Data-Driven Prototyping and Implementation of File Processing Programs Using the Data Transform Method

C.J. Lucena*
R.C.B. Martins*
D.D. Cowan**

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

Research Report CS-84-28
September 1984

*Pontificia Universidade Catolica, Rio de Janeiro, Brazil

**Dept. of Computer Science, University of Waterloo,
Waterloo, Ontario, Canada, N2L 3G1

Data-Driven Prototyping and Implementation of File Processing Programs Using the Data Transform Method

Lucena, C.J., Martins, R.C.B.,

Pontificia Universidade Catolica,
Rio de Janeiro, Brazil

Cowan, D.D.,

University of Waterloo,
Waterloo Ontario Canada

ABSTRACT

This paper describes a substantial extension to the method proposed by Jackson whereby a program is designed from a description of its input and output data. By extending the concepts of input and output both structure-clash and backtracking problems can be integrated into the basic Jackson method. This technique is called the Data Transform Method [DTM]. The paper describes how DTM may be applied to file-processing problems and shows that a program schema results which can be used to assist with the semi-automatic production of software. Finally a specific problem involving backtracking is developed to illustrate how DTM is used in practice.

Keywords: data-driven programming, data transformations, data transform method, file processing, Jackson method, program design, program prototyping

"I do not know why, but I have never seen a machine that, however perfect in the philosopher's description, is perfect in its mechanical functioning. Whereas a peasant's billhook, which no philosopher has ever described, always functions as it should..." Umberto Eco "The Name of the Rose"

1. Introduction

The Data Transform Method [DTM] is an extension of the ideas of Jackson [1] and Warnier [2] which use the specifications of the input and output data to assist in determining the structure of a program.

Preliminary descriptions of the data transform method (DTM) together with accompanying examples have already been published in the literature. In [3] the DTM is described in conjunction with a classical programming problem, the sorting problem, to illustrate the concepts and show how to handle a problem that involves some backtracking. In [4] attention was centered on the solution of some programming examples, classified by Jackson [1] as structure-clash problems.

The goals of the previous publications have been to show how the limitations of a powerful technique such as Jackson's basic method can be overcome and how his basic method can be used as the basis for a far more general program-design approach, namely the DTM approach.

The present paper describes again the ideas associated with the DTM approach but uses a more concise formalism and examines the question of how prototypes of programs developed through DTM can be transformed into production programs. As in previous publications the

presentation is based on a typical file processing example; namely the problem of delimited strings [1].

The DTM uses the idea of applying data transformations to the problem data as they are specified at the problem statement level, and so uses the notions of problem reduction and problem decomposition as they are defined in the theory of problems [5]. The DTM transforms the space of solutions of the original problem into a space in which the problem can be decomposed into a set of sub-problems which are solvable by Jackson's basic method. The transformation steps are standard and put the problem in a canonical form that can be described as being an executable specification. The prototype constructed in this manner is a program design which is correct by construction.

The DTM extends the applicability of the Jackson basic method, since important problems that are not handled directly through the method (such as backtracking and structure clashes, can now be programmed through a uniform approach. The following results, which are stated here without proof, have been proven [6] and serve to characterize the relationship between the DTM and Jackson's basic method:

(i) The class of Jackson-solvable problems is properly included in the class of g.s.m.-(general sequential machine) solvable problems.

(ii) The class of g.s.m.-solvable problems is properly included in the class of DTM-solvable problems

(iii) The class of structure-clash problems and the class of backtracking problems are properly included in the class of DTM solvable problems.

(iv) The class of pushdown-transducers solvable problems such as the recognition of ww^R, is included in the class of DTM-solvable problems.

It often happens that, a loss in efficiency accompanies the increase in generality. Although a DTM canonical solution is meant to be an executable specification or rapid prototype for a file processing problem it is important to stress that the specification as produced also allows for the derivation of an efficient final implementation. While the prototype can be developed with computer assistance, through a dialogue between a simple software system and the programmer, the process of generating an efficient implementation, although it can be guided in part by a series of well-defined steps, depends ultimately on the particular application under consideration. Section 4 presents the derivation of a practical implementation for the prototype developed in Section 3.

Before moving to Section 2, where the DTM approach is presented through a concise formalism, an abstract machine model of DTM is described which provides an early intuitive description of the method.

File-processing programs can be presented in an abstract manner as a program P which accepts as input a file of elementary data items and produces a file of entities called a report. A picture showing one input file with each elementary item designated by a subscripted e and a report designated by l_n is shown in Figure 1. Even multiple input and output files can be viewed in this manner.

This abstract view of a program P can be modified so that a new program P_1 accepts the elementary data items but its output is now a file or sequence of partial reports. The file of partial reports is created by recording as output each step in the report creation where the last partial report is the complete report or output. Such a picture is shown in Figure 2 where each partial report is designated by a subscripted l. It is clear that the original program could be obtained from P_1 by the simple step of discarding all but the last element of the output file.

Another level of abstraction is introduced into the DTM model where a program P_2 processes as input a stream of data called a paired-I/O-unit and produces as output a paired-I/O-unit. This abstraction is shown in Figure 3. The paired-I/O-units each contain the input and output which exists at a point in the operation of the program. Figure 3 only shows the initial input and final output and so the paired-I/O-units show a full input of elementary items and an

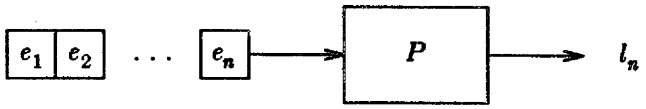

Figure 1

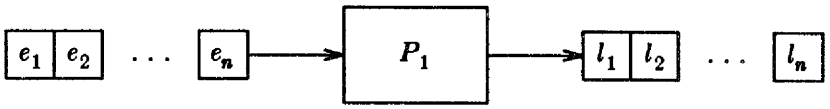

Figure 2

empty output and a full output of partial reports and an empty input. In other words, before the programs start operation the output is empty and after the program is finished the input is exhausted or empty. This pairing of input and output into paired-I/O-units provides a symmetry between input and output data structures and hence is the way the DTM avoids the problem of structure clashes. Program P_1 should be derivable from program P_2 since P_1 would just discard one-half of the paired-I/O-unit.

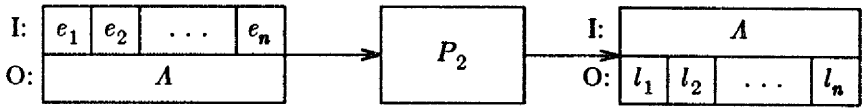

Figure 3

One final level of abstraction is necessary in the DTM. This new level is shown in Figure 4 where the program P_3 is introduced and is shown accepting a paired-I/O-unit as input and producing a sequence of paired-I/O-units as output. The method of generating the output is also shown in the Figure. Program P_3 is decomposed into a number of programs P_{3_i} which incrementally process the paired-I/O-unit at the input and gradually transform it into the sequence of paired-I/O-units at the output.

At each stage P_{3_i} the input is contcatenated with the results of that stage to produce the output. This means that the input and output are both sequences containing a complete history of the computation to that point and the final output is a complete history of the computation. Since there is now a complete history of the computation present at each stage, there is no need to backtrack. It should be clear that program P_2 can be derived from P_3 by retaining only the last member of the output sequence.

The DTM as shown in the Figures is expanded in the next few sections. A detailed description of the approach and the coding of problems involving backtracking is presented.

The DTM is primarily mechanical in nature except for the construction of the components which constitute P_3. Hence software tools can be developed which should allow computer-aided development of software.

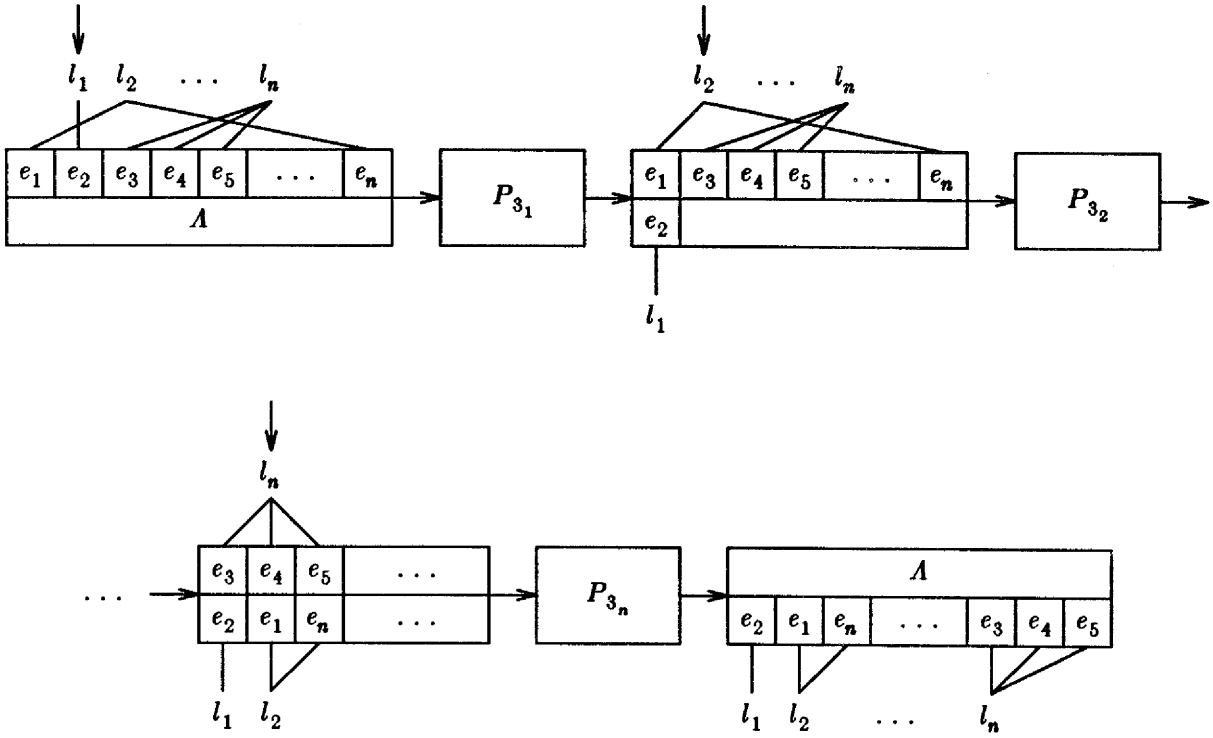

Figure 4

2. The Data Transform Method

DTM has been described in earlier versions of the present formalism [3,4]. When proposing a new programming methodology, in particular one that claims to extend the power of an existing method (Jackson's method), it is important to characterise precisely the problem-solving approach on which it is based. This can be done when we think about programs as solving problems and use concepts from the theory of problems [5].

A problem is a structure $P = <D,O,q>$ where the elements of D are the problem data, the elements of O are the solutions and q is a binary relation between D and O representing the problem's conditions.

A program solves a problem P if it defines a relation between D and O such that

$$\forall d \in D \; q(d;f(d)) \qquad (1)$$

A solution space S for a problem is the set of all solutions, that is,

$$S(P) = \{f : D \rightarrow O \mid f \subseteq q\}.$$

A derivation of a program using DTM requires that input ($d \in D$) and output ($o \in O$) specifications be given and a program be constructed in which equation (1) holds.

Jackson's basic method tries to determine initially a direct mapping between a data structure for d and a data structure for o, where $(d;o) \in p$. A difficulty occurs in solving certain classes of problems because backtracking and structure clashes arise. DTM starts by expressing the abstract notions of $d \in D$ and $o \in O$ instead of attempting to look for a data representations for these two entities.

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 [7].

Program reduction and decomposition are applied in a way which will leave us with a set of Jackson-solvable problems.

The reduction of a problem $P_0 = <D_0,O_0,q_0>$ to a problem $P_1 = <D_1,O_1,q_1>$ is a pair of functions $P_{01} = \{ins,ret\}$ where *ins* (for insert) is an unary function from D_0 to D_1, $ins:D_0 \rightarrow D_1$ and *ret* (for retrieve) is an unary function from O_1 to O_0, $ret:O_1 \rightarrow O_0$, such that, for all $f_1 \in S(P_1)$, $ret.f_1.ins \in S(P_0)$. A sufficient condition for the pair $\{ins,ret\}$ to be a reduction of P_0 to P_1 is that $ret.q_1.ins \subseteq q_0$, where

$$ret.q_1.ins = \{(d_0;o_0) \in D_0 \times O_0 \mid \exists\ o_1 \in O_1,\ o_0 = ret(o_1) \wedge (ins(d_0);\ o_1 \in q_1\}.$$

The first step of DTM consists of defining D_1 and O_1 as the cartesian product of D_0 and O_0, that is, $D_1 = O_1 = D_0 \times O_0$, and the functions ins and ret such that

$$\forall d_0 \in D_0,\ ins(d_0) = (d_0;A)$$

and

$$\forall d_1 \in D_1\ \forall o_0 \in O_0,\ ret(d_0,o_0) = o_0$$

where A stands for a well defined element in O (usually the empty sequence). In other words the reduction through *ins* and *ret* makes use of the data type constructor cartesian-product (record) which is one of the three basic constructors proposed by Hoare [7]. The input and output data of P_1 have now, trivially, the same structure (independent of any chosen representations for d and o). Figure 5 illustrates this step.

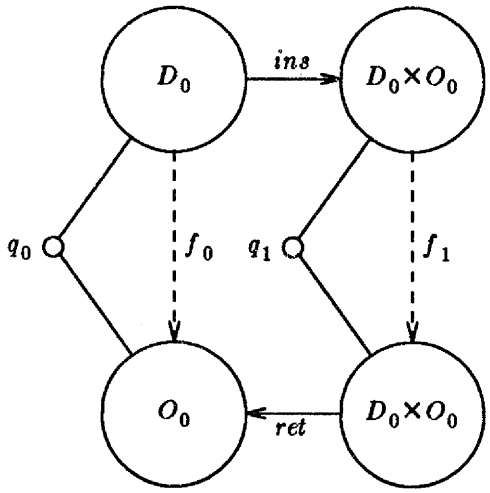

Figure 5

The second step of DTM consists of defining a new reduction $T_{12} = \{make,last\}$ of the problem $P_1 = <D_1,O_1,q_1>$ to a problem $P_2 = <D_2,O_2,q_2>$, where

$$D_1 = O_1 = D \times O$$

$$D_2 = O_2 = D_1^*;$$

make: $D_1 \rightarrow D_1^*$, builds a unitary sequence from a given argument, and *last*:$D_1^* \rightarrow D_1$, returns the last element of a sequence. This reduction is needed to avoid the backtracking problem since the sequence mechanism provides a history of the computation.

Figure 5 now takes the form presented in Figure 6.

We now describe a standard program schema that decomposes the problem P_2 into a series of smaller problems that can be solved through Jackson's basic method. The schema is illustrated in the diagram presented in Figure 7.

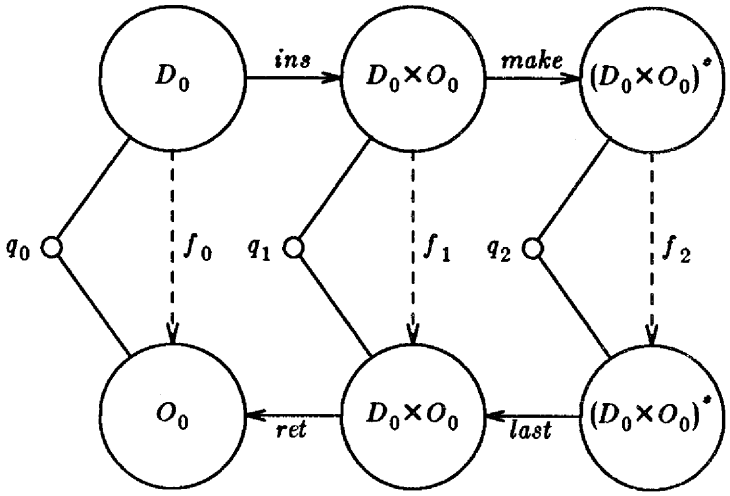

Figure 6

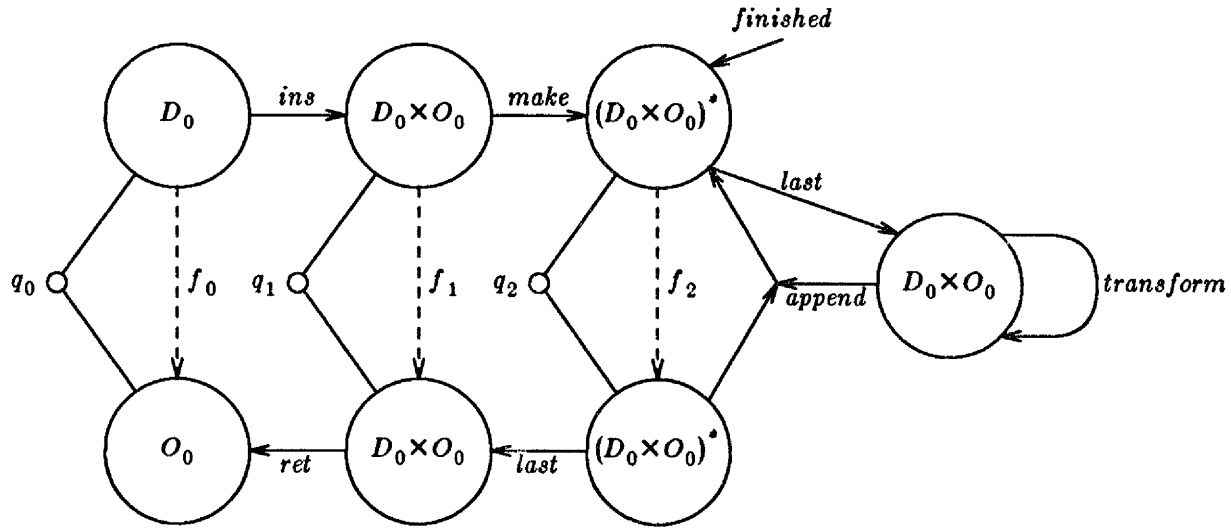

Figure 7

A Pascal-like pseudo-code description of the program schema described in Figure 7 is presented in Figure 8.

The function *update* for the class of file processing problems, is defined as

Program *schema*:

```
type D0 = seq of objects1;
     O0 = objects2;
     D0×O0 = record
                i:D0;
                r:O0
             end;
     (D0×O0)* = seq of D0×O0;

var x,d0:D0;
    y,o0:O0;

Procedure P2;

  var x3:(D0×O0)*;

  begin
    x3 ← x2;
    while not finished(x3) do
      x3 ← update(x3);
    y2 ← x3
  end{P2};

Procedure P1;

  var x2,y2:(D0×O0)*;

  begin
    x2 ← make(x1);
    P2;
    y1 ← last(y2)
  end{P1};

Procedure P;

  var x1,y1:D0×O0;

  begin
    x1.i ← x;
    x1.r ← Λ;
    P1;
    y ← y1.r;
  end{P};

begin
  x ← copy(d0);
  P ;
  o0 ← copy(y)
end{schema}.
```

Figure 8

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

where *transform* is a function from $D_0 \times O_0$ to $D_0 \times O_0$ which contributes to the solution of the problem.

The function *append* has the usual meaning of the operator normally associated with the type sequence, that is

$$append: (D_0 \times O_0)^* \times (D_0 \times O_0) \rightarrow (D_0 \times O_0)^*$$

and

$$append((l_1, \ldots, l_n), l) = (l_1, \ldots, l_n, l)$$

A correctness criterion for the program schema can be expressed as follows:

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

(ii) $\forall x_3 \in (D_0 \times O_0)^*$, $smllr(transform(x_3).i, x_3.i)$

(iii) *smllr* is a well-founded relation in $D_0 \times D_0$ such that any $d_0 \in D_0$ is in a finite *smllr* chain starting at Λ: $smllr(\Lambda, d_0^1), smllr(d_0^1, d_0^2) \ldots smllr(d_0^n, d_0)$ (this is the usual criterion for a 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 previous conditions. We can now state the partial correctness conditions for the class of programs.

(v) $\forall x_3 \in (D_0 \times O_0)^*$, $finished(x_3) \Rightarrow q_2(x_3, make(o_0.i, \Lambda)$

(vi) $\forall x_3 \in (D_0 \times O_0)^*$ $q_2(x_3, make(o_0.i, \Lambda)) \Rightarrow q(last(x_3).r, d_0)$

Intuitively, the relation *smllr* guarantees that in each step the *transform* function contributes something to 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_0 by the application of *ins* and *make* Condition (vi) ensures that the reduction from the original problem P_0 to P_2 is good, i.e., that the element from x_3 obtained by the application of *ret* and *last* is the solution to the original problem with input d_0.

3. The Delimited Strings Problem

Jackson in his book "Principles of Program Design" [1] proposes the following problem (pg. 130; problem 10). A program component is to be designed which will analyse a character string, recognizing and printing two substrings, S_1 and S_2. S_1 is terminated by a character "@", and S_2 by a character "&"; the complete string is terminated by a character "%".

On entry to the component two items of input data are available: the complete string, and a pointer or subscript which points to a current location in the string. S_1 is defined to be the substring whose first character is the character pointed at and whose last character is the terminating "@"; S_2 has as its first character the character following the terminating "@" of S_1, and its last character is a terminating "&". Either or both strings may be empty of non-terminating characters.

It is known that the complete string is terminated by a "%" sign within 100 characters of the current location on entry: this fact can, and should, be relied on. However, it is not known that correct substrings S_1 and S_2 are present. If both are present and correct, a report should be printed in the form

GOOD STRING
S_1=xxxxxxxxx@

S_2=yyyyyy&

otherwise the string should be printed from the current location up to the terminator, in the form

BAD STRING
CHAR-001=f
CHAR-002=g
.
.
.
CHAR-nnn=%

To design a solution to the problem using DTM, we need to characterize first the abstract domains of D_0 and O_0. This can be done through the next program shown in Figure 9.

```
Program delimited strings;

  type D0 = file of char;
       O0 = partial-report;
       D0×O0 = record
                 i:D0;
                 r:O0
               end;
       (D0×O0)* = file of (D0×O0);

  var x,d0:D0;
      y,o0:O0;

  begin
    x ← copy(d0);
    firstred(x,y);
    o0 ← copy(y)
  end.
```

Figure 9

The procedure *firstred* is responsible for the reduction $P_{01} = \{ins,ret\}$ and is listed in Figure 10.

```
Procedure firstred(x:D0; var y:O0);

  var x1,y1:D0×O0;

  begin
    x1.i ← x;
    x1.r ← Λ;
    secondred(x1,y1);
    y ← y1.r
  end;
```

Figure 10

The assignments making use of the selectors i and r play the role of the functions *ins* and *ret*. The procedure *secondred* is responsible for the second step towards placing a program design in a canonical form. The code which looks very much like the previous procedure is in Figure 11.

Procedure *secondred*(x_1:$D_0 \times O_0$;var y_1:$D_0 \times O_0$);

var x_2,y_2:$(D_0 \times O_0)^*$

begin
  $x_2 \leftarrow make(x_1)$;
  firstdecomp(x_2,y_2);
  $y_1 \leftarrow last(y_2)$
end;

Figure 11

At this point we have finished the reduction phase and are able to proceed to the first step in the decomposition which will be expressed by the procedure *firstdecomp* in Figure 12.

Procedure *firstdecomp*(x_2:$(D_0 \times O_0)^*$, var y_2:$(D_0 \times O_0)^*$);

var x_3:$(D_0 \times O_0)^*$

begin
  while not $length(last(x_3).i) = 0$ do
    $x_3 \leftarrow update(x_3)$;
  $y_2 \leftarrow x_3$
end;

Figure 12

The functions *update* and *transform* in Figure 13 complete the canonical form used to express the program design. In fact, in the procedure *firstdecomp* we have already characterized the predicate *finished* as *length* (*last* (x_3).i) = 0 as required in the problem statement.

The program designer can ignore the details of this method of program development. Only the notion of a DTM abstract machine needs to be kept in mind since all the steps described previously that led to the canonical form can be easily mechanized. In this case the details were left in the text for a better understanding of the underlying ideas.

The canonical form which provides the setting for the program design can already guarantee that the program will terminate as long as the procedure *process* guarantees that

$$length(x_6) < length(x_5).$$

The question of producing the report for the delimited strings problem is made quite simple if we choose to center our attention on a character as the basic logical entity.

The procedure *process* in Figure 14 will be assigned the task of constructing the report y_6 from the input x_5. The construction of this report will depend on the state of the report.

The construction of this report will depend upon the history of the computation which will be contained in a variable called *type*:

(i) if *type* has the value "#" then only characters other than @, & and % have been read.

(ii) if *type* has the value "@" then string S_1 is complete and S_2 has been partially read.

```
Function update(x3:(D0×O0)*):(D0×O0)*;

  var y3:(D0×O0)*;
      x4:(D0×O0);

  begin
    y3 ← x3;
    x4 ← last(y3);
    x4 ← transform(x4)
    update ← append(y3,x4)
  end;
end;


Function transform(x4:D0×O0):D0×O0;

  var x5,x6:D0;
      y5,y6:O0;

  begin
    x5 ← x4.i;
    y5 ← y4.r;
    process(x5,y5,x6,y6);
    transform.i ← x6;
    transform.r ← y6
  end;
```

Figure 13

(iii) if *type* has the value "&" then string S_2 is complete.

(iv) if *type* has the value "c" then the entire string has been read correctly.

(v) if *type* has the value "w" then the string is incorrect. The reading and removal of a character from x_5 determines the state of the report and its form.

Note that the functions *first* and *tail* used at the beginning of the procedure *process* have the usual meaning; *first* removes the first element in a sequence and *tail* returns the entire sequence without the first element.

The reader should note that if a direct mapping between the I/O data structures had been tried without previously generalizing the problem a backtracking situation would have arisen which is not Jackson-solvable. By transforming the problem using DTM a data-driven prototype has been produced because the procedure *process* can be constructed using Jackson's basic method. We illustrate this by presenting in Figure 15, 16 and 17 the I/O diagrams and the process diagram using Jackson's notation.

Since process guarantees that the length of x_6 is always strictly smaller than x_5, the termination of the program is always assured. Since the length zero is reached in the states "c" or "w" it is easy to develop an argument that shows the total correctness of the program design.

```
Procedure process(x5:D0;y5:O0;var x6:D; var y6:O);

 type state = ("#","@","&","C","w");
      partial-report = case state of
        "#" : record
                "@" : file of char;
                "%" : file of char
              end;
        "@" : record
                "@" : file of char;
                "&" : file of char;
                "%" : file of char
              end;
        "c" : record
                "@" : file of char;
                "&" : file of char
              end;
        "w" : record
                "%" : file of char
              end
        otherwise
      end;

 var type:state;
     letter:char;
     y5,y6:partial-report,

 begin
  letter := first(x5);
  x6 := tail(x5);

  {state determination}

  case letter of
    "@" : if type = "#"{undefined}
          then type := "@"
          else type := "#";
    "&" : if type = "@"
          then type := "C" {correct}
          else type := "#";
    "%" : if type = "&"
          then type := "c"
          else type := "w"; {error}
    otherwise
          if type ≠ "@" or "&" or "%"
          then type := "#";
 end;
```

```
{report generation}

case type of
  "#" : begin
          y6.@ := append(y5.c,letter);
          y6.% := append(y5.%,letter)
        end;

  "@" : begin
          y6.@ := y5.@;
          y6.& := append(y5.&,letter);
          y6.% := append(y5.%,letter)
        end;

  "c" : begin
          y6.@ := y5.@;
          y6.& := y5.&
        end;

  "w" : begin
          y6.% := y5.%
        end;
  otherwise
 end;
end;
```

Figure 14

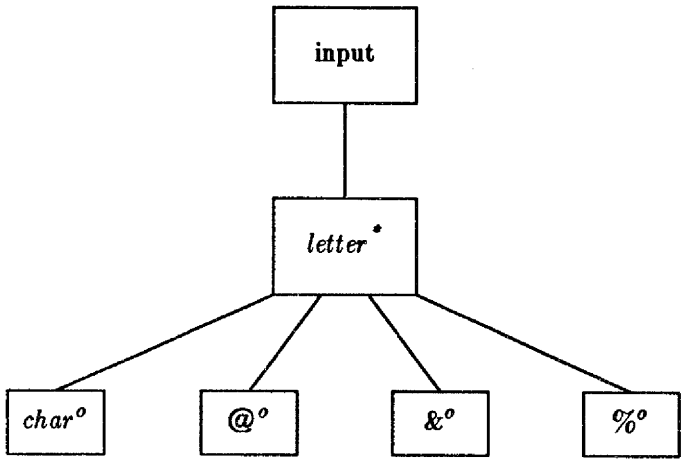

Figure 15

4. Derivation of an Implementation from the Program Design.

It is not the goal of this paper to elaborate on the advantages of rapid data-driven prototyping. This issue has been extensively discussed in the literature, and has been one of the major motivations for the formulation of DTM. In the present section it is illustrated how a production program can be produced from a DTM design.

Initially Jackson's notation is used to outline the typical prototype produced by DTM through the steps described in the previous section.

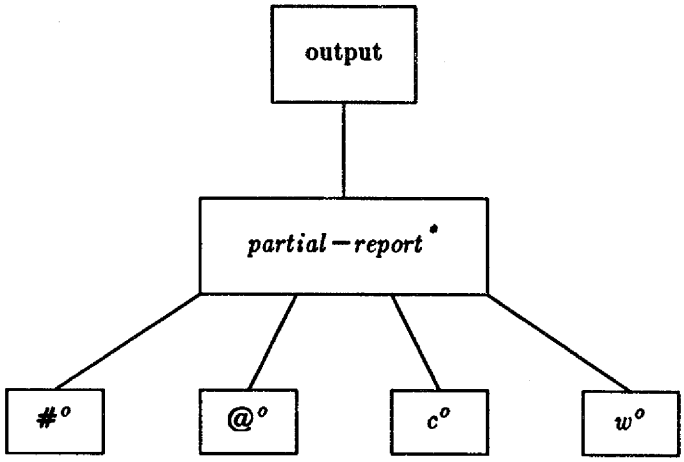

Figure 16

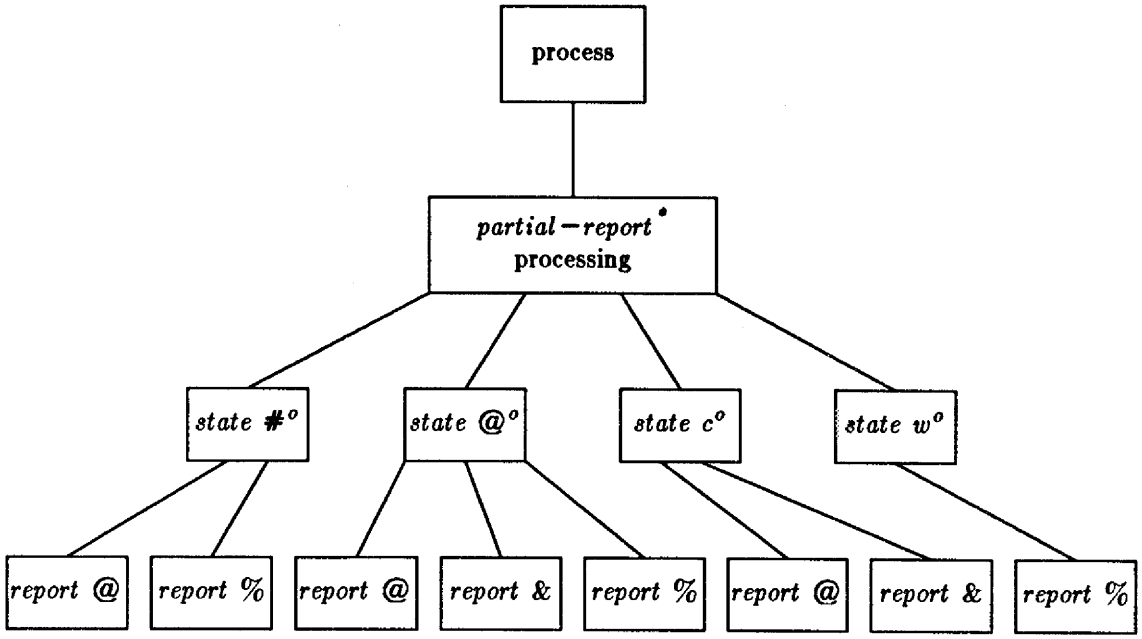

Figure 17

Looking at Figure 18 it appears that the program can be substantially reduced in size by removing the overhead that was necessary to simulate a DTM abstract machine.

Since the two reductions defined by {*ins*, *ret*} and {*make*, *last*} always occur in the program schema (Figure 8), the procedures *firstred* and *secondred* can be combined into a single procedure called *reduce*. The new version of the schema with this transformation is shown in Figure 19.

Before proceeding it is important that the reader be aware that the method is not being simplified or optimized but rather the implementation of the method for a Pascal-like machine is being developed. If, for instance, a fundamentally different base-machine was available, say a functional machine, the method would still hold but implementation procedures would look quite different.

In file-processing applications the history of the computation is often not required since the program is usually handling the last component of the sequence. Since after testing a data-driven prototype for a file-processing application we normally cease to be interested in the sequence of

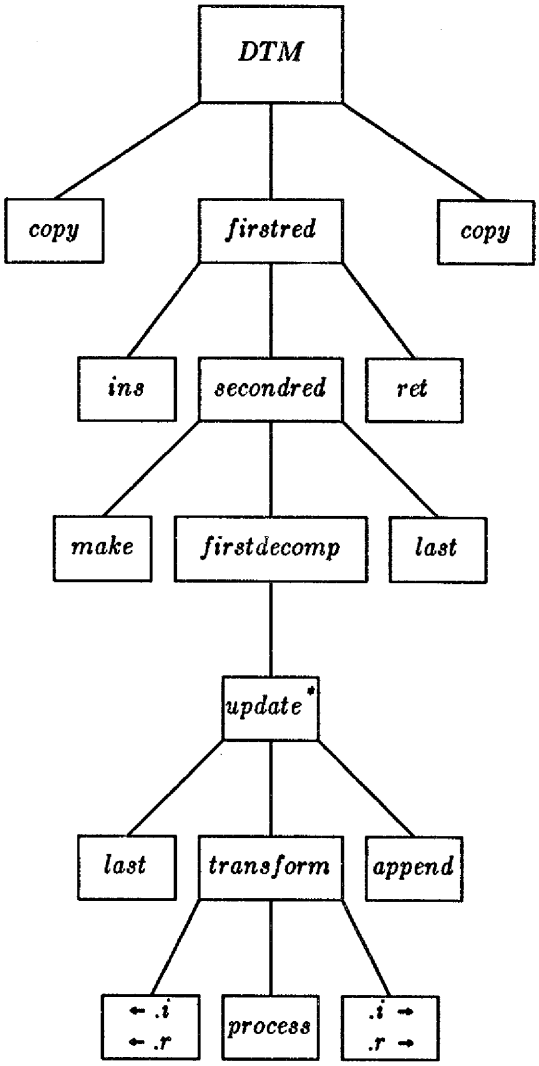

Figure 18

computation produced by the executable specification, we note that the procedures *firstdecomp* and *update* may be merged and that the type $(D \times O)^*$ can be eliminated. These modifications lead to the program schema in Figure 20.

Finally the invocation of the procedures *update* and *reduce* can be replaced with the body of the procedures in the main program body. The program schema now takes the form shown in Figure 21.

Using this latest version of the program schema and replacing the general names of the variables by the names of variables used in the solution of the problem of delimited strings presented in Section 3, the schema is as shown in Figure 22.

```
Program schema;

 type D0 =
      O0 =
      (D0×O0) = record
                  i:D0;
                  r:O0
                end
      (D0×O0)* = seq of (D0×O0)

 var d0,x:D0;
     o0,y:O0;
     x4,x1,y1:D0×O0;
     y3,x2,y2,x3:(D0×O0)*;

 procedure update(D×O)*;
   begin
     y3 ← x3;
     y4 ← last(x3);
     x4 ← transform(x4);
     update ← append(y3,x4)
   end {update};

 procedure firstdecomp;
   begin
     x3 ← x2;
     while not length(last(x3).i) = 0 do
       x3 ← update(x3);
     y2 ← x3
   end {firstdecomp};

 procedure reduce;
   begin
     x1.i ← x;
     x1.r ← Λ;
     y2 ← make(x1);
     firstdecomp;
     y1 ← last(y2);
     y ← y1.r
   end {reduce};

 begin
   x ← copy(d0);
   reduce;
   o0 ← copy(y)
 end {schema}
```

Figure 19

```
Program schema;

  type D0 =
       O0 =
       (D0×O0) = record
                   i:D0;
                   r:O0
                 end;

  var d0,x:D0;
      o0,y:O0;
      x3,x2,y2:(D0×O0);

  procedure update;

    begin
      x3 ← x2;
      while not length(x3.i) = 0 do
        x3 ← transform(x3);
      y2 ← x3
    end {update};

  procedure reduce;

    begin
      x2.i ← x;
      x2.r ← Λ;
      update;
      y ← y2.r
    end {reduce}

  begin
    x ← copy(d0);
    reduce;
    o0 ← copy(y)
  end {schema}
```

Figure 20

```
Program schema;

  type D0 =
       O0 =
       (D0×O0) = record
                   i:D0;
                   r:O0
                 end;

  var d0,x:D0;
      o0,y:O0;
      x2:D0×O0;

  begin
    x ← copy(d0);
    x2.i ← x;
    x2.r ← Λ;
    while not length(x2.i) = 0 do
      x2 ← update(x2);
    y ← x2.r;
    o0 ← copy(y)
  end {schema};
```

Figure 21

```
Program delimitedstrings;

 type
  state = ("#","@","c","w");
  partial−report = case state of
       "#" : record
               "@" : file of char;
               "%" : file of char
             end;
       "@" : record
               "@" : file of char;
               "&" : file of char;
               "%" : file of char
             end;
       "c" : record
               "@" : file of char;
               "&" : file of char
             end;
       "w" : record
               "%" : file of char
             end;
       otherwise
      end;

  D0 = file of char;
  O0 = partial−report;
  D0×O0 = record
            i:D0;
            r:O0
          end;

var
  d0,x:D0;
  o0,y:O0;
  x2:D0×O0;

Procedure process(var x2.i:D0, var x2.r:O0);

  var
    type:char;
    letter:char;

  begin
    letter := first(x2.i);
    x2.i := tail(x2.i);
    type := "#";
    case letter of
      "@" : if type = "#"
            then type := "@"
            else type := "#";
      "&" : if type = "@"
            then type := "c"
            else type := "w";
      "%" : if type = "&"
```

```
            then type := "c"
            else type := "w";
        otherwise
            if type ≠ "@" or "&" or "%"
            then type := "#";
      end

      case type of
        "#" : begin
                x2.r.% := append(x2.r.@,letter);
                x2.r.% := append(x2.r.%,letter)
              end;
        "@" : begin
                x2.r.@ := x2.r.@;
                x2.r.& := append(x2.r.&,letter);
                x2.r.% := append(x2.r.%,letter)
              end;
        "c" : begin
                x2.r.@ := x2.r.@;
                x2.r.& := x2.r.&
              end;
        "w" : begin
                x2.r.% := x2.r.%
              end;
        otherwise
      end
    end;

    x := copy(d0);
    x2.i := x;
    x2.r := Λ;
    while not length(x2.i) = 0 do
      process(x2.i,x2.r);
    y := x2.r;
    o0 := copy(y)
  end {delimitedstrings}
```

Figure 22

5. Conclusions

The paper has shown that a prototype program can be derived using the Jackson approach embedded in the Data Transform Method (DTM). Since most of the prototype can be characterized by a program schema it is only necessary to concentrate on the development of the lower level procedure called "process". We believe that this method of program formulation and prototyping simplifies program development. There is also an extra benefit in that total program correctness is almost a by-product.

Since there is one program schema which can be followed for every file-processing problem it should be clear that the program development process can be partially automated. In fact several prototyping tools have already been created and tested. Future work will concentrate on developing a user-friendly interface based on expert systems.

6. Acknowledgements

The work described in this paper was sponsored by several agencies and the authors gratefully acknowledge the assistance of NSERC, WATFAC, and FINEP.

7. References

[1] Jackson, M.; "Principles of Program Design", A.P.I.C., Studies in Data Processing, No. 12, Academic Press, London, 1975.

[2] Warnier, J.D., "Logical Construction of Programs," New York, Van Nostrand Reinhold, 1979.

[3] Lucena, C.J., Martins, R.C.B., Velosos, P.A.S., Cowan, D.D., "The Data Transform Programming Method: An Example For File Processing Problems", pp. 388-397, Proceedings of the 7th International Conference on Software Engineering, Orlando, Florida, 1984.

[4] Lucena, C.J., Martins, R.C.B., Veloso, P.A.S., Cowan, D.D., "A Theoretical proposal for a CASD System Extending Jackson's Method for Program Construction", to appear in Advanced Automation, edited by Julius T. Tou, Plenum Publishers, 1984.

[5] Veloso, P.A.S., Veloso, S.R.M., "Problem Decomposition and Reduction: Applicability, Soundness, Completeness; Trapp, R., Klir, J., Pickler, F. (eds); Progress in Cybernetics and Systems Research, Vol. VIII, (Proc. of the 5th EMSCR, Vienna, 1980): Hemisphere Publ. Co. 1980.

[6] Martins, R.C.B., "The Data Transform Method", Doctoral Thesis, Computer Science Department, Catholic University, Rio de Janeiro, Brazil (in Portuguese).

[7] Hoare, C.A.R., "Notes on Data Structuring" in Dahl, Dijkstra, E.W., Hoare, C.A.R., Structured Programming, Academic Press, 1972.