An Automation of Jackson's Structured Programming

T.H. Tse*

A program development system has been written based on Jackson's structured programming methodology. It accepts input and output data structures and generates pseudo-code. Executable operations other than computations can be generated automatically. The number of input and output data structures can be unlimited since they can be accepted in turn by the system. Backtracking is solved by accepting *"quit if"* operations in the data structures. The system also supports program inversion, which is essential for structure clash and for multi-user interactive systems.

Keywords and phrases: automatic programming, software development, software engineering, structured programming.
CR categories: D.1.2, D.2, D.2.2, K.6.3.

1. BACKGROUND

Various popular methodologies have been developed for information systems designers, such as Jackson's structured programming (Jackson, 1975, 1980; and Ingevaldsson, 1979), Jackson's system development (Jackson, 1983), structured analysis (DeMarco, 1978; and Weinberg, 1980), structured design (Yourdon and Constantine, 1979), structured systems analysis (Gane and Sarson, 1979) and structured systems development (Orr, 1977) Most of them, however, are only manual tools developed from experience. There is a need to computerize these tools.

On the other hand, there are other projects in automating the system development process. Examples are ISDOS (Teichroew and Hershey, 1977), SAMM (Stephens and Tripp, 1978), SARA (Campos and Estrin, 1978), SREM (Alford, 1982), UDS2 (Biggerstaff, 1979) and USE (Wasserman, 1982). But they are often developed independently of the popular tools we have mentioned, although some attempts of integration have been made (Yamamoto, 1981). As a result, practising systems designers may find them difficult to use (Davis, 1982; and Martin, 1984).

To remedy this situation, a research project has been set up by the author to automate the popular development methodologies. Various data-driven structured design methods are being studied (Tse, 1985).

The present paper reports the findings on an automation of Jackson's structured programming (JSP). In Section 2 we will outline the procedure for program development according to the JSP methodology. Then in Section 3 we will describe how the procedure is automated.

2. OUTLINE OF JSP

JSP was developed as a technique for writing structured programs from given input and output data structures. It consists of a well-defined set of steps to match the input and output data structures and to merge them into a program structure, which is then converted into pseudo-code. The method has proved to be superior to other functional decomposition techniques (Griffiths, 1978).

In this section we will briefly describe the procedure recommended in JSP. For the convenience of interested readers, suggestions for further reference are given whenever a key concept is introduced. Unless otherwise stated, (19xx: yy-zz) refers to page yy-zz of Jackson (19xx).

2.1 Basic JSP Procedure

We will illustrate the basic steps in JSP through an example.

Example 1

Information on customer orders is stored up in an order file. Each order consists of a header record holding the order number and the customer number, followed by a group of detail records. Each detail record carries the amount debited for a different product. We are required to produce a report showing the total amount debited for each order.

2.1.1 Input and Output Data Structures

The first step in JSP is to specify the structures of the input and output data. Data structures are defined in terms of three basic constructs – sequence, iteration and selection (1975: 17-32, 1983: 86-91) – just as in structured programming (Bohm and Jacopini, 1966, and Dahl *et al.*, 1972). The constructs are represented graphically by trees in Jackson structure diagrams, as shown in Figure 1. Thus the input and output structures of Example 1 are drawn as two trees, as shown in Figure 2.

* *The author is with the Centre of Computer Studies and Applications, University of Hong Kong, Pokfulam Road, Hong Kong.* *Manuscript received July, 1984; revised May, 1985.*

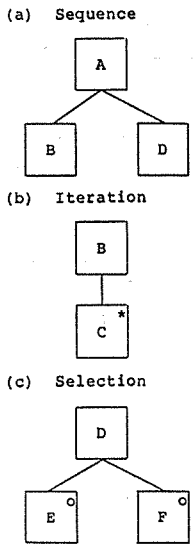


Figure 1 Basic Constructs

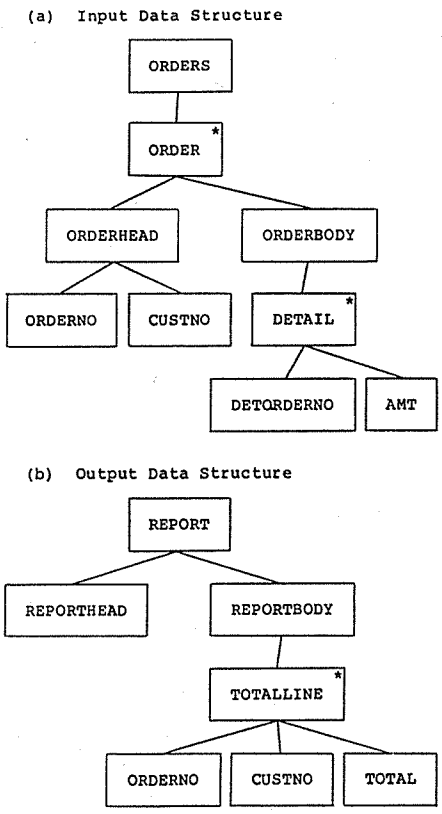


Figure 2 Data Structures

2.1.2 Structural Correspondence

We then find the nodes in the data structures that correspond to each other. An input node A will be regarded as corresponding to an output node B if one instance of A is processed to give an instance of B (1975: 67-70). The structural correspondence for Example 1 is shown in Figure 3. "Order", for instance, corresponds to "totalline" because each order is processed to give a total line.

2.1.3 Program Structure

The program structure is then prepared by first drawing the corresponding nodes and then adding in other nodes that do not correspond (1975: 67-70). This is illustrated in Figure 4.

An exception to this procedure is when the input consists of two or more sequential files arranged in different orders. In this case a structure clash is said to occur (1975: 151-166), and can be resolved by introducing intermediate files or by program inversion (see Section 2.4 below).

2.1.4 Executable Operations

Very often, the target program has to perform operations which are not obvious from the data structures. They are called executable operations in JSP (1975: 45, 51 and 1980: 396). They are divided into five types: *start/stop, open/close, read/write, assignment* and *computational* operations. The next step is to list out the appropriate statements for these five types of operations in a numbered checklist, e.g.

1. stop;
2. open;
3. open (report);
4. close (orders);
5. close (report);
6. read (orders);
7. write (reporthead);
8. write (totalline);
9. report.orderno: = orders.orderno;
10. report.custno: =orders.custno;
11. total: = total + amt;
12. total: = 0;

2.1.5 Allocation of Operations

The executable operations are allocated to the appropriate nodes in the program structure (1975: 46-47 and 52-55), as illustrated in Figure 5. Jackson suggests that this can be done by noting the frequency of execution of each operation. For example, an open statement is executed only once at the beginning of the program. A read statement is executed at the beginning of the program as well as at the end of each iteration block where the input record is involved.

Executable operations have remained a major weakness of JSP. They have to be invented from nowhere and then allocated to the program structure in terms of meaningless numbers. An unreadable program structure such as Figure 5 usually results. This weakness further justifies the need for automation.

2.1.6 Pseudo-code

The program structure is then converted into Jackson pseudo-code (1975: 47-48 and 1983: 122-127), as shown in Figure 6. The pseudo-code can be translated into Pascal, Cobol or whatever programming language required.

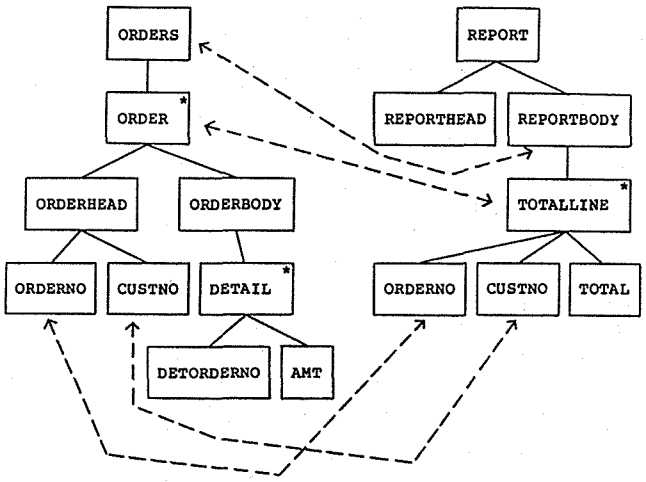


Figure 3 Structural Correspondence

2.2 Backtracking

At the beginning of a selection or iteration, we normally check whether some condition is satisfied before taking the appropriate action. But sometimes we may not be fully aware of the situation at the very start. For example, we may need to exit from the middle of an iteration when a running total exceeds a pre-defined ceiling. Backtracking is used in JSP for such cases (1975: 122-129 and 1983: 203-207). We make an assumption (e.g. running total < ceiling), and then follow the program procedure based on that assumption. Later on, however, if the assumption proves to be wrong, we will have to abandon the procedure.

The statement we need for backtracking is a **"quit if"** command. Consider for instance

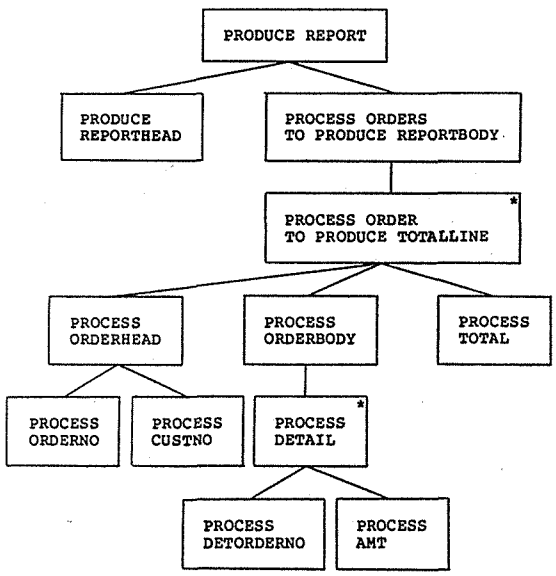


Figure 4 Program Structure

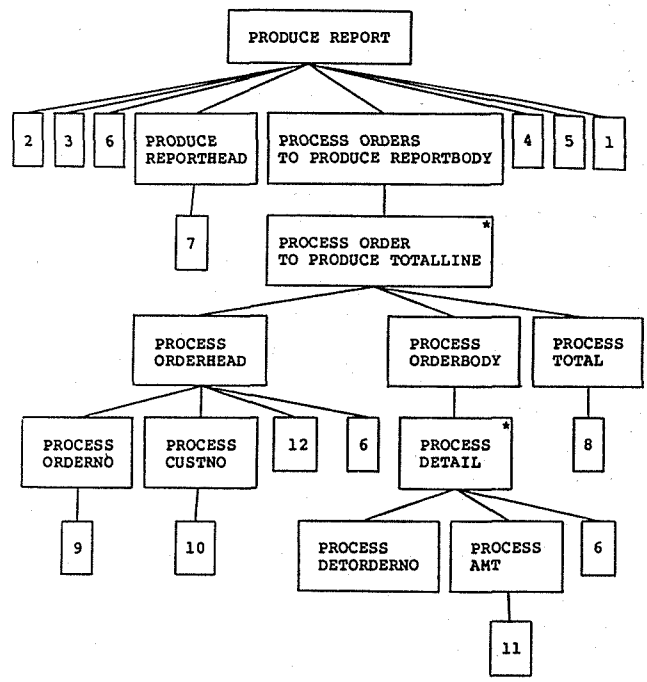


Figure 5 Adding Executable Operations

```
B
    C
    B quit if (cond1)
    D
B end.
```

This indicates that if (cond1) is true, then we have to exit from B.

Sometimes an alternative routine, say E, is required when (cond1) is found to be true. Then the original routine and the alternative routine are labelled respectively by the keywords **posit** and **admit**, as shown below:

```
B posit
    C
    B quit if (cond1)
    D
B admit
    E
B end.
```

2.3 Interactive Input

When interactive input is required for a program, Jackson (1983: 343-348) suggests that each "read" statement should be followed by an iteration of feedbacks, thus:

```
read (orders);
validate itr while (invalid)
    write (diagnostics);
    read (orders);
validate end.
```

2.4 Program Inversion

There are some situations where the above JSP procedures do not work effectively by themselves. Examples are:

- when we prepare a program running under a multi-tasking environment, such as one that support multi-user interactive input.
- when there is a structure clash in sequential access, that is to say, when the sequential files are in different orders.

Program inversion is recommended in JSP to solve these problems (1975: 171-183 and 1983: 263-272). A program is inverted to become a program module, which can be called by a scheduler for as many times as necessary. The inverted module will "remember" the statement it has been executing before returning control to the scheduler. The next time when the module is called, it will resume from that point.

For example, the procedure for inverting a program with respect to an input file is as follows:

- Delete the open statement and the first read statement
- Insert a multi-way branch and the initial pointer address at the beginning of the module, thus:

  goto tp (textpointer);
  tp (1):

Replace each statement and the close statement by:

  textpointer: = N;
  return;
  tp (N):

where N starts from 2 and is incremented by 1 for every "return".

- Change the stop statement to "return".

Readers may refer to an interesting summary in King (1982), where a Jackson structure diagram is used to describe the program inversion process.

3. AUTOMATION OF JSP

A program development system has been written to accept definitions of the input and output data structures. Matching and merging are performed automatically to produce program structures in the form of pseudo-code. The system supports executive operations, backtracking and program inversion. It consists of a suite of medium sized Pascal programs, currently running in batch mode on a PDP 11/70 under RSTS/E. The main features of the system are described in this section.

3.1 Specification of Data Structures

In order to define the data structures in a language familiar to the average systems designer, our specification language combines the features of the Cobol data division and the Jackson pseudo-code. Each node in the data structure corresponds to a line in the specification language, consisting of components as described below. The program structure generated by the system will have a similar syntax.

3.1.1 Level Number

The level number is used to indicate the relative position of the node in the data structure. A Cobol-like convention is used. Level 01 indicates the root, and the child of a node of level L will have a level of L + 1.

```
report seq
    open (orders);
    open (report);
    read (orders);
    write (reporthead);
    orders-reportbody itr while not endof (orders)
        report.orderno := orders.orderno;
        report.custno := orders.custno;
        total := 0;
        read (orders);
        orderbody itr while (detorderno = orders.orderno)
        and not endof (orders)
            total := total + amt;
            read (orders);
        orderbody end
        write (totalline);
    orders-reportbody end
    close (orders);
    close (report);
    stop;
report end
```

Figure 6 Jackson Pseudo-Code

3.1.2 Node Name

The name is used for the matching of nodes and can be any string of alphanumerical characters.

3.1.3 Keyword

The keyword is used to distinguish one type of node from another. It helps the system to determine the appropriate action required when data structures are matched to produce a program structure. Five keywords are used, as described below. Please note that the keyword **end**, normally used in JSP to specify the end of a tree, is not required by our system.

(a) **seq**
This indicates a sequence. For example, Figure 1(a) can be specified as

```
02  A seq
    03  B
    03  D
```

(b) **itr**
This indicates an iteration. For example, Figure 1(b) can be specified as

```
03  B itr
    04  C.
```

(c) **sel** and **alt**
These keywords are used in a selection. Figure 1(c) can be represented by

```
03  D sel
    04  E alt
    04  F alt.
```

(d) rec
Consider the following example:

```
02  G rec
    03  H
    03  I.
```

The keyword **rec** indicates that G is an input or output record. A read or write statement will be generated in the program structure as appropriate.

3.2 Structural Correspondence

3.2.1. Some Basic Definitions

Before we describe how the system deals with structural correspondence, let us formally define a few useful terms.

(a) Given a node X of level L, a *subordinate* of X is a node satisfying two conditions:
- It must have a level > L.
- It must immediately follow X, or a subordinate of X.

Consider for instance

```
02  A
    03  B
        04  C
    03  D
        04  E
        04  F
02  G.
```

The nodes B to F are the subordinates of A.
(b) A *tree* with *root* X (or simply a *tree* X) is defined as the set consisting of the node X together with all its subordinates. In the above example, the nodes A to F form a tree with root A (or simply a tree A).

(c) A *subtree* of X is a tree satisfying two conditions:

- Its root must have a level of L + 1.
- It must immediately follow X, or a subtree of X.

In the example above, the subtrees of A are the trees

```
03  B
    04  C
```

and

```
03  D
    04  E
    04  F.
```

3.2.2 *Matching of Data Structures*

The system regards the input and output data structures as two trees, which are matched and merged to produce a single tree representing the program structure. The algorithm is shown in Figure 7, and the main concepts are explained below.

When two trees are matched, we check first of all whether the roots correspond. In general we assume that two nodes with identical names will correspond to each other, but the user can also specify that two nodes with different names should correspond.

If the roots, say A and B, correspond, we combine them into one node A-B in the program structure. As for their subordinates, there are two possible courses of action:
(a) We continue matching the subtrees of A and B.
(b) We do not continue matching, but simply reproduce the trees A and B as two distinct program subtrees under A-B.

Consider, for instance, two trees

```
02
    A
    03  C
    03  D
```

and

```
02  B
    03  E
    03  F.
```

```
procedure structural correspondence
begin
read input node;
read output node;
match trees (1, 1, 1);
end;

procedure match trees (M, N, P)
(*  Match the next input tree whose root is of level  M,
    with the next output tree whose root is of level  N,
    to form a program tree whose root is of level  P.  *)
begin
if (root of input tree corresponds to root of output tree)
and (match subtrees = yes) then
    begin
    write combined input/output node (level P);
    read input node;
    read output node;
    while (input level > M) and (output level > N) do
        match trees (M+1, N+1, P+1);
    while (input level > M) do
        copy input tree (M+1, P+1);
        (*  Copy the next input tree whose root is of level
            M+1  to a program tree whose root is of level
            P+1.  *)
    while (output level > N) do
        copy output tree (N+1, P+1);
    end

else if (root of input tree corresponds to root of output
tree) and (match subtrees = no) then
    begin
    write combined input/output node (level P);
    write input node (level P+1);
    read input node;
    while (input level > M) do
        copy input tree (M+1, P+2);
    write output node (level P+1);
    read output node;
    while (output level > N) do
        copy output tree (N+1, P+2);
    end

else if (root of input tree corresponds to some other node
in this output tree) then
    begin
    write output node (level P);
    read output node;
    while (input level >= M) and (output level > N) do
        match trees (M, N+1, P+1);
    while (output level > N) do
        copy output tree (N+1, P+1);
    end

else if (root of input tree corresponds to a node in some
other output tree) then
    copy output tree (N, P)

else if (root of output tree corresponds to some other node
in this input tree) then
    begin
    write input node (level P);
    read input node;
    while (input level > M) and (output level >= N) do
        match trees (M+1, N, P+1);
    while (input level > M) do
        copy input tree (M+1, P+1);
    end

else if (a node in this input tree corresponds to some
output node) and not (a node in this output tree corresponds
to some input node) then
    copy output tree (N, P)

else copy input tree (M, P);

end;
```

Figure 7 Algorithm for Structural Correspondence and Matching of Trees

Action (a) will produce a program tree

```
02  A-B
    03  C-E
    03  D-F
```

whereas action (b) will produce

```
02  A-B
    03  A
        04  C
        04  D
03  B
    04  E
    04  F.
```

TABLE 1 Treatment of Correspondence Cases

	Root of Input Tree	Root of Output Tree	Root and Subordinates of Program Tree	Continue Matching Subtrees?
1.1	A seq	B seq	A-B seq	Yes
1.2	A seq	B itr	A-B seq A seq B itr	No
1.3	A seq	B sel	A-B seq A seq B sel	No
1.4	A seq	B alt	A-B alt	Yes
1.5	A seq	B rec	A-B rec	Yes
2.1	A itr	B seq	A-B seq A itr B seq	No
2.2	A itr	B itr	If the conditions agree: A-B itr Otherwise: A-B seq A itr B itr Issue a warning message in this case	Yes No
2.3	A itr	B sel	A-B seq A itr B sel	No
2.4	A itr	B alt	A-B alt A itr B seq	No
2.5	A itr	B rec	A-B seq A itr B rec	No
3.1	A sel	B seq	A-B seq A sel B seq	No
3.2	A sel	B itr	A-B seq A sel B itr	No
3.3	A sel	B sel	A-B sel	Yes
3.4	A sel	B alt	A-B alt A sel B seq	No
3.5	A sel	B rec	A-B seq A sel B rec	No
4.1	A alt	B seq	A-B alt	Yes
4.2	A alt	B itr	A-B alt A seq B itr	No
4.3	A alt	B sel	A-B alt A seq B sel	No
4.4	A alt	B alt	A-B alt Issue a warning message if the conditions do not agree	Yes
4.5	A alt	B rec	A-B alt subA-B rec	Yes
5.1	A rec	B seq	A-B rec	Yes
5.2	A rec	B itr	A-B seq A rec B itr	No
5.3	A rec	B sel	A-B seq A rec B sel	No
5.4	A rec	B alt	A-B alt subA-B rec	Yes
5.5	A rec	B rec	A-B rec	Yes

Whether action (a) or (b) is taken depends on the combination of keywords in A and B. In a recent project report, Law (1984) proposes 64 combinations, and even then there are ambiguities left. The present author has modified Jackson's keywords into five (see Section 3.1.3 above), and reduced the number of combinations to 25, as shown in Table 1. For 11 of these cases, action (a) should be taken. Otherwise procedure (b) should be used.

3.2.3 Conditions for Selection and Iteration

The user defines the condition for selection in the data structure after the keyword **alt**. If two **alt** nodes correspond, say "A **alt** (cond1)" and "B **alt** (cond2)", the system will check whether the strings (cond1) and (cond2) are the same. If so, a single condition will result in the program structure. Otherwise a warning message will be given. A similar warning message will also be issued by the system if the conditions of two corresponding **itr** nodes do not agree.

3.2.4 Re-Packing

The system carries on the matching until all the nodes in the input and output trees have been scanned. The resulting program tree may contain some unnecessary hierarchies such as sequences within sequences. A final re-packing is done by the system to simplify the program structure. For example,

02 G **itr**
  03 H **seq**
    04 I
    04 J
  03 K **seq**
    04 L

becomes

02 G **itr**
  03 I
  03 J
  03 L.

3.3 Executable Operations

The system will automatically generate statements for start/stop, open/close, read/write and assignment operations, but not for computational operations. The details are as follows:

3.3.1 Start/stop, Open/close and Read/write Operations

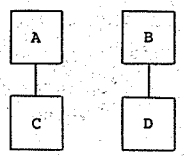
Open statements are generated at the beginning of the pseudo-code. Close and stop statements are generated at the end. A write statement is generated at the end of each output **rec** block. For sequential access, a read statement is generated at the end of each input **rec** block, and also after each "open input" statement. For direct access, a read statement is generated at the beginning of each input **rec** block.

3.3.2 Assignment Operations

Elementary nodes are defined as nodes without subordinates. Whenever two elementary nodes, say A and B, correspond to each other, an assignment statement

B: = A;

(a) Data Structures



where
- A corresponds to B
- match subtrees = no
- C corresponds to D.

(b) Resulting Program Structure

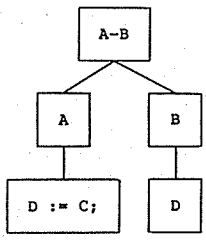


Figure 8 Non-Matched Subtrees

is generated automatically by the system. When an elementary node corresponds to a non-elementary node, no assignment statement is generated, and matching is done according to Table 1 as usual.

Assignment statements are also automatically generated in another situation. When the roots of two trees correspond, the system may or may not be required to match the subtrees. In the latter case, information on node correspondence in the subtrees may be lost. To solve the problem, whenever an elementary input node in a non-matched subtree corresponds to some elementary output node, the system will generate an assignment statement, as shown in Figure 8.

3.3.3 Computational Operations

It is not possible for the development system to "guess" any computation. They have to be defined by the user. In the original version of JSP, computational statements are added to the program structure after the latter has been prepared. In our automated version, this practice is not maintained for two reasons:

- The user is not familiar with the program structure generated automatically by the system.
- Adding statements at the end defeats the purpose of automation.

Instead, computational operations are defined in the data structures by the user. They are distinguished from other types of nodes by the absence of a keyword and the presence of a semi-colon (;). In Example 1, for instance, the operation

total: = total + amt;

can be added under "amt".

A computational statement is not matched with any other node, but is simply reproduced at the appropriate place in the program structure.

(a) Input Data Structure

```
01  orders itr while not endof (orders)
    02  order seq
        03  orderhead rec
            04  orderno
            04  custno
            04  total := 0;
        03  orderbody itr while (detorderno =
            orders.orderno) and not endof (orders)
            04  detail rec
                05  detorderno
                05  amt seq
                    06  total := total + amt;
```

(b) Output Data Structure

```
01  report seq
    02  reporthead rec
    02  reportbody itr while not endof (orders)
        03  totalline rec
            04  orderno
            04  custno
            04  total
```

Figure 9 Specification of Data Structures

```
01  report seq
    02  open (custs);
    02  open (orders);
    02  open (report);
    02  read (orders);
    02  write (reporthead);
    02  custs-orders-reportbody itr while not endof (orders)
        03  report.orderno := orders.orderno;
        03  read (custs (orders.custno));
        03  report.custno := orders.custno;
        03  cust1 posit
            04  cust1 quit if not (found);
            04  report.name := custs.name;
        03  cust2 admit
            04  report.name := "warning";
        03  total := 0;
        03  read (orders);
        03  orderbody itr while (orders.orderno =
            report.orderno)
            04  total := total + amt;
            04  read (orders);
        03  write (totalline);
    02  close (custs);
    02  close (orders);
    02  close (report);
    02  stop;
```

Figure 10 Multiple Input Data Structures and Backtracking

Example 1 Revisited

As an illustration, the input and output data structures of Example 1 are re-specified following Sections 3.1 and 3.3, and are shown in Figure 9. The program structure generated automatically by the system closely resembles Figure 6, apart from minor syntactical deviations such as the use of level numbers and the omission of **end** nodes, as described in Section 3.1.

3.4 Multiple Input/Output Data Structures

The system can handle multiple input and/or multiple output data structures. Matching and merging is done as usual on two of the data structures, resulting in a provisional program tree. The provisional result is then matched with a third data structure to give a more accurate program tree. The system will continue in this way, accepting data structures one at a time, until the final program structure is obtained.

Example 2

Suppose the requirements for Example 1 are modified. The records of all the customers are held in a separate direct access file. When an order is read, the program must read the appropriate cus-

```
01  report seq
    02  goto tp (textpointer);
    02  tp (1):  continue;
    02  validate itr while (invalid)
        03  write (diagnostics (jobno));
        03  textpointer := 2;
        03  return;
        03  tp (2):  continue;
    02  write (reporthead (jobno));
    02  custs-orders-reportbody itr while not endof (orders)
        03  report.orderno := orders.orderno;
        03  read (custs (orders.custno));
        03  report.custno := orders.custno;
        03  custl posit
            04  custl quit if not (found);
            04  report.name := custs.name;
        03  cust2 admit
            04  report.name := "warning";
        03  total := 0;
        03  textpointer := 3;
        03  return;
        03  tp (3):  continue;
        03  validate itr while (invalid)
            04  write (diagnostics (jobno));
            04  textpointer := 4;
            04  return;
            04  tp (4):  continue;
        03  orderbody itr while (orders.orderno =
            report.orderno)
            04  total := total + amt;
            04  textpointer := 5;
            04  return;
            04  tp (5):  continue;
            04  validate itr while (invalid)
                05  write (diagnostics (jobno));
                05  textpointer := 6;
                05  return;
                05  tp (6):  continue;
        03  write (totalline (jobno));
    02  textpointer := 7;
    02  return;
    02  tp (7):  continue;
    02  return;
```

Figure 11 Interactive Input and Program Inversion

tomer record and place the customer name on to the report as part of the total line.

One extra input data structure – the customer file – will have to be specified to the development system. Pseudo-code is generated automatically and is shown in Figure 10.

3.5 Backtracking

To specify backtracking, the user can insert **quit if** statements into the data structures as he sees fit. The system will place them in the appropriate positions in the program structure. The user can also add **posit** and **admit** nodes into the data structures. The system will match them as if they are **alt** nodes.

An example of backtracking is included in Figure 10.

3.6 Interactive Input

If interactive input is required in a program, the system will generate an iteration, writing out diagnostics according to Section 2.3.

3.7 Program Inversion

The program inversion procedure is quite tedious and hence error prone. Furthermore, the final product – the inverted module – is quite unreadable and looks like an assembly language program to many people. To solve the problem, our system will invert programs automatically on request. As a result, the inversion process is transparent to the user.

Example 2 Revisited

As an illustration, we have converted the program in Example 2 to accept interactive input, and have further inverted it for a multi-user environment. The result is shown in Figure 11.

4. FURTHER STUDIES

We have chosen to generate Jackson pseudo-code from the present system because we would like to test it with the examples from Jackson. It should be possible to modify the system and generate programs in a high level language, but further experimentation would be required.

Besides serving as a development system for JSP, the present system should also be a useful aid for Jackson's system development (JSD). For example, once the entity structures have been defined, part of the initial model step (1983: 121-170) and the function step (1983: 171-245) can be undertaken by the system. The program inversion function of our system would also be useful in the implementation step (1983: 256-332). Further studies should be made in these directions.

5. CONCLUSIONS

The program development system under study can accept input and output data structures and generate pseudo-code according to Jackson's structured programming methodology. Executable operations other than computations can be generated automatically. The number of input and output data structures can be unlimited since they can be accepted in turn by the system. Backtracking is solved by accepting **"quit if"** operations in the data structures. The system also supports program inversion, which is essential for structure clash and for multi-user interactive systems.

ACKNOWLEDGMENTS

The author is indebted to S.F. Law, then at the University of Hong Kong, for his assistance in implementing an earlier version of the system. He is also grateful to Richard Kaczynski of the London School of Economics, University of London, for his technical assistance during a major revision of the paper. Part of this research was done at the London School of Economics, University of London, under a Commonwealth Academic Staff Scholarship.

References

ALFORD, M.W. (1982): Software requirements engineering methodology (SREM) at the age of two, *Advanced System Development Feasibility Techniques,* Couger, J.D., Colter, M.A. and Knapp, R.W. (eds.), Wiley, New York, pp. 385-398.

BIGGERSTAFF, T.J. (1979(: The unified design specification system (UDS2), *Proceedings of Conference on Specifications of Reliable Software,* IEEE, New York, pp. 104-118.

BOHM, C. and JACOPINI, G. (1966): Flow diagrams, Turing machines and languages with only two formation rules, *Communications of ACM,* Vol. 0, No. 2, pp. 366-371.

CAMPOS, M.I. and ESTRIN, G. (1978): Concurrent software system design supported by SARA at the age of one, *Proceedings of 3rd International Conference on Software Engineering,* IEEE, New York, pp. 230-242.

DAHL, O.J., DIJKSTRA, E.W. and HOARE, C.A.R. (1972): *Structured Programming,* Academic Press, London.

DAVIS, A.M. (1982): The design of a family of application-oriented requirements languages, *Computer,* Vol. 15, No. 5, pp. 21-28.

DeMARCO, T. (1978): *Structured Analysis and System Specification,* Prentice-Hall, Englewood Cliffs, New Jersey.

GANE; C. and SARSON, T. (1979): *Structured Systems Analysis: Tools and Techniques,* Prentice-Hall, Englewood Cliffs, New Jersey.

GRIFFITHS, S.N. (1978): Design methodologies – a comparison, *Structured Analysis and Design, Vol. 2, Infotech State of the Art Report, Maidenhead, England, pp. 133-166.*

INGEVALDSSON, L. (1979): *JSP: a Practical Method of Program Design,* Input Two-Nine, London.

JACKSON, M.A. (1975): *Principles of Program Design,* Academic Press, London.

JACKSON, M.A. (1980): Constructive methods of program design, *Tutotial on Software Design Techniques,* Freeman, P. and Wasserman, A.I. (eds.), IEEE, New York, pp. 394-412. Also *Lecture Notes in Computer Science,* Vol. 44 (1976), Springer-Verlag, Verlin, pp. 236-262.

JACKSON, M.A. (1983): *System Development,* Prentice-Hall, Englewood Cliffs, New Jersey.

KING, D. (1982): Jackson design methodology, *Software Engineering Notes,* Vol. 7, No. 2, pp. 33-34.

LAW, S.F. (1984): Data structure design methodology, SC 1301 Project Report, Centre of Computer Studies and Applications, University of Hong Kong, Hong Kong.

MARTIN, J. (1984): *An Information Systems Manifesto,* Prentice-Hall, Englewood Cliffs, New Jersey.

ORR, K.T. (1977): *Structured Systems Development,* Yourdon, New York.

STEPHENS, S.A. and TRIPP, L.L. (1978): Requirements expression and verification aid, *Proceedings of 3rd International Conference on Software Engineering,* IEEE, New York, pp. 101-108.

TEICHROEW, D. and HERSHEY, E.A. III (1977): PSL/PSA: a computer-aided technique for structured documentation and analysis for information processing systems, *IEEE Transactions on Software Engineering,* Vol. SE-3, No. 1, pp. 41-48.

TSE, T.H. (1985): Towards a unified algebraic view of structured analysis and design models, Technical Report, TR-A6-85, Centre of Computer Studies and Applications, University of Hong Kong, Hong Kong.

WASSERMAN, A.I. (1982): The user software engineering methodology: an overview, *Information Systems Design Methodologies: a Comparative Review,* Olle, T.W., Sol., H.G. and Verrijn-Stuart, A.A. (eds.), North-Holland, Amsterdam, pp. 591-628.

WEINBERG, V. (1980): *Structured Analysis,* Prentice-Hall, Englewood Cliffs, New Jersey.

YAMAMOTO, Y. (1981): An approach to the generation of software life cycle support systems, Ph.D. Thesis, University of Michigan, Ann Arbor, Michigan.

YOURDON, E. and CONSTANTINE, L.L. (1979): *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design,* Prentice-Hall, Englewood Cliffs, New Jersey.

Biographical Note

T.H. Tse obtained his B.Sc. degree from the University of Hong Kong in 1970, and his M.Sc. from the University of London in 1979. He is currently a lecturer in computer science at the University of Hong Kong. His research interests include software engineering and formal aspects of information processing.

Mr Tse is a member of the British Computer Society, the British Institute of Management and the Institute of Data Processing Management. He is a council member of the Vocational Training Council in Hong Kong. He was awarded an MBE by the Queen in 1982.