NEWTON--A DYNAMIC PROGRAM ANALYSIS TOOL CAPABILITIES SPECIFICATION

by

Jonathan D. Feiber
Richard N. Taylor
Leon Osterweil

Department of Computer Science
University of Colorado
Boulder, Colorado 80309

CU-CS-200-81 February, 1981

This material is based upon work supported by the National Science Foundation under Grant No. MCS8000017, and DOE Grant No. DE-AC02-80ER10718.

I. Introduction

This paper contains a specification for a dynamic program testing aid for Fortran programs, called Newton. Like most contemporary dynamic testing tools Newton aids the testing process by effecting the instrumentation of subject programs and then collecting, organizing and reporting the results of executing the instrumented program. This process is illustrated in Figure 1.1.

The Newton specification has been developed over a period of several months of reading, evaluation and original thought. It has been significantly influenced by earlier work in this area. Most specifically, the early work of Fosdick [FOSD74], Ramamoorthy [RAMA75], Fairley [FAIR75], Stucki [STUC73], and Brown [BROW73] established the pattern for subsequent dynamic testing systems. Stucki [STUC75] and Fairley [FAIR75] first most cogently advanced the notion of dynamic assertion checking. Taylor [TAYL80] improved upon these ideas and formalized the assertion language upon which Newton's is based.

Perhaps the most original aspects of Newton are those which are present because Newton, unlike most earlier dynamic test tools, is designed to be integrated into a comprehensive tool environment, called TOOLPACK. Thus it is expected that Newton will not be directly accessible by end users, but rather will be invoked through the TOOLPACK user interface. The usage of Newton, and other TOOLPACK tools, will, moreover, be enhanced by the support of other system utilities such as the TOOLPACK data base/information management system and diagnostic output browsing facilities [OSTE81]. Because of this, the Newton specification focuses more on the capabilities to be provided and less on human interfaces.

For example, Newton users will have considerable flexibility in selection from among a considerable number of instrumentation options. This has typically been considered necessary, but a definite drawback is that it can be intimidating, especially to novice users. This situation is helped considerably by thinking of Newton as being embedded in TOOLPACK. Any set of selected options can be given a name and stored in the TOOLPACK data base as a Test Options Packet. Experienced users, or the system creators can initially create these packets and store them

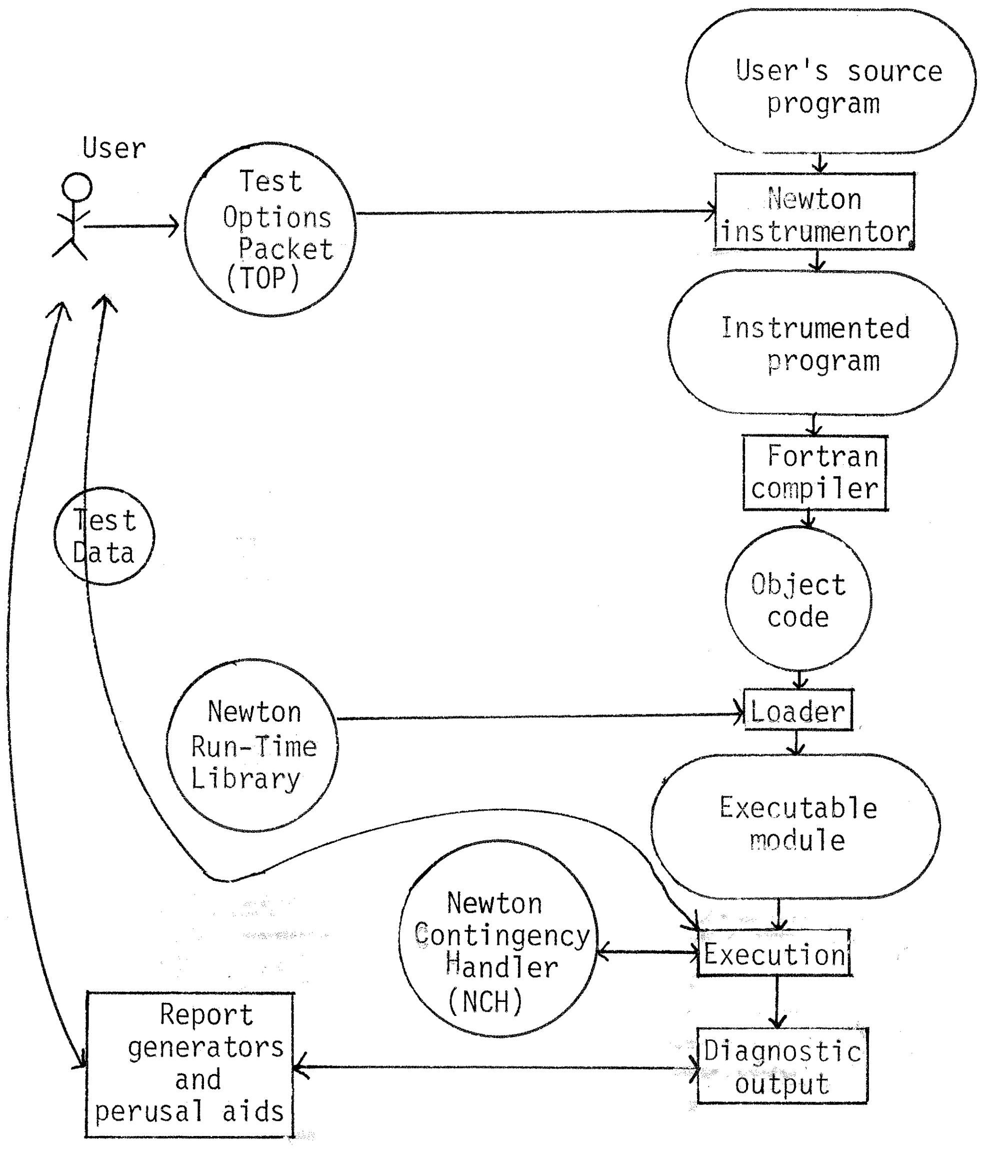

Figure 1.1: Use of Dynamic Test Tools

under write protection for later use by novice users having no need to understand the option specification language. More advanced or sophisticated users can copy and then modify these packets or create their own packets according to their own particular requirements. With these usage scenarios in mind we have permitted ourselves the luxury of an extensive command language.

Similarly, with the prospect of a facility for browsing a diagnostic output data base in mind, we have permitted ourselves the luxury of hypothesizing extensive and detailed outputs from the imbedded test probes. The effectiveness of many earlier dynamic test systems was severely reduced because their output was voluminous, unstructured and inaccesible by tools for browsing. Other tools strove for comprehensibility by restricting diagnostic output. In Newton we enable the user to specify the emission of complete and comprehensive outputs from embedded probes. These outputs are then considered to go into a structured data base. The data base is to be queriable effectively with the aid of interactive browsers either after termination of the run, or during the execution of the test run in response to user specifiable breakpoint commands, or after detection of a program error on assertion violation.

2.0 User Communication with Newton

Newton requires that the user specify a text file containing syntactically correct FORTRAN 77 source text, a Newton Test Option Packet.

2.1 Test Option Packet

The Test Option Packet (TOP) provides program analysis control over the user FORTRAN 77 source text. Each TOP is a named set of program analysis commands and program analysis scope information.

2.1.1 Test Option Packet Soruces

LIBRARY TOPs - Newton maintains 2 distinct libraries of TOPs that are available to the user. A system library of general purpose TOPs is supplied with Newton; and Newton provides the facilities for creation and maintenance of user libraries of TOPs.

USER TOPs - individual Newton users may create and edit TOPs via a set of Newton command language primitives and an editor. These user TOPs as well as the system TOPs become a part of the TOOLPACK data base, and remain throughout the life of the database. Users must assign unique names to each of their own TOPs.

3.0 Newton Capabilities

Below are outlined the test options available to the Newton user. Newton test options are divided into two catagories:

TOP analysis commands - these are Newton analysis features that are controlled via Program Analysis Commands (PACs) within the TOP.

Embedded Analysis Commands - these are Newton program assertions and directives that are embedded within the user FORTRAN 77 source text in the form of special comments.

3.1 TOP PAC's

A TOP PAC has two parts, the analysis command itself and Range Specifiers.

Program Analysis Commands (PAC) are single line directives that appear within the Test Option Packet to control program analysis options. The presence or absence of a program analysis command controls the status of that analysis capability. The necessity of certain capabilities being active in some form at all times causes Newton to assume a default status for certain capabilities. Command scope may be limited by use of a named program range specifier. The Program Analysis Commands are order independent, except that range specifiers must be defined before being used.

3.1.1 Command Syntax

Program Analysis Commands have the following syntax:

COMMAND NAME {,option,option,...,option}{;program unit range}
program unit range :: = program unit range name | program unit | program unit dewey decimal range, dewey decimal range,...,dewey decimal range |
program unit range, program unit range
array range :: = array|array(ℓbound:ubound, ℓbound:ubound,...,ℓbound:ubound)|
array range, array range

Each command is assigned a unique command name, some commands require control options for specifying use. Program unit range specifications may reference other program unit range specifications by name (see Section 3.3).

3.1.2 Source Text Statement Numbering

Newton allows the user to choose from three alternative methods for numbering executable FORTRAN 77 source text statements. Each method allows the user to specify an optional increment for statement numbering. Only one numbering method may be used throughout the user FORTRAN 77 source text. Statement numbering is used in reporting analysis results.

3.1.2.1 Absolute Sequential Order

Statement numbering via an ascending integer statement number throughout the user's FORTRAN 77 source text (Figure 3.1) by specifying the PAC:

ABSOLUTE NUMBERING {,increment}

3.1.2.2 Routine Relative Sequential Order

Statement numbering via a six character identifier (local routine name, blank filled) and an ascending integer relative to routine location (Figure 3.2) by specifying the PAC:

ROUTINE NUMBERING {,increment}

3.1.2.3 Dewey Decimal

Statement numbering via a six character identifier (local routine name, blank filled) followed by the lowest hierarchical level of neighboring dewey decimal numbers found as labels within the assertion language for that routine (see below), and then by an ascending integer relative to routine location (Figure 3.3) by specifying the PAC:

DEWEY DECIMAL NUMBERING {,increment}

3.1.3 Program Analysis Commands

3.1.3.1 Sequence and Counting Options

Newton provides the user with various options for monitoring program execution sequence and statement execution frequency. These capabilities are provided by inserting instruments in the source code prior to each

FIGURE 3.1

sample report resulting from the use of the program analysis command:

ABSOLUTE NUMBERING,25

```
                    ---------------------------------------------------

                     PROGRAM LISTING                           STATEMENT NUMBER


      PROGRAM TEST
C
C --  CALCULATE SIGMA FOR 2 ARRAYS OF 10 VARIABLES EACH
C
      COMMON /STORAG/ A(10),B(10)
      DATA A/12.9,34.9,2.8,65.9,3.89,12.12,4.9,6.9,12.0,9.0/
      DATA B/99.0,56.9,89.0,120.0,32.0,45.0,56.1,45.0,79.2,56.9/
      CALL SIGMA(A,ASIGMA)                                          0
      CALL SIGMA(B,BSIGMA)                                          25
      WRITE(6,10) A,ASIGMA,B,BSIGMA                                 50
 10   FORMAT(1X,10F7.2,4X,1F8.4)
      END                                                           75


      SUBROUTINE SIGMA(ARRAY,ZSIGMA)
      DIMENSION ARRAY(10)
      CALL MEAN(ARRAY,ZMEAN)                                        100
      SUM = 0.0                                                     125
      DO 75 I=1,10                                                  150
      SUM = SUM + ((ARRAY(I)-ZMEAN)*(ARRAY(I)-ZMEAN))               175
 75   CONTINUE                                                      200
      ZSIGMA = SQRT( SUM/(10.0-1.0) )                               225
      RETURN                                                        250
      END                                                           275


      SUBROUTINE MEAN(VARS,RMN)
      DIMENSION VARS(10)
      TOTAL = 0.0                                                   300
      DO 98 I=1,10                                                  325
      TOTAL = TOTAL + VARS(I)                                       350
 98   CONTINUE                                                      375
      RMN = TOTAL/10.0                                              400
      RETURN                                                        425
      END                                                           450
```

FIGURE 3.2

sample report resulting from the use of the program analysis command:

ROUTINE NUMBERING,25

```
                -----------------------------------------------------

                        PROGRAM LISTING                        STATEMENT NUMBER


      PROGRAM TEST
C
C --  CALCULATE SIGMA FOR 2 ARRAYS OF 10 VARIABLES EACH
C
      COMMON /STORAG/ A(10),B(10)
      DATA A/12.9,34.9,2.8,65.9,3.89,12.12,4.9,6.9,12.0,9.0/
      DATA B/99.0,56.9,89.0,120.0,32.0,45.0,56.1,45.0,79.2,56.9/
      CALL SIGMA(A,ASIGMA)                                             TEST 0
      CALL SIGMA(B,BSIGMA)                                             TEST 25
      WRITE(6,10) A,ASIGMA,B,BSIGMA                                    TEST 50
 10   FORMAT(1X,10F7.2,4X,1F8.4)
      END                                                              TEST 75


      SUBROUTINE SIGMA(ARRAY,ZSIGMA)
      DIMENSION ARRAY(10)
      CALL MEAN(ARRAY,ZMEAN)                                           SIGMA 0
      SUM = 0.0                                                        SIGMA 25
      DO 75 I=1,10                                                     SIGMA 50
      SUM = SUM + ((ARRAY(I)-ZMEAN)*(ARRAY(I)-ZMEAN))                  SIGMA 75
 75   CONTINUE                                                         SIGMA 100
      ZSIGMA = SQRT( SUM/(10.0-1.0) )                                  SIGMA 125
      RETURN                                                           SIGMA 150
      END                                                              SIGMA 175


      SUBROUTINE MEAN(VARS,RMN)
      DIMENSION VARS(10)
      TOTAL = 0.0                                                      MEAN 0
      DO 98 I=1,10                                                     MEAN 25
      TOTAL = TOTAL + VARS(I)                                          MEAN 50
 98   CONTINUE                                                         MEAN 75
      RMN = TOTAL/10.0                                                 MEAN 100
      RETURN                                                           MEAN 125
      END                                                              MEAN 150
```

FIGURE 3.3

sample report resulting from the use of the program analysis command:

DEWEY DECIMAL NUMBERING

```
              --------------------------------------------------

                       PROGRAM LISTING                         STATEMENT NUMBER



        PROGRAM TEST
C
C --    CALCULATE SIGMA FOR 2 ARRAYS OF 10 VARIABLES EACH
C
        COMMON /STORAG/ A(10),B(10)
        DATA A/12.9,34.9,2.8,65.9,3.89,12.12,4.9,6.9,12.0,9.0/
        DATA B/99.0,56.9,89.0,120.0,32.0,45.0,56.1,45.0,79.2,56.9/
        CALL SIGMA(A,ASIGMA)                                       TEST 0-1
        CALL SIGMA(B,BSIGMA)                                       TEST 0-2
        WRITE(6,10) A,ASIGMA,B,BSIGMA                              TEST 0-3
 10     FORMAT(1X,10F7.2,4X,1F8.4)
        END                                                        TEST 0-4


        SUBROUTINE SIGMA(ARRAY,ZSIGMA)
        DIMENSION ARRAY(10)
C       ASSERT 1.1
        CALL MEAN(ARRAY,ZMEAN)                                     SIGMA 1.1-1
C       ASSERT 1.2
        SUM = 0.0                                                  SIGMA 1.2-1
        DO 75 I=1,10                                               SIGMA 1.2-2
C       ASSERT 1.2.1
        SUM = SUM + ((ARRAY(I)-ZMEAN)*(ARRAY(I)-ZMEAN))            SIGMA 1.2.1-1
C       END ASSERT 1.2.1
 75     CONTINUE                                                   SIGMA 1.2-3
C       END ASSERT 1.2
        ZSIGMA = SQRT( SUM/(10.0-1.0) )                            SIGMA 1.1-2
        RETURN                                                     SIGMA 1.1-3
C       END ASSERT 1.1
        END                                                        SIGMA 0-1


        SUBROUTINE MEAN(VARS,RMN)
        DIMENSION VARS(10)
C       ASSERT 1.1
        TOTAL = 0.0                                                MEAN 1.1-1
        DO 98 I=1,10                                               MEAN 1.1-2
C       ASSERT 1.1.1
        TOTAL = TOTAL + VARS(I)                                    MEAN 1.1.1-1
C       END ASSERT 1.1.1
 98     CONTINUE                                                   MEAN 1.1-3
        RMN = TOTAL/10.0                                           MEAN 1.1-4
        RETURN                                                     MEAN 1.1-5
C       END ASSERT 1.1
        END                                                        MEAN 0-1
```

event of the type being monitored. These instruments, when executed create sequence monitoring packets, each consisting of the calling program unit location (program unit statement number and execution count) and the type of the event.

When event sequencing reporting is specified by the user, these sequence monitoring packets are stored on circular lists, with the most recent events being placed at the head of the list and the oldest entry being automatically deleted. The user may optionally specify the lengths of the various lists. When event count reporting is specified by the user, the packets are used only to create and maintain the requested counters and are not stored.

3.1.3.1.1 Statement Execution Sequence

Newton allows the user to capture and store the sequential order of statement execution as a list of source text statement numbers (Figure 3.4) by specifying the PAC:

EXECUTION SEQUENCE {,limit} {;<u>range</u>}

Where limit is an integer upper bound on the number of statement execution entries retained in the circular list. Each time a statement within the specified <u>range</u> is executed its statement number is added to the list. Statements that fall outside the <u>range</u> are indicated by a special marker reserved for that purpose in the execution sequence list.

3.1.3.1.2 Statement Execution Frequency

Newton allows the user to specify that an integer count of the number of times each statement was executed (Figure 3.5) be reported by specifying the PAC:

EXECUTION COUNT {;<u>range</u>}

Each time a statement within the <u>range</u> is executed its integer count is incremented by one.

3.1.3.1.3 Statement Type Frequency

Newton allows the user to store an integer execution frequency count for types of FORTRAN 77 statements, arithmetic operations, and subprogram calls (Figure 3.6) by specifying the PAC:

STATEMENT TYPE COUNT, <u>Keyword</u>, <u>Keyword</u>,...,<u>Keyword</u>,{;<u>range</u>}

FIGURE 3.4

sample report resulting from the use of the program analysis command:

```
EXECUTION SEQUENCE,125;TEST,SIGMA
```

the following report assumes the program and statement numbering in FIGURE 3.2

```
               -------------------------------------------------

                    EXECUTION SEQUENCE BY STATEMENT NUMBER

               LAST STATEMENT EXECUTED IS THE LAST ELEMENT IN LIST

               ******* DENOTES STATEMENTS EXECUTED OUTSIDE OF RANGE


          TEST 0                          SIGMA 75                        SIGMA 100
          SIGMA 0                         SIGMA 100                       SIGMA 75
          *******                         SIGMA 75                        SIGMA 100
          SIGMA 25                        SIGMA 100                       SIGMA 75
          SIGMA 50                        SIGMA 75                        SIGMA 100
          SIGMA 75                        SIGMA 100                       SIGMA 75
          SIGMA 100                       SIGMA 125                       SIGMA 100
          SIGMA 75                        SIGMA 150                       SIGMA 75
          SIGMA 100                       TEST 25                         SIGMA 100
          SIGMA 75                        SIGMA 0                         SIGMA 75
          SIGMA 100                       *******                         SIGMA 100
          SIGMA 75                        SIGMA 25                        SIGMA 75
          SIGMA 100                       SIGMA 50                        SIGMA 100
          SIGMA 75                        SIGMA 75                        SIGMA 75
          SIGMA 100                       SIGMA 100                       SIGMA 100
          SIGMA 75                        SIGMA 75                        SIGMA 125
          SIGMA 100                       SIGMA 100                       SIGMA 150
          SIGMA 75                        SIGMA 75                        TEST 50
          SIGMA 100                                                       TEST 75
```

FIGURE 3.5

sample report resulting from the use of the program analysis command:

EXECUTION COUNT

```
EXECUTION COUNT                              PROGRAM LISTING


                                      PROGRAM TEST
                         C
                         C --         CALCULATE SIGMA FOR 2 ARRAYS OF 10 VARIABLES EACH
                         C
                                      COMMON /STORAG/ A(10),B(10)
                                      DATA A/12.9,34.9,2.8,65.9,3.89,12.12,4.9,6.9,12.0,9.0/
                                      DATA B/99.0,56.9,89.0,120.0,32.0,45.0,56.1,45.0,79.2,56.9/
      1                               CALL SIGMA(A,ASIGMA)
      1                               CALL SIGMA(B,BSIGMA)
      1                               WRITE(6,10) A,ASIGMA,B,BSIGMA
                         10           FORMAT(1X,10F7.2,4X,2F8.4)
      1                               END


                                      SUBROUTINE SIGMA(ARRAY,ZSIGMA)
                                      DIMENSION ARRAY(10)
      2                               CALL MEAN(ARRAY,ZMEAN)
      2                               SUM = 0.0
      2                               DO 75 I=1,10
     20                               SUM = SUM + ((ARRAY(I)-ZMEAN)*(ARRAY(I)-ZMEAN))
     20                  75           CONTINUE
      2                               ZSIGMA = SQRT( SUM/(10.0-1.0) )
      2                               RETURN
                                      END


                                      SUBROUTINE MEAN(VARS,RMN)
                                      DIMENSION VARS(10)
      2                               TOTAL = 0.0
      2                               DO 98 I=1,10
     20                               TOTAL = TOTAL + VARS(I)
     20                  98           CONTINUE
      2                               RMN = TOTAL/10.0
      2                               RETURN
                                      END
```

FIGURE 3.6

sample report resulting from the use of the program analysis command:

```
STATEMENT TYPE COUNT,ASSIGNMENT,CONTROL;MEAN
```

```
        ------------------------------------------------

                 STATEMENT TYPE FREQUENCY COUNT


        STATEMENT TYPE                       FREQUENCY COUNT


        ASSIGNMENT                                 26

        CONTROL                                    22
```

Where Keyword is one of the following identifiers: ASSIGNMENT, CONTROL, I/O, PROGRAM UNITS, OPERATORS. These Keywords are used to specify for which type of FORTRAN 77 statements Newton should store an execution frequency count. Each time a type of FORTRAN 77 statement specified by a Keyword is executed within the range, its count is incremented by one.

3.1.3.1.4 Program Unit Calling Sequence

Newton allows the user to maintain a subroutine calling sequence list for program execution (Figure 3.7) by specifying the PAC:

```
PROGRAM UNIT SEQUENCE {,limit} {;range}
```

Where limit is an integer upper bound on the number of calling sequence entries retained in the circular list. Each time a program unit is called within the range specified, the program unit call is added to the program unit calling sequence list. Program unit calls that fall outside the range are indicated by a special marker reserved for that purpose in the program unit calling sequence list.

3.1.3.1.5 Program Unit Call Frequency

Newton allows the user to maintain count of the number of calls on each program unit (Figure 3.8) by specifying the PAC:

```
PROGRAM UNIT COUNT {;range}
```

Each time a program unit is called within the range its integer count is incremented by one.

3.1.3.1.6 Transfer of Control Sequence

Newton allows the user to monitor the transfer of control sequence within a range by maintaining a list of statement pairs consisting of the originating statement number and the destination of transfer of control statement number (Figure 3.9). This is done by specifying the PAC:

```
TRANSFER SEQUENCE {,limit}{;range}
```

where limit is an integer upper bound on the number of pairs of statement number entries retained in the circular list. Statement numbers that fall outside the range are indicated by a special marker reserved for that purpose in the transfer of control sequence buffer.

FIGURE 3.7

sample report resulting from the use of the program analysis command:
PROGRAM UNIT SEQUENCE

the following report assumes the program and statement numbering in FIGURE 3.2

```
                ------------------------------------------------

                        PROGRAM UNIT CALLING SEQUENCE

           LAST PROGRAM UNIT CALLED IS THE LAST ELEMENT IN LIST

           ******* DENOTES PROGRAM UNITS CALLED OUTSIDE OF RANGE


 CALLING STATEMENT NUMBER / EXECUTION COUNT           PROGRAM UNIT CALLED


                TEST 0 / 1                                   SIGMA

                SIGMA 0 / 1                                  MEAN

                TEST 25 / 1                                  SIGMA

                SIGMA 0 / 2                                  MEAN
```

FIGURE 3.8

sample report resulting from the use of the program analysis command:

PROGRAM UNIT COUNT

the following report assumes the program and statement numbering in FIGURE 3.2

```
--------------------------------------------------

              PROGRAM UNIT CALLING FREQUENCY

   ******* DENOTES PROGRAM UNITS CALLED OUTSIDE OF RANGE


SUBROUTINE NAME                    CALL COUNT


   SIGMA                               2

   MEAN                                2
```

FIGURE 3.9

sample report resulting from the use of the program analysis command:

TRANSFER SEQUENCE,23;SIGMA,1.2

the following report assumes the program and statement numbering in FIGURE 3.3

```
--------------------------------------------------

              TRANSFER OF CONTROL SEQUENCE

     LAST TRANSFER OF CONTROL IS THE LAST ELEMENT IN LIST

     ******* DENOTES TRANSFER OF CONTROL OUTSIDE OF RANGE


******* / SIGMA 1.2-1                    SIGMA 1.2-3 / *******
SIGMA 1.2-1 / SIGMA 1.2-2
SIGMA 1.2-2 / SIGMA 1.2.1-1
SIGMA 1.2.1-1 / SIGMA 1.2-3
SIGMA 1.2-3 / SIGMA 1.2.1-1
SIGMA 1.2.1-1 / SIGMA 1.2-3
SIGMA 1.2-3 / SIGMA 1.2.1-1
SIGMA 1.2.1-1 / SIGMA 1.2-3
SIGMA 1.2-3 / SIGMA 1.2.1-1
SIGMA 1.2.1-1 / SIGMA 1.2-3
SIGMA 1.2-3 / SIGMA 1.2.1-1
SIGMA 1.2.1-1 / SIGMA 1.2-3
SIGMA 1.2-3 / SIGMA 1.2.1-1
SIGMA 1.2.1-1 / SIGMA 1.2-3
SIGMA 1.2-3 / SIGMA 1.2.1-1
SIGMA 1.2.1-1 / SIGMA 1.2-3
SIGMA 1.2-3 / SIGMA 1.2.1-1
SIGMA 1.2.1-1 / SIGMA 1.2-3
SIGMA 1.2-3 / SIGMA 1.2.1-1
SIGMA 1.2.1-1 / SIGMA 1.2-3
SIGMA 1.2-3 / SIGMA 1.2.1-1
SIGMA 1.2.1-1 / SIGMA 1.2-3
```

3.1.3.1.7 Transfer of Control Frequency

Newton allows the user to maintain an integer frequency count for each entry in the transfer sequence list (Figure 3.10) by specifying the PAC:

TRANSFER COUNT { ;range}

3.1.3.2 Program Unit Parameter Monitoring

Newton provides the user with three options for monitoring program unit parameters during program unit calls. In each case a parameter monitoring list is built. Each entry in a parameter monitoring list consists of the program unit location of the call (program unit statement number and statement execution count) and the program unit parameters passed. These parameter monitoring list entries are stored as a circular list, with current events being placed at the head of the list. The user may specify an optional limit for the length of the list. Program unit calls that fall outside the range are indicated by a special marker reserved for that purpose in the program unit parameter monitoring sequence list.

3.1.3.2.1 Parameter Monitoring by Value

Newton allows the user to maintain a list each of whose entries consists of the values of all parameters passed to and returned from a program unit for each program unit call encountered (Figure 3.11) by specifying the PAC:

PARAMETER VALUES {,NO ARRAY} {,limit} {;range}

NO ARRAY is a keyword that allows the user to conserve internal storage by not requiring storage of values for arrays. Limit is an optional integer upper bound on the number of parameter monitoring entries retained in the circular list. Undefined values are denoted by a special marker reserved for that purpose in the parameter value sequence

3.1.3.2.2 Parameter Monitoring by Names

Newton allows the user to maintain a list of actual parameter names used at each program unit call within the range (Figure 3.12) by specifying the PAC:

PARAMETER NAMES {,limit}{;range}

FIGURE 3.10

sample report resulting from the use of the program analysis command:

TRANSFER COUNT;SIGMA,1.2

the following report assumes the program and statement numbering in FIGURE 3.3

```
--------------------------------------------------

            TRANSFER OF CONTROL FREQUENCY

   ******* DENOTES TRANSFER OF CONTROL OUTSIDE OF RANGE


TRANSFER PAIR                              COUNT


******* / SIGMA 1.2-1                        1

SIGMA 1.2-1 / SIGMA 1.2-2                    1

SIGMA 1.2-2 / SIGMA 1.2.1-1                  1

SIGMA 1.2.1-1 / SIGMA 1.2-3                 10

SIGMA 1.2-3 / SIGMA 1.2.1-1                  9

SIGMA 1.2-3 / *******                        1
```

FIGURE 3.11

sample report resulting from the use of the program analysis command:

PARAMETER VALUES,NO ARRAY,100

the following report assumes the program and statement numbering in FIGURE 3.2

```
                    --------------------------------------------------

                          PARAMETER MONITORING BY VALUE

                LAST PROGRAM UNIT CALL IS THE LAST ELEMENT IN LIST

                ******* DENOTES PROGRAM UNIT CALL OUTSIDE OF RANGE

                XXXXXXX DENOTES VALUE UNDEFINED AT CALLING LOCATION


STATEMENT NUMBER / EXECUTION COUNT    PARAMETER VALUE (PASSED) / (RETURNED)


              TEST 0 / 1                      (A,XXXXXXX) / (A,25.88)

              SIGMA 0 / 1                     (ARRAY,XXXXXXX) / (ARRAY,16.53)

              TEST 25 / 1                     (B,XXXXXXX) / (B,76.31)

              SIGMA 0 / 2                     (ARRAY,16.53) / (ARRAY,67.90)
```

FIGURE 3.12

sample report resulting from the use of the program analysis command:

PARAMETER NAMES,100

the following report assumes the program and statement numbering in FIGURE 3.2

```
---------------------------------------------------

              PARAMETER MONITORING BY NAME

      LAST PROGRAM UNIT CALL IS THE LAST ELEMENT IN LIST

      ******* DENOTES PROGRAM UNIT CALL OUTSIDE OF RANGE


STATEMENT NUMBER / EXECUTION COUNT               PARAMETER NAME


        TEST 0 / 1                               (A,ASIGMA)

        SIGMA 0 / 1                              (ARRAY,ZMEAN)

        TEST 25 / 1                              (B,BSIGMA)

        SIGMA 0 / 2                              (ARRAY,ZMEAN)
```

where limit is an integer upper bound on the number of parameter name entries retained in the circular list. Newton uses a special marker to indicate which parameters are expressions.

3.1.3.2.3 Parameter Bindings

Newton allows the user to maintain a list of the value bindings of formal parameters (Figure 3.13) by specifying the PAC:

PARAMETER BINDINGS {,limit}{;range}

where limit is an integer upper bound on the number of parameter binding entries retained in the circular list. The information reported at the program unit calling location is the binding of the formal parameters to names. Newton may have to trace back through several program unit calls in order to determine what the bindings in effect at this call are.

3.1.3.3 Value Evolution Monitoring

Newton provides the user with the following options to monitor the evolution of values in program variables by constructing various value evolution monitoring lists. Each entry in a value evolution monitoring list consists of the program unit location where a new value was assigned (program unit statement number and execution count) and the assigned value or array index. These value evolution entries are stored as a circular list, with current events being placed at the head of the list. The user may specify an optional limit for the length of the list.

Value evolution is divided into two categories: arrays and simple variables.

3.1.3.3.1 Arrays

3.1.3.3.1.1 Value Evolution by Index

Newton allows the user to monitor the value evolution of program arrays by storing a sequential history of evolution for each array element. A new value evolution entry is added to the appropriate sequence each time an assignment to a particular array element takes place within the range (Figure 3.14) by specification of the PACs:

ARRAY VALUE, array range {,limit}{;range}

FIGURE 3.13

sample report resulting from the use of the program analysis command:

PARAMETER BINDINGS,100

the following report assumes the program and statement numbering in FIGURE 3.2

```
          --------------------------------------------------

                    PARAMETER MONITORING BY BINDING

          LAST PROGRAM UNIT CALL IS THE LAST ELEMENT IN LIST
          ******* DENOTES PROGRAM UNIT CALL OUTSIDE OF RANGE


    STATEMENT NUMBER / EXECUTION COUNT               PARAMETER BINDING


            TEST 0 / 1                               (A,ASIGMA)
            SIGMA 0 / 1                              (A,ASIGMA)
            TEST 25 / 1                              (B,BSIGMA)
            SIGMA 0 / 2                              (B,BSIGMA)
```

FIGURE 3.14

sample report resulting from the use of the program analysis command:

ARRAY VALUE,DISPLAY,IARRAY,100

on the following sample program text:

```
      PROGRAM DISPLAY
      DIMENSION IARRAY(3)
             .
             .
             .
      DO 10 I=5,8                                          DISPLAY 100
      DO 10 J=1,3                                          DISPLAY 110
      IARRAY(J) = I*J                                      DISPLAY 120
 10   CONTINUE                                             DISPLAY 130
             .
             .
             .
      END
```

```
           ------------------------------------------------

                        VALUE EVOLUTION BY SEQUENCE

            LAST ARRAY ELEMENT SET IS THE LAST ELEMENT IN LIST

            ******* DENOTES ARRAY ELEMENTS SET OUTSIDE OF RANGE


      STATEMENT NUMBER / EXECUTION COUNT                         VALUE

  ARRAY ELEMENT IARRAY(1)
                DISPLAY 120 / 1                                    5
                DISPLAY 120 / 4                                    6
                DISPLAY 120 / 7                                    7
                DISPLAY 120 / 10                                   8

  ARRAY ELEMENT IARRAY(2)
                DISPLAY 120 / 2                                   10
                DISPLAY 120 / 5                                   12
                DISPLAY 120 / 8                                   14
                DISPLAY 120 / 11                                  16

  ARRAY ELEMENT IARRAY(3)
                DISPLAY 120 / 3                                   15
                DISPLAY 120 / 6                                   18
                DISPLAY 120 / 9                                   21
                DISPLAY 120 / 12                                  24
```

or:

ARRAY VALUE,ALL{,limit}{;range}

where array range allows the specification of range within program arrays. ALL is a keyword that allows reference to all arrays not explicitly referenced in previous array value PACs. Limit is an integer upper bound on the number of array value entries retained in the circular list.

3.1.3.3.1.2 Value Evolution by Sequence

Newton also allows the user to monitor the value evolution of program arrays by storing a sequential history of evolution for each program array. In this case a list entry consists of a pair, the first element of which is the value assigned and the second element of which is the array index to which it was assigned. A new value evolution entry is added to an array's sequence each time an assignment to an array element takes place within the range (Figure 3.15). This monitoring is effected by specification of the PACs:

ARRAY INDEX, array range {,limit}{;range}

or:

ARRAY INDEX,ALL{,limit}{;range}

where array range allows the specification of range within program arrays. ALL is a keyword that allows reference to all arrays not explicitly referenced in previous array index PACs. Limit is an integer upper bound on the number of array index entries retained in the circular list.

3.1.3.3.2 Value Evolution of Simple Variables

Newton allows the user to monitor the value evolution of simple program variables by storing a sequential history of evolution for each program variable. A new value evolution list entry is added to a program variable sequence each time an assignment is made to the specified variable within the range (Figure 3.16) by specification of the PACs:

VARIABLE,program unit,variable,variable,...,variable{,limit}{;range}

or:

VARIABLE,ALL{,limit}{;range}

where ALL is a keyword that causes the monitoring of all variables not explicitly named in previous variable PACs. Limit is an integer upper bound on the number of variable entries retained in the circular list.

FIGURE 3.15

sample report resulting from the use of the program analysis command:

ARRAY INDEX,DISPLAY,IARRAY,100

on the following sample program text:

```
      PROGRAM DISPLAY
      DIMENSION IARRAY(3)
              .
              .
              .
      DO 10 I=5,8                                                DISPLAY 100
      DO 10 J=1,3                                                DISPLAY 110
      IARRAY(J) = I*J                                            DISPLAY 120
10    CONTINUE                                                   DISPLAY 130
              .
              .
              .
      END
```

```
                  VALUE EVOLUTION BY INDEX

        LAST ARRAY ELEMENT SET IS THE LAST ELEMENT IN LIST

        ******* DENOTES ARRAY ELEMENTS SET OUTSIDE OF RANGE


STATEMENT NUMBER / EXECUTION COUNT / ARRAY ELEMENT              VALUE

        DISPLAY 120 / 1 / IARRAY(1)                               5
        DISPLAY 120 / 2 / IARRAY(2)                              10
        DISPLAY 120 / 3 / IARRAY(3)                              15
        DISPLAY 120 / 4 / IARRAY(1)                               6
        DISPLAY 120 / 5 / IARRAY(2)                              12
        DISPLAY 120 / 6 / IARRAY(3)                              18
        DISPLAY 120 / 7 / IARRAY(1)                               7
        DISPLAY 120 / 8 / IARRAY(2)                              14
        DISPLAY 120 / 9 / IARRAY(3)                              21
        DISPLAY 120 / 10 / IARRAY(1)                              8
        DISPLAY 120 / 11 / IARRAY(2)                             16
        DISPLAY 120 / 12 / IARRAY(3)                             24
```

FIGURE 3.16

sample report resulting from the use of the program analysis command:

VARIABLE,MEAN,TOTAL,100

the following report assumes the program and statement numbering in FIGURE 3.2

```
          ---------------------------------------------------

                            VALUE EVOLUTION

          LAST VARIABLE SET IS THE LAST ELEMENT IN LIST

          ******* DENOTES VARIABLE SET OUTSIDE OF RANGE


STATEMENT NUMBER / EXECUTION COUNT / VARIABLE NAME              VALUE

          MEAN 0 / 1 / TOTAL                                     0.0
          MEAN 50 / 1 / TOTAL                                    12.9
          MEAN 50 / 2 / TOTAL                                    47.8
          MEAN 50 / 3 / TOTAL                                    50.6
          MEAN 50 / 4 / TOTAL                                    116.5
          MEAN 50 / 5 / TOTAL                                    120.39
          MEAN 50 / 6 / TOTAL                                    132.51
          MEAN 50 / 7 / TOTAL                                    137.41
          MEAN 50 / 8 / TOTAL                                    144.31
          MEAN 50 / 9 / TOTAL                                    156.31
          MEAN 50 / 10 / TOTAL                                   165.31
          MEAN 0 / 2 / TOTAL                                     0.0
          MEAN 50 / 11 / TOTAL                                   99.0
          MEAN 50 / 12 / TOTAL                                   155.9
          MEAN 50 / 13 / TOTAL                                   244.9
          MEAN 50 / 14 / TOTAL                                   364.9
          MEAN 50 / 15 / TOTAL                                   396.9
          MEAN 50 / 16 / TOTAL                                   441.9
          MEAN 50 / 17 / TOTAL                                   498.0
          MEAN 50 / 18 / TOTAL                                   543.0
          MEAN 50 / 19 / TOTAL                                   622.2
          MEAN 50 / 20 / TOTAL                                   679.1
```

3.1.3.4 Variable Extremal Value Monitoring

Newton provides the user with the following options to monitor the extremal values of program variables. Each entry in an extremal value list consists of the program unit location where the assignment of an extremal value occurred (program unit statement number and execution count) and the assigned value. These extremal value entries are stored as a circular list, with current events being placed at the head of the list. The user may specify an optional limit for the length of the list.

3.1.3.4.1 Minimum and Maximum Values

Newton allows the user to maintain a list of the minimum and maximum values assigned to program variables (Figure 3.17) by specification of the PACs:

```
MIN/MAX{,array range}{program unit,variable,variable,...,variable,
  {,limit}{;range}
```

or:

```
MIN/MAX,ALL{,limit}{;range}
```

where array range allows the specification of range within program arrays. ALL is a keyword that allows reference to all program variables not explicitly referenced in previous minimum and maximum value PACs. Limit is an integer upper bound on the number of minimum and maximum value entries retained in the circular list.

3.1.3.4.2 First and Last Values

Newton allows the user to store the first and last values assigned to program variables (Figure 3.18) by specification of the PACs:

```
FIRST/LAST{,array range}{,program unit,variable,variable,...,
  variable}{,limit}{;range}
```

or:

```
FIRST/LAST,ALL{,limit}{;range}
```

where array range allows the specification of range within program arrays. ALL is a keyword that allows reference to all program variables not explicitly referenced in previous first and last value PACs. Limit is an integer upper bound on the number of first and last value entries retained in the circular list.

FIGURE 3.17

sample report resulting from the use of the program analysis commands:

```
MIN/MAX,SIGMA,ZMEAN,SUM,ZSIGMA,100
MIN/MAX,MEAN,TOTAL,RMN,100
```

the following report assumes the program and statement numbering in FIGURE 3.2

```
          --------------------------------------------------

                        MINIMUM AND MAXIMUM VALUE

          ******* DENOTES VARIABLE SET OUTSIDE OF RANGE


NAME / STATEMENT NUMBER / EXECUTION COUNT                        VALUE

        ZMEAN / SIGMA 0 / 1                             MINIMUM: 16.53
        ZMEAN / SIGMA 0 / 2                             MAXIMUM: 67.90

        SUM / SIGMA 25 / 1                              MINIMUM:     0.0
        SUM / SIGMA 75 / 20                             MAXIMUM: 52411.9

        ZSIGMA / SIGMA 125 / 1                          MINIMUM: 25.88
        ZSIGMA / SIGMA 125 / 2                          MAXIMUM: 76.31

        TOTAL / MEAN 0 / 1                              MINIMUM:   0.0
        TOTAL / MEAN 50 / 20                            MAXIMUM: 679.09

        RMN / MEAN 100 / 1                              MINIMUM: 16.53
        RMN / MEAN 100 / 2                              MAXIMUM: 67.90
```

FIGURE 3.18

sample report resulting from the use of the program analysis commands:

```
FIRST/LAST,SIGMA,ZMEAN,SUM,ZSIGMA,100
FIRST/LAST,MEAN,TOTAL,RMN,100
```

the following report assumes the program and statement numbering in FIGURE 3.2

```
                ------------------------------------------------

                              FIRST AND LAST VALUE

                *******  DENOTES VARIABLE SET OUTSIDE OF RANGE


        NAME / STATEMENT NUMBER / EXECUTION COUNT                        VALUE

                ZMEAN / SIGMA 0 / 1                              FIRST: 16.53
                ZMEAN / SIGMA 0 / 2                               LAST: 67.90

                SUM / SIGMA 25 / 1                               FIRST:     0.0
                SUM / SIGMA 75 / 20                               LAST: 52411.9

                ZSIGMA / SIGMA 125 / 1                           FIRST: 25.88
                ZSIGMA / SIGMA 125 / 2                            LAST: 76.31

                TOTAL / MEAN 0 / 1                               FIRST:   0.0
                TOTAL / MEAN 50 / 20                              LAST: 679.09

                RMN / MEAN 100 / 1                               FIRST: 16.53
                RMN / MEAN 100 / 2                                LAST: 67.90
```

3.1.3.5 Program Error Conditions

Newton allows the user to check for certain program execution errors. Detection of any of the following error conditions results in error information being stored in Newton's diagnostic data base and program control being transferred to the Newton Contingency Handler.

3.1.3.5.1 Subscript Range Checking

Newton allows the user to effect a check of each index used in an array subscripting operation: to be sure it is within the range declared for that array. This is accomplished by specification of PACs:

SUBSCRIPT RANGE,array range{;range}

or:

SUBSCRIPT RANGE,ALL{;range}

where array range allows the specification of range within program arrays. ALL is a keyword that effects monitoring of all arrays not explicitly named in previous subscript range PACs.

3.1.3.5.2 Division by 0

Newton allows the user to check for a 0 divisor before each division operation by specifying the PAC:

DIVISION{;range}

3.1.3.5.3 Underflow/Overflow Errors

Newton allows the user to check for underflow and overflow in arithmetic operation by specifying the PAC:

OVERFLOW/UNDERFLOW{;range}

3.1.3.5.4 Square Root of Negative Numbers

Newton allows the user to check for a negative argument before each square root operation by specifying the PAC:

NEGATIVE SQRT{;range}

3.1.3.5.5 Common and Natural Log of Nonpositive Argument

Newton allows the user to check for a negative or zero argument before each common or natural log operation by specifying the PAC:

NONPOSITIVE LOG{;range}

3.2 Embedded Analysis Commands

3.2.1 Program Assertions and Value Keeping Statements

Newton gives the user the capability to use program assertions and value keeping. Assertions and value keeps are special program statements that are used to capture the intent of the program. Newton requires that program assertions and value keeping statements be embedded in the form of special comments within the user source text.

3.2.1.1 Newton Assertion Language Definition

3.2.1.1.1 Notation

The grammar used to describe the assertion and statistics gathering languages is a variant of BNF, described below.

i) Nonterminals are underlined, e.g., <u>assert statement</u>

ii) Terminals composed of Latin letters are printed in upper case and enclosed in quotes, e.g., "ASSERT"

iii) Items which are optional are enclosed in braces, e.g., {"GLOBAL"}

iv) Items suffixed with an asterisk (*) may appear zero or more times

v) Items suffixed with a plus sign (+) may appear one or more times

vi) Multiple productions corresponding to a single non-terminal are listed on successive lines. The non-terminal and the ::= sign only appear on the first production.

3.2.1.1.2 Assert Statement Grammar

```
ASSERT statement ::="C "{special label}"ASSERT"{"GLOBAL"}ext-logical-exp
                         {control}
special label ::= integer "."{integer{"."integer}*}{"."}
ext-logical-exp ::= value{relop value}*
                    expression list{"NOT"} "IN" range{range}*
control ::= "ON""TRUE"program unit
              "ON""FALSE"program unit
value ::= {quantifier} relational expression
          "(" value ")"
expression list ::= arithmetic expression {arithmetic expression}*
relop ::= ".CAND."
          ".COR."
quantifier ::= "FOR""ALL" quantifier completion
               "EXISTS" quantifier completion
quantifier completion ::= integer variable range{,integer variable range}*
range ::= "(" constant {"TO"constant}")"
end ASSERT stmt ::= "C""END""ASSERT" special label
statistics value ::= {name} "$""("special label")"
```

3.2.1.1.3 Context Sensitive Rules

1. No two ASSERT statements may have the same special label.

2. Multiple ASSERT GLOBAL/END ASSERT statements are possible, and nesting is required.

3. ASSERT/END ASSERT with same label must be in sequential order with END ASSERT following ASSERT

4. The special label on assert statements must begin in columns 3-5.

3.2.1.1.4 Assertion Semantics

1. Any and all ASSERT statements may be labeled with a Dewey decimal number. Their instrumentation may be controlled by an external mechanism which references these numbers.

2. The ASSERT GLOBAL statement specifies a condition which much continuously hold over a range of the program. This range is demarcated by the ASSERT statement and the END ASSERT statement whose special labels match. If no such END ASSERT statement exists or if the ASSERT statement is unlabeled, the assertion applies to all program text following the ASSERT statement in the current static scope (at the sub-program level). The expressions listed must not reference any variables where scope is smaller than the range of the assertion. Note that if subscripted variables (or otherwise parameterized expressions) are used in the assertion, the entire expression is re-evaluated each time a check is required. Thus, if the expression was A(I) = 1, then the subscript I is evaluated anew at each check point.

3. "Threshold" control may be achieved in the following manner. The special value VIOLATE(special label) may be used within any comparison in an assertion. Its value is the number of times the referenced assertion has been violated. If no reference (special label) is provided, the number of violations of the current assertion is taken.

4. Special value statistics value may be used within any extended logical expression. statistics value allows the value of any FORTRAN 77 expression which is saved in a KEEP to be referenced in an assertion.

The optional name which precedes the label allows a particular value to be referenced out of several saved at the KEEP (there may have been a list of expressions to KEEP). The name supplied must be textually identical to one of the expressions listed in the KEEP.

5. Quantifiers on comparisons allow the formation of powerful assertions. The quantifier completions presented allow for looping constructs. When used with FOR ALL, the assertion must hold true as the integer variable assumes each of the values specified in the range. When multiple integer variables and ranges are specified, the assertion must hold for every combination of integer-variable and value. When used with EXISTS, the assertion must hold true for at least one integer-variable and value (or combination thereof, if several integer variable ranges are specified).

6. The IN range specification indicates that each value specified in the expression list must lie within one of the ranges provided. A range may consist of a single value. Ranges may only be specified for integer and real valued expressions.

3.2.1.1.5 Assertion Violations

When an assertion violation occurs, the user has the option to have control transferred to a user specified program unit (the program unit is named in the ASSERT statement), or if no routine is specified control is transferred to the Newton Contingency Handler. If control is transferred to a user program unit, the program unit must contain either a RETURN statement to resume program execution or a STOP statement to terminate program execution.

3.2.1.2.1 Value Keeping Statement

```
value keeping statement ::= "C"{special label}"KEEP" "GLOBAL"function
   "C"{special label}"KEEP"expression{qualifier}
end keep statement ::= "C""END""KEEP" special label
qualifier ::= "IF"comparison
label stmt ::= "C"special label
end label stmt ::= "C""END"special label
```

Context Sensitive Rules.

1. No two KEEP statements may be labeled with the same number.

2. Multiple KEEP GLOBAL / END KEEP pairs are possible, and nesting is not required.

3. label stmt / end label stmt must be in sequential order with end label stmt following label stmt

4. The special label on value keeping statements must begin in columns 3-5.

3.2.1.2.2 Value Keeping Statement Semantics

1. All KEEP statements may be labeled with a Dewey decimal number. As such they are individually named and their instrumentation may be controlled in a sophisticated manner by an external mechanism.

2. The KEEP GLOBAL statement specifies a list of functions which are to be called after every (applicable) statement within the textual scope defined by the KEEP GLOBAL statement and the END KEEP statement whose special labels match. If no matching END KEEP is found, such a statement is generated at the end of the current textual scope (at the subprogram level).

3. The functions which may be invoked at each (appropriate) statement are as follows:

   normal function — A general FORTRAN 77 function which will be called after the execution of each statement. This provision is in keeping with the overall criterion of providing a general syntax. Implementation restrictions, however, are almost certain. The function name may be followed by a list of parameters. The scope of the parameters must be consistent with the scope of the keep. statistics value is a legal parameter.

4. If GLOBAL is not specified, the KEEP statement refers only to the program state defined at the point of the KEEP.

5. The expressions in the expression list may be any computable expression (including FORTRAN normal functions) and is subject to the rules provided for functions in rule 3 above, statistics value is a legal part of an expression.

6. If a KEEP statement has a qualifier phrase, the information requested will be kept only if the condition is met. Evaluation of the condition is subject to the extensions and restrictions applied to normal functions in rule 3 above.

3.2.1.3 Sample Usages of the Assertion and Value Keeping Facility

1)
```
C       ASSERT A.EQ.B+C
```
A simple arithmetic relationship which must be true at the point of assertion placement.

2)
```
C       ASSERT  A.LE.5 .CAND. F(X).EQ.F(Z)
```
Two arithmetic relationships. The second relationship is checked (causing evaluation of the functions) if and only if $A \leq 5$.

3)
```
C       ASSERT VIOLATE.LT.4.CAND. F(X).EQ.0
```
F(X) will only be compared with zero if this assertion has been violated at most four times.

4)
```
C       ASSERT GLOBAL X.GT.0
```
X must remain positive from the assertion through the end of the current scope (either procedure, task, or program end).

5)
```
C   1.1 ASSERT GLOBALX.GT.0
     :
     :      X must remain positive throughout this region
     :
    END ASSERT 1.1
```

6)
```
    ASSERT X IN(1 TO 6)(12)
```
The condition $1 \leq X \leq 6$ or X-12 must be satisfied.

7)
```
C   1.1 KEEP X
```
The current value of X is retained for later use in an assertion.

8)
```
C       ASSERT X$(1.1).EQ.X
```
Asserts that the last value of X stored at KEEP 1.1 is equal to the current value of X.

9)
```
C       ASSERT $(1.1).EQ.X
```
Same as example 8. This syntax is valid if KEEP 1.1 only retained variable X.

10)
```
C       KEEP X IF F(X).LE.5
```
The value of X will be retained only if $F(X) \leq 5$.

11) C   ASSERT X.LT. 0. .COR. ERROR

This example illustrates how special processing may be performed on assertion violation. If X is not less than zero then (presumably) something has gone awry in the program. In order to gather as much information as possible, a user-supplied function is called which may, for example, print out a helpful message.

12) C   ASSERT FOR ALL I(1 TO 10), J(1 TO 5) A(I).LT.B(J)

This assertion is equivalent to the logical conjunction of the following assertions.

```
A(1).LT.B(1)
A(1).LT.B(2)
   .
   .
   .
A(1).LT.B(5)
A(2).LT.B(1)
A(2).LT.B(2)
   .
   .
   .
A(10).LT.B(5)
```

In other words, each element of A must be less than every element of B.

13) C   ASSERT FOR ALL I(1 TO 10) A(I).LE.A(I+1)

This asserts that the first 11 elements of A are sorted in ascending order.

14) C   ASSERT EXISTS N (4 TO 100) A**N.EQ.B**N+C**N

This assertion declares that there exists at least 1 value of N between 4 and 100 inclusive such that $A^N = B^N + C^N$, for values A,B, and C. (This assertion will fail, of course, if A,B, and C are integers and $(A)(B)(C) \neq 0$).

15) C   KEEP MAX(X,$(22.))

Assuming that function MAX returns the larger of its two arguments, this KEEP will retain the maximum value of X which occurs at this statement. (MAX must also be able to detect that $(22.) is undefined on the first call of the function. This ability is implementation dependent).

3.2.2 Program Breakpoint

Program breakpoints give the user the ability to interrupt program execution and examine the contents of Newton's diagnostic Database via the Newton Contingency Handler. The user also has the option to transfer control to a user specified program unit when a breakpoint is encountered. If control is transferred to a user program unit, the named program unit controls program execution.

Breakpoints may also be contingent upon user defined events (an event being the same as an assertion language ext-logical-exp). Breakpoint events are evaluated over the range of the entire local routine when the keyword GLOBAL is present.

Embedded Command: C BREAKPOINT {"GLOBAL"}{ext-logical-exp}{program unit}

3.2.2.1 Sample Usages of Program Breakpoint

```
1)   C       BREAKPOINT
```

Unconditionally halt program execution and transfer control to the Newton Contingency Handler whenever this command is encountered.

```
2)   C       BREAKPOINT   GLOBAL   X.GT.0   DEBUG
```

If at any time during execution of the current routine the value of X becomes greater than 0, program control is transferred to the user routine DEBUG

```
3)   C       BREAKPOINT FORALL I(1 TO 10), J(1 TO 5) A(I,J).LE.0
```

As execution passes the site of this statement, check all values of array A within the specified indices and if any value is less than or equal to 0 transfer control to the Newton Contingency Handler

3.2.3 Program Timing

Newton provides the user with program "STOPWATCHES" to monitor program execution. The user is given access to and control over variables in the Newton diagnostic Database which can be manipulated as though they were stopwatches. This is done by means of embedded command of the form:

```
     C     STOPWATCH, Keyword,Keyword,...,Keyword;name
```

where name is the user defined name of the program STOPWATCH. STOPWATCH names are separate from program variables and must be distinct from program variable names. Keyword is either NEW, RESET, START, ELAPSED, or STOP. RESET resets the "stopwatch" to zero. START has the effect of beginning the measurement of elapsed time by the named STOPWATCH by reading and storing the clock in the host system which measures the amount of time remaining out of the user's maximum time limit specification (i.e., the user's "countdown timer.") ELAPSED updates the elapsed time in the STOPWATCH by reading the countdown timer again, computing the difference, and storing it. STOP updates the elapsed time and stops timing for the STOPWATCH. New STOPWATCHES are declared when the user assigns a previously unused STOPWATCH name and the keyword NEW to the STOPWATCH command. New STOPWATCHes are set to zero when they are declared.

3.2.3.1 Sample Usages of Program Timing

```
1)   C      STOPWATCH, NEW, START; CLOCK
```
A new STOPWATCH variable CLOCK is declared and timing is begun.

```
2)   C      STOPWATCH, ELAPSED; CLOCK
```
The amount of measured elapsed time since the last start of STOPWATCH CLOCK is placed in CLOCK.

```
3)   C      STOPWATCH, STOP; CLOCK
```
Same as example 2 except the STOPWATCH CLOCK is now stopped.

```
4)   C      STOPWATCH, RESET, START; CLOCK
```
STOPWATCH CLOCK is reset to zero and timing is begun.

3.3 Limiting Program Analysis Scope

Newton gives the user the ability to limit the scope of program analysis. The user may use a program analysis command that references dewey decimal number labels within the assertion language to define and name a range of program source text. This range may be used on PAC's to limit the application of Newton program analysis. If no range name is present a PAC is active over the entire execution unit. The user may specify the PAC:

PROGRAM UNIT RANGE,Name{;program unit,program unit,...,
program unit}{;program unit, dewey decimal range,
dewey decimal range, ..., dewey decimal range}

Name is the user declared range.

A special PAC is available to limit Newton evaluation of embedded commands:

EMBEDDED COMMANDS;range

Global assertions are inactive unless assertions are active over the entire range.

4.0 Newton Contingency Handler

The Newton Contingency Handler (NCH) is available to users to examine and change certain program attributes during program execution. Users may enter the NCH via:

BREAKPOINT— execution of certain types of breakpoint commands within the user source text results in user entry to the NCH.

EXECUTION ERROR— Newton detection of a test option violation (Newton execution error) results in user entry to the NCH.

ASSERTION VIOLATION — certain types of assertion language violations result in user entry to the NCH.

Below are the options available to users while in the NCH:

4.1 Queries

Newton allows the user to browse through program attributes with the following options.

4.1.1 Sequence and Counting

The user is allowed to examine any information returned from the sequence and counting program analysis options up to the point of program execution interruption.

4.1.2 Source Text

The user is allowed to examine the program source text.

4.2 Information Saving During Breakpoints

The user has the capability to save program information during a breakpoint. A "saved" breakpoint consists of the value of every program variable in the symbol tables being written as a record in the Breakpoint Saved Table (BST). The maximum number of records in the BST is a TOOLPACK defined parameter. BST records are maintained as a circular list with new entries placed at the beginning of the list.

4.3 Value Changing

The user has the capability to change the value of any program variable.

If the NCH detects that the user has changed the value of any variable when the user NCH session is over the NCH forces a special system breakpoint save. This system breakpoint save is entered into the BST as a non-deletable record. If the BST fills with non-deletable records the NCH will not allow any further value changing.

4.4 Return Options

The user has the option to:

- return to user program execution
- return to TOOLPACK control

4.5 NCH Non-Interactive Users

NCH non-interactive users are treated as a special case and are routed depending on the entry mode to the NCH:

NCH entry via breakpoint — the system initiates a user breakpoint save and returns to user program execution.

NCH entry via Newton execution error — the system initiates a user breakpoint save and returns to TOOLPACK control mode.

NCH entry via assertion violation — the system initiates a user breakpoint save and an entry to a special assertion violation table and returns to user program execution.

Acknowledgments

The Newton assertion language definition is adapted from a paper written by Richard Taylor.

The University of Colorado Software Validation Group provided many helpful suggestions and comments.

William Mesch assisted in the preparation of a preliminary version of Newton.

REFERENCES

[BROW73] J. R. Brown, A. J. DeSalvio, D. E. Heine, and J. G. Purdy, "Automated Software Quality Assurance," in Program Test Methods, (W. C. Hetzel, ed.) Prentice-Hall, Englewood Cliffs, N.J., 1973, pp. 181-203.

[FAIR75] R. E. Fairley, "An Experimental Program Testing Facility," Proc. First National Conf. on Software Eng., IEEE Cat. #75CH0992-8C pp. 47-55 (1975).

[FOSD74] L. D. Fosdick, "BRNANL - A Fortran Program to Identify Basic Blocks in Fortran Programs," Department of Computer Science, University of Colorado, Boulder, Colo., Technical Report, CU-CS-040-74.

[OSTE81] L. J. Osterweil, "TOOLPACK - A preliminary Design and Architecture," Dept. of Computer Science, University of Colorado, Boulder, Colo., Technical Report (to appear Spring 1981).

[RAMA75] C. V. Ramamoorthy and S.-B. F. Ho, "Testing Large Software With Automated Software Evaluation Systems," IEEE Transactions on Software Engineering, SE-1 pp. 46-58 (March 1975).

[STUC73] Stucki, L. G., "Automatic Generation of Self-Metric Software," Proceedings IEEE Symposium on Computer Software Reliability, New York, N.Y. (April 1973) IEEE #73CH0741-9CSR, pp. 94-100.

[STUC75] L. G. Stucki and G. L. Foshee, "New Assertion Concepts in Self-Metric Software," Proceedings 1975 International Conference on Reliable Software, IEEE Cat. #75-CH0940-7CSR pp. 59-71.

[TAYL80] R. N. Taylor, "The Design of Dynamic Assertion Verification Language," ACM SIGPLAN Notices, (January 1980).