CANADIAN COMPUTER CONFERENCE

SESSION '78

LA CONFÉRENCE CANADIENNE DE L'INFORMATIQUE

ASSISES '78

PROCEEDINGS

LES RÉSUMÉS DES CONFÉRENCES

MONTRÉAL, QUÉBEC

MAI 17-19, 1978

TOWARDS LOGIC AND PERFORMANCE ANALYSIS OF UNWRITTEN PROGRAMS

G. Belkin, E. M. Jaworski
Computer Science Department
Concordia University
Montreal, Quebec

INTRODUCTION

This paper describes a set of "tools" which provide the programmer with certain aids that we feel can be valuable in the designing of computer programs. Our "system" consists of a formalized table notation, and an interpretive processor with which the tables can be executed and analyzed. The table notation stems from basic decision table concepts [9], and is partially described in another paper, "Primitive Logical Constructs Considered Harmful in Structured Programs", by Fancott and Jaworski [4]. Our processor is experimental, and does not yet contain many of the features which we would like, but we believe that the design methods which we propose are feasible and do, in fact, aid in the program design process. The aim of this paper is simply to demonstrate some of these "tools" that our notation and processor can provide to the computer programmer.

NOTATION

The table notation which our processor handles is an early version of that presented in [4], and will be recognized as a form of limited-entry decision table [9]. Some of the more-apparent differences between this notation and conventional decision table notation are:

(a) Tables are linked in a structured fashion,

(b) A table need not have any conditions, in which case it is treated in a manner analagous to a block in most structured programming languages,

(c) Conditions and actions must be written in APL notation -- (this is a restriction of the experimental processor),

(d) Action "entries" (indications of which actions are to be performed by a particular rule) are expressed as numbers, which specify the order of action execution, and

(e) A number of "special" actions have been defined, of which we shall only have cause to see the "DO" and "REPEAT" actions in this paper. The "DO" action is nothing more than a "call" to execute another table, and is synonymous to the "PERFORM" discussed in [4]. The "REPEAT" simply causes the current table to be re-evaluated, creating a loop.

PROCESSOR

The "processor" is really a collection of APL functions which together provide for definition, display, execution, and analysis of tables, in an interactive manner. Eventually, it is hoped that more powerful table editing and manipulation functions will be added to the system which, for ease of reference,

will be called TAP (Table Analysis Package). As the currently-implemented version of TAP is just an experimental subset of an envisioned system, we shall refer to it as TAP/X.

TAP/X FEATURES

To demonstrate features of TAP/X, we shall test and analyze a problem specification and program of Naur [10], presented by Goodenough and Gerhart [5]. The problem is one of text reformatting, and its specifications are given as follows:

Input properties:

"I1) Input is a stream of characters, where the characters are classified as break and nonbreak characters. A break character is a BL (blank), NL (new line indicator), or ET (end-of-text indicator).
I2) The final character in the text is ET.
I3) A word is a nonempty sequence of nonbreak characters.
I4) A break is a sequence of one or more break characters."

Output properties:

"O1) A new line should start only between words and at the beginning of the output text, if any.
O2) A break in the input is reduced to a single break character in the output.
O3) As many words as possible should be placed on each line (i.e., between successive NL characters).
O4) No line may contain more than MAXPOS characters (words and BL's).
O5) An oversize word (i.e., a word containing more than MAXPOS characters) should cause an error exit from the program (i.e., a variable ALARM should have the value TRUE)."

The authors of [5] correctly identify numerous errors or problems contained in Naur's original program, and then set out to correct them. The program, which they claim satisfies the problem specifications, is shown in fig. 1. We apparently must assume that (1) MAXPOS is defined (as the maximum number of characters per output line), (2) BUFFER is defined (and is at least MAXPOS characters in length), and (3) INCHARACTER and OUTCHARACTER are either defined language constructs or procedures.

Before attempting to devise test data for their program, the authors decided to convert the program to a limited-entry decision table, thus making the various "cases" of the program more evident. The table which they produce has been reproduced as fig. 2, restated in the notation of TAP/X, but otherwise unchanged.

Without explaining exactly how this table representation was arrived at, the authors proceed to use it as a model of their program, in order to select test data. But does it actually represent their program? Where are the initialization statements (statements 1,2, and 3 of the original program)? Where is the first execution of INCHARACTER(CH), which in the original program occurs before any of the condition tests? Where is the test of the ALARM variable, which


```
Alarm := false;
bufpos := 0;
fill := 0;
repeat
   readcharacter(CH);
   if CH = BL ∨ CH = NL ∨ CH = RT
   then begin
      if bufpos ≠ 0
      then begin
         if fill + bufpos < MAXPOS ∧ fill ≠ 0
         then begin
            outcharacter(BL);
            fill := fill + 1;end
         else begin
            outcharacter(NL);
            fill := 0   end
         for k := 1 step 1 until bufpos do
            outcharacter(buffer[k]);
         fill := fill + bufpos ;
         bufpos := 0  end end
   else
      if bufpos = MAXPOS
      then Alarm := true;
      else begin
         bufpos := bufpos + 1;
         buffer[bufpos] := CH  end
until Alarm ∨ CH = RT ;
```

Fig. 1. Corrected version of Naur's program.

DISPLAY FORMAT

FORMAT1 TEXT FORMATTER

(CH=BL)∨(CH=NL)	T	T	T	T	N	N	N	N	N	N
CH=RT	-	-	-	-	T	T	T	T	N	N
BUFPOS=0	T	T	T	N	T	T	T	N	-	-
(FILL+BUFPOS)<MAXPOS	T	T	N	-	T	T	N	-	-	-
FILL=0	T	N	-	-	T	N	-	-	-	-
BUFPOS=MAXPOS	-	-	-	-	-	-	-	-	T	N
OUT←OUT,BL	1	-	-	-	1	-	-	-	-	-
FILL←FILL+1	2	-	-	-	2	-	-	-	-	-
OUT←OUT,NL	-	1	1	-	-	1	1	-	-	-
FILL←0	-	2	2	-	-	2	2	-	-	-
OUT←OUT,BUFFER(1:BUFPOS)	3	3	3	-	3	3	3	-	-	-
FILL←FILL+BUFPOS	4	4	4	-	4	4	4	-	-	-
BUFPOS←0	5	5	5	-	5	5	5	-	-	-
ALARM←1	-	-	-	-	-	-	-	-	1	-
BUFPOS←BUFPOS+1	-	-	-	-	-	-	-	-	-	1
BUFFER(BUFPOS)←CH	-	-	-	-	-	-	-	-	-	2
CH←READINPUT	6	6	6	1	-	-	-	-	-	3
INPUT←READINPUT	7	7	7	2	-	-	-	-	-	4
REPEAT	8	8	8	3	-	-	-	-	-	5
* COMPLETED	-	-	-	-	4	4	4	1	2	-

Fig. 2. Decision table restated in TAP/X notation.

in the original program controls the repeat until loop? All of these have been mysteriously removed during the conversion to table format. This is not to say that the table in fig. 2 is incorrect, but simply that it is not a true representation of the program in fig. 1. In fact, with the addition of the initialization steps:

```
ALARM := false;
BUFPOS := 0;
FILL := 0;
INCHARACTER(CH);
```

the table in fig. 2 satisfies the problem specifications, as claimed.

Although Goodenough and Gerhart apparently had to rely on manual evaluation of their table to ensure its "correctness", we were able to present the table to TAP/X, and interactively examine its operation. To illustrate this process, we have selected two of the authors' test data sets and executed the table with the processor's execution-flow trace feature activated. The results of these runs (figs. 3(a) and 3(b), respectively), verify that the exercised rule sequence for each case is as stated in Goodenough and Gerhart's paper. Further executions with other test data also gave results which satisfied the problem specifications.

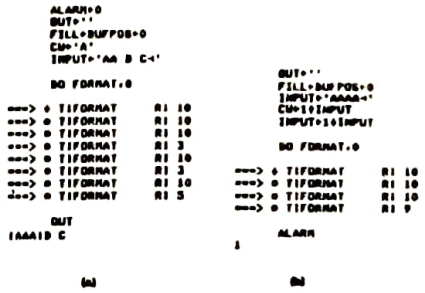

Fig. 3. Test runs of program from fig. 2.

Although we have concluded that the table in fig. 2 appears to satisfy the stated problem specifications, has this really told us anything about the coded program in fig. 1? Since we have seen that the table is not an accurate representation of the program, we must conclude that our observations thus far apply only to the table itself.

Returning to the "coded" version of the program (fig. 1), we find that, using our notation, we can convert it to tables in such a way that it is accurately represented. Such a translation is presented in fig. 4. Numerous executions of this set of tables revealed that they do indeed seem to satisfy the stated requirements of the problem, and thus, so does the program.

```
DISPLAY ALL

FORMAT1  INITIALIZATION

-----------------------
OUT+''                1
ALARM+0               2
BUFPOS+0              3
FILL+0                4
BUFFER+MAXPOS+' '     5
DO CONTROL.0          6
CH+1@INPUT            7
INPUT+1@INPUT         8
DO PROCESS.0          9

CONTROL1  GET CHARACTER

ALARM v CH=ET       N Y
-------------------- - -
CH+1@INPUT          1 -
INPUT+1@INPUT       2 -
DO PROCESS.0        3 -
REPEAT              4 -
@ END TABLE         - 1

PROCESS1  PROCESS INPUT STREAM

(CH=BL)v(CH=NL)v(CH=ET)          Y Y Y N N
BUFPOS+0                         Y Y N - -
(FILL+BUFPOS)<MAXPOS)^FILL=0     Y N - - -
BUFPOS=MAXPOS                    - - - Y N
-------------------------------------------
BUFPOS+0                         6 6 - - -
FILL+0                           - 2 - - -
OUT+OUT.BL                       1 - - - -
FILL+FILL+1                      3 - - - -
OUT+OUT.NL                       - 1 - - -
K+1                              2 2 - - -
DO PRINTW.0                      4 4 - - -
FILL+FILL+BUFPOS                 5 5 - - -
ALARM+1                          - - - 1 -
BUFPOS+BUFPOS+1                  - - - - 1
BUFFER[BUFPOS]+CH                - - - - 3
@ END TABLE                      7 7 1 2 2

PRINTW1  PRINT WORD

K=BUFPOS              N Y
----------------------- -
OUT+OUT.BUFFER[K]     1 1
K+K+1                 2 -
REPEAT                3 -
```

Fig. 4. Program from fig. 1 rewritten in TAP/X notation.

As an experiment in program design using our table notation, another solution to the problem was developed directly from the problem specifications. Although many formal "refinements" [12] were applied during its design, we present in fig. 5 only the first "completed" version of our solution. Notice that the design is somewhat different, in that we have chosen to consider each character in the context of the previous character, to determine the "case" of the problem.


```
FORMAT1  INITIALIZATION

-                    -
---------------------
FILL+0               1
BUFPOS+0             2
OUT+NL               3
PC+BL                4
CH+BL                5
DO PROCESS.0         7
BUFFER+MAXPOS*' '    6

PROCESS1  PROCESS INPUT STREAM

CH = ET                    N N N - - Y
(CH = BL) v (CH = NL)      N N N Y Y -
(PC = BL) v (PC = NL)      Y N N N Y -
BUFPOS = MAXPOS            - Y N - - -
---------------------------------------
PC+CH                      1 - 1 - 1 -
BUFPOS+1                   2 - - - - -
BUFPOS+BUFPOS+1            - - 2 - - -
BUFFER[BUFPOS]+CH          3 - 3 - - -
DO BREAK.0                 - - - 1 - 1
CH+1@INPUT                 4 - 4 2 2 -
INPUT+1@INPUT              5 - 5 3 3 -
ALARM+0                    - - - - - 2
ALARM+1                    - 1 - - - -
REPEAT                     6 - 6 4 4 -
@ END TABLE                - 2 - - - 3

BREAK1  PRINT CONTROLLER

BUFPOS = 0                 Y N N
(FILL + BUFPOS) < MAXPOS   - Y N
---------------------------------
PC+CH                      - 4 4
OUT+OUT.BL                 - 1 -
OUT+OUT.NL                 - - 1
DO PRINTW.0                - 3 3
FILL+FILL + BUFPOS         - 5 -
FILL+BUFPOS                - - 5
BUFPOS+0                   - 6 -
K+K+1                      - 2 2
@ END TABLE                1 7 6

PRINTW1  PRINT WORD

K = BUFPOS                 Y N
---------------------------------
REPEAT                     - 3
OUT+OUT.BUFFER[K]          1 1
K+K+1                      - 2
@ END TABLE                2 -
```

Fig. 5. Program developed from program specification.

The first test execution of this solution is shown in fig. 6. As this example illustrates, the result is incorrect, in that it contains two "new-line" (NL) characters at the beginning of the output string. This example also serves to further demonstrate the execution tracing facility of TAP/X. Note that the trace messages are indented according to the "abstraction" level [12] of each table. (In TAP/X, this corresponds to the "level" of the table on the current execution path). Displaying the trace messages in this manner permits one to easily see how execution "moves" from one level to another, and shows at each point which tables are still "active" (still in control of execution). The problem of deciding where to place

trace messages in a program [1] is eliminated since the structure inherent in the table notation causes all possible "branches" in a program to be table "transfers", and every table is automatically "traced" as it is entered. (Further planned additions to TAP/X include the ability to selectively trace only certain tables).

```
      INPUT> 'AAA B C+'

      DO FORMAT,0

===> 0 TIFORMAT      RI 1
   ===> 0 TIPROCESS     RI 5
   ===> 0 TIPROCESS     RI 1
   ===> 0 TIPROCESS     RI 3
   ===> 0 TIPROCESS     RI 3
   ===> 0 TIPROCESS     RI 4
      ===> 0 TIBREAK       RI 3
         ===> 0 TIPRINTN      RI 2
         ===> 0 TIPRINTN      RI 2
         ===> 0 TIPRINTN      RI 1

   ===> 0 TIPROCESS     RI 1
   ===> 0 TIPROCESS     RI 4
      ===> 0 TIBREAK       RI 3
         ===> 0 TIPRINTN      RI 1

   ===> 0 TIPROCESS     RI 1
   ===> 0 TIPROCESS     RI 4
      ===> 0 TIBREAK       RI 2
         ===> 0 TIPRINTN      RI 1


      OUT
 AAAB C
```

Fig. 6. Test run no. 1.

Before attempting to discover the cause of the error in fig. 6, we decided to execute the tables with other test data, to check for further errors. In the second test (fig. 7), we see that the output string contains a leading blank (after the NL character), which is inconsistent with the specifications.

Although the set of tables being considered is not very complex, and could probably be checked manually at this point, we have chosen to present the methods by which TAP/X might aid in the location of the cause of these errors.

Fig. 7 shows the result of table execution with the "full" action trace feature activated. In this mode, every action is displayed immediately before it is executed. Looking at this list of actions, we see that the first entry to the output buffer (point A in fig. 7) is the NL character, which seems reasonable in view of the specifications. The next entry to the output buffer (point B) is a blank, which is obviously why the result contains a leading blank! Resorting to another TAP/X feature, we "disable" table BREAK, within which the leading blank is being inserted. This "disable" feature permits selective removal (isolation) of a table within a set of tables. Fig. 8 shows execution with the same test data as in fig. 7, but with table BREAK disabled. When TAP/X attempts to execute the action "DO BREAK,0" in table PROCESS, it informs us of the disabled status of that table, and invites us to take over manually, using the full interactive capability of APL, and having access to all program variables in their current states. From the previous example, we know that rule 2 of table BREAK

```
      INPUT← ' A B ='

      DO FORMAT.0

===> ◊ TIFORMAT     RI 1
         FILL←0
         BUFPOS←0
         OUT←NL
         PC←DL
         CW←DL
         BUFFER←MAXPOS↑'
         DO PROCESS.0

   ===> ◊ TIPROCESS    RI 5
            PC←CW
            CW←1↓INPUT
            INPUT←1↓INPUT
            REPEAT

   ===> ◊ TIPROCESS    RI 5
            PC←CW
            CW←1↓INPUT
            INPUT←1↓INPUT
            REPEAT

   ===> ◊ TIPROCESS    RI 1
            PC←CW
            BUFPOS←1
            BUFFER[BUFPOS]←CW
            CW←1↓INPUT
            INPUT←1↓INPUT
            REPEAT

   ===> ◊ TIPROCESS    RI 4
            DO BREAK.0

      ===> ◊ TIBREAK      RI 2
               OUT←OUT,DL
               K←1
               DO PRINTW.0

         ===> ◊ TIPRINTW     RI 1
                  OUT←OUT,BUFFER[K]
                  ◊ END TABLE

               PC←CW
               FILL←FILL + BUFPOS
               BUFPOS←0
               ◊ END TABLE

            CW←1↓INPUT
            INPUT←1↓INPUT
            REPEAT
```

```
===> ◊ TIPROCESS    RI 1
         PC←CW
         BUFPOS←1
         BUFFER[BUFPOS]←CW
         CW←1↓INPUT
         INPUT←1↓INPUT
         REPEAT

===> ◊ TIPROCESS    RI 4
         DO BREAK.0

   ===> ◊ TIBREAK      RI 2
            OUT←OUT,DL
            K←1
            DO PRINTW.0

      ===> ◊ TIPRINTW     RI 1
               OUT←OUT,BUFFER[K]
               ◊ END TABLE

            PC←CW
            FILL←FILL + BUFPOS
            BUFPOS←0
            ◊ END TABLE

         CW←1↓INPUT
         INPUT←1↓INPUT
         REPEAT

===> ◊ TIPROCESS    RI 4
         DO BREAK.0

   ===> ◊ TIBREAK      RI 1
            ◊ END TABLE

         ALARM←0
         ◊ END TABLE

   OUT
 A B
```

Fig. 7. Test run with action trace.

would be executed next, and upon examination of the variables BUFPOS and FILL, we can verify that this is correct. ("BUFPOS=0" is false, and since "MAXPOS=3, "(0+1)<3" is true). But since rule 2 inserts a blank and rule 3 does not, we would like, if possible, to force rule 3 to be executed next.

```
      EXEC TRACE ON!
  TRACE ON ( 1)

      DISABLE BREAK
  TABLE! BREAK       DISABLED

      INPUT←' A B ='

      DO FORMAT.0

===> ◊ TIFORMAT     RI 1
   ===> ◊ TIPROCESS    RI 5
   ===> ◊ TIPROCESS    RI 5
   ===> ◊ TIPROCESS    RI 1
   ===> ◊ TIPROCESS    RI 4

 **TABLE! BREAK       DISABLED
   CALLED FROM TABLE! PROCESS   (VIA DO)
   USER ACTION REQUESTED ('EXIT' CONTINUES)

BUFPOS
1

FILL
0

OUT
|

FILL←MAXPOS

ENABLE BREAK
 TABLE! BREAK       ENABLED

REPEAT

      ===> ◊ TIBREAK      RI 1
         ===> ◊ TIPRINTW     RI 1


   ===> ◊ TIPROCESS    RI 1
   ===> ◊ TIPROCESS    RI 4
      ===> ◊ TIBREAK      RI 2
         ===> ◊ TIPRINTW     RI 1


   ===> ◊ TIPROCESS    RI 1
   ===> ◊ TIBREAK      RI 1


      OUT
|A B
```

Fig. 8. Test run with table BREAK 'disabled' then 'enabled'.

To ensure that rule 3 is selected, we decided to set FILL=MAXPOS, so that "(FILL+BUFPOS)<MAXPOS" will be false. Having done this, we re-enabled table BREAK, and continued execution. (The REPEAT action causes table BREAK to be executed, now that it is re-enabled).

The result of this test indicates that we have succeeded in eliminating the leading blank, but we now have two NL characters at the start of the output. This is easily understood, though, since fig. 8 shows that prior to executing table BREAK we already had an NL character in the output buffer, and rule 3 of BREAK inserts an NL character as its first action (see fig. 5).

After further examination of fig. 7, we observed that the variables OUT and FILL are not modified (after initialization) until table BREAK is executed. Since initially forcing rule 3 of BREAK seems to correct the leading-blank problem, we can therefore set FILL initially to MAXPOS instead of zero. Similarly, since rule 3 of BREAK (which will now always be executed initially) inserts an NL character, we can refrain from starting the output buffer with an NL character, and initialize OUT to "empty". These two corrections to the initialization table eliminated the problems, and subsequent testing of the tables failed to reveal any

other errors.

There is another feature of TAP/X which has proven itself very useful in our experiments in table testing — a cumulative count of executions of each table, and of each rule within each table. It is interesting to note that sometimes, after numerous tests of a set of tables, a display of these execution statistics shows that there is yet a rule somewhere which has never been executed. This may be due to insufficient testing, and knowledge of it may lead to discovery of a problem which could otherwise remain undetected for some time. Or it may be due to incorrect logic specification within a table, which is itself an important discovery.

```
TABLE: FORMAT                    TABLE: BREAK
   TEC=  10                         TEC=  107
   REC=  10                         REC=  0 73 34

TABLE: PROCESS                   TABLE: PRINTN
   TEC=  613                        TEC=  473
   REC=  100 1 360 98 29 9          REC=  107 366

              TEC = Table Execution Count
              REC = Rule  Execution Count
```

Fig. 9 Execution Statistics for program in fig. 5

An example of these execution statistics is given in fig. 9, showing the counts accumulated over 10 test runs of the tables in fig. 5. If we assume that the test data was representative of actual data which might be processed by this sort of program, then the observation that rule 3 of table PROCESS has the

highest frequency of execution could lead us to a better understanding of the problem, the program, and/or the data.

CONCLUSIONS

What has all this to do with "unwritten" programs? While it is true that any program or program segment which we wish to analyze must be written in some form, TAP/X allows us to analyze a table at a high level of abstraction, even if the lower-level tables have not yet been designed. This is made possible by creating "dummy" low-level tables and disabling them. We can then execute the higher-level table, manually "simulating" the "unwritten" tables as required. (Although we may not yet know how to program the lower levels, we do know what results we expect of them!)

There is also another interpretation of the word "unwritten" as used here — programs developed with the aid of TAP/X are unlikely to be left in their designed form, especially considering the lack of a compiler for that notation. Thus the programs, although they have been designed and tested, have yet to be "coded" in some conventional language. TAP/X can help in this area as well, since the execution statistics which it maintains can guide the programmer towards producing an efficiently-coded program.

The features described in this paper represent only some of those "tools" which we have already implemented in TAP/X. Others include tracing of selected data values through program execution, counts of condition tests and action executions actually performed for each rule of a table, additional table-linkage facilities permitting the creation of more complex program loops, and capability of representing a condition as a table call and explicitly returning a true or false value for that condition.

In summary, we feel that there is a need for testing and analysis of programs during the design stage of development, and that the TAP methods described offer one possible approach to this problem.

REFERENCES

[1] B. W. Boehm, "Some Steps Toward Formal and Automated Aids to Software Requirements Analysis and Design", Information Processing 74, North-Holland Publishing Company, 1974, pp. 192-197.

[2] J. Cohen, and C. Zuckerman, "Two Languages for Estimating Program Efficiency", Communications of the ACM, 17, 6, June 1974, pp. 301-308.

[3] W. E. Fairley, "An Experimental Program-Testing Facility", IEEE Transactions on Software Engineering, SE-1, 4, December 1975, pp. 350-357.

[4] T. Fancott, and W. L. Jaworski, "Primitive Logical Constructs Considered Harmful in Structured Programs", Conference Proceedings, Canadian Computer Conference, Session 478, Montreal.

[5] J. B. Goodenough, and S. L. Gerhart, "Toward a Theory of Test Data Selection", IEEE Transactions on Software Engineering, SE-1, 2, June 1975, pp. 156-173.

[6] P. Henderson, and R. A. Snowdon, "A Tool for Structured Program Development", Information Processing 74, North-Holland Publishing Company, 1974, pp. 204-207.

[7] J. C. Huang, "An Approach to Program Testing", Computing Surveys, 7, 3, September 1975, pp. 113-128.

[8] M. M. Jaworski, "An Interactive System for the Generation of Programs from Decision Tables", Computer Aided System Simulation, Analysis & Design, (CASSAD 70), University of Houston, 1970.

[9] M. Montalbano, "Decision Tables", Science Research Associates, Inc., 1974.

[10] P. Naur, "Programming by Action Clusters", BIT, 9, 3, 1969, pp. 250-258.

[11] K. H. Kim Ramamoorthy, and M. T. Chan, "Optimal Placement of Software Monitors Aiding Systematic Testing", IEEE Transactions on Software Engineering, SE-1, 4, December 1975, pp. 403-411.

[12] N. Wirth, "On the Composition of Well-Structured Programs", Computing Surveys, 6, 4, December 1974, pp. 247-259.