A Data-Directed Approach to Program Construction

D.D. Cowan*, J.W. Graham, and J.W. Welch
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

C.J.P. Lucena
Departmento de Informatica
Pontificia Universidade Catolica
Rio de Janeiro - Brazil

Research Report CS-78-02

Revised June 1979

*Research supported in part by Canadian National Research Council Grant A2655.

A DATA-DIRECTED APPROACH TO PROGRAM CONSTRUCTION

D.D. Cowan, J.W. Graham, and J.W. Welch
Computer Science Department
University of Waterloo
Waterloo Ontario Canada

Carlos J.P. Lucena
Departmento de Informatica
Pontificia Universidade Catolica
Rio de Janeiro - Brazil

ABSTRACT

The present paper discusses a method of program construction based on the specification of the data types. The input and output data types and the mapping between them are specified at a high level of abstraction and this non-procedural specification is used to develop a program schema. The data type and mapping specifications are modified to include a concrete representation of the data and these are used to expand the program schema into a program. A graphical representation for data and program specifications is also introduced and it is shown how this can simplify the techniques and be very useful in program construction. The method is illustrated by developing two programs - the line justifier program described by Gries and the bubblesort.

KEY WORDS: Program construction, program derivation, program specification, program schema, data types.

1. INTRODUCTION

There are large collections of programs, particularly in industry, which have a long lifetime but which are frequently being modified. In fact a large percentage of the programmers currently employed are working on this "maintenance" task. Since modifications are so important it is reasonable to develop a disciplined approach to programming which makes modification as easy as possible. Various attempts at such methods are exemplified by Dijkstra's Structured Programming [1], Wirth's Systematic Programming [2], and Jackson's techniques [3], to mention just a few of the practitioners and proponents.

It has been observed that many of the changes in typical data processing programs are caused by changes in the structure of the data to be processed and by the accompanying actions which must occur when there is a change in the structure. Hence, if a program or system of programs can be made to resemble the data that is being processed, then modifications to the data might be easily reflected in the program.

This paper describes a disciplined approach to programming which leads to programs which can be systematically maintained. This approach resembles the approach used by Jackson [3]. This paper tries to put in a more formal perspective many of the concepts suggested by Jackson, and introduces a graphical representation of data and programs which we have found to be very

helpful in their design, construction and documentation. Specifically this representation allows the design of programs in which there is a one-to-one correspondence between portions of the input and output data structures and the program structure. This class of programs as mentioned before, covers a wide variety of applications.

In developing our approach to creating programs, we take two specific views of the programming process. First we adopt the approach which has been described by Hoare [1, 4], namely that the structuring of data should be handled by the following three mechanisms: direct or Cartesian product, discriminated union and sequence. Types are either unstructured (or primitive) or structured in many levels through the use of these three mechanisms.

Second we use the approach first described by Jackson [3] that a program is a transformation, mapping, or set of productions which transforms the input data as described by a data type specification into the output data which can also be described by a data type specification. In other words any input data which fits the definitions prescribed by the input data specification will be transformed by the program into some output data which fits the output data specification. Figure 1 illustrates this second view.

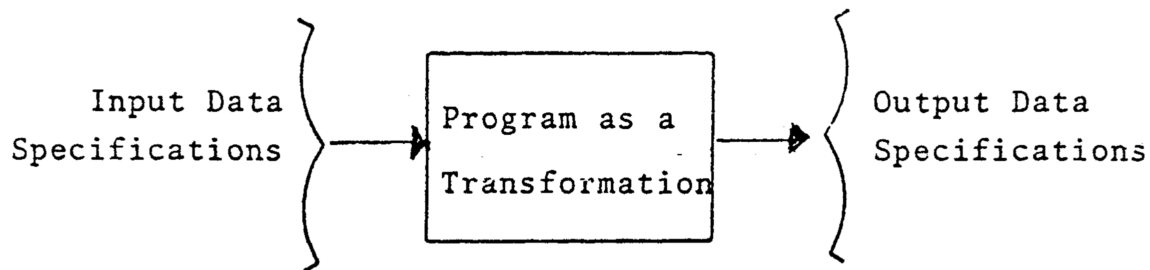


Figure 1

Both Jackson [3] and Hoare [4] observe that the control structures which are required to handle data structures of types record, sequence and discriminated union are precisely the simple sequencing of statements, a looping construct such as the while do, and the case or if then else statement respectively. This observation will also be used in the development of our programs.

In order to present our ideas in a clear manner we shall work two examples. The first example, presented by Gries [5] is used to illustrate the notation and also to introduce the concept of levels of data abstraction into our method.

The second example, a sort program, is used to illustrate further how the method might be applied to a more complex program.

2. THE LINE JUSTIFIER PROBLEM: CONSTRUCTION OF A PROGRAM SCHEMA

As mentioned previously, our view of program derivation will be presented through the construction

of two programs. This section discusses the derivation of a program for an example suggested by Gries [5]. We first illustrate all the techniques by deriving a program which operates on an abstract specification of the input and output data. After that, one concrete representation of the data will be used to illustrate how one can move through levels of abstraction and develop an operational program.

2.1 The Problem Statement

A line-justifier is the part of a text editor which inserts blanks between the words in a line in order to avoid having blanks at either the right or left-hand end of the line. We wish to construct a line-justifier program according to the following specifications:

(i) The program accepts a numbered left-justified line having more than one word in which there will be just one blank between words and possibly several blanks after the last word.

(ii) The program will produce as output a justified line, that is a line in which the extra blanks to the right of the last word will have been distributed in the spaces between the words on the line. The difference between the number of blanks in two arbitrary intervals between words will be at most one. When there is a difference the number of blanks between words will be the same up to a given

word in the line; and after this word the number of blanks between words will again be uniform, but there will be either one more or one less than the previous number of blanks. For aesthetic reasons the even lines will have more blanks at the beginning of the line and the odd lines more blanks toward the end.

2.2 Data Type Specifications

As indicated in the problem statement in the previous section the line-justifier program manipulates objects called lines. Specifically it manipulates even-lines and odd-lines. At a high level of abstraction these lines will be the lowest level entities manipulated by our program and hence will be used as the terminal symbols in our input data specification.

In a corresponding fashion the lowest level entities produced by our abstract program are lines with justification performed from the left or the right. These lines will be called just-left line and just-right line respectively and will be used as terminal symbols in the output specification. The input specification can be described as a text which consists of repeated instances of line which is either even-line or odd-line. A definition of the input specification is given in Figure 2. Similarly we can define the output data as a justified text which consists of

repeated instances of justified-line which is either a just-left line or a just-right line. A definition of this output specification is given in Figure 3.

type text = sequence line

type line = (even-line, odd-line)

Note: The comma designates a discriminated union Hoare [1].

Figure 2

type justified-text = sequence justified-line

type justified-line = (just-left-line, just-right-line)

Figure 3

text	→	justified-text
line	→	justified-line
even-line	→	just-left-line
odd-line	→	just-right-line

Figure 4

2.3 Programs as Transformations

Many programs can be conceived directly as transformations. Specifically they transform elements of the input data type to elements of the output data type. In the current example the abstract program which justifies lines transforms text into justified text,

line into justified line, even-line into just-left line, and odd-line into just-right line. This transformation process can be described by a set of mappings between domains of the input and output data types to be used in the program and the set of mappings for this example is shown in Figure 4. Not all programs yield such a straight forward set of productions but the second example will illustrate other situations.

2.4 Construction of the Program Schema from the Type Specifications and the Mappings

The data definition and the set of mappings suggest the organization of the abstract program for line justification of text and the actual programs can be produced in a straight-forward manner. We shall now describe an informal procedure which we use to produce a program from these specifications. The mappings in Figure 4 show that there is a one-to-one correspondence between all domains of the input data types and the output data types.

Using this mapping we can construct our program in several steps:

(i) At the highest level of the mapping the domain of text is mapped into the domain of justified text. At the lowest level the mapping must be realized by two functions which convert even-line into just-left-line

2.4 Construction of the Program Schema from the Type Specifications and the Mappings

(i) cont'd.

and odd-line into just-right-line. These two operations or functions are called just-left (line) and just-right (line) respectively.

The reader should note that the word "line" has been used in three different ways in this paper; "line" has been used to designate a data type, the domain of the data type line in the mapping, and to represent a variable of type line. We hope that the reader is not confused by these multiple uses of the same name for different but related concepts; it was done in order to minimize the number of different names used throughout the paper.

(ii) Both line and justified-line are expressed as alternations (discriminated unions) and there is a direct mapping between them. Hoare [4] and Jackson [3] observed that data structures defined in this way are controlled in a program by the <u>if then else</u> construct. Hence, if we define the mapping

line → justified-line

as

process-line (line)

then we can follow the structure implied by the type definitions to produce

```
process-line (line)
if even (line)
then
     just-left (line)
else
     just-right (line)
fi
end
```

Of course, elaboration of the type definition required the invention of a Boolean function even (line) to determine if the number associated with a line is even.

(iii) Similarly we write the production for text as process-text (text). Using the type definitions and the mappings and observing that repetitions are controlled by while do constructs [3, 4] we can attempt to construct the next step in our program. This becomes

```
process text (text)
while
do
   process-line (line)
od
end
```

The problem arises when we attempt to construct a predicate for the while statement. This predicate should contain a mechanism for transforming text into a sequence of lines and then testing when there are no more lines

to be processed. We invent a function read (text) where read removes one of the repeated elements from text and returns it as a value. If text is empty then the value nil is returned. The predicate for while then becomes

```
line: = read (text) ≠ nil
```

These sequencing functions become somewhat more complex in other advanced examples but this is beyond the scope of the present discussion. Our program now becomes

```
process-text(text)
while line: = read (text) ≠ nil
do
   process-line (line)
od
end
```

(iv) Using a direct substitution for process-line we arrive at the following program

```
process-text (text)
while line: = read (text) ≠ nil
do
   if even (line)
   then
       just-left (line)
   else
       just-right (line)
   fi
od
end
```

3. INTRODUCTION OF A DATA REPRESENTATION: CONSTRUCTION OF THE ACTUAL PROGRAM.

So far in the discussion we have used a very abstract description of line. In this section we are

going to associate a specific model of a line with our more abstract program schema.

By definition, our program receives as input a line expressed in a given representation and produces as output a line expressed in the same representation. We shall think of a line as a 6-tuple of natural numbers having the following components:

- p - number of blanks in the left-most intervals of the line.
- q - number of blanks in the right-most intervals of the line.
- t - index of the word after which the number of blanks changes.
- n - number of words on a given line.
- s - number of extra blanks at the end of the line.
- z - line number.

Note that the program will not handle the text but rather a representation which contains only the information relevant to the specific problem.

We can now define an extended set of data specifications to express the lower-level information in this representation. We shall use the symbols p,q,t,n,s and z to represent the six components of the Cartesian product for six-tuple. These new extended data specifications and mappings are shown in Figure 5, 6, and 7.

```
type text         = sequence line
type line         = (even-line, odd-line)
type even-line    = six-tuple
type odd-line     = six-tuple
type six-tuple    = (p;q;t;n;s;z)
```

Note: The semi-colon(;) designates the Cartesian product.

Figure 5

```
type justified-text    = sequence justified-line
type justified-line    = (just-left-line, just-right-line)
type just-left-line    = six-tuple
type just-right-line   = six-tuple
```

Figure 6

```
text        →   justified-text
line        →   justified-line
even-line   →   just-left-line
odd-line    →   just-right-line
six-tuple   →   six-tuple
```

Figure 7

Using the extended set of definitions we can now construct a program from the original program schema.

(i) At the lowest level there must be assignments which convert the various components of the six-tuple for even-line and odd-line into the components of the six-tuple for just-left-line and just-right-line. The record structure (Cartesian product) suggests the structuring of control through a sequence of statements. The sequence of assignments are shown next; the details are left to the reader. The names used in the data definitions are also used as the names in the assignment. All the elements of the six-tuple can ultimately be defined as being of type integer (primitive in most programming languages).

In order to be brief the assignments are written in terms of the components of the six-tuple. The components on the left-hand side of the assignment should qualify either just-left-line or just-right-line and the components on the right-hand side qualify either even-line or odd-line.

For just-left-line and even-line we have

```
q: = s/(n-1);                p: = q+1;
t: = mod(s,(n-1))+1;         n: = n;
s: = 0;                      z: = z;
```

For just-right-line and odd-line we have

```
p: = s/(n-1);                q: = p+1;
t: = n-mod(s,(n-1));         n: = n;
s: = 0;                      z: = z;
```

(ii) The Boolean function even (line) can now be replaced by mod(z,2) = 0 since we know the concrete representation of line and hence the method of numbering it.

Since the program schema has already been constructed we can modify it by including this extra information. We then produce the following program:

```
process-text (text)
while line: = read (text) ≠ nil
do
   if mod(z,2) = 0
   then
        q: = s/(n-1)
        p: = q+1
        t: = mod(s,(n-1))+1
        n: = n
        s: = 0
        z: = z
   else
        p: = s/(n-1)
        q: = p+1
        t: = n-mod(s,(n-1))
        n: = n
        s: = 0
        z: = z
   fi
 od
 end
```

Again in this program we have not qualified the components p,q,t,n,s and z with the names of the structures to which they apply for the sake of brevity.

In any program they should be qualified in order to distinguish which variables are actually modified by the assignments.

4. A GRAPH MODEL FOR THE DATA DEFINITION AND THE MAPPINGS BETWEEN TYPE DOMAINS

Although the previous method of constructing programs provides a systematic approach we have found it more convenient for purposes of visual representation to draw the data definitions and mappings between type domains as a linear graph, specifically a tree. The construction of these graphical representations will be described by example.

The graphical format of the data specification of Figure 2 is shown in Figure 8. This form can be described as being a construct of type text which contains zero or more repeated instances of the type line. The type line is one of two types, it is an even-line or an odd-line. The asterisk (*) in the diagram indicates repetition and the two question marks (?) separated by a dashed line indicate an alternative. Similarly, the output data specifications of Figure 3 are presented in Figure 9. They can be described in exactly the same manner as the specifications in Figure 8.

One can see by examination of the two diagrams that there is a one-to-one correspondence between the

two definitions and hence the mappings can be described directly on the input specification. The mappings are shown in Figure 10. The word "process" has been written in front of each type and the asterisk (*) and question marks (?) have been replaced by letters. These letters represent the predicates which are to be tested at these points in the program and are amplified below the diagram. The type of test is shown in the case of a repetition.

Graphical Representation of Input Data Specifications of Figure 2

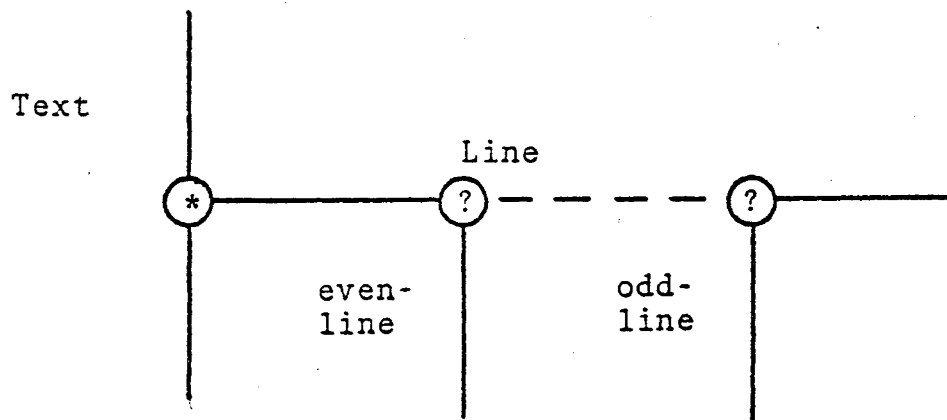


Figure 8

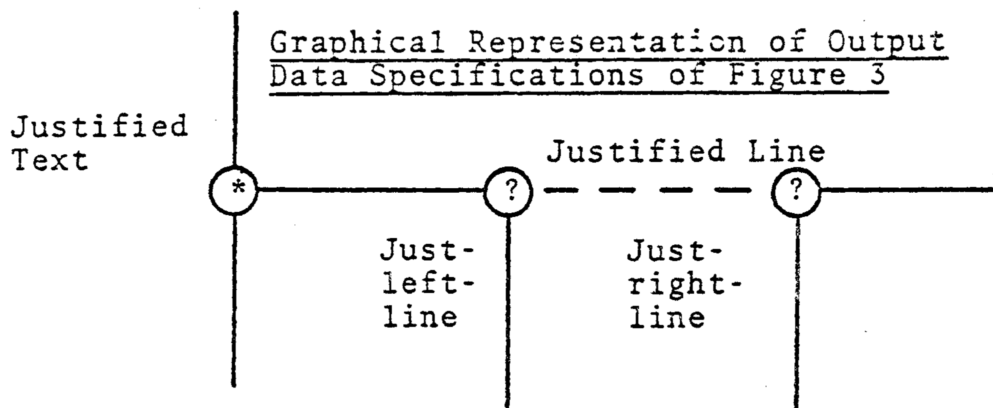
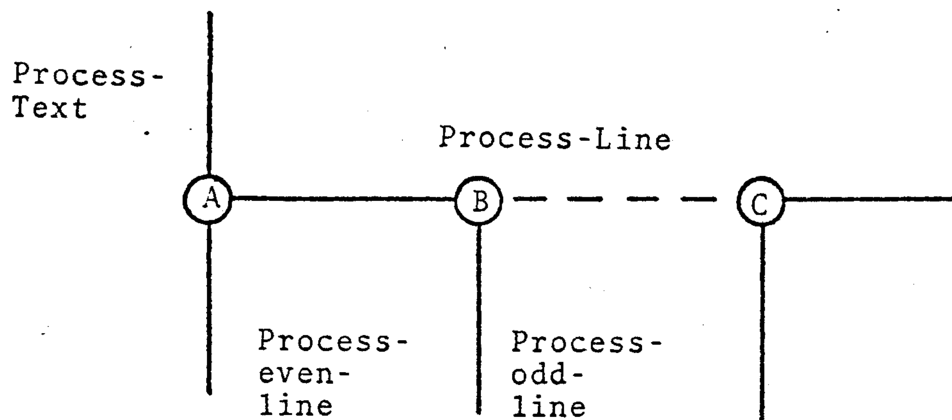


Figure 9



A: while line:= read (text) ≠ nil

B: even (line)

C: not even (line)

Figure 10

The program can now be constructed directly from the diagram. We shall describe it verbally since the program was written earlier. The procedure process-text consists of repeated occurrences of the procedure process-line under control of the predicate line: = read (text) ≠ nil. Process-line is composed of two procedures process-even-line and process-odd-line; only one of these procedures is executed under control of the Boolean function even (line). Of course process-even-line and process-odd-line are just the routines just-left and just-right in the earlier version of the program.

5. A SORTING PROGRAM: CONSTRUCTION OF A PROGRAM SCHEMA

This section discusses the derivation of a program for an internal sort, specifically we shall eventually derive the bubblesort. We illustrate our techniques by deriving a program schema which operates on an abstract definition of the input and output data. This will provide us with a program schema from which we can derive more than one internal sort. After that derivation one concrete representation will be used to refine the sort and to illustrate how one can move through levels of abstraction and develop an operational program.

5.1 Statement of the Problem - Some Preliminary Analysis

A sort is a program which orders sets. Specifically we shall construct an internal sort that is a program which takes some small finite unordered subset of the integers and permutes it into an ordered subset, using the usual ordering.

In order to use our techniques successfully on the sorting problem we must first place it in the correct context. A sorting procedure takes an unordered set and gradually converts it into an ordered set. In effect there is a sequence of partially ordered sets A_i which are processed by a sequence of programs P_i. After K applications of the programs the final set A_{K+1} will be sorted. This interpretation of sorting is shown in Figure 11. We immediately recognize that the programs P_i are identical and we interpret a sorting procedure which takes a set A of partially ordered sets and transforms this set into a set B of partially ordered sets one of which is completely sorted. If we represent the members of A by A_i and the members of B by B_j then A has K members $A_1, A_2 \ldots A_K$ and B also has K members $B_1, B_2 \ldots B_K$ and further $B_i = A_{i+1}$. With this preliminary analysis we can now construct a formal description of the input and output for our program in a form similar to Hoare's [1] specification of data types.

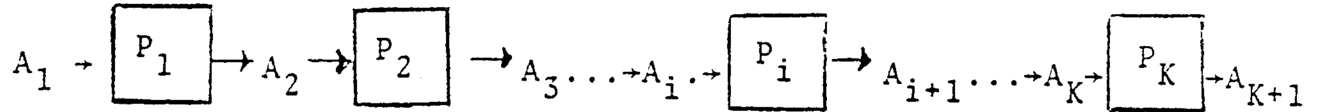


Figure 11

5.2 Data Type Specification

As indicated in the problem statement in the previous section the sort program manipulates a set of sets of numbers which are partially sorted. Let us call these sets of numbers arrays of numbers, or more simply arrays. At this point the word array is not used in the representation sense, (as implied by most programming languages) although later in the paper we shall use it as a representation for the set we wish to sort.

At a high level of abstraction these arrays will be the lowest level entities manipulated by our program, and hence will be used as the terminal types in our input data specification. In a corresponding fashion the lowest level entities produced by our abstract program are partially-ordered arrays. These partially-ordered-arrays will be used as terminal data-types in the output-data specification. The input specification and output specifications shown in Figures 12 and 13 are in a form similar to Hoare's [1] specification of data types.

<u>type</u> set = <u>sequence</u> of array

Figure 12

<u>type</u> partially-ordered-set = <u>sequence of</u> partially-ordered-array

Figure 13

set → partially-ordered-set
array → partially-ordered-array

Figure 14

5.3 Construction of a Program Schema

Many programs can be viewed directly as transformations of the input data-types to the output data-types. In the current example the abstract program which sorts the set transforms the type set into partially-ordeered-set and the type array into partially-ordered-array. This transformation is described by the set of mappings shown in Figure 14. The mappings are mappings from the domain of one type to the domain of the other type. We now describe the steps of an informal method for constructing a program schema from the specifications of Figures 12, 13 and 14.

(i). To construct a program we note that there must exist a function which converts an array into a partially-ordered-array, since there is a direct mapping between them. This operation we call change (array) and it replaces the mapping

array → partially-ordered-array.

The reader should note that the name "array" has been used for three different but related concepts in this paper; "array" has been used to designate a data type, the domain of the data type array in the mapping, and to represent a variable of type array.

(ii) Both set and partially-ordered-set are represented by sequences and there is a direct correspondence between them. Hoare [4] and Jackson [3] observed that structures defined in this way are controlled in a program by a while-do construct. Hence if we use sort (set) to represent the mapping

set → partially-ordered-set,

then we can attempt to construct the next step in our program.

This becomes

```
sort (set)
while
do
   change (array)
od
end.
```

A problem arises when we attempt to construct a predicate for the while statement. This predicate should contain a mechanism for transforming the set into a sequence of arrays and then testing when there are no more arrays to be sorted. However, we observed earlier that this set of arrays is somewhat artificial and that really there is only one array which is gradually transformed into a sorted array.

Hence the argument of sort is array and the only predicate we need is one that tests if the array is sorted. We shall use the Boolean function ordered (array) to determine if the while is finished. Ordered (array) will return the value "true" if the array is sorted.

The program now becomes

```
sort (array)
initialize
while not ordered (array)
do
    change (array)
od
end
```

The statement "initialize" indicates that some variables may have to have values before the predicate can be tested. This statement was not introduced into the previous example because it was not necessary and would only provide a complication at that stage. However, programs do require this statement although it is often implicit in the construction.

6. INTRODUCTION OF A DATA REPRESENTATION

In this section we transform the abstract program schema into a procedure by introducing the usual representation of an array; this transformation occurs in two stages. First the array is divided into two parts, an unsorted-part and a sorted-part. Because the unsorted-part is going to be ordered, it is described in more

detail as a sequence of overlapping pairs (o-pair) and the overlapping pair is expanded as a discriminated union of good overlapping pair (g-o-pair) or bad overlapping pair (b-o-pair). The extended type specifications and mappings for the first stage are presented in Figures 15, 16 and 17 where the type array is shown as a record whose components are separated by a semi-colon(;) and an o-pair is a discriminated union whose parts are separated by a comma (,). In the second stage the type array will be defined explicitly as an array of integers. The next few steps illustrate the method of constructing the program for the first step.

```
type set           = sequence of array
type array         = (unsorted-part; sorted-part)
type unsorted-part = sequence of o-pair
type o-pair        = (g-o-pair, b-o-pair)
```

Figure 15

```
type partially-ordered-set   = sequence of partially-ordered-array
type partially-ordered-array = (unsorted-part; sorted-part)
```

Figure 16

```
set           → partially-ordered-set
array         → partially-ordered-array
unsorted-part → unsorted-part
sorted-part   → sorted-part
```

Figure 17

(i) There is a mapping between each type in the output specification and a corresponding input specification. There is no correspondence at the level of overlapping pairs since although sorted-parts are mapped into sorted-parts and unsorted-parts into unsorted-parts, each pass of the process over the set may make the unsorted-part smaller and the sorted-part larger.

(ii) We now construct procedures for mapping of the two parts of array. These are decrease (unsorted-part) and increase (sorted-part). With these new procedures the program becomes

```
sort (array)
initialize
while not ordered (array)
do
   decrease (unsorted-part)
   increase (sorted-part)
od
end.
```

(iii) The unsorted-part is composed of a sequence of overlapping pairs and must be a program under control of a while-do construct. The predicate must check whether the end of the unsorted-part has been reached. Hence the code for decrease (unsorted-part) is

```
decrease (unsorted-part)
initialize
while not end (unsorted-part)
do
   process-o-pair
od.
```

(iv) The type o-pair consists of a discriminated union of two types and is processed using the if-then-else control structure [3, 4]. Process-o-pair becomes

```
if bad (o-pair)
then
     process-b-o-pair
else
     process-g-o-pair
fi.
```

The program we have constructed now has the following form:

```
sort (array)
initialize
while not ordered(array)
do
     initialize
     while not end (unsorted-part)
     do
        if bad (o-pair)
        then
          process-b-o-pair
        else
          process-g-o-pair
        fi
     od
     increase (sorted-part)
od
end.
```

At this point decisions must be made about the actual form of the procedures and predicates thus forcing us into a final choice of sorting method, namely the bubblesort.

To make sure the next level of program is equivalent to the higher level presented previously, we need to re-state the concepts used in the higher level in terms of the new notation. This has been done in Figure 18 where type array is effectively defined as being structured as an array. Note that the name "array" is used as the name of a type and also as the structuring mechanism.

```
type set          = sequence of array
type array        = (unsorted-part; sorted-part)
type sorted-part  = array 1..j-1 of integer
type unsorted-part = array j..n-1 of o-pair
type o-pair       = (g-o-pair, b-o-pair)
type g-o-pair     = (integer; integer)
type b-o-pair     = (integer; integer)
```

Figure 18

We now decide that if a is of type array then we have a bad overlapping pair (b-o-pair) if

$$a_i > a_{i+1}$$

and the values of a_i and a_{i+1} will be interchanged. Process-b-o-pair will be implemented by a procedure swap (x,y). If an o-pair is of type g-o-pair then no processing needs to occur.

Hence process-o-pair becomes

```
if a_i > a_{i+1}
then
    swap (a_i, a_{i+1})
fi.
```

Since the representation of the array has been specified, it is now possible to construct the predicate end (unsorted-part) and its initialization. The bubble-sort starts with an index i = n-1 (the index of the last overlapping pair in the unsorted-part) and terminates when i < j since the end of the unsorted-part would have been reached.

Hence while not end (unsorted-part) can now be replaced with

```
i:=n
while (i:=i-1)≥j
```

which is equivalent to a for-loop

```
for i:=n-1 downto j.
```

The final step in the construction of our bubble-sort program is the construction of the predicate ordered (array).

This predicate will be false only when j>n-1 because the sorted-part of the array will be the full array. The statement

```
while not ordered (array)
```

can be replaced by

```
j:=1
while j≤n-1.
```

This is not quite enough since there must be a method of incrementing j, for the procedure to terminate. This is the function played by the procedure increase (sorted-part). It is simply replaced by j:=j+1.

Since j is incremented each time through the while loop the while can be replaced by

for j:=1 to n-1.

The entire program (without declarations and with variable name "a" substituted for array) can now be written as:

```
sort (a)
for j:=1 to n-1
do
     for i:=n-1 downto j
     do
       if a_i > a_i+1
       then
         swap (a_i,a_i+1)
       fi
     od
 do
 end.
```

7. A GRAPHICAL REPRESENTATION OF THE PROGRAM SCHEMA

The graphical format of the input data type specification of Figure 12 is shown in Figure 19. This form can be described as being a construct of type set which contains zero or more repeated occurrences of type array. The asterisk (*) in the diagram indicates repetition. Similarly the output data type specification of Figure 13 is presented in Figure 20. It can be described in a similar manner to the specification in Figure 12.

One can see by examination of the two diagrams that there is a one-to-one correspondence between the two type specifications and hence the mappings can be described directly on the input type specification. The productions are shown in Figure 21. The word "process" has been written in front of each type and the asterisk (*) has been replaced by a letter. This letter represents the predicate which is to be tested at this point in the program and is amplified below the diagram. The type of test in the case of a repetition is also indicated.

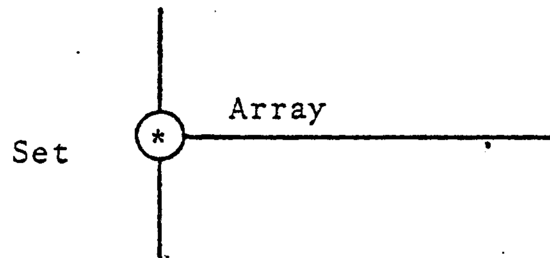


Figure 19

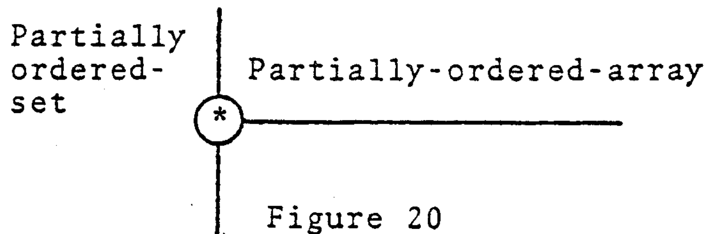
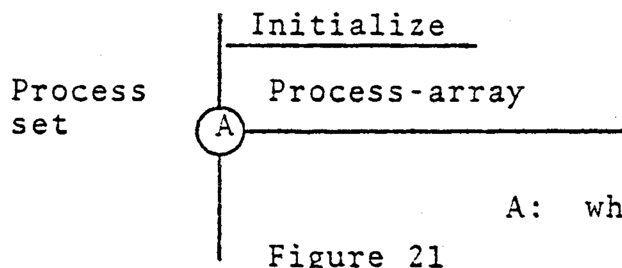


Figure 20



A: while not ordered (array)

Figure 21

The program can now be constructed directly from the diagram. We shall describe it verbally since the program was written earlier. The procedure process-set consists of the statement initialize followed by repeated occurrences of the procedure process-array under control of the predicate ordered (array). Process-array is of course change (array). The predicate was obtained using the same arguments as we have used previously. With this approach it is possible to write down the program directly from the diagram and in fact we normally use the diagram to specify our program and then code it directly. The input and output type-specification diagrams are called data structure diagrams. The diagram showing the mappings is called a program structure diagram.

7.1 Introducing A Data Representation into the Graphical Program Schema

One can easily modify the diagrams of Figures 19, 20 and 21 to include a representation of the array which holds the integers to be sorted. The input type and output type specification are shown in Figures 22 and 23. This input data-type specification can be described as being a construct of type set which is a sequence of instances of type array. An array consists of a tuple or record which has two components, a

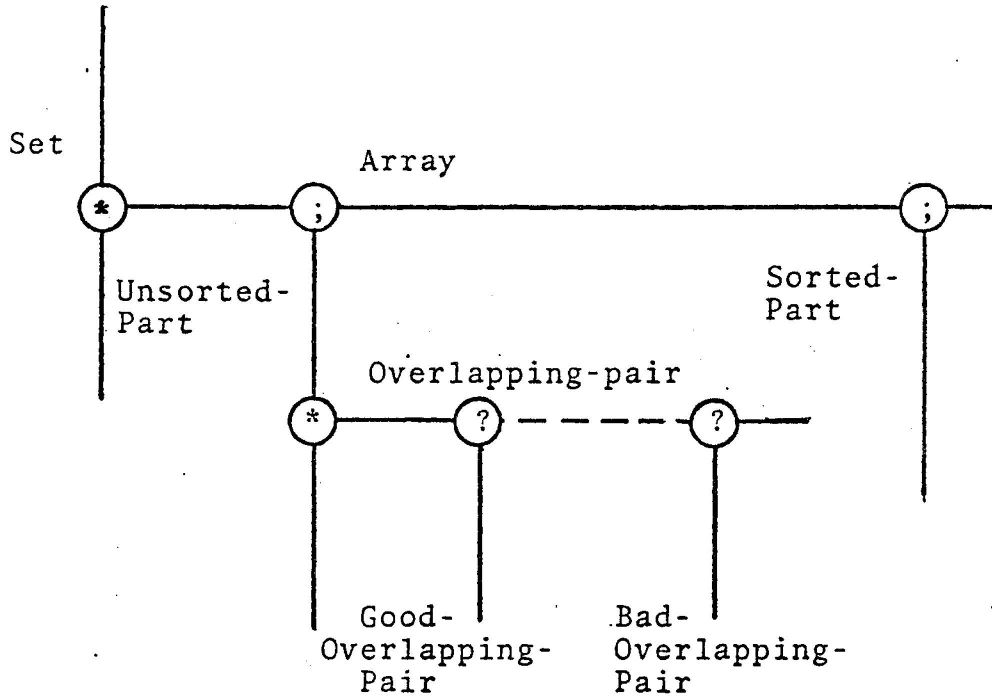


Figure 22

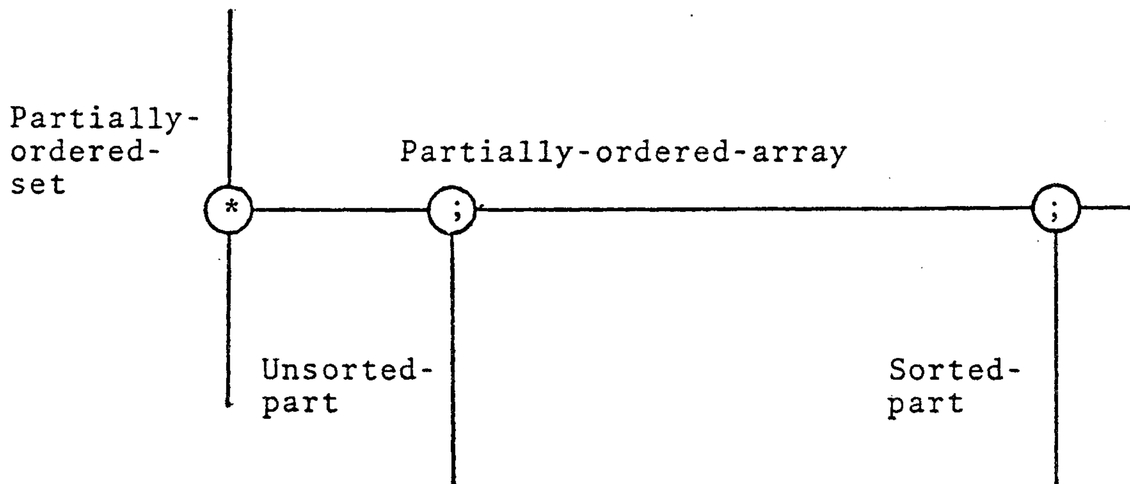


Figure 23

sorted-part and unsorted-part. The unsorted-part is further defined as consisting of a sequence of instances of type overlapping-pair and overlapping-pairs are discriminated unions or alternatives of two types good-overlapping-pair and bad-overlapping-pair. The only extra notation introduced is the use of the semi-colon (;) to denote components of the tuple or record. The output type specification can be described in the same manner. The mappings are described in Figure 24 and are obtained by writing the word process before each input type specification. The predicates are denoted by letters replacing the asterisks and question marks in the diagram and are further described below the program structure diagram.

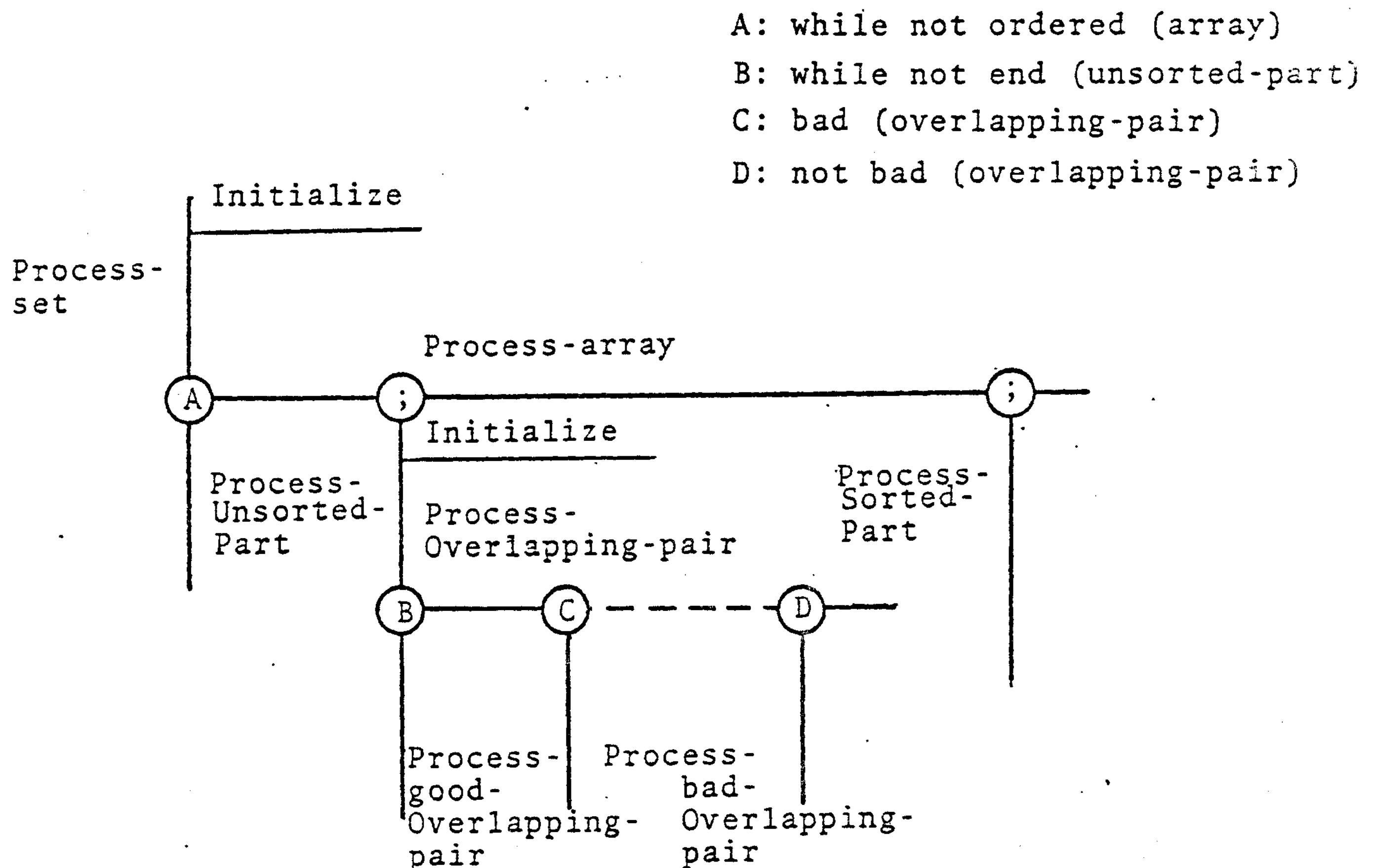


Figure 24

The procedure process-bad-overlapping-pair is replaced by the procedure swap and the process-sorted-part by increase (sorted-part). Of course the predicates were obtained by arguments similar to those given previously.

Using these diagrams to represent the input and output type-specifications and also the mappings is quite advantageous. The relationships among the various types is more evident in a two-dimensional diagram than in a one-dimensional string as exemplified by the type-specifications shown earlier in the paper. In our opinion this makes it easier to recognize relationships, construct predicates and hence produce working programs.

The diagrams can also be drawn very quickly; this has a number of advantages:

(i) It is easy to use them to illustrate data structures and programs and hence they are a useful instructional tool.

(ii) It is easy to discard incorrect attempts at constructing a program rather than "patching" them since there is a very small investment in drawing the diagrams.

(iii) It is easier to maintain programs constructed in this way since there is a small investment in redrawing the diagram.

CONCLUSIONS

We have discussed the construction of an Algol-like program by considering it to be a mapping between input and output data types. Specifically we have followed a procedure consisting of a number of well-defined steps.

First we specified abstract input and output data types and the mapping between them. We then used this combination to derive a program schema. As a second step the abstract data types are expanded into concrete representations by choosing ones which can be implemented in most Algol-like languages. Of course the mapping between the data types is expanded to include the concrete representations. Finally, we expand the program schema into an actual program by using the concrete representations and the mapping between them.

Having constructed the program using a formal model of the types and mapping, we introduce an informal graphical representation which can be used to construct this program schema, and the program. The graphical representation has been found to be quite convenient for expressing ideas about programs and as a tool to aid in program development.

BIBLIOGRAPHY

[1] O-J. Dahl, E.W. Dijkstra and C.A.R. Hoare (1972) Structured Programming pp 1-174 Academic Press.

[2] N. Wirth (1973) Systematic Programming: An Introduction Prentice-Hall.

[3] M.A. Jackson (1975) Principles of Program Design Academic Press.

[4] C.A.R. Hoare (1975) Data Reliability pp 528-533 Proceedings International Conference on Reliable Software April 1975.

[5] D. Gries (1976) An Illustration of Current Ideas on the Derivation of Correctness Proofs and Correct Programs. IEEE Transactions on Software Engineering Vol SE-2 No.4.

[6] Dijkstra E.W.(1975) Guarded Commands, Non-determinacy and a Calculus for the Derivation of Programs. Proceedings of the International Conference on Reliable Software April 1975

[7] Dijkstra E.W.(1976) A Discipline of Programming Prentice-Hall.

[8] Manna Z. and Waldinger R. (1975) Knowledge and Reasoning in Program Synthesis. Artificial Intelligence Journal Vol. 6. No. 2.