# THE DATA TRANSFORM PROGRAMMING METHOD: AN EXAMPLE FOR FILE PROCESSING PROBLEMS

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

*Departamento de Informática, PUC/RJ-Rio de Janeiro,Brasil
+Department of Computer Science,University of 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 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.

## 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. Thus, 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 had little or no impact in everyday data processing apllications 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 handle only 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[4] established a correspondence between the class of programs available to treatment by his method and the



0270-5257/84/0000/0388$01.00©1984 IEEE

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 machin). 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[5], 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 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 of data processing problems rather than 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[5] 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[6] were instrumental for the formulation and improvement of the original ideas in Cowan and Lucena[5]. Some authors have proposed a programming approach where the transition between successive versions of a program is done according to formal rules called program transformation (see, for instance, [12], [13], [14] and [15]). According to this approach programs are considered as formal objects which can be manipulated by transformations 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[7]. The method reduces the original problem to a set of sub-problems that can be solved through the direct application of Jackson's method, thereby producing 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 and 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). The data transform method is presented through some concepts in problem solving theory and theory of data types, associated with informal arguments.

## THE DATA TRANSFORM METHOD

Programs solve problems. According to Veloso[6] 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 specifications for D, for O and for q, to construct a program P such that (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 D to a representation of o O). As it was pointed out in the introduction, 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 that 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 entities. This approach, of course, became a standard procedure in many programming methodologies but is not very common in the context of data-direct programming. The strategy for program derivation through the data transform method consists of applying the concepts of problem reduction and decomposition together with Hoare's general data type construction mechanisms[7]. Problem reduction and decomposition are applied in a way which will leave us with a set of Jackson solvable problems at 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 $P=<D,O,q>$ and write $P \overset{\Gamma}{\rightarrow} P_1$ if we can define
a unary function insert, ins: $D \rightarrow D_1$ and
a unary function retrieve, retr: $O_1 \rightarrow O$
such that the program defined by

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

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 $D_1xO_1$, P is a solution to $P$ (a total function from D to O), $P_1$ is a solution to $P_1$ (a total function from $D_1$ to $O_1$), and that the functions ins and retr need to be defined in such a way that the condition expressed in (2) is satisfied.

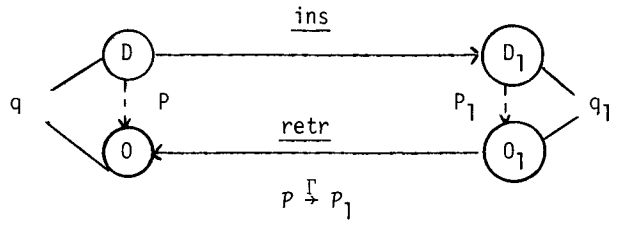


Figure 1

The first step of the data transform method consists of defining both $D_1$ and $O_1$ as the cartesian product of D and O; ins such that $\underline{ins}(d)=(d,o_0)$ for some $o_0 \in O$; retr such that $\underline{retr}(d,o_n)=o_n$. In other words, the reduction through ins and retr makes use of the data type constructor cartesian product (record) which is one of the 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.

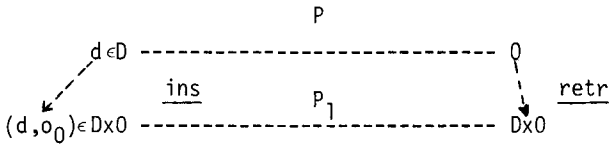


Figure 2

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 $O_1^*$ 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, namely, make: builds an unitary sequence from a given argument; last: returns the last element of the sequence. Figure 1 would now be replaced by the situation pictured in Figure 3.

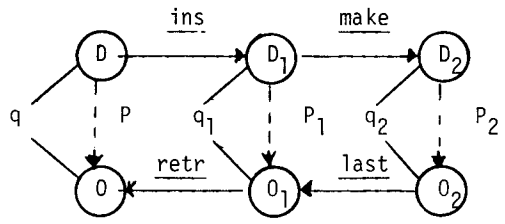


Figure 3: $P \overset{\Gamma}{\rightarrow} P_1 \overset{\Gamma}{\rightarrow} P_2$

The diagram in Figure 2 can now be expanded in the following way

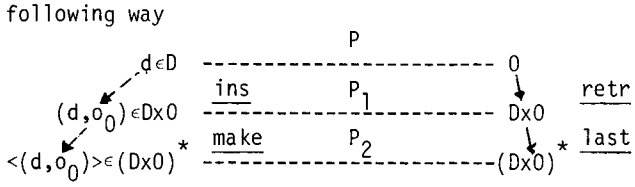


Figure 4

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 [6]. If we take the problem $P_2=<D_2,O_2,q_2>$, a n-ary decomposition $\Delta$ of $P_2$, $P_2 \rightarrow \Delta$, consists of

i) n functions $\underline{decmp}_i$: $D_2 \rightarrow D_2$, i=1,...,n;

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

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

iv) a unary relation $\underline{easy}$ $D_2$

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

$$P_2(d_2)=\begin{cases} \underline{immd}(d_2) & \text{if } \underline{easy}(d_2) \\ \underline{combine}(d_2,\underline{sol}_1\ \underline{decmp}_1(d_2),\ldots, & \\ \quad \ldots,\underline{sol}_n\ \underline{decmp}_n(d_2)) & \text{otherwise} \end{cases} \quad (3)$$

where $\underline{sol}$ stands for the part of the solution of $P_2$ $\underline{contributed}$ by each decomposition, defines a solution for problem $P_2$. Intuitively, if the problem is simple ($\underline{easy}$), that is, gsm computable, decomposition is not necessary and we have a direct ($\underline{immd}$)solution. Otherwise the solution for $P_2$ is obtained through the combination ($\underline{combine}$) of the solutions ($\underline{sol's}$) to the programs $P_2^1, P_2^2,\ldots,P_2^n$ which correspond to the solutions. The decomposition $\underline{process}$ 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.

## 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 parameters.

The program schema below defines the family of programs (in the sense of [8]) that can be obtained by the data transform method as specialized for file processing programming, when we have $\underline{one}$ application of the first step of the method followed by $\underline{one}$ application of the second step.

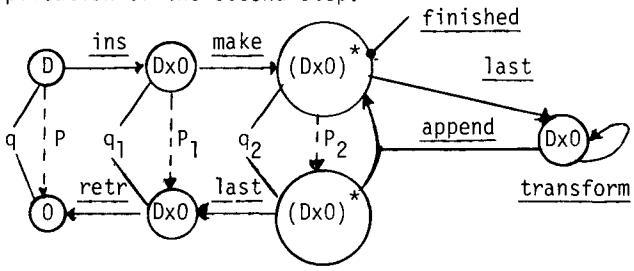


Figure 5: Diagram for file processing problems solution by the data transform method.

The notation used in Figure 6 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 the Appendix. In the program Schema the selectors i and r simulate the function $\underline{ins}$ and $\underline{retr}$ and the symbol $\Lambda$ stands for the null sequence. The program schema only creates an instance of the input data to allow the application of the method.

The function $\underline{update}$ for the class of file processing problems, can be defined as

$\underline{update}(x_3)$ = $\underline{append}(x_3, \underline{transform}(\underline{last}(x_3)))$

where transform is a function from DxO to DxO

```
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 6: Program Schema for File Processing Programming through the Data Transform Method

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: $(DxO)^* \times (DxO) \rightarrow (DxO)^*$

and append $((p_1,\ldots,p_n),p) = \langle p_1,\ldots,p_n,p\rangle$

A Correctness Criterion for the Method

We first state a termination condition for the program schema displayed in Figure 6. We have:

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

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

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

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

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

Intuitively, smllr guarantees that in each step transform contributes some more for the solution. The smllr relation, which is well founded, characterizes the $\Lambda$ 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.

## THE SORTING PROBLEM

We have selected the sorting problem as our 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 Sections 2 and 3. It would be harder to do the same in a short paper with a problem with a more complex definition.

Let A be a totally ordered set, $d=<a_1,a_2,\ldots,a_n>^{\chi} \in D$ a finite sequence of elements from A and $o=<b_1,b_2,\ldots,b_n>^{\chi} \in O$ 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 6 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 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)*;
   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

---

*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.

Figure 7:

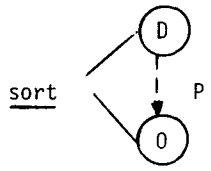


Figure 7: Sort

We are now ready to apply the first step of the method. It is graphically represented in Figure 8. We want now to model the situation expressed in Figure 8, through a program P.

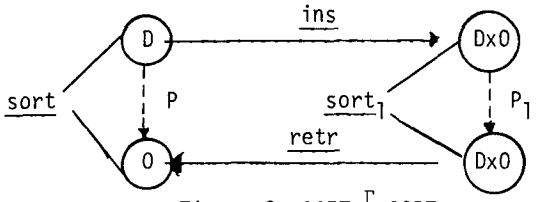


Figure 8: SORT $\Gamma$ $SORT_1$

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 3.

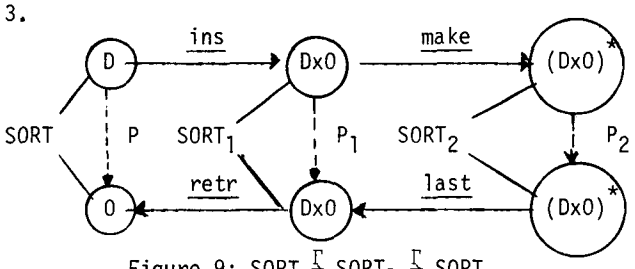


Figure 9: SORT $\Gamma$ $SORT_1$ $\Gamma$ $SORT_2$

$P_1$ 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 correctness of the program (in particular termination).

The next step is a first decomposition of $P_2$.Recall that we are only interested here in solving problems that can be classified as file processing applications. For this purpose the following decomposition can be proposed. The notation we use is widely employed in the literature about abstract data types[9]. It bears a natural similarity wity 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 9. It should be noted that Figure 10 contains a diagram which is typical of file processing programs.

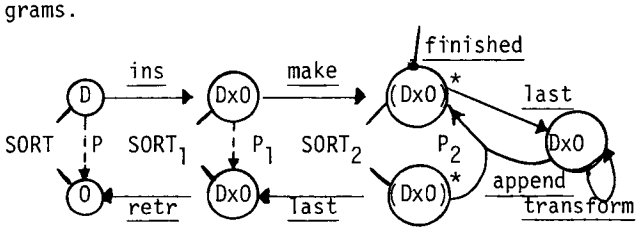


Figure 10: SORT→SORT$_1$→SORT$_2$↑(last,transform,append)

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 11).

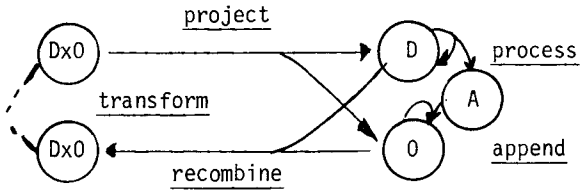


Figure 11

This decomposition step can be thought of as being coupled to the diagram in Figure 10 (note the dots to the left of the diagram in Figure 11). 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. The functions first and tail have their usual meaning when applied to sequences (see glossary in the Appendix).

```
Procedure transform(x4:Dx0):Dx0;
   var x5,x6:D;
       y5,y6:0;
       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}
```

We need now to specify the predicate finished so as to satisfy the correctness conditions presented in section 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:

$$\forall x_3:(Dx0)^*,\ \underline{finished}(x_3) \leftrightarrow \underline{length}(\underline{last}(x_3).i=0)$$

To satisfy the correctness criterion expressed in section 2 we need to define a well founded relation smllr. We propose the following:

$$(\forall d_1,d_2):D, \underline{smllr}(d_1,d_2) \leftrightarrow \underline{length}(d_1) < \underline{length}(d_2)$$

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). We also note that smllr has been defined via "<", thus being well founded, 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 may have 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.

## CONCLUSIONS

This paper introduced the data transform programming method and applied it to the solution of a simple and classical programming problem. The example meant to compare our approach with Jackson's method, since his method cannot directly solve problems of the class we have dealt with. We have shown the solution of a toy application of file processing programming, which often deals with far more complex situations. The full power of the method can be better assessed through its applications to larger problems. When we deal with aspects of real-world data processing problems such as making verification accessible to practitioners, providing programming standards for large programming teams and enhancing the quality of documentation and maintenance practices of the method can be fully appreciated. The present work is a major extension of the work published in [5]. Many interesting developments of the present work are in sight. A software environment to support the method proved to be an interesting feature. The works by Coleman, Hughes and Powell[10] and Logrippo and Skuce[11] also follow this general direction although they are restricted to the automation of Jackson's basic method, which we extend here.

We believe, as Cheatham[13], that for a large, long-lived software project, the existence of an ac-

curate, readable and executable 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 at an efficient version for the solution at hand through a set of well defined program transformations. Some interesting theoretical results have also been produced[16]. 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, a recursive solution can be contemplated) as well as 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

### Glossary of Functions

copy - copies the arguments and produces another instance of the type

first - exhibits the first element of a sequence, that is, first($<a_1,a_2,\ldots,a_n>$) = $= a_1$ 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($<a_1,a_2,\ldots,a_n>$) = $a_n$

make - constructs an unitary sequence, i.e., make($a_1$) = $<a_1>$

tail - constructs a sequence by removing the first element of the original sequence, i.e., tail ($<a_1,a_2,\ldots,a_n>$) = $<a_2,\ldots,a_n>$

recombine - constructs an ordered pair from two given elements, i.e., recombine($a_1,a_2$) = $(a_1;a_2)$

## BIBLIOGRAPHY


[1] M.A. Jackson, Principles of Program Design. London: Academic Press, 1975.

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

[3] - E. Yourdon, L.L. Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and System Design. New York: Yourdon Press, 1978.

[4] J.W. Hughes, "A Formalization and Explanation of the Michael Jackson's Method of Program Design", Software-Practice and Experience. V. 9, 1979.

[5] D.D. Cowan, J.W. Graham, J.W. Welch, C.J. Lucena, "A data-directed approach to program construction", Software-Practice and Experience. Vol. 10, 1980.

[6] P.A.S. Veloso, S.R.M. Veloso, "Problem decomposition and reduction: applicability, soundness, completeness"; R. Trappl. J. Klir, F. Pichler (eds); Progress in Cybernetics and Systems Research, Vol. VIII, Washington, DC Hemisphere Publ. Co. 1980.

[7] C.A.R. Hoare, "Notes on data structuring". ; Dahl, O., J., Dijkstra, E.W., Hoare, C.A.R. (eds), Structured Programming. Academic Press: 1972.

[8] D.L. Parnas, "Designing software for Ease of Extension and Contraction". IEEE Trans. Software Engineering, Vol. SE-5, nº 2, 1979.

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

[10] D. Coleman, J.W. Hughes, M.S. Powell; "A Method for the Syntax Directed Design of Multiprograms". IEEE Trans. on Software Engineering, Vol SE-7, nº 2, 1981.

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

[12] M. Broy, P. Pepper; "Program Development as a Formal Activity". IEEE Transactions of Software Engineering Vol SE-7, nº 1, 1981.

[13] T.E. Cheatham, G.H. Holloway and J.A. Townley , "Program Refinement by Transformation". Proceedings of the 5th International Conference on Software Engineering, 1981.

[14] S.L. Gerhart,"Correctness-Preserving Program Transformations".Proc. ACM Symp. on Principles of Programming Languages, 1975.

[15] J.J. Arsac,"Syntatic Source to Source Transforms and Program Manipulation".CACM, Vol 22, nº 1, 1979.

[16] R.C.B. Martins, "The Data Transform Method" (in Portuguese). Ph.D. Thesis, Computer Science Department, Pontifícia Universidade Católica, Rio de Janeiro, 1983.