PUC

Series: Monografias em Ciência da Computação

Nº 5/83

THE DATA TRANSFORM PROGRAMMING METHOD AND FILE PROCESSING PROBLEMS

by

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

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 – CEP-22453
RIO DE JANEIRO – BRASIL

PUC-RJ - DEPARTAMENTO DE INFORMÁTICA

Series: Monografias em Ciência da Computação, Nº 5/83

Series Editor: Antonio L. Furtado April, 1983

THE DATA TRANSFORM PROGRAMMING METHOD AND FILE PROCESSING PROBLEMS*

by

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

* Research partly sponsored by FINEP, CNPq and IBM do Brasil.

+ Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada.

RESUMO

Este trabalho apresenta um novo método de programação, chamado o método dos transformadores de dados. Particularmente, apresenta-se a especialização do método necessária para tratar-se de aplicações referentes a processamento de arquivos sequenciais. Faz-se uma comparação direta com o trabalho de Jackson [1] apresentando se soluções uniformes para problemas que não podem ser resolvidos pelo método básico de Jackson.

O novo método consiste na aplicação de transformações de dados a formulação abstrata do problema, de acordo com as noções de redução e decomposição de problemas. As transformações de dados são expressas em termos de programa através de um conjunto básico de construtores. O método reduz o problema original a um conjunto de subproblemas que pode ser resolvido através da aplicação direta do método de Jackson. Produz-se uma solução correta por construção.

Palavras chaves: engenharia de software, método de Jackson, fluxo de dados, teoria da programação, teoria de problemas.

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 be 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.

"What are the data?;
What is the unknown?;
What is the condition?"
G. Polya

1. Introduction

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 in industry. The work of Jackson [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 the 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 problems 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.

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 3 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.

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 multithreading problem and the matrix transposition problem illustrates an ordering clash.

We try to make this work self-contained by reviewing Jackson's method and Yourdon and Constantine's data flow design methodology.

The data transform method is presented through some concepts in problem solving theory and theory of data types, associated with informal arguments.

2. The Jackson Method

Following the formalization of the method by Hughes [5] in terms of generalized sequential machines (gsm's), we outline a correspondence between the class of programs developed by using the basic Jackson method and gsm computable functions between regular languages.

Jackson pointed out that input (and output), may often be regarded as possibly infinite languages (over some primitive set of data values). A Jackson tree provides a finite representation of such a language and can be used to represent only a regular language. One can show therefore, that Jackson trees are an alternative notation for regular expressions as are data type definitions using the types sequence, cartesian product and discriminated union [8]. In fact, all the notations above are capable of representing only restricted forms of regular expressions if we assume the usual conventions concerning the priority of the regular expression operations "*", "U" and "." (if we do not make this assumption and proceed strictly according to the formal definition of these operations, there is an exact one to one correspondence).

We proceed to outline the four notations below by defining how a given regular expression is represented in the other two notations.

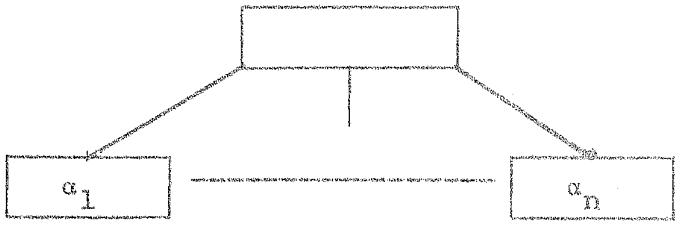
i) A single terminal symbol a is represented as follows:

(a)

Jackson-tree Hoare-Wirth notation

Figure 1

ii) A regular expression $(\alpha_i,\ldots,\alpha_n)$ which is a concatenation of regular expressions is represented as follows:



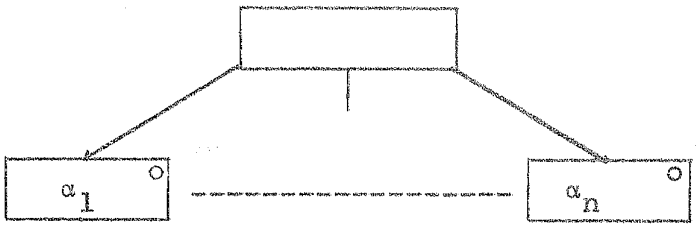
$(\alpha_1;\ldots;\alpha_n)$

Jackson tree

Hoare-Wirth notation

Figure 2

iii) An expression $(\alpha_1 \cup \alpha_2 \cup \ldots \cup \alpha_n)$ which is the union of regular expressions is represented as follows:



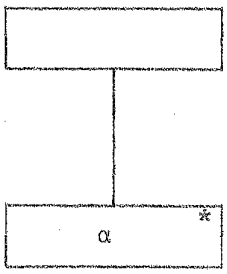
$(\alpha_1,\ldots,\alpha_n)$

Jackson tree

Hoare-Wirth notation

Figure 3

iv) An expression α^* which is the iteration of a regular expression is represented as follows:



seq of α

Jackson tree

Hoare-Wirth notation

Figure 4

The reader should note that bracketing of regular expressions with respect to the same operation has been abandoned in the above notations to create more compact notations. This reduces the size of our various notations without loss of correctness. Note also that this corresponds to generalizing the usual binary operation "." and "U" to a family of operations to any finite number of arguments.

Now the method requires that correspondences are identified between the specifications of the input and that of the output in terms of correspondences between substructures in the two specifications. This is done in a bottom-up fashion so that the translation of a node in the tree (or graph) depends only on its descendents (i.e., the sub-expression or subtree defined by the node).

The correspondence effectively defines a desired translation between the nodes of the input specification tree and those of the output specification tree.

As it was proven in [5], for a given characterization of gms's it can be shown that Jackson's basic method gives rise exactly to transformations which are gsm computable.

3. Data Flow Design

Data flow design has been proposed by Yourdon and Constantine [3] as a program or programming system design methodology. As we did with the Jackson method we will now outline the central concepts in data flow design.

The purpose of the methodology is to identify the primary processing functions of the system, the high-level inputs to those functions, and the high-level outputs. It then creates high-level modules within the hierarchy to perform each of these tasks: creation of high-level inputs, transformation of inputs into high-level outputs and the processing of those outputs. Clearly, data flow design is an information flow model rather than a procedural model.

Like other information flow models, transform analysis makes use of a graph model of computation. It is called a data flow graph. The nodes of the graph are called transforms. Each node represents a data transformation (to be accomplished later by a module or a program) from a data representation to another. The data elelemts are represented by labelled arrows connecting the nodes. Figure 5 shows a transform with a single input stream and a single output stream.

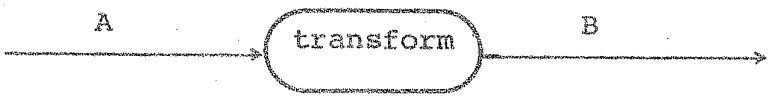


Figure 5

A transform may require (or accept) elements of more than one input data streams in order to produce its outputs. An asterisk "*" or a disjunction symbol "+" indicate simultaneous requirements of data elements or a mutually exclusive situation, respectively.

Yourdon and Constantine [3] have proposed a data flow design methodology composed essentially of the following four steps. In what follows we have tried to capture the central aspects of each step.

Step 1: Consists of the statement of the problem as a data flow graph; the authors recommend that the designer should be concerned first with the "main" data paths dealing with primary inputs.

Step 2: Identification of the initial and final data elements. Initial and final data elements are those high-level elements of data which are furthest removed from physical input and output, respectively.

Step 3: This step is further subdivided into four parts. Having identified the initial and the final data elements of the system:

i) specify a "main" module which, when activated, will perform the entire task of the system by calling upon subordinates.

ii) for each initial data element feeding a central transform an initial module is specified as an immediate subordinate to the main module.

iii) for each final data element emerging from any central transform a subordinate final module is defined which will accept the final data element.

iv) for each central transform or functionally cohesive[1] composition of central transforms, we specify a subordinate transform module which will accept from the main module the appropriate input data and transform it into the appropriate output data.

Yourdon and C. note at this point that there is a simple (usually one-to-one) correspondence between the initial data flow graph and the module diagram that can be associated to it.

Step 4: This step consists of the factoring of the initial, final and transform modules until the ultimate physical input and output are reached as well as the detailed transform modules detected during the analysis of the problem (those for which it is not possible to state a transform with any

---

[1] Because of the goal of the present work, we have omitted many aspects of the method being described, such as cohesiveness, coupling, functional strength etc., which are not directly related to the subject of this paper.

clearly discernible sub-tasks).

Yourdon and C. point out that the objective of transform analysis is to make the program structure reflect the structure of the problem statement as a data flow graph.

4. The Data Transform Method

The General Method

Programs solve problems. According to Veloso [7] a problem is a structure $P = \langle D,O,q \rangle$ 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 $d \epsilon D$ and an output specification for $o \epsilon O$ to construct a program P such that equation (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 $d \epsilon D$ to a representation of $o \epsilon O$). As it was pointed out in section 2, 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 $d \epsilon D$ and $o \epsilon O$, 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 (section 2 and [8]).

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 $P \rightarrow P_1$ if we can define
an unary function insert, ins: D → D_1 and
an unary function retrieve, retr: O_1 → O
such that the program defined by P(d) = retr(P_1(ins(d)) (2)
solves P when P_1 solves P_1.
In Figure 7 below we illustrate this situation. Note that
q is a subset of DxO
q_1 is a subset of D_1xO_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.

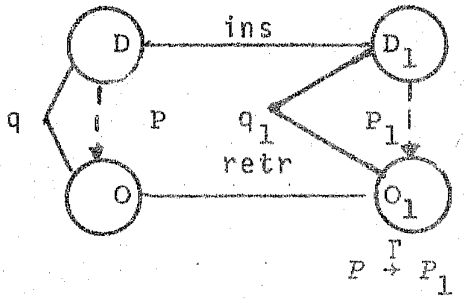


Figure 7

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)=0. 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 8 below further clarifies the previous considerations.

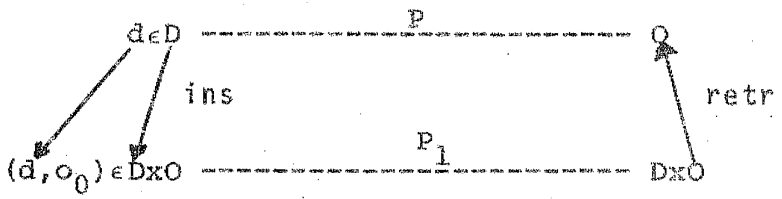


Figure 8

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 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 $P_2=<D_2,O_2,q_2>$ 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 sequence from a given argument
last: returns the last element of the sequence

Figure 7 would now be replaced by the situation pictured in Figure 9.

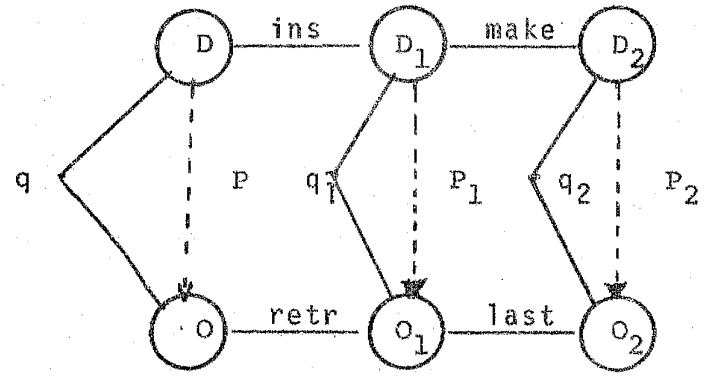


Figure 9: $P \overset{\Gamma}{\downarrow} P_1 \overset{\Gamma}{\downarrow} P_2$

The diagram in Figure 8 can now be expanded in the following way

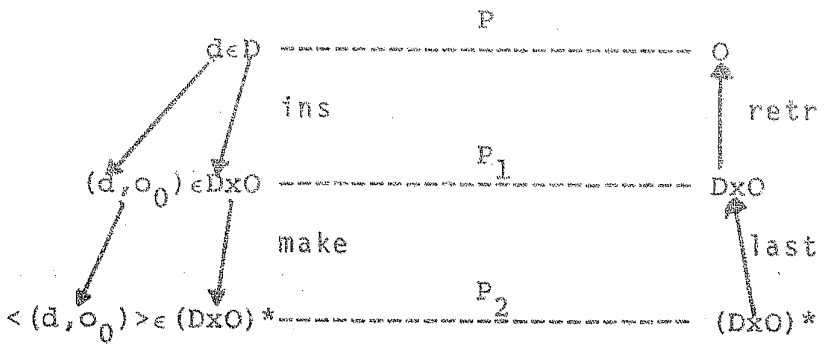


Figure 10

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 $P_2=<D_2,O_2,q_2>$, a n-ary decomposition Δ of P_2, $P_2\uparrow\Delta$, consists of

i) n functions $decmp_i: D_2 \rightarrow D_2$, i=1,...,n;

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

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

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 $P_{2_1}, P_{2_2},\ldots,P_{2_n}$ 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.

5. 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.

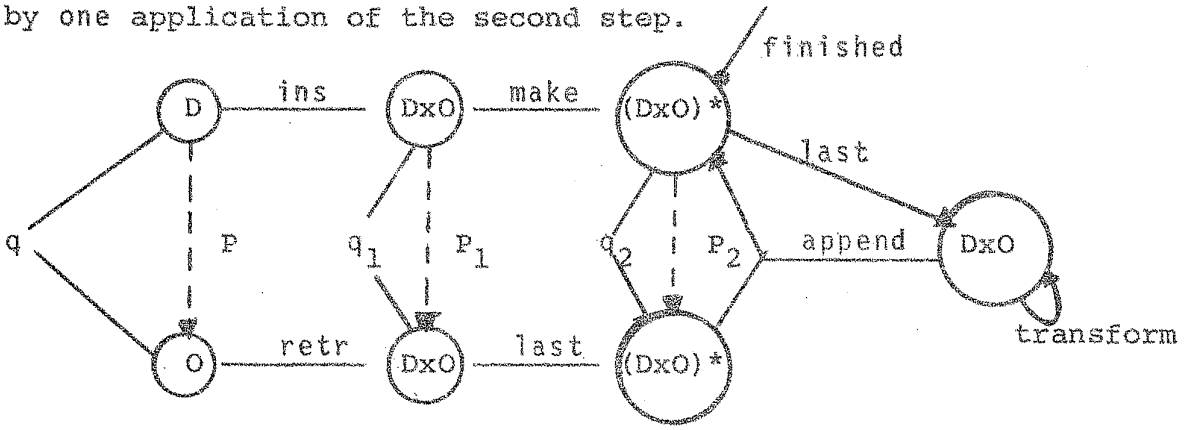


Figure 11.a - Diagram for file processing problems solution by the Data Transform Method

The notation used in Figure 11.b below is Pascal-like. The programs that constitute Schema are presented in the order of their derivation, therefore violating a Pascal syntax rule.

From now on, any standard function not defined in the text is explained in the glossary of functions in Appendix I.

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

```
Program schema;
   type D = seq of objects1;
   type O = objects2;
   type DxO = record i:D;
                     r:O
                     end;
   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 11.b - Program Schema for File Processing Programming through the Data Transform Method

data to 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

$$append: (DxO)^* \times (DxO) \rightarrow (DxO)^*$$
$$\text{and append } ((p_1,\ldots,p_n),p) = (p_1,\ldots,p_n,p)$$

A Correctness Criterion for the Method

We define initially the termination condition for the program schema displayed in Figure 11.b. 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 necessarily be reached to accomplish the termination of the program.

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.

6. 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]. We will also take advantage of the conciseness of the sorting problem statement to illustrate through its development via the data transform method all the details of the theory presented in Section 4. It would be harder to do the same with a problem with a more complex definition (such as the ones presented in the next sections).

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

i) $\{a_1,\ldots,a_n\} = \{b_1,\ldots,b_n\}$

ii) $(\forall i,\forall j, 1 \leq i < j \leq n) \Rightarrow b_i < b_j$

For simplification purposes we assume that $a_i \neq a_j$ for all $i \neq j$ and $d \neq \Lambda$.

As in Figure 11.b 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[2]. Program sort can be defined as follows:

```
Program sort;
   type D = seq of Aobjects;
        O = seq of Aobjects;
        (DxO) = record i:D;
                       r:O
                end;
        (DxO)* = seq of(DxO)*;
```

[2] That is, we will apply steps 1 and 2, therefore placing the problem in our canonical form, and then examine the solution at hand to see if further reductions or decompositions are necessary.

```
var  x,d:D;
     y,o:O;
     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.

Graphically, what we have done so far, leaves us with the situation shown in Figure 12.

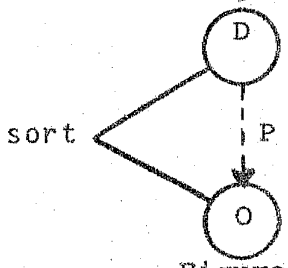


Figure 12:Sort

We are now ready to apply the first step of the method. It is graphically represented in Figure 13. We want now to model the situation expressed in Figure 13, through a program P.

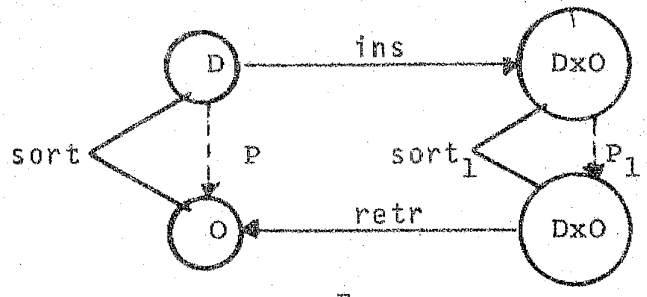


Figure 13: SORT $\sqsubseteq$ $SORT_1$

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 which corresponds to the abstract notion introduced in Figure 9.

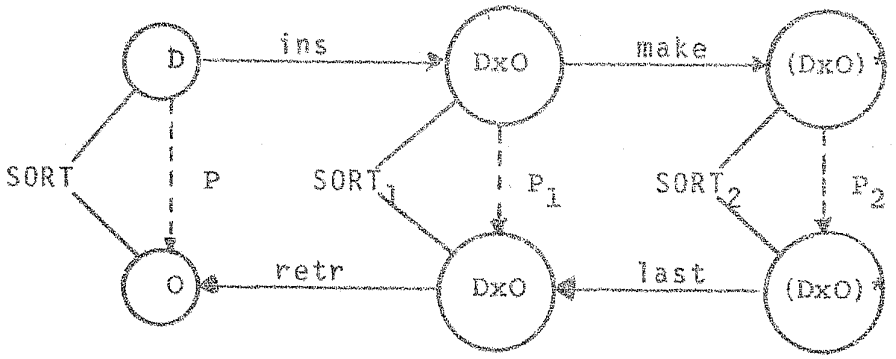


Figure 14: SORT $\downarrow$ SORT$_1$ $\downarrow$ SORT$_2$

P_1 can be expressed 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. For didactic purposes we shall add the first decomposition to the diagram in Figure 14. It should be noted that Figure 15 contains a diagram which is typical of file processing programs.

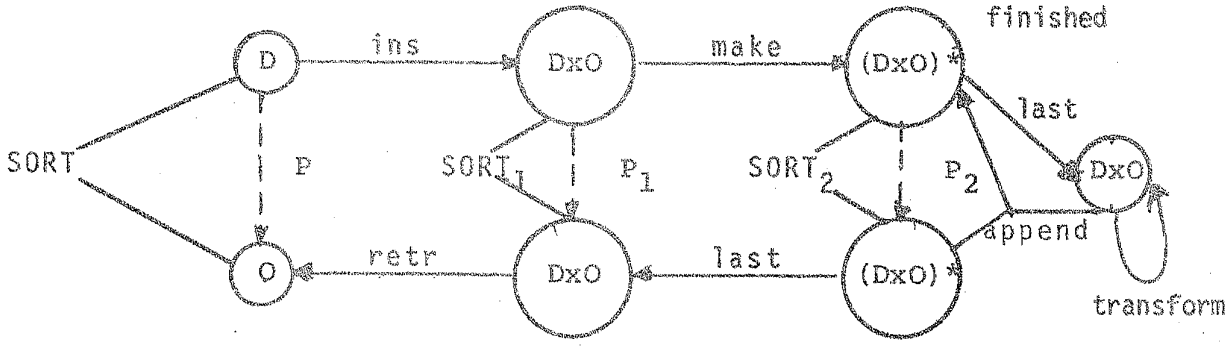


Figure 15 - SORT $\overset{\Gamma}{\downarrow}$ $SORT_1$ $\overset{\Gamma}{\downarrow}$ $SORT_2$ $\uparrow$(last,transform,append)*

We are now ready to express programs P_2 and update as follows:

```
Procedure P2;
   var x3: (DxO)*;
      begin
           x3 ← x2;
           while not finished(x3) do
                x3 ← update(x3);
           y2 ← x3;
      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 16).

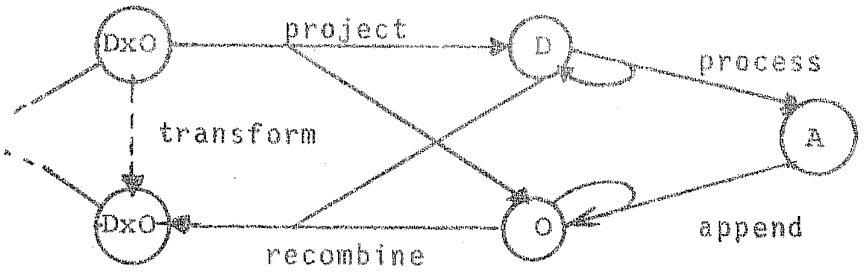


Figure 16

This decomposition step can be thought of as being coupled to the diagram: - Figure 15 (note the dots to the left of the diagram in Figure 16). 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 program.

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 (see glossary in Appendix I).

We need now to specify the predicate finished so as to satisfy the correctness conditions defined in 4.3. 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:

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

To satisfy the correctness criterion expressed in 4.3 we need to define a well-founded relation smllr. We propose the following:

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

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 4.3. 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, since 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. The reader will find in the Appendix II a complete version of the program derived for the sorting problem.

7. 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 follows.

> "An input file on paper tape contains the texts of a number of telegrams. The tape is accessed by a "read block" instruction, which reads into main storage a variable-length character string delimited by a terminal EOB character: the size of a block cannot exceed 100 characters, excluding the EOB. Each block contains a number of words, separated by space characters; there may be one or more spaces between adjacent words, and at the beginning and end of a block there may (but need not) be one or more additional spaces. Each telegram consists of a number of words followed by the special word "ZZZZ"; the file is terminated by a special end-file block, whose first character is EOF. In addition, there is always a null telegram at the end of the file, in the block preceeding the special end-file block: this null telegram consists only of the word "ZZZZ". Except for the fact that the null telegram always appears at the end of the file, there is no particular relationship between blocks and telegrams: a telegram may begin and end anywhere within a block, and may span several blocks; several telegrams may share a block.
>
> The processing required is an analysis of the telegrams. A report is to be produced showing for each telegram the number of words it contains and the number of those words which are oversize (more than 12 characters). For purposes of the report, "ZZZZ" does not count as a word, nor does the null telegram count as a telegram."

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 programs Transform and Process.

The change in the Transform function is minor and the program 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 17).

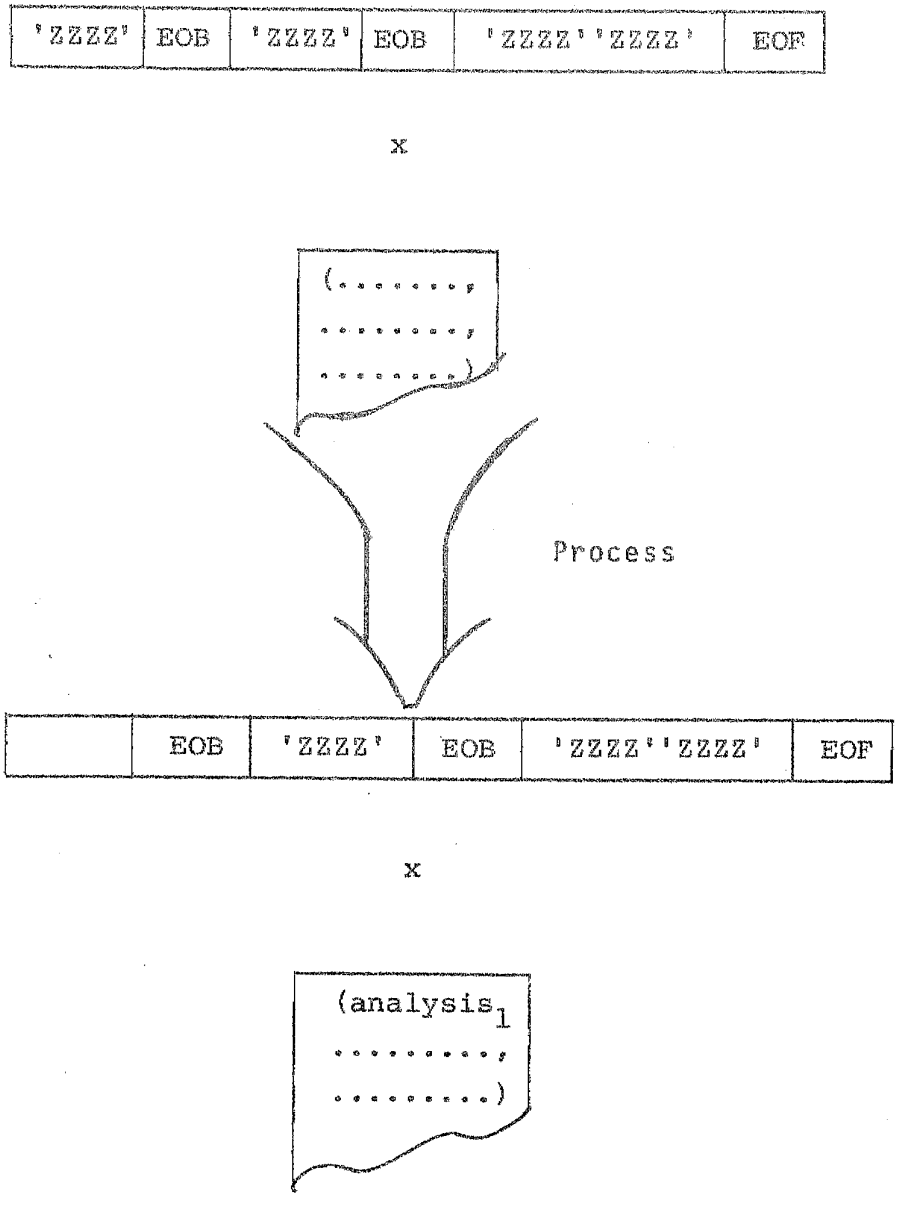


Figure 17

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. Whin 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     x5 not 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:

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

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 in-

put 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.

Although the previous level of decomposition may be considered satisfactory, a reader could possibly feel more confortable with a further decomposed solution. We will illustrate this possibility by decomposing Process one more time. Process can be decomposed into three sub-problems. The first, get-telegram, reads the input tape block by block (x_5) and within each block it looks for the word "ZZZZ". Once "ZZZZ" is found, the rest of the block which is being processed is appended to the output tape (x_6).

The second sub-problem, get-tape, reads the rest of the input tape (x_5) and transfers its contents to the output tape (x_6).

The third sub-problem, analysis, makes the analysis of one telegram. The input of this sub-problem is a telegram which consists of a sequence of words, and the output is a report about the analysis of one telegram, which can be in turn recognized as a file processing problem (and therefore further reduced).

The new version of Process becomes:

```
Procedure Process;
   type T: seq of words;
      var t1: T;
          r1:telegram-analysis;
         begin
              Get-telegram;
              Get-tape;
              Analysis
         end {Process};
```

```
Procedure Get-telegram;
        begin
             x6 ← Λ;
             get(first block in x5);
             if  first word is 'ZZZZ'
             then r1 ← Λ
             else
                 begin
                      t1 ← first word;
                      while telegram not finished  do
                           begin
                                while(block and telegram not empty)do
                                     if word is ¬ 'ZZZZ'
                                     then t1 ← append (t1,word)
                                     else telegram finished;
                                while(block not empty)do
                                     x6 ← rest of block;
                                get(another block)
                           end
                 end
        end {Get-telegram};
Procedure Get-tape;
        begin
             while(x5 not empty)do
                  begin
                       x5 ← append(x6,block in x5);
                       get (block in x5)
                  end
        end {Get-tape};
Procedure Analysis;
   type T: seq of words;
        R: telegram-analysis;
        (TxR):record i:T;
                     r:O
               end
        (TxR)*:seq of(TxR)*;
   var x',t1:T;
       y',r:R;
       x1',y1':TxO;
       x2',y2',x3':(TxO)*;
```

```
Procedure P2';
        begin
             x3' ← x2';
             while(last(x3').i ≠ 0) do
                  x3 ← updateT(x3');
             x2' ← x3'
        end {P2'};
Procedure P1';
        begin
             x2'← make(x1');
             P2';
             y1'← last(y2)
        end {P1'};
Procedure P';
        begin
             x1'.i ← x';
             x1'.r ← Λ ;
             P1';
             y ← y1'.r
        end {P'};
begin
     x'← copy(t1);
     P;
     r ← copy(y')
end {Analysis};
```

Note that for the sake of clarity we have redefined the types T and R. $Update_T$ could be defined in such a way that each of its executions would perform the required analysis of one word of a telegram.

8. The System Log Problem

The System Log Problem is a structure clash problem which Jackson classifies as a multi-threading problem. The problem as stated by Jackson is the following (page 160) of [1].

> "A time-sharing system collects information about system usage. This information consists of records, one for each log-on, log-off, program-load and program-unload. When a user of the system logs on, he is allocated a unique job-number for that session: one user would receive two different job-numbers if he logged on two different occasions. The system ensures that no user can log on unless the terminal is free (that terminal), and that he cannot log off unless he has previously logged on. Further, he is allowed only one active program at any one time: he must unload that program before he can load another or load the same program again.
>
> The collected information is written to magnetic tape. The records contain the following information:
>
> log-on record: code "N";job-number;time of logging on;
>
> log-off record;code "F";job-number;time of logging off;
>
> program-load record: code"L";job-number;program-id;time of loading;
>
> program-unload:code "U";job-number;program-id;time of unloading.
>
> The records are written in strict chronological sequence".

The instance of the data that will be used for the transform application of the reductions and decompositions used by the data method will be generated by the following SYSTEMLOG program.

```
Program Systemlog;
   type D = seq of job records;
        O = seq of job reports;
        (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 {Systemlog}.
```

As in the previous section we will skip the steps of the method that takes us from the first step to the decomposition when Transform need to be defined. We need now to specify the programs Transform and Process. As before, the change needed by the Transform function is trivial.

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

We shall look now for a Process program which is gsm solvable at this level of decomposition. We use the abstraction that the input tape is a sequence of job records (seq of job records). We are therefore, overlooking details such as the fact that there are various types of records associated to the same job (log-on records N, log-off records F, program-load records L, and program-unload records U) and that the records refering to the same job are not contiguous in the tape. All the same, we will deal

with the problem, one job record at a time (the same way we looked at a telegram at a time in the previous example). We will produce, as before, a gsm solvable program. Figure 18 illustrates the procedure just described in a graphical form.

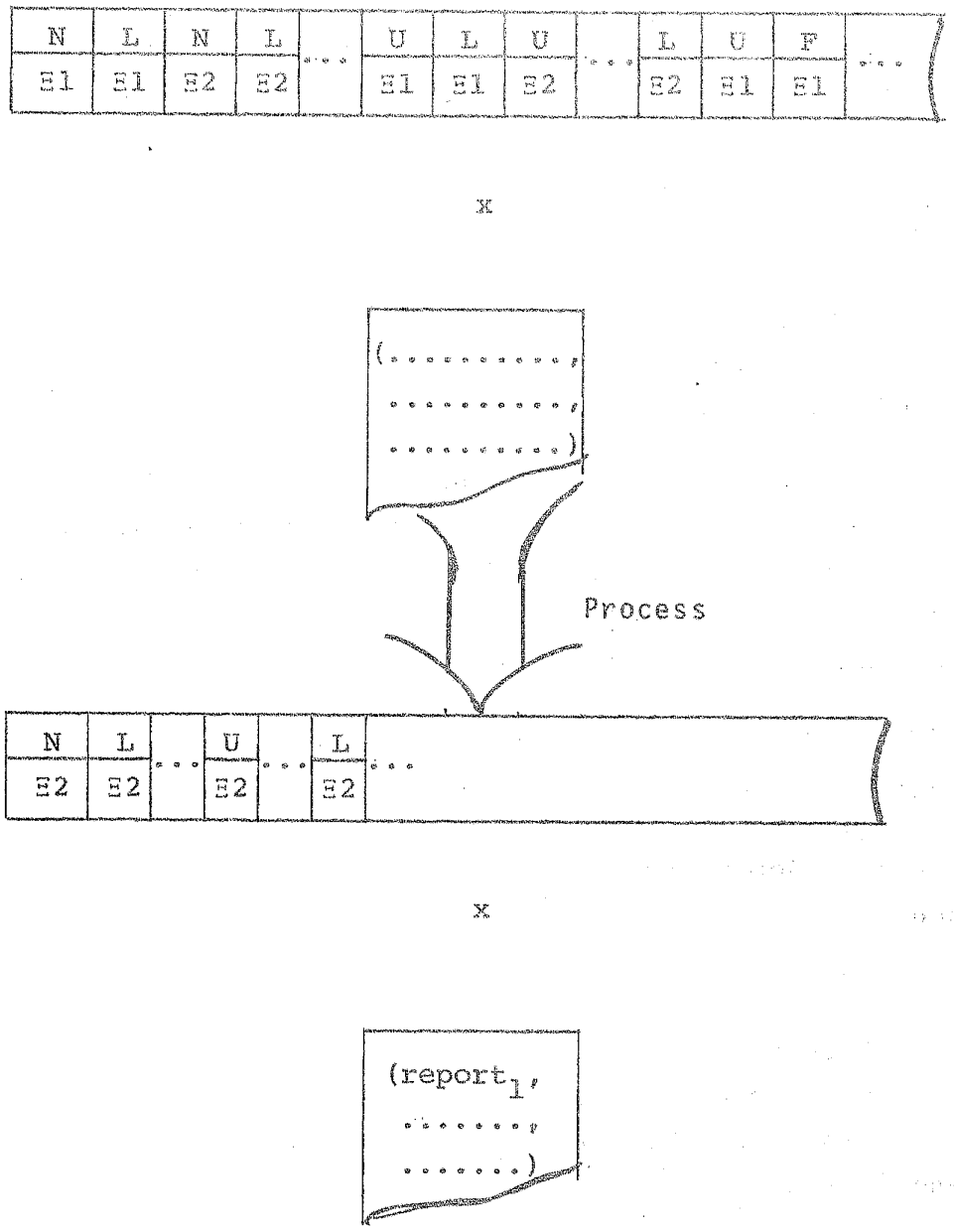


Figure 18

The core of the Process program scans the tape collecting and processing information about one job, while constructing an output tape with the information about this job supressed.

One possible schematic version for process could be the following:

```
Procedure Process;
      begin
           x6 ← Λ;
           get (first register in x5);
           get (another register from x5);
           while (x5 not empty) do
                begin
                     if job-number of reg.=job-number of first reg.
                     then
                         analysis of job-report
                     else
                         x6 ← append(x6,register);
                     get (another register from x5)
                end
      end {Process};
```

Termination and correctness are as simply dealt with as before as soon as finished is defined as in the previous cases.

9. The Matrix Transposition Problem

This last example completes the number of structure clashes problems presented in Jackson. In his terminology this problem is called an ordering clash problem.

We will state the problem in the following simple way. One tape contains the elements of an mxn matrix recorded by line. The transpose problem will display the elements of the same matrix by column. In other words, we will find the transpose A^T of a matrix A.

As before, we will define a program TRANSPOSE 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 Transpose;
   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:O;
      begin
           x ← copy(d);
           P;
           y ← copy(y)
      end {Transpose}.
```

The nucleus of the Process problem looks very much like the system log basic problem with the difference that when scanning the input tape for a column we can determine exactly where each element of the column is located. Figure 19 illustrates the approach taken in this case.

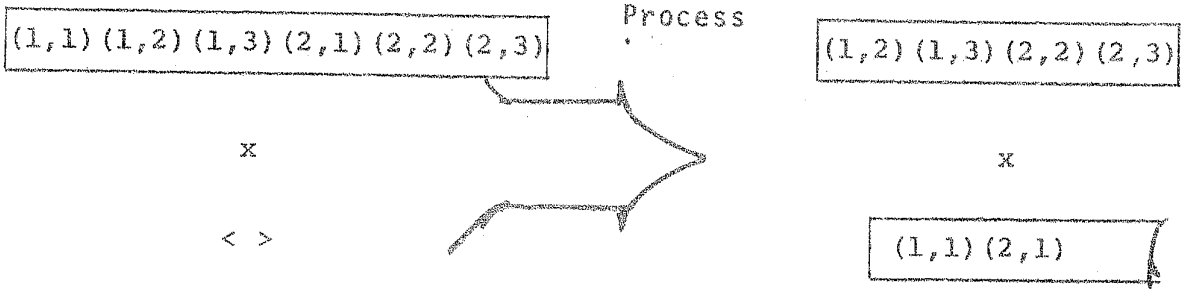


Figure 19

One possible schematic version for Process is presented as follows:

```
Procedure Process;
   begin
        get (first element in x5);column ← first element of x5;
        x6 ← Λ;
        get (x5);
        while(x5 not empty) do
             begin
                  if(element x5 is in the same column as the first)
                  then
                       column ← append(column, element)
                  else
                       x6 ← append(x6,element);
                  get (one element of x5)
             end
   end {Process};
```

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. When choosing this criterion for exemplification we realized that althrough 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 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 5 to 8 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 arrive 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.

Appendix 1

Glossary of Functions

copy - copies the arguments produce another instance of the type

first - exhibits the first element of a sequence, that is, first $(<l_1,l_2,\ldots,l_n>) = l_1$ and the original sequence is not changed

get - exhibits and removes the first element of a sequence

last - exhibits the last element of a sequence, i.e., last $(<l_1,l_2,\ldots l_u>) = l_n$

make - constructs an unitary sequence, i.e., make$(l_1)=<l_1>$

tail - constructs a sequence by removing the first element of the original sequence, i.e., queue$(<l_1,l_2,\ldots,l_n>)=<l_2,\ldots,l_n>$

recombine - constructs an ordered pair from two given elements, i.e., recombine$(l_1,l_2) = (l_1;l_2)$

Appendix II

```
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,x5,x6:D;
       y,o,y5,y6:O;
       x1,y1,x4 :DxO;
       y3,x3,x2,y2:(DxO)*;
       minimum : Aobjects;
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};
```

```
Procedure Transform:DxO;
        begin
             x5 ← x4.i;
             y5 ← x4.r;
             Process;
             y6 ← append(y5,minimum);
             Transform ← recombine(x6,y6)
        end {Transform};
Procedure Update:(DxO)*;
        begin
             y3 ← x3;
             x4 ← last(x3);
             x4 ← transform(x4);
             update ← append(y3,x4)
        end {update};
Procedure P2;
        begin
             x3 ← x2;
             while length(last(x3).i)≠ 0 do
                  x3 ← update(x3);
             y2 ← x3
        end {P2};
Procedure P1;
        begin
             x2 ← make(x1);
             P2;
             y1 ← last(y2)
        end {P1};
Procedure P;
        begin
             x1.i ← x;
             x1.r ← Λ;
             P1;
             y ← y1.r
        end {P};
        begin
             x ← copy(d);
             P;
             o ← copy(y)
        end {sort}.
```

Bibliography

1. Jackson, M.A. Principles of Program Design. London: Academic Press, 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 and System Design. Yourdon Press, 1978.

4. Chand, D.R., Yadav, S.B. Logical Construction of Software. CACM, 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. Vol.10, Waterloo, 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 (Proc. of 5th EMCSR, Vienna, 1980): 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. S E. 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., and 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.