ICAA '83

一九八三年
國際高等自動化會議論文集

Proceedings of the International Conference on Advanced Automation—1983

中華民國七十二年十二月十九日～二十一日

Dec. 19-21, 1983

中央研究院資訊科學研究所

中華民國・台北市

Institute of Information Science, Academia Sinica

Taipei, Taiwan, Republic of China

一九八三年
國際高等自動化會議論文集

Proceedings of the International Conference on Advanced Automation—1983

中華民國七十二年十二月十九日～二十一日

Dec. 19-21, 1983

中央研究院資訊科學研究所

中華民國 • 台北市

Institute of Information Science, Academia Sinica

Taipei, Taiwan, Republic of China

A THEORETICAL PROPOSAL TO A CASD SYSTEM EXTENDING THE JACKSON'S METHOD

C.J.P.Lucena+; R.C.B.Martins+; P.A.S.Veloso+; D.D.Cowan§

+ Department of Computer Science, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro,Brazil
§ Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada

ABSTRACT

This paper presents a new programming method, called the data transform programming method. In particular, we present a specialization of data transform programming to deal with file processing applications. Direct comparison is made with Jackson's approach [1] by the presentation of uniform solutions to problems that cannot solved through his basic method. The new method consists of the application of data transformations to the abstract problem statement, following the formal notions of problem reduction and problem decomposition. Data transformations are expressed in programming terms through a basic set of data type constructors. The method reduces the original problem to a set of sub-problems that can be solved through the direct application of Jackson's method. It produces a solution which is correct by construction.

Key-words: Software engineering, Jackson's method, data-flow design, theory of programming, theory of problems.

1. INTRODUCTION

As computer costs go down the use of computer assistance in the process of problem solving increases. In fact, Computer Assisted Design (CAD) is rapidly catching up in most technological areas. Very recently, as the software development process became better known, a new area has been receiving widespread attention: Computer Assisted Software Design (CASD). Most of the work in the CASD area can be roughly classified into two categories: systems to support the activity of programming-in-the-large (systems level programming) and systems to aid the process of programming-in-the-small (module level programming). The present work describes a methodology that can be used as a basis for a CASD system.

It has been observed that many of the changes in typical data processing applications, often called file processing programs, are caused by the changes in the structure of the data to be processed or to be output as the result of processing and by the accompanying actions which must occur to reflect these changes in the structure of the input/output data. Hence, if a program or system of programs can be designed to reflect the structure of the data that is being processed, then modifications to the data might more easily be reflected in the modifications of the program necessitated by these changes. The above ideas were captured by experienced practitioners who have formulated programming methodologies that have considerably influenced today's programming practices industry. The work of Jackson's [1], Warnier [2] and Yourdon and Constantine [3], are often quoted as some of the most important in this area.

As in many engineering areas, also in the area of software engineering, most of the research work in theory (in particular in programming theory) takes a long time to influence industry. In fact, most of the work in formal program derivation has little or no impact in routine data processing applications programming. On the other hand, since file processing programs have not been sufficiently studied from the formal point of view, experienced practitioners lack the tools to express their ideas about programming methodology in a rigorous way. Even the very successful propositions by Jackson, Warnier, and Yourdon and Constantine could only be made precise through exhaustive exemplification. Very often, subtle aspects of these methodologies have not been expressed at the precision level that is achieved, for instance, in most of the literature about program synthesis.

Data transform programming deals with the class of problems that can be solved by the basic Jackson method. It can also solve, through a uniform approach, problems that Jackson can only handle through major departures from his basic method. The formalization of data transform programming was made possible through the association of the notion of data abstraction to file processing programming and through the utilization of formal definitions for concepts such as program decomposition and program reduction borrowed from the areas of logic and problem solving.

In order to put the original Jackson basic method on a more formal basis, Hughes [5] establishes a correspondence between the class of programs available to treatment by his method and the formal language concept of generalized sequential machine. It turns out that Jackson's basic method gives rise to transformations which are gsm computable (in sense that the required transformation can be performed by a generalized sequential machine). That, of course, explains why Jackson's basic method cannot solve backtracking problems (multiple passes over the input) and problems that he calls structure clashes problems. Jackson solves the latter problems by using ad hoc solutions and the technique of program inversion (preparation of a program to be used, for the same function, as a subroutine to another program).

Cowan and Lucena [6], by introducing a new factor (abstract levels of specification for data and program and the subsequent implementation thereof in terms of more concrete levels of abstraction) into Jackson's method have solved the sorting problem to illustrate how the exercise of thinking abstractly about a problem can lead to novel solutions or solutions which were thought to be unavailable due to shortcomings of a given method. We were left with the problem of showing that the many aspects of the structure clash problem, namely conflict of order, multithreading and boundary conflict problems [1] could be solved uniformily through the same or a similar approach. The idea was to consider that since these form an important class of typical data processing problems they should be solved through a set of prescribed rules which are common to the whole class data processing of problems and not through exceptions to the rules of a basic method. We have also investigated the problem of whether or not the original approach by Cowan and Lucena [6] could be generalized and formalized as a method. The informal notion of data-flow design by Yourdon and Constantine [3], together with the formal notion of problem solving by Veloso and Veloso [7] were instrumental for the formulation and improvement of the original ideas in Cowan and Lucena [6].

Some authors have proposed a programming approach where the transition between successive versions of a program is done according to formal rules called program transformations (see, for instance, [13], [14], [15] and [16]). According to this approach programs are considered as formal objects which can be manipulated by transformation rules. The data transform method involves the application of data transformations to the abstract problem statement, following the formal notions of problem reduction and problem decomposition. Data transformations are expressed in programming terms by using the basic set of data type constructors proposed by Hoare (see section 2 and [8]). The method reduces the original problem to a set of sub-problems that can be solved through the direct application of Jackson's method. It produces a solution which is correct by construction.

The present paper formulates the data programming method and applies it to the sorting problem (unsolvable by the basic Jackson method) and to other examples proposed by Jackson to illustrate the shortcomings of his method. These other examples are particular cases of the structure clash problem. The telegram problem illustrates a boundary clash situation, the system log problem is an example of a multireading problem and the matrix transposition problem illustrates an ordering clash. Since the present paper aims at bridging some of the gap between theory and practice in programming, we have tried not to write it as a mathematical paper. In Section 2 where we describe the method in a somewhat formal way, may be skiped in a first reading. Further formalizations and proofs are to be found in accompanying papers.

2. THE DATA TRANSFORM METHOD

The General Method

Programs solve problems. According to Veloso and Veloso [7] a problem is a structure P=<D,O,q> with two sorts, where
the elements of D are the problem data,
the elements of O are the solutions (outputs)
and q is a binary relation between D and O.

A program P solves a problem if P defines a total function between D and O such that

$$(\forall d{:}D)\ q(P(d),d) \qquad (1)$$

holds. To derive a program through the data method consists of, given an input specification for $d \in D$ and an output specification for $o \in O$ to construct a program P such that formula (1) holds.

Certain data-directed design approaches, such as Jackson's, proceed as above by trying to find at the beginning of the derivation process a direct mapping between the input data structures and the output data structures (a mapping from a representation of $d \in D$ to a representation of $o \in O$). For some situations it is not possible to solve some problems through Jackson's basic method (problems which are not gsm computable). The data transform method proposes a canonical form for the expression of programs that include trivially problems which are solvable through the Jackson basic method and that is amenable to simple transformations which lead to solutions to problems which are not Jackson solvable.

The data transform method starts by expressing the abstract notions of $d \in D$ and $o \in O$, instead of trying to look for data representations for these two entities. This approach, of course, became a standard procedure in many programming methodologies but is not very common in the context of data-directed programming. The strategy for program derivation through the data transform method consists of applying the concept of problem reduction and decomposition while using Hoare's general data type construction mechanisms. Problem reduction and decomposition is applied in a way which will leave us with a set of Jackson solvable problems in hand. In the process of decomposing the problem the method bears some similarity with Yourdon and Constantine's data flow design.

We say a problem P_1=<D_1,O_1,q_1> is a reduction of =<D,O,q> and write $P \overset{\Gamma}{\to} P_1$ if we can define a unary function insert, ins: $D \to D_1$ and a unary function retrieve, retr: $O_1 \to O$ such that the program defined by

$$P(d) = retr(P_1(ins(d))) \qquad (2)$$

solves P when P_1 solves P_1.

In Figure 1 below we illustrate this situation. Note that q is a subset of DxO, q_1 is a subset of $D_1 x O_1$, P is a solution to P (a total function between D and O), P_1 is a solution to P_1 (a total function between D_1 and O_1) and that the functions ins and retr need to be defined in such a way that the composition expressed in (2) is satisfied.

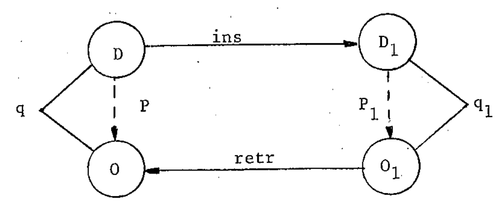

Figure 1: $P \overset{\Gamma}{\to} P_1$

The first step of the data transform method, consists of defining D_1 and O_1 as the cartesian product of D and O; ins such that ins(d)=(d,o_0) for some $o_0 \in O$; retr such that retr(d,o_n)=O. In other words, the reduction through ins and retr makes use of the data type constructor cartesian product (record) which is one of the three basic constructors proposed by Hoare [8]. Intuitively it avoids the problem of structure clashes between the input and output spaces which sometimes occur when the basic Jackson method is directly applied. The input and output data of P_1 have now, trivially, the same structure (independently of any chosen representations for d and o). Figure 2 below further clarifies the previous considerations.

This first step is clearly an intermediate step in the reduction process and is basically motivated by the existence of the structure clash type of problems in

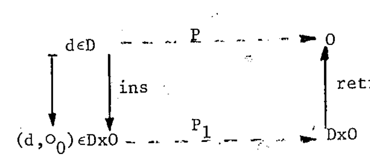

Figure 2

a data-directed programming type of solution. A trivial case, in practice, would be the one for which it is possible to define compatible data structures for d and o. That is, a situation in which P is gsm solvable.

The method requires a second step whenever P_1 is not a simple problem, but requires for instance, modularization or the treatment of backtracking or recursive situations. The second step of the data transform method consists of defining a new reduction $P_2 = \langle D_2, O_2, q_2 \rangle$ of P_1. In this step we will make use of the sequence (file) data type constructor. We will define D_2 as D_1^*; O_2 as O_2^* and the function ins from D_1 to D and retr from O_1 to O as being, respectively, the functions make and last which have the normal meaning of these operators when applied to sequences, that is, make: builds an unitary sequente from a given argument; and last: returns the last element of the sequence. Figure 1 would now be replaced by the situation pictured in Figure 3.

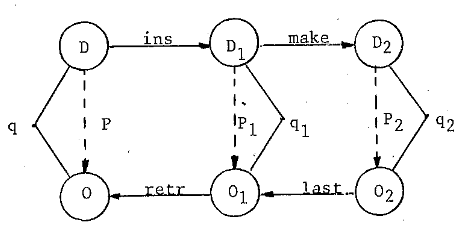

Figure 3: $P \overset{\Gamma}{\to} P_1 \overset{\Gamma}{\to} P_2$

The outcome of this step is a program P_2 which we want to decompose into simpler programs. Let us be more precise about what we mean by decomposition [7]. If we take the problem $P_2 = \langle D_2, O_2, q_2 \rangle$, a n-ary decomposition Δ of P_2, $P_2 \uparrow \Delta$, consists of

i) n functions $decmp_i : D_2 \to D_2$, $i=1,\ldots,n$;

ii) a(n+1) ary function $merge: D_2 x O_2^n \to O_2$;

iii) a unary function $immd: D_2 \to O_2$

iv) a unary relation $easy \subseteq D_2$

We call items (i) to (iv) a good n-ary decomposition of P_2 iff

$$P_2(d_2) = \begin{cases} immd(d_2) \text{ if } easy(d_2) \\ combine(d_2, sol_1[decmp_1(d_2)], \ldots \\ \quad \ldots, sol_n[decmp_n(d_2)]) \text{ otherwise} \end{cases} \qquad (3)$$

where sol stands for the part of the solution of P_2 contributed by each decomposition. Intuitively, if the problem is simple (easy), that is, gsm computable, decomposition is not necessary and we have a direct (immd) solution. Otherwise the solution for P_2 is obtained through the combination (combine) of the solutions (sol's) to the programs $P2_1, P2_2, \ldots, P2_n$ which correspond to the solutions. The decomposition process is guided by a data flow design type of analysis while we try to identify as many gsm solvable problems as possible. If one or more of the identified programs are not gsm computable, steps 1 and 2 and decomposition are applied to all programs at hand and applications of steps 1 and 2.

3. THE DATA TRANSFORM METHOD FOR FILE PROCESSING PROGRAMMING

We are mainly interested here in an important specialization of the data transform method to deal with file processing programming. These problems are identified in association with the data transform method as problems for which the inputs for P are always entities of the general type (files) and as problems for which the constitutive programs of P_2 (obtained by decomposition) are always similar, in the sense that a while statement can drive a copy of them by changing the necessary inputs through its parameter. The program schema below defines the family of programs (in the sense of [9]), that can be obtained by the data transform method as specialized for file processing programming, when we have one application of the first step of the method followed by one application of the second step.

The notation used in Figure 5 below is Pascal-like. The programs that constitute Schema are presented in the order of their derivation, therefore violating a Pascal syntax rule. In the program Schema the selectors i and r simulate the function ins and retr and the symbol Λ stands for the null sequence. The program schema only creates an instance of the input data to

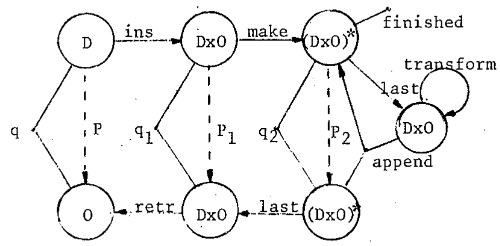

Figure 4 - Diagram for file processing problems solution by the Data Transform Method

allow the application of the method.

The function update for the class of file processing problems, has been defined as $update(x_3)=append(x_3, transform(last(x_3)))$ where transform is a function from DxO to DxO which contributes to the solution of the problem. Refer to definition of $P_2(d_2)$ in equation (3).

The function append has the usual meaning of the operator with the same name, normally associated to the type sequence that is append: $(DxO)^* X(DxO) \rightarrow (DxO)^*$; where append $((p_1,\ldots,p_n),p)=(p_1,\ldots,p_n,p)$

A Correctness Criterion for the Method

We define initially the termination condition for the program schema, displayed in Figure 5. We have:

i) $update(x_3)=append(x_3,transform(last(x_3)))$

ii) $\forall x_3 \in (DxO)^*, smllr(transform(x_3).i, x_3.i)$

iii) smllr is a well founded relation in DxD such any $d \in D$ is in a finite smllr chain starting at Λ: $smllr(\Lambda,d_1)\ smllr(d_1,d_2)\ldots\ smllr(d_n,d)$ (that is usual for file processing program)

iv) $last(x_3).i=\Lambda \leftrightarrow finished(x_3)=true$

Transform and finished must be specified so as to satisfy the above conditions. We can now state the partial correctness condition for the class of programs.

v) $\forall x_3 \in (DxO)^*, finished(x_3) \Rightarrow q_2(x_3, make(d.i.\Lambda))$

vi) $\forall x_3 \in (DxO)^*, q_2(x_3, make(d.i,\Lambda)) \Rightarrow q(last(x_3).r,d)$

Intuitively, the relation smllr guarantees that in each step the transform function contributes some more for the solution of the problem. The smllr relation, which is a well founded relation, characterizes the empty element as a distinguished element that will necessarili be reached to accomplish the termination of the program.

```
Program schema;
   type D = seq of objects1;
   type O = objects2;
   type DxO = record i:D;
                     r:O
   type(DxO)* = seq of DxO;
   var x,d:D;
   var y,o:O;
    begin
       x ← copy(d);
       P ;
       o ← copy(y)
    end {schema}.
Procedure P;
   var x1,y1:DxO;
    begin
       x1.i ← x; x1.r ← Λ;
       P1;
       y ← Y1.r
    end {P};
Procedure P1;
   var x2,y2:(DxO)*;
    begin
       x2 ← make(x1);
       P2;
       y1 ← last(y2)
    end {P1};
Procedure P2;
   var x3:(DxO)*;
    begin
       x3 ← x2;
       while not finished (x3)do
       x3 ← update (x3);
       y2 ← x3
    end {P2};
```

Figure 5 - Program Schema for File Processing Programming through the Data Transfor Method

Condition (v) guarantees that when the program stops x_3 is the solution of the problem for which the input is obtained from d by the application of ins and make and condition (vi) ensures that the reduction from the original problem P to P_2 is good, i.e., that the element from x_3 obtained by the application of retr and last is the solution to the original problem with input d.

4. THE SORTING PROBLEM

We have selected the sorting problem as our first example for a number of reasons. First of all, the problem is very well known and therefore the reader can concentrate all the attention in the problem solving method and compare it with the many available solutions to the problem. Second, since sorting exemplifies a situation of backtracking (or at least some backtracking) it illustrates a case where Jackson's basic method cannot be directly applied [1].

Let A be a totally ordered set, $d=<a_1,a_2,\ldots,a_n> \in D$ a finite sequence of elements from A and $o=<b_1,b_2,\ldots,b_n> \in O$ a finite sequence of elements from A. To sort means to solve a problem SORT=<D,O,q> such that $q(o,d)$ is defined by $\{a_1,\ldots,a_n\}=\{b_1,\ldots,b_n\}$ and $(\forall i,\forall j, 1\leq i<j\leq n) \Rightarrow b_i > b_j$. For simplification purpose we assume that $a_i \neq a_j$ for all $i \neq j$ and $d \neq \Lambda$.

As in Figure 5 we will define a Program Sort that will create an instance of the data that will be used for the application of the data method. Program sort can be define as follows:

```
Program sort;
   type D = seq of Aobjects;
        O = seq of Aobjects;
        (DxO = record i:D;
                      r:O
              end;
        (DxO)* = seq of (DxO)*;
   var  x,d:D;
        y,o:D;
        begin
             x ← copy(d);
             P;
             o ← copy(y)
        end {sort}.
```

Of course, identifiers such as (DxO) and (DxO)* are not available in standard Pascal syntax. They are used here for compatibility with the mathematical notation. The notation seq of Aobjects stands for a sequence of objects.

We are now ready to apply the first step of the method. Procedure P can then be expressed as:

```
Procedure P;
   var x1,y1:DxO;
      begin
          x1.i ← x;
          x1.r ← Λ;
          P1;
          y ← y1.r
      end {P};
```

Note that the selectors i and r simulate the functions ins and retr.

We now apply step 2 expressing P_1 as follows:

```
Procedure P1;
   var x2,y2:(DxO)*;
      begin
          x2 ← make(x1);
          P2;
          y1 ← last(y2
      end {P1};
```

Functions make and last need to be expressed in PASCAL notation, following their usual definitions for files. Note that so far we have only organized the solution of the problem so as to put it in our canonical form. Later we will indicate how the above structure for the problem solution will actually help establishing the correction of the program (in particular termination). Next step is a first decomposition of P_2. Remember we are only interested in this paper to solve problems that can be classified as file processing applications. For that purpose the following decomposition can be proposed. The notation we use is widely applied in the literature about abstract data types [10]. It bears a natural similarity with Yourdon and Constantine's data flow graphs because when decomposing we are detecting the transformations to be applied on the data. We are now ready to express procedure P_2 and update as follows:

```
Procedure P2;
   var x3: (DxO)*;
      begin
```

```
        x3 ← x2;
        while not finished(x3) do
              x3 ← update(x3);
        y2 ← y3 ;
     end {P2};
Procedure update(x3:(DxO)*):(DxO)*;
   var x4:DxO;
       y3:(DxO)*;
     begin
        y3 ← x3;
        x4 ← last(x3;
        x4 ← transform(x4);
        update ← append(y3,x4)
     end {update};
```

For the next level of decomposition we will separate the input structure from the output structure and will remove one input element, "transform" it and place it in the output. This idea can be expressed graphically through the following diagram

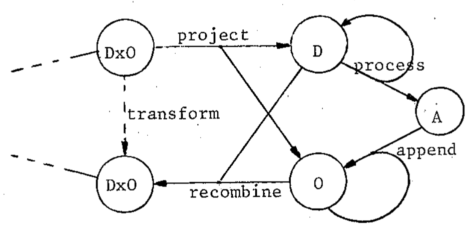

Figure 6

This decomposition step can be throught of as being coupled to the diagram: - Figure 4 (note the dots to the left of the diagram in Figure 6). The function project stands for the first and second projection of the cartesian product (simulated by the selectors i and r in the following transform program). The function recombine constructs an ordered pair from two given elements. It should be clear that project, recombine and append are gsm solvable. We need now to define process in such a way that in each pass of its execution process reduces the input and expands the output while contributing to the solution of the problem. Hopefully we will be able to define process so as to be gsm solvable (otherwise we would need to further decompose process). Since the sorting problem is very well known it is simple to identify the central operation of process so as to make it gsm solvable. This operation consists of selecting the minimal element of the input sequence and append it to the end of the output sequence. The operation then determines a sequence of one pass scannings over the input, leading therefore to a gsm solvable problem. We can at this point present the code for transform and process.

```
Procedure transform(x4:DxO):DxO;
   var x5,x6:D;
       y5,y6:O;
       minimum:Aobjects;
     begin
        x5 ← x4.i;
        y5 ← x4.r;
        Process;
        y6        ←    append(y5,minimum);
        transform ←    recombine(x6,y6)
     end {transform}
Procedure Process;
     begin
        minimum ← first(x5);
        x5    ←    tail(x5);
        x6    ←    Λ;
          while not (x5=Λ) do
               if minimum<first(x5)then
                 begin
                    x6 ← append(x6,first(x5))
                    x5 ← tail(x5)
                 end
                                    else
                 begin
                    x6 ← append(x6,minimum);
                    minimum ← first(x5);
                    x5 ← tail(x5)
                 end
     end {Process}
```

The functions first and tail have their usual meaning when applied to sequences. We need now to specify the predicate finished so as to satisfy the correctness conditions defined in 3. For that we note that process reduces in each pass the length of the first component of the ordered pair which is being "transformed". It naturally suggests that this process terminates whenever the length of the first component becomes zero. We can now define finished as:

$\forall x_3 \in (DxO)^*, finished(x_3) \leftrightarrow length(last(x_3).i) = 0$

To satisfy the correctness criterion expressed in 3 we need to define a well-founded relation smllr. We propose the following:

$(\forall d_1, d_2) \in D, smllr(d_1, d_2) \Leftrightarrow length(d_1) < length(d_2)$

An informal argument can be expressed as follows. Given the way process was constructed, length(transform(x_3).i) < length(x_3.i) and that proves condition (ii) of 3. We also have that smllr has been defined as "<" which is a well founded relation, which proves condition (iii). The definition of finished matches condition (iv) and finally the condition (v) for partial correctness can be shown by induction on the way the output sequence is constructed (in each step we introduce the next possible smallest element).

The reader must have noticed that in the problem solution the first reduction which seemed artificial, sence the sorting problem cannot be characterized as a structure clash problem, has in fact been instrumental for proving the termination of the program. In fact, recall that finished and smllr have been defined on the first component of an input-output ordered pair.

5. THE TELEGRAM ANALYSIS PROBLEM

The classical telegrams analysis problem, often used as an example of structure clash, boundary clash in Jackson's [1] terminology, has been defined in his book (page 155).

As before, we will define a program TELEGRAM that will create an instance of the data that will be used for the application of the reductions and decompositions that will take us to our canonical form.

```
Program Telegram
   type D = seq of Telegrams;
        O = seq of Telegram-analysis;
        (DxO) = record i:D;
                       r:O
                       end;
        (DxO)* = seq of (DxO);
   var x,d:D;
       y,O:O;
      begin
           x ← copy(d);
           P;
           y ← copy(y)
      end {Telegram}.
```

The solution of the problem follows exactly the same steps used in the sorting example up to the point where we need to define the procedures Transform and Process. The change in the Transform function is minor and the procedure can be expressed as follows:

```
Procedure Transform(x4:DxO):DxO;
   var x5,x6:D;
       y5,y6:O;
       report:telegram-analysis;
     begin
          x5 ← x4.i;
          y5 ← y4.r;
          Process;
          y6 ← append(y5,report);
          Transform ← recombine(x6,y6)
     end {Transform};
```

We are now going to derive Process. According to the data transform method we need Process to be gsm solvable or decomposable in gsm solvable programs. Recall that the method makes use of the notion of data abstraction. In particular, Process will deal with seq of telegrams. It means, in practice, that we are focusing in the concept of a Telegram instead of reasoning at the block "level" as Jackson does. The core of the program Process, which is dealing with the cartesian product of the sequence of telegrams with sequence of telegram analysis can be represented graphically by the following picture.

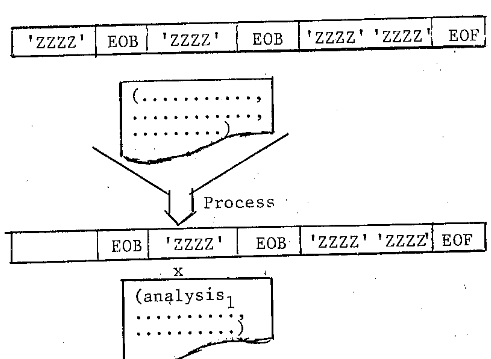

Figure 7

To implement Process, it is necessary to scan the tape block by block. Within each block Process must analyse word by word and compute each one for report purposes. When finding the end of a telegram before the end of a block, Process places the rest of the block as the first block in the output tape. The processing of words through this approach involves no prediction and therefore Process is gsm solvable. One possible schematic version for Process could be the following:

```
Procedure Process;
      begin
          x6 ← Λ;
          get(first block in x5);
          if first word in block is 'ZZZZ'
          then report ← Λ
          else
            begin
            initialize report;
            while(telegram not empty)do
              begin
              while(telegram not empty and
                     block not empty)    do
                     analysis of a word in report;
              while (block not empty)    do
                     construct the first block
                     in x6;
              get (another block in x5)
              end;
            while(x5not empty)do
            begin
                 append(x6,block from x5);
                 get (block in x5)
            end
          end
      end {Process};
```

As in the sorting problem we need now to characterize the predicate finished. It so happens that it takes the same form as in the sorting example, that is:
$\forall x_3 \in (DxO)^*, finished(x_3) <\!\!-\!\!> length(last(x_3).i) = 0$

That, of course, is so because we are dealing with a standard file processing problem, as defined by the data transform method. We reach this standard form for the termination procedure because the first problem reduction (cartesian product) leaves us with the input data to be processed as the first component of the product. The input data is always reduced (each execution of Process has at least an operation get) and saved and therefore the program terminates when the input part of product is empty. For the proof of correctness of the program we proceed as in the sorting example after verifying the inner simple details of the operations "initialize report" and "analysis of words" in the Process program.

6. CONCLUSIONS

We have presented in this paper the data transform programming method and applied it to the solution of some classical programming problems. The choice of the examples was meant to compare clearly our approach with Jackson's method, since his method cannot solve directly the problems we have dealt with. Our method also solves Jackson's system log and matrix transposition problems [17]. When choosing this criterion for exemplification we realized that although the examples used are not solvable through Jackson's basic method they are trivial applications of file processing programming, which often deals with far more complicated situations. This could have probably given the impression to the reader that we are using a theory that is too general to deal with the present problems. Note that the full power of the method can better be left felt through its applications. When we deal with large problems such as making verification accessible to practitioners, providing programming standards for large programming teams and enhancing documentation and maintenance can be assessed. We plan to design other publications meant to evaluate data transform programming as applied to real problems. On the other hand, we are confident that starting with situations even simpler than the ones that appear in sections 4 to 7 we are able to illustrate the potential of data programming for teaching purposes.

The present work is a major extension of the work published in [6]. Still, many interesting developments of the present work are in sight. Partly automating the method is one possible research direction. The work by Coleman, Hughes and Powell [11]and Logrippo and Skuce [12] follow this general direction although they are restricted to Jackson's basic method. We believe, as [14], that for a large, longlived software project, the existence of an accurate, readable model or specification, such as the one produced by the data transform method, can be as important as the existence of an efficient implementation of it. We are presently working on a refinement procedure that will allow us to an efficient version for the solution at hand through a set of well defined program transformations.

Some interesting theoretical results are currently being pursued. They are related to the formal characterization of the class of problems which are solvable through the general version of the data transform method (when, for instance, the recursion problem can be contemplated) and of the class of problems defined by the specialization of the data transform method to file processing programming, which we have examined in this paper.

REFERENCES

1. Jackson,M.A. Principles of Program Design. London: Academic Pres, 1975.

2. Warnier,J.D. Logical Construction of Programs. New York: Van Nostrand Reinhold, 1974.

3. Yourdon,E.;Constantine,L.L. Structured Design: Fundamentals of a Discipline of Computer Program ans System Design. Yourdon Press, 1978.

4. Chand,D.R.; Yadav,S.B. Logical Construction of Software. CAGM, V.23, N10, 1980.

5. Hughes,J.W. A Formalization and Explanation of the Michel Jackson Method of Program Design. Software-Practice and Experience. V.9, 1979.

6. Cowan,D.D.;Graham,J.W.;Welch,J.W.;Lucena,C.J. A Data-directed Approach to Program Construction. Software-Practice and Experience. V.10, 1980.

7. Veloso,P.A.S.; Veloso,S.R.M. Problem Decomposition and Reduction: Applicability, Soundness, Completeness; Trappl,R.;Klir,J.;Pichler,F. (eds.) Progress in Cybernetics and Systems Research. Vol.VIII, Hemisphere Publ. Co. 1980.

8. Hoare,C.A.R., Notes on Data Structuring. in Dahl, O.J.;Dijkstra,E.W.;Hoare,C.A.R. Structured Programming. Academic Press. 1972.

9. Parnas,D.L. Designing Software for Ease of Extension and Contraction. IEEE Trans. SE.Vol. SE-5, nº 2, 1979.

10. Goguen,J.A.;Thatcher,J.W.;Wagner,E.G.;Wright,J.F. An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types, in Yeh,R.T. (ed) Current Trends in Programming Methodology, vol.IV.

11. Coleman,D.;Hughes,J.W.;Powell,M.S. A Method for the Syntax Directed Design of Multiprograms. IEEE Trans. on S.E., vol. SE 7, Nº 2, 1981.

12. Logrippo,L.;Skuce,D.R. File Structures, Program Structures, and Attributed Grammars. Technical Report TR82-02, Computer Science Department, University of Ottawa, 1982.

13. Broy,M.;Pepper,P. Program Development as a Formal Activity. IEEE Transactions on Software Engineering Vol SE-7, Nº 1, 1981.

14. Cheatham,T.E.;Holloway,G.H.;Townley,J.A. Program Refinement by Transformation. Proceedings of the 5th International Conference on Software Engineering, 1981.

15. Gerhart,S.L. Correctness-Preserving Program Transformations. Proc. ACM Symp. on Principles of Programming Languages, 1975.

16. Arsac,J.J. Syntatic Source to Source Transforms and Program Manipulation. CACM, Vol 22, Nº 1, 1979.

17. Lucena,C.J.P.;Martins,R.C.B.; Veloso,P.A.S. and Cowan,D.D. The Data Transform Programming Method and File Processing Problems. Technical Report 5/83. Computer Science Department. Pontificia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 1983.