ≠ m →
        if trans↑.key = ckey →
            if trans↑.action = ins → insert(new,trans,valid)
            [] trans↑.action = upd → update(new,trans,valid)
            [] trans↑.action = del → delete(new,trans,valid)
            fi;
            get(trans); i := eof(trans)
        [] trans↑.key ≠ ckey → m := i
        fi
    od;
    if valid → put(new)
    [] ¬valid → skip
    fi;
od;
copyold(old,new,j,N)
{R}
```

Several comments about this implementation are in order. The logical structure reflected in the development of this algorithm is clearly visible in the final implementation. With successive generalizations the guards for a previous development step do not change. What may change is whether the guard protects a mechanism that is applied iteratively or a mechanism that is part of an alternative construct. This is important for constructive proof development. The choice of orienting mechanism for this problem may have seemed unusual. If instead, a single transaction for a single key had been chosen as the focus the same algorithm structure would have resulted, only the original development step would have been a compound and more complex step. Therefore while identifying the orienting mechanism is important, other choices for the original development step can be acceptable provided that they are able to establish the postcondition at least under restricted conditions.

5.4. Fermat's Factoring Algorithm

In the previous three problems the structure of the data had a strong influence on the development and structure of the final program. With Fermat's factoring problem [14] it is not possible to rely on the structure of the data for guidance in the development of the program.

Fermat provided a set of relations that may be used to find the largest factor **u** of an odd integer **n** that is less than or equal to the square root of n [15]. For such an n the following relations hold:

$$n = u \times v \quad \textit{where } u \le v \tag{1}$$

$$n = x^2 - y^2 \quad \textit{where } 0 \le y < x \le n \tag{2}$$

$$x = (u + v)\,\mathbf{div2} \tag{3}$$

$$y = (v - u)\,\mathbf{div2} \tag{4}$$

$$u = x - y \tag{5}$$

$$v = x + y \tag{6}$$

To find the largest factor of n less than or equal to $\sqrt{n}$ it is necessary to find values of x, y and u that satisfy relations (2) through (5). More formally,

Postcondition R:

$$\mathbf{E}(x,y:0 \leq y \leq x \leq \lfloor\sqrt{n}\rfloor : n = x^2 - y^2 \wedge u = x - y \wedge v = x + y \wedge n = u \times v)$$

R_0. Square Root - Orienting Step

In solving the factoring problem it is first necessary to find values of **x** and **y** that satisfy:

$$n = x^2 - y^2 \qquad (2)$$

From these values, **u** and **v** values may be obtained using (5) and (6).

There are two variables in the relation (2). In searching for an orienting mechanism it is necessary to consider whether or not it is possible to establish *R* by changing a *single* variable. The only way to do this is to hold y at zero and try to find an x to satisfy (2) (i.e. if n is a perfect square). The obvious way to do this is to generate successive squares of x. Summing the odd sequence is sufficient for this purpose. Defining the odd sequence relations

$$xx = 2x + 1$$

$$yy = 2y + 1$$

yields

$$u = (xx - yy) div 2$$

$$v = (xx + yy) div 2$$

The orienting mechanism can therefore be simply implemented as follows.

SQUARE ROOT $\{n = x_0^2\}$

```
var r,xx,yy:integer;
    r := 0; xx := 1; yy := 1;
    do r + xx ≤ n →
        r := r + xx;
        xx := xx + 2
    od;
    if r < n → "further processing"
    [] r = n → skip {R established by assignment below}
    fi;
    u = (xx - yy) div 2
    {$R_0$}
```

This mechanism can only establish *R* if n is a perfect square. It therefore has two possible states of termination.

R_1. Composite Relation $\{n = x_0^2 - y_0^2\}$ - First Generalization

The situation where the orienting mechanism terminates with r < n will need to be accommodated in a general solution to the factoring problem. This implies that a non-zero value of y is needed to satisfy the relation (2) - that is the second variable must be introduced into the computation to establish R.

To do this successive squares for y can be generated again using the odd sequence sum.

COMPOSITE RELATION

```
var r,xx,yy:integer;
    r := 0; xx := 1;  yy := 1;
    do r+xx ≤ n →
        r := r+xx;
        xx := xx+2
    od;
    if r < n →
        r := r+xx;
        xx := xx+2;
        repeat
            r := r−yy;
            yy := yy+2
        until r ≤ n;
        if r < n → "further processing"
        [] r = n → skip {R established by assignment below}
        fi
    [] r = n → skip
    fi;
    u := (xx−yy) div 2
    {R_1}
```

Again this mechanism cannot guarantee to establish R in the general case. It will only establish R if n = $(\lfloor\sqrt{n}\rfloor + 1)^2 - y^2$.

R_2. Composite Relation $\{n = x_i^2 - y_j^2\}$ - Second Generalization

If the mechanism developed in the previous refinement step terminates with r < n further processing is required. Examination of the R_1 mechanism reveals that there is already a mechanism that can be applied when the condition r < n exists. The mechanism created in the most recent development step can therefore be iteratively applied to establish R.

COMPOSITE RELATION

```
var r,xx,yy:integer;
    r := 0; xx := 1; yy := 1;
    do r+xx ≤ n →
        r := r+xx;
        xx := xx+2
    od;
    do r< n →
        r := r+xx;
        xx := xx+2;
        repeat
            r := r−yy;
            yy := yy+2
        until r ≤ n
    od;
    u := (xx−yy) div 2
    {R}
```

It is interesting to compare this solution with an alternative implementation given below:

```
r := 0; xx := 1; yy := 1;
do r ≠ n →
    if r < n → r := r+xx; xx := xx+2
    [] r > n → r := r−yy; yy := yy+2
    fi
od;
u := (xx−yy) div 2
```

What a comparison between the two solutions brings out very clearly is the influence of program control structure on efficiency. With the second implementation **two** tests $r \neq n$ and either $r < n$ or $r > n$ are made with each iteration. The structure of the first implementation avoids this. The first solution recognizes two other things that are ignored by second solution. Firstly that x must initially be increased until it at least reaches a value equal to the integer square root of n. Only then is it necessary to give consideration to the introduction of y into the computation. Secondly after the square root magnitude for x is reached subsequent unit increases in x will almost invariably be accompanied by multiple increments in y. Recognition of these factors ensures that the first solution is the more efficient solution. This example serves to underline that textual simplicity should not be confused with mechanistic simplicity. It is suggested that the program structure of the first solution matches the transformations that are applied to the data. This is not the case with the second implmentation. The first solution may be described as a fully **iteratively resolved** solution to the problem in the same way as the solutions to earlier problems described have been iteratively resolved. What is interesting to observe here is that the method of inductive stepwise refinement can provide iteratively resolved solutions to problems where there are no data structure queues to guide the development.

6. EVALUATION

Broadly there are two main schools of program development, those dependent on top-down functional decomposition and those based on data structure dependence [16]. The method of program development described here is goal-directed. At the same time it attempts to constructively exploit functional dependencies that are either explicit or implicit in data and data relationships. As such the method has a leg in both of the main schools of program development.

For all the examples presented the steps in the development have not been formal. Instead we have relied heavily on the choice of the stages of development rather than formal proofs to provide a correctness argument for the final program. A structure has however been preserved in the development and the implementation which makes it straightforward to attach constructive formal proofs should that be necessary.

Programs developed by inductive stepwise refinement exhibit, in Jackson's terms [3], a structure consistent with structure in the data imposed either by the preconditions or the postconditions. Structure at the module level is constructive in that modules designed at one level of refinement frequently find direct application at subsequent higher levels of refinement. While the method of development proposed exploits structural dependencies in the data it does not rely on them. It therefore has application beyond the domain of serial data processing including instances where recursion may be relevant (e.g. in the solution of the eight-queens problem [17] the initial step focusses on placing a single queen in the general case on the board - the incorporation of recursion follows directly).

To try to assess the emphasis that the method places on both design and program efficiency it is useful to refer to a principle put forward by Zipf [18]. In Zipf's terms the constructive principle attempts to ensure that progress towards the final solution is made by, at each stage, always applying the principle of least effort. Resulting programs appear to achieve their goal by the least mechanistic effort for a given algorithmic strategy. This should however not be interpreted as suggesting that the method will necessarily always produce the most efficient program although this is frequently the case.

One of the best measures of how systematic and effective a given method of program development is, is whether consistent designs are obtained when it is applied by different people [19]. Limited experience suggests that consistent designs are usually obtained with the proposed method.

In trying to assess the weaknesses of the method probably the most crucial one is that the set of rules for deciding whether one mechanism is more elementary than another is not comprehensive enough. This problem can usually be partly alleviated by detailed accurate data analysis and modelling. In some problem domains, different or additional rules would obviously be needed to guide the development process. This deficiency does not necessarily impede the application of the method but rather it may influence the structure of the result. The more powerful and comprehensive the rule set available, the better should be the structural integrity of the resulting programs.

7. CONCLUSIONS

Methods for program development, including the present one, are only useful while they are applied with a critical eye. Methods and methodologies by their very nature are always developed within a limited context and with limited experience and we must always be wary about the thought of their mechanical application.

Much of what has been discussed in this paper is concerned with program quality. The overriding intent has been to explore the use of tools and strategies which may be helpful in improving the quality of both the design process and the designed product. In this pursuit a method has been sought that consistently partitions both the design process and the finished product. The measures of quality in the finished product have been structural and semantic clarity, generality, and mechanistic (as distinct from textual) simplicity.

References

1. Dijkstra, E.W., "A Discipline of Programming", Prentice-Hall, Englewood Cliffs N.J. (1976)
2. Gries, D., "The Science of Programming", Springer-Verlag, N.Y. (1981)
3. Jackson, M.A., "Principles of Program Design", Academic Press, London (1975)
4. Websters Twentieth Century Dictionary, 2nd Ed. World Publishing Co. N.Y. (1960)
5. Polya, G., "How to Solve It", Princeton University Press, Princeton, N.J. (1971)
6. Polya, G., "Induction and Analogy in Mathematics", Princeton University Press, Princeton, N.J. (1954)
7. Polya, G., "Patterns of Plausible Inference", Princeton University Press, Princeton, N.J. (1954)
8. Turski, W.M., "Computer Programming Methodology", Heyden, London (1978)
9. Floyd, R.W., "Paradigms of Programs", ACM Turing Award Lecture - 1978, Comm. ACM, 22, 455-460 (1979)
10. Barter, C.J., "Data Transformations and Program Transformations" in "Programming Language Systems", M.C. Newey, R.B. Stanton, G.L. Wolfendal (Eds), Australian National University Press, Canberra (1978)
11. Dromey, R.G., "Forced Termination of Loops", Software Practice and Experience (submitted)
12. Henderson, P., and Snowden, R., "An Experiment in Structured Programming", BIT, 12, 38 (1972)
13. Jones, C.B., "Software Development - A Rigorous Approach", Prentice-Hall, London (1980)
14. Knuth, D.E., "The Art of Computer Programming Vol II: Seminumerical Algorithms",Addison-Wesley, Reading, Mass. (1969)
15. Dickson, L.E., "History of the Theory of Numbers", Chelsea, N.Y. (1952)
16. Griffiths, S.N., "Design Methodologies - A Comparison" in "Structured Analysis and Design", Infotech International Ltd., Maidenhead, England
17. Naur, P., "An Experiment in Program Development", BIT, 12, 347 (1972)
18. Zipf, G.K., "Human Behaviour and the Principle of Least-Effort", Hafner, N.Y. (1965)
19. Berglund, G.D., "A Guided Tour of Program Design Methodologies", Computer, **14**, No. 10, 13 (1981)