Full-screen testing of interactive applications

by M. E. Maurer

This paper describes the dialog test functions of the Interactive System Productivity Facility/Program Development Facility program product, with emphasis on the full-screen design that makes it unique. Perspective is provided by a brief summary of the test facilities available in the predecessor System Productivity Facility program product (SPF) and the requirements that led to their enhancement.

The last ten years have seen a dramatic increase in the number of on-line interactive data processing applications, at the expense of traditional batch production systems. However, the increased sophistication and ease of use found in these end user systems have not often been matched in the tools used by programmers to develop them. The growing programmer shortage and today's emphasis on programming quality and productivity are focusing more attention on this deficiency. Businesses are beginning to treat systems development as a regular application in itself,[1] and more programming tools are becoming available to support development. Many more integrated tools are needed to bring the programming development process up to par with existing end user applications.

Most of a programmer's work is divided among the activities of design, coding, testing, and documentation. The project described in this paper has concentrated on the testing phase—specifically, the testing of interactive applications—and has tried to apply to it the same usability characteristics and state-of-the-art technology found to be successful in current user applications. This work was part of ongoing development efforts to enhance the System Productivity Facility (SPF) program product,[2] a widely used program development tool. The resulting dialog test function was shipped in the Interactive System Productivity Facility/Program Development Facility (ISPF/PDF or PDF) program product, which is an SPF follow-on. Figure 1 summarizes the evolution of the SPF product set and points out where dialog test fits in.

Because of the large amount of programmer time spent on detection and correction of programming errors—twenty-five percent of the development time reported in one study[3] and three times the time spent on coding reported in another study[4]—it is no surprise to find close attention paid to the productivity of the debugging process.

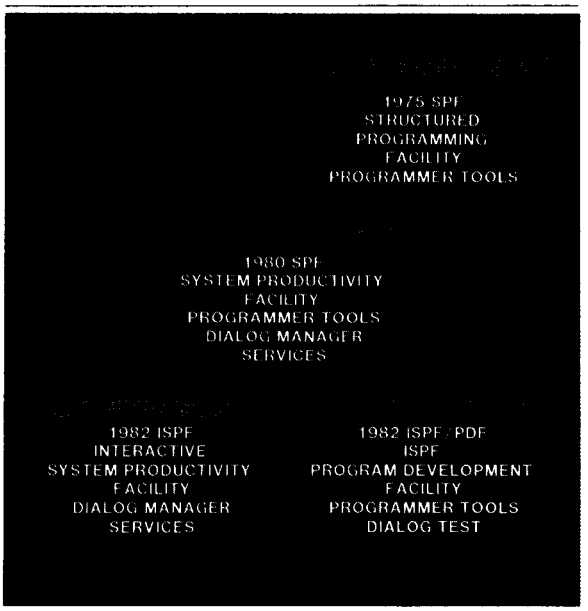
Interactive debugging products are available for such machines and operating systems as the Program Control System on TSS/360, Interactive Debug on System/34, TSO TEST for MVS, and CP/CMS Debug for VM. Even compilers have interactive debugging versions—the PL/I Checkout Compiler, for example. What is new in the current effort is the exploitation of the power of a full-screen terminal to provide much more usable debugging assistance, and provide it at the symbolic level at which the programs are coded.

Furthermore, the implementation of the support as a consistent extension to the program development tool already used by the programmer for coding, compiling, and documentation gives the programmer familiar and established externals and a coherent and more complete package. Let us briefly describe the underlying SPF product to explain the derivation and applicability of the current effort.

History

SPF was first introduced in 1975 as the Structured Programming Facility, which is a collection of programming development tools that have greatly simplified many programming tasks. SPF prompted the user to enter required information on a sequence of display screens and retained much of that information from one session to the next. Included in SPF were data-set utilities, job submission functions, operating system command interface, and full-screen edit and browse functions with four-way scrolling.

Figure 1 Evolution of the SPF products



Our experiences in developing such applications suggested requirements for debugging assistance.

Within a few years, users were learning that there were underlying service routines in SPF that could be even more valuable than the tools for those who were developing interactive applications. These services could be used to free the programmer from display device and many operating system dependencies and to facilitate data handling, allowing the programmer to concentrate on application-specific processing. Such interactive applications became known as *dialogs* (referring to conversations between the user at the terminal and the program in the computer). The collection of services was then termed the *dialog manager*.

Demand for a formalized offering of this function resulted in the development of a new SPF in 1980, the System Productivity Facility program product. This "new" SPF included all the capability of the previous product, plus the externalization of the dialog manager services.

The current project was begun to address problems of moving user-written ISPF applications into production efficiently and effectively. Our experiences in developing such applications suggested requirements for debugging assistance. Our observations of other testing tools, when compared with the technologies in user applications, showed that there was much opportunity for improvement of their user interfaces. The rudiments of debugging functions were part of the old SPF product, but it had to be greatly expanded to address the needs of large-scale system developers.

This paper concentrates on ways in which the dialog testing functions incorporated in ISPF/PDF have advanced the state of the programming testing art. First, we examine how a programmer uses ISPF services. We then analyze testing requirements and describe the environment that ISPF offers for testing. Finally, we describe ways in which the new product has been able to satisfy these requirements in innovative ways. The reader is referred to the available product documentation for additional detail.[5–7]

Coding ISPF applications

A programmer may code an ISPF application in any one or a combination of the following languages: the command procedure language of the host system (EXEC2 for VM/CMS or CLIST for MVS/TSO) or a supported programming language (including PL/I, COBOL, FORTRAN, or assembler language). The command procedures or programs are called *application* or *dialog functions*, and they embody the processing of the application. When the programmer needs a service provided by ISPF, he codes a call to that ISPF service in his function. Services available to the ISPF application perform the following functions:

- Direct the flow of control among the dialog functions and the panels.
- Display predefined panels (screen images) and messages to the end user.
- Define and maintain ISPF symbolic variables to pass data among the dialog functions, the dialog manager (ISPF), and the end user.
- Maintain application data in specialized files, called *tables*. Tables are logically similar to two-dimensional arrays in which each row corresponds to a record in a traditional data set, and each column corresponds to a field in that record.
- Produce tailored output by processing control statements and performing variable substitution in a special input file called a *skeleton*.
- Provide generalized edit and browse facilities for application data.

A user dialog, therefore, can contain many kinds of parts: functions, panels, dialog variables, tables, messages, and skeletons. Let us use an example to illustrate the relationship among these parts in a sample ISPF application.

Figure 2 shows part of a dialog that displays a panel to an end user and saves the data entered on the panel in an ISPF table. There are interactions among the application functions coded by the programmer, the ISPF dialog manager, the data sets it uses, and the end user at a terminal. The reference numbers in the figure identify each major interaction.

At reference number 1 in Figure 2, the dialog invokes the ISPF display service. The display service obtains the requested panel from the panel library at 2 and displays the panel at the terminal. The display service interprets the user responses at 3, and then returns control to the application at 4, when the user enters the END command. Next, the application invokes the table add service at 5, which writes the data to the application's table at 6, and returns control to the application at 7.

Notice that the application programmer has only to invoke the display service. There is no need to understand how to obtain the panel from the library, how to send it to the terminal for display, or how to obtain or interpret the user response. In these ways ISPF services increase the productivity of programmers who are developing interactive applications. The programmer does have to code the panel, however (including the screen image and processing logic his application requires), and be sure it is in the panel library for ISPF to find.

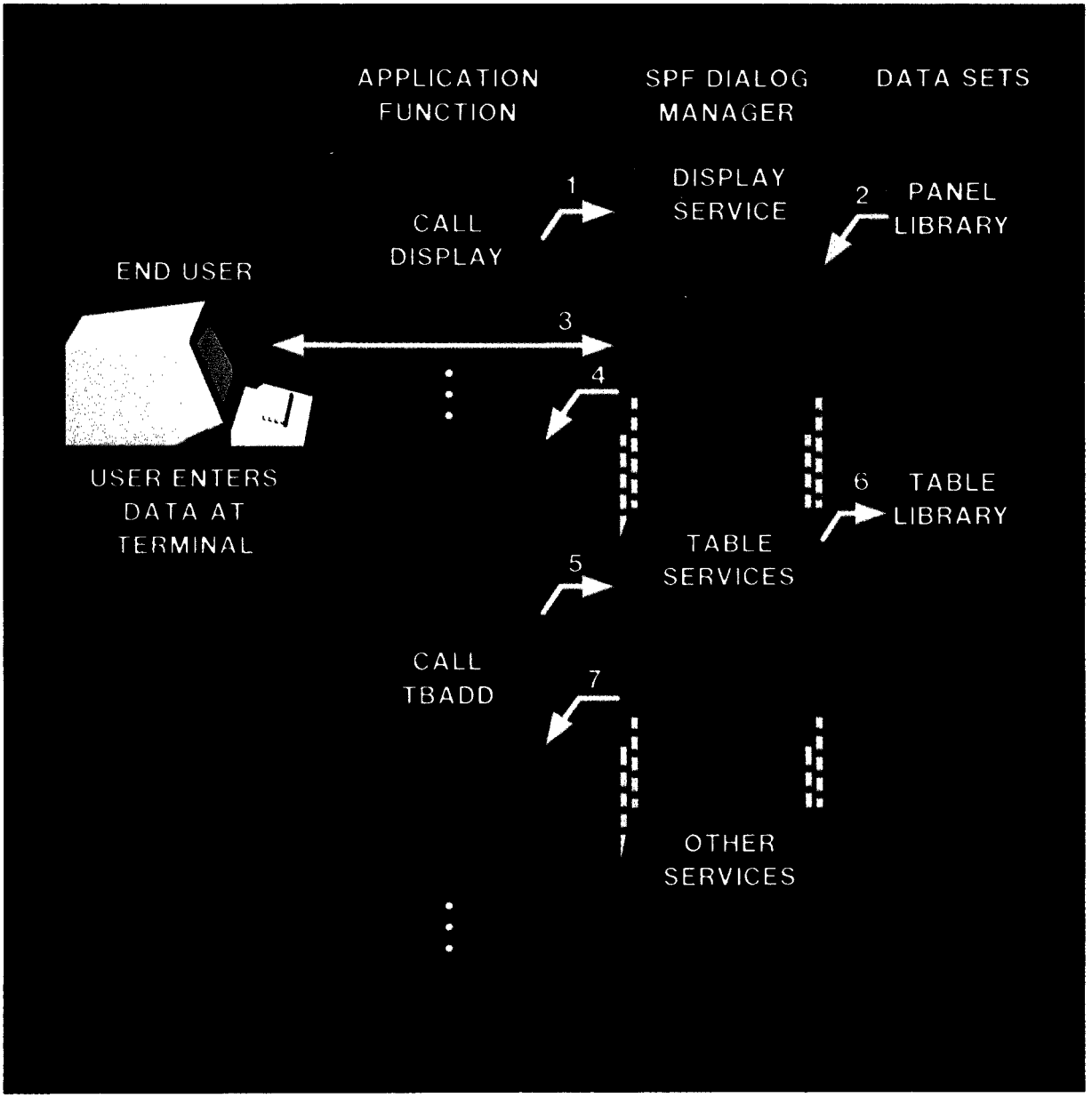
Notice that the application programmer has only to invoke the display service.

With this example in mind, we examine the testing assistance that would help the programmer debug ISPF applications like this, and the usability characteristics ISPF should have.

Requirements

As with most interactive products, user requirements for testing dialogs fall into two basic categories: (1) functional requirements that describe the product capabilities, and (2) interface requirements that describe the user interface to the function. In many ways, the interface requirements are more important than the functional requirements, because a product that is not easy to use will not be used, regardless of its function. Indeed, this is the area we identified as significant for enhancement in the current project. We wanted to give the tester facilities comparable to those in current end user applications.

Figure 2 Sample ISPF application interactions



The functional requirement for dialog test is to help a tester debug the ISPF-specific parts of applications—panels, messages, file skeletons, tables, and ISPF services used. The programmer developing the application illustrated in Figure 2, for example, needs a way to test the panel for proper coding. The product should provide convenient ways to simulate missing, incomplete, or defective parts of a dialog. In the example above, it should be possible to test the subroutine that updates the table, even if the subroutine that displays the panel to the user is not yet written. The debugging tool must help the

programmer test both the individual parts of an application and the integrated and complete system.

This function should complement existing operating system debugging aids used to debug non-ISPF application processing logic. It has to be sufficiently flexible to complement different styles of testing, yet provide enough guidance so that a novice tester becomes productive quickly.

Dialog test, therefore, makes extensive use of selection and data entry panels instead of commands.

The general process of debugging any program suggests some other kinds of function that dialog test should provide. For instance, on the most basic level, a tester executes a program and hopes it runs correctly. When it fails, the tester tries to determine where, how, and why it failed. He typically goes through an iterative process, narrowing down the source of an error. It is helpful to stop the test at key points in the execution to examine variables or modify their values, to examine input to and output from critical or suspect subroutines, and to trace control flow or changes to data areas. This view of testing provides the justification for many of the functions in dialog test.

Implementing dialog test as an ISPF application and part of PDF provides a firm basis for satisfying the user interface requirements. By retaining the time-tested externals of SPF, we could provide consistency and familiarity to the user. Dialog test, therefore, makes extensive use of selection and data entry panels instead of commands. It also has a similar panel design and (where applicable) control flow and screen manipulation commands and processes identical to those of SPF. As an ISPF application itself, dialog test has available the same facilities as end user applications.

In addition, we recognized the importance of letting the developer conduct the debugging sessions at the symbolic level at which the application is written. Unlike many existing debugging aids, data and locations in programs should be referenced by name, rather than by computations of hexadecimal addresses. Data should be displayed in translated characters whenever possible, not in memory format. The burden of remembering lengthy or complex data should be assumed by the system. Dialog test should present lists of items for selection by the user rather than requiring him to remember a name to tell the program. We wanted to avoid introducing a new complex debugging language and to minimize the amount of training required to use the function. This means that dialog test should present testing options as selections on panels rather than forcing the tester to remember and enter commands. Listing the testing options would also help the tester remember the available facilities, thereby encouraging him to take advantage of all the power of the system when doing a test.

Discussions with internal users who had built ISPF dialog applications verified these requirements. Before designing the specific functions, we had to ensure an adequate execution environment for testing dialogs. SPF had very limited test functions that required major restructuring for this ISPF/PDF project.

Base test support

The SPF product provided test support in two ways. It allowed the tester to set general execution conditions to facilitate testing through parameters with which SPF was invoked. It gave specific debugging functions through options on panels during a test session.

Specifying TEST and TRACE when SPF is first invoked results in the following key execution characteristics:

- Dialog objects are re-fetched from their libraries when needed, so that the latest changes can be used immediately.
- Extra information, which is useful for debugging, is shown on screen image printouts and tutorial panel displays and is written to the log.

Figure 3 SPF support selection panel

```
------------------------  SUPPORT SELECTION MENU  ----------------------------
SELECT OPTION ===>_

   1  TEST PANEL      - DISPLAY PANEL AS USER WOULD SEE IT
   2  TEST FUNCTION   - INVOKE DIALOG FUNCTION OR SELECTION MENU
   3  TEST VARIABLES  - SET OR DISPLAY VARIABLES FOR TEST FUNCTION
   4  CONVERT MENUS   - CONVERT SELECTION/TUTORIAL MENUS TO NEW FORMAT
   5  CONVERT MSGS    - CONVERT MESSAGES TO NEW FORMAT
   6  TEST MENU       - TEST OLD FORMAT SPF MENUS
   7  TRACE MODE      - SET TRACE MODE FOR SCREEN 1
```

- The tester can continue execution after a severe error, so that more than one problem can be discovered in a single test session.

The execution functions are implemented as part of the SUPPORT option on the SPF Primary Option Menu. This selection displays the panel shown in Figure 3, on which the first three options are testing functions, and the remainder help a programmer convert to new SPF facilities. With the test panel function (option 1 in Figure 3), the programmer can visually verify the format of his panel, and then type data into input fields. The programmer identifies the panel, using the same parameters that would be coded in a dialog to invoke the DISPLAY service. Option 2 lets the tester identify and give control to the function to be tested, by entering data for the parameters of the SELECT service in appropriately labeled fields. Finally, variable names can be entered on a list with option 3 to display their current values, to modify their values, or to define new variables.

Although offering some help to the dialog developer, this test function was limited by restrictions on the nature and scope of reference of the test variables and by requiring the tester to remember the names of the dialog variables of interest. For ISPF/PDF, we were interested in eliminating these constraints on the dialog tester and adding significant additional function.

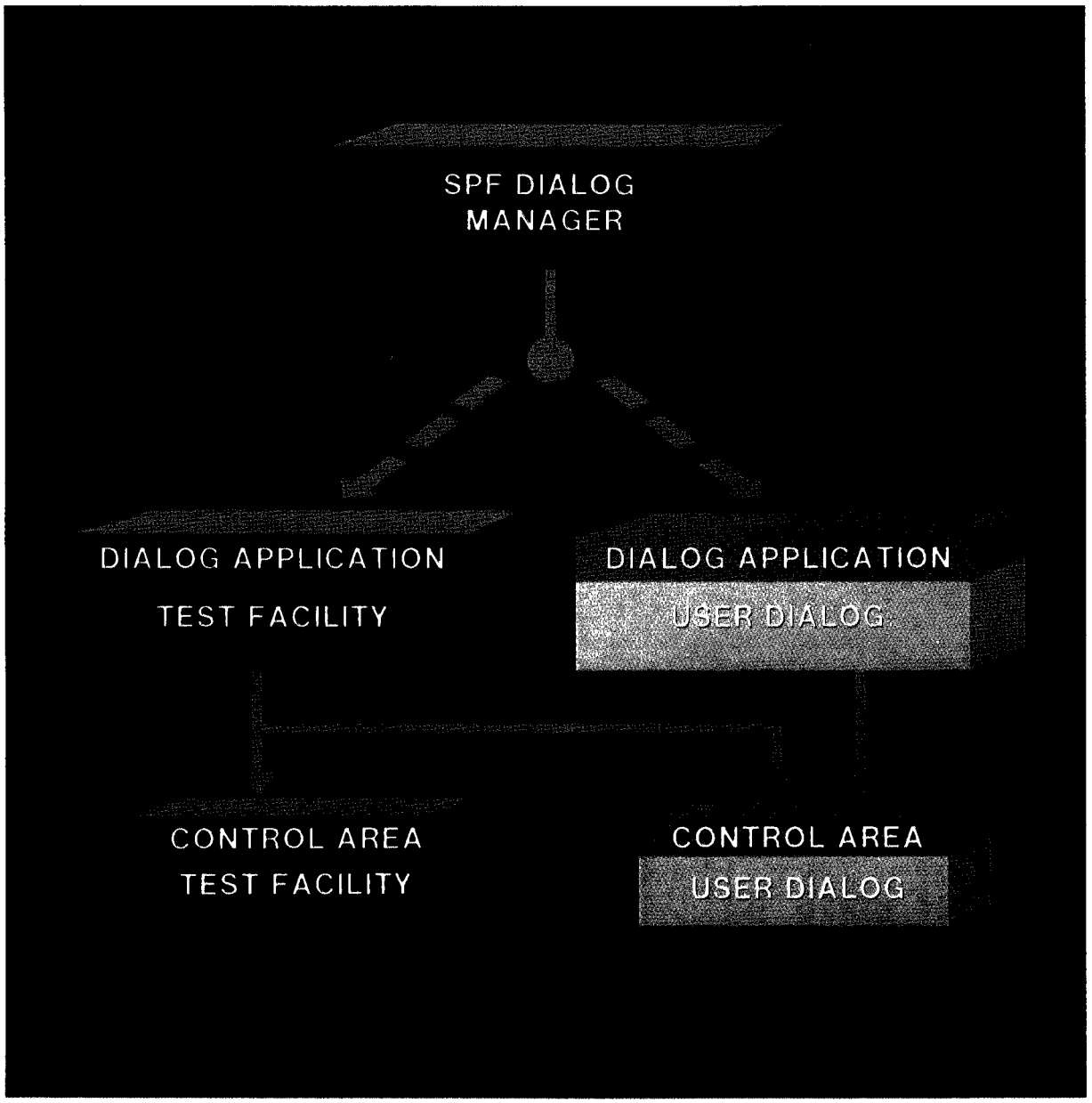
Extended test environment

The goal for the new test function was to simulate an actual production environment for the user dialog being tested. The user dialog should not require modification for test, and ISPF variables should be accessible to the tester just as they are to the program being tested. The dialog test facilities should execute as an ISPF application under the control of the dialog manager, but give control to the user dialog (another ISPF application) as though it were running alone on ISPF.

Implementation of a *test environment* accomplishes these objectives. Critical to this implementation is a careful separation of dialog variables and control structures between dialog test and the dialog being tested. However, the dialog manager has knowledge only of the active dialog at any particular time, and it is not aware that control is switched back and forth between the two dialogs.

Figure 4 illustrates this problem. The ISPF dialog manager controls an application running under it, and creates control structures and dialog variables for the application. As this application, the dialog test facility "shadows" itself and separates its control areas from a similar set belonging to the dialog being tested. At appropriate times, dialog test causes control to be passed to this shadow application or takes back control.

Figure 4 Dialog manager/dialog test relationship



In addition, any operation on user data performed through a dialog test option is treated as an extension of the user program. This means, for example, that if the tester changes a variable value (using the dialog test variables option), the variable retains the new value when the dialog being tested is given control. This is consistent with the expected uses of dialog test options to substitute for missing pieces of an application or to correct problems caused by the dialog being tested.

With this internal test environment in place, we could design a test function to satisfy the detailed requirements for function and usability.

Implementation

ISPF is implemented so that the tester need never set initial parameters to do testing. Some of the old function obtained from those parameters is now given to the tester automatically when the test facility is selected. Other functions have been replaced with greatly expanded capability and flexibility that are available interactively to the tester.

The old display panel and invoke function options have been retained, with some change to reflect new capabilities of the underlying ISPF services and the enhanced test environment. The variables option has been completely redesigned to increase its power and usability.

New test functions for ISPF/PDF, derived from an analysis of the typical debugging process, include the following:

- The ability to set breakpoints where execution of the application being tested is suspended, to allow the use of other test facilities.
- The ability to trace the usage of dialog services and dialog variables.
- The ability to browse trace output in the ISPF log during test.
- The ability to examine and update ISPF tables.
- The ability to invoke interactively any dialog service except CONTROL.

The old SPF SUPPORT option has been redesigned as the DIALOG TEST option, with a primary option menu that presents selections for the dialog test functions, as shown in Figure 5. This approach lets the user conduct the test by choosing the desired options, in the order required for the application

Most important is that the user's interface to these functions be as simple, flexible, and consistent as possible.

Figure 5 Dialog test primary option menu

```
---------------------  DIALOG TEST PRIMARY OPTION MENU  -------------------
OPTION ===> _

   1  FUNCTIONS         - Invoke dialog functions/selection menus
   2  PANELS            - Display panels
   3  VARIABLES         - Display/set variable information
   4  TABLES            - Display/modify table information
   5  LOG               - Browse ISPF log
   6  DIALOG SERVICES   - Invoke dialog services
   7  TRACES            - Specify trace definitions
   8  BREAKPOINTS       - Specify breakpoint definitions
   T  TUTORIAL          - Display information about Dialog Test
   X  EXIT              - Terminate dialog testing

Enter END command to terminate dialog testing.
```

problems at hand and for his particular style of testing. Categorizing and listing the functions this way reduces training requirements by reminding testers of the available options.

Most important is that the user's interface to these functions be as simple, flexible, and consistent as possible. Many test options—variables, tables, breakpoints, and traces—require sets of information that the tester usually enters and updates during the test session. We have recognized this commonality and have made it the most important area in which to concentrate our usability improvements. Described next is the basic design for all these areas, illustrated by the variables option. This is followed by a description of the other new test functions.

Externals

The manner in which the test functions are provided to the user has undergone many revisions, because of the importance of having the right user interface and the importance we place on optimizing the usability of the product. Following the successful precedent of the SPF product and the requirements for dialog test, it became clear that the testing options should be presented to the user as lists, and that the user would be prompted to enter required data. The types of panels to be used for the prompting, and their organization, were critical decisions.

Data entry panels were planned first, to capture the information needed for each function or to display information to the tester. These are panels with labeled fields that indicate to the user the information being requested or displayed. Figure 6 shows our initial panel design approach to define and initialize a variable. This design takes advantage of the whole screen and clearly identifies information.

Limitations to such a design become obvious when an attempt is made to change or delete a previously specified variable, or to examine all variables entered so far, or to determine the number of interactions required to create ten variables. We wanted to make it easy for the user to examine or change any variable or many variables, with a minimum number of interactions, and without remembering exact variable names.

Many functions of dialog test, in addition to variables, share these same requirements. A scrollable selection list, which is familiar from other parts of PDF, satisfies some of the objectives. That list, however, must be more than just selectable, because

Figure 6 Initial design of the define variable panel

```
------------------------  DEFINE VARIABLE  -----------------------------------
COMMAND ===>

   ENTER THE DESCRIPTION OF THE NEW VARIABLE TO BE DEFINED:

   VARIABLE NAME     ==> _______

   VARIABLE POOL     ==> -                  F = Function
                                            S = Shared
                                            P = Profile

   ATTRIBUTE         ==> -                  T = Truncated

   VARIABLE VALUE    ==> ------------------------------------------------------
------------------------------------------------------------------------------
```

Figure 7 Final design of the test variables panel

```
----------------------  VARIABLE DISPLAY AND SET  ---- LINE 000001 COL 001
COMMAND ===>_                                                SCROLL ===> P

ADD AND CHANGE VARIABLES.  UNDERSCORES NEED NOT BE BLANKED.
ENTER END COMMAND TO FINALIZE CHANGES.

      VARIABLE  P A VALUE

 '''' EMPLBLDG  F   101
 '''' EMPLDEPT  F   a28
 '''' EMPLDIV_  F   Manufacturing
 '''' EMPLFNAM  F   Mary
 '''' EMPLINIT  F   D
 '''' EMPLLNAM  F   Doe
 '''' EMPLLOC_  F   Poughkeepsie
 '''' EMPLNUM_  F   12345678
 '''' EMPLOFF_  F   1-B-34
 '''' EMPLPHON  F   111-2222
 '''' Z_______  S N
```

update capability is necessary for many of the displayed fields.

It became apparent that *table display panels* of ISPF could be implemented to satisfy all these objectives. These panels present tables of information in two-dimensional arrays. Here each row corresponds to one entity (such as a variable) and each column contains one item of data describing that entity (such as its value). The table data may be scrolled, so that there can be as many rows as needed. The application presenting a table display panel—dialog test, in this case—can be written to support multiple updates of data in any of the columns. The addition of a column for the entry of *line commands* provides a way for even more flexible manipulation of the data by the user.

The final design of the variables test panel used to display, modify, or define and initialize variables is illustrated in Figure 7. The display panel shows all the variables defined for the dialog being tested, so that the user need not remember and enter the names of the variables. The names are arranged in the following pool order to match their search order from a user program: (1) function pool, (2) shared pool, and (3) profile pool. Within each pool, the names are arranged alphabetically. While the user is examining or setting a specific variable, it is easy to browse the other variables and possibly spot an error.

The system retains all specifications that the user has entered or that the dialog has created, and automatically displays them when the appropriate test option is selected. The user simply overtypes any data on the display that are to be changed or else uses line commands to manipulate an entire row.

The leftmost column of the test variable panels is a line command area, as in the PDF Editor. New variables are defined by inserting new lines in the display, using the I line command, and entering descriptions in the appropriate columns. Other test options permit the use of appropriate line com-

Figure 8 Help panel for test variables option

```
TUTORIAL ------------------------ VARIABLES -------------------------- TUTORIAL
OPTION ===> _

                          ----------------------------
                         |                            |
                         |          VARIABLES         |
                         |                            |
                          ----------------------------

  The VARIABLES option allows you to display and modify the function, shared,
  and profile pool variable values for the function currently in execution.
  In addition new variables may be created.

The following topics are presented in sequence, or may be selected by number:

    1 - Definitions                      4 - Primary Commands
    2 - Variables Panel                  5 - Line Commands
    3 - Manipulating Variables           6 - Usage Notes


------ CUR PANEL = ISR7V000   PERV PANEL = ISR7V000   LAST MSG ISPD241 ------
```

mands, including insert, delete, and repeat. The LOCATE primary command makes it easy to find an item such as a variable in the displays.

This implementation exhibits all the usability characteristics identified in the requirements analysis. Values of variables are readily examined because the system automatically presents an ordered list of all known variables when the function is requested and identifies them by the symbolic name assigned by the programmer. Variable values are easily modified by overtyping the field after the proper entry is found either by scanning the list or by using the locate command. New variables are easily defined by inserting a line and typing the name, pool, and value. Many of these operations can be performed with a single interaction.

The tester needs the ability to display variables both to verify correct dialog execution and to pinpoint the location where incorrect data are being created. The tester may want to change the values of variables to correct errors so that a test can proceed. It may also be necessary to force different paths through the code. Further, a tester may find it necessary to define new variables to substitute for missing or incomplete parts of an application.

As in all PDF, detailed tutorial information for test support is available through the HELP command. Here, too, attention is given to usability. The tutorial for each suboption in test starts with a selection panel to categorize the available information into definitions (if applicable), panel description, task instructions, commands, and usage notes to aid in quickly finding the answer to a question. Figure 8 shows the tutorial page displayed when help is requested from the variables panel.

All tutorials that describe panels make it easy to find the explanation for a particular part of a panel by intensifying the words from it that identify each

Figure 9 Help panel for variables test panel

```
TUTORIAL ---------------- VARIABLES - VARIABLES PANEL ------------------ TUTORIAL
COMMAND ===> _

 The VARIABLES PANEL allows you to change variable values and create new
 variables. Each row of the scrollable display represents a variable, as
 follows:

  ''''          - The line command area (see the Variables Line Commands topic).

  VARIABLE      - The variable name.  This field is required.

  P             - The pool in which the variable exists:
                    f  - function.
                    s  - shared.
                    p  - profile.
                  This field is required.

  A             - Attributes of the variable, if any:
                    N  - non-modifiable variable.
                    T  - truncated variable.
                  This field is non-modifiable.
```

field, offsetting their descriptions to the right, and listing them in the same order as the original panel. This is illustrated in Figure 9 by the tutorial page for the panel description of the variables test panel (obtained by selecting 2 on the panel shown in Figure 8).

Breakpoints

Existing operating system debugging facilities, such as TSO TEST or CMS DEBUG, give the tester the ability to specify locations in programs that are known as *breakpoints*. At breakpoints, execution may be suspended in order to examine and manipulate program and test data. This capability is provided for dialog test and includes the ability to use any of the test options at a breakpoint. This way, the tester has maximum flexibility to respond to the unpredictability of a testing session. At a breakpoint, the tester can use all the test options to analyze the execution of a dialog. Then, on the basis of this finding, he can modify the rest of the test, using those same options.

The logical place to allow breakpoint interruptions for ISPF applications is at an invocation of dialog services. A breakpoint BEFORE execution of a service allows the tester to validate the input. A breakpoint AFTER such an execution allows the modification of the service return code to force execution of different paths through the application.

Additional flexibility derives from allowing the tester to identify specific dialog functions during which the breakpoint is to be considered or ignored, and to control the specific conditions under which a breakpoint can occur. For example, the tester may specify that a breakpoint is to occur only in a particular function or only if a particular service has completed with a given return code, or only if it was invoked with a specified parameter value.

Specifications for these conditions for breakpoints, like variables, are entered on a table display panel. Here each row is a new breakpoint definition and each column contains another part of the description of the breakpoint. Since many breakpoints may be needed for only specific and possibly separate parts of a test, an ACTIVE column lets the tester disable a breakpoint. The breakpoint may be re-enabled later without having to re-enter all its data. The ACTIVE column and other columns default to the most commonly used values, so that all the tester has to do is identify the ISPF service name to completely define a breakpoint.

The user requires access to trace data during a test in order to use that information (perhaps at a breakpoint) to influence his next actions.

Whenever the tester selects the BREAKPOINTS option, all the breakpoints previously defined are displayed. He can then modify breakpoint specifications by overtyping selected data, as well as define new breakpoints by inserting new rows in the display. This design is very different from a testing tool like TSO TEST, where the programmer must calculate the hexadecimal address of the location of a breakpoint. Also, in a tool like TSO TEST, no identification of the required information is given, and all existing breakpoints are not automatically displayed while new ones are being defined or updated.

When the dialog being tested is executing and a breakpoint is reached, the tester is presented with a panel that identifies the call within the dialog where the breakpoint is taken. The displayed panel contains selections for all the dialog test options. Using these options, the tester learns more about the problem being analyzed, and thus can use this information dynamically to refine the conduct of the test. When the tester is ready to resume execution of the interrupted dialog, he has only to select a GO option.

Traces

Detailed information about the use of dialog variables and dialog services is a valuable debugging aid provided through the TRACE option. Since tracing may degrade performance and create large amounts of output, however, the tester must be able to limit the scope of a trace to only those data that are needed.

The single SPF TRACE parameter was replaced in ISPF/PDF by a powerful and flexible tracing capability to track both variables (that are referenced, set, or changed) and dialog service calls throughout an application or within specific functions.

Definitions of traces are entered on table display panels with all the function previously described for variables and breakpoints. The trace output for a variable includes the name of the variable, its value, the pool in which it is defined, the type of reference to it, and the ISPF service that caused the reference. For a service call, the output identifies the application, screen, and function in which the call occurred, the starting and ending point of the service, and the parameters with which the service is invoked.

In the design of trace support, it was difficult to resolve the questions of where to direct trace output and how to make it accessible to the user. One candidate for the output was a new trace data set. Opposing this was the fact that it is generally undesirable to require additional data sets for the user to define and manage. The ISPF transaction log was another possibility, but the log could not be accessed during the ISPF session. The user requires access to trace data during a test in order to use that information (perhaps at a breakpoint) to influence his next actions.

This problem was resolved by enhancing the log interface so that the ISPF log could be accessed during the test session in an easy and familiar way. The BROWSE LOG option gives the user the PDF Browse facility with the ISPF log. Additionally, the tester can use the split screen capability to browse the log on one screen while entries are being recorded into it on another. Using the log also provides a time stamp on each trace entry.

Figure 10 Typical trace output

```
BROWSE LOG - SPFLOG2.LIST ----------------------------LINE 000000 COL 013 092
COMMAND ===>_

TIME            *** SPF TRANSACTION LOG ***                    USERID: Z73MEM

14:50    START OF ISPF LOG - - -  - SESSION # 176 ------------------

14:54    DIALOG TRACE ------------ - APPLICATION(ISR)  FUNCTION(T1) SCREEN(1)

14:56     TBCREATE BEGIN ... .... - ISPEXEC TBCREATE T1 KEYS(EMPLNUM EMPLNAME)

14:58     TBCREATE END.. ... .... - ISPEXEC TBCREATE T1 KEYS(EMPLNUM EMPLNAME)

14:60       ..RETURN CODE (4)     -

14:62      EMPLNUM. POOL (F) .... - VALUE(876914)

14:64       ..GET BY TBADD        -

14:70    DIALOG TRACE ------------ - APPLICATION(ISR)  FUNCTION(T2) SCREEN(1)

14:72     END OF ISPF LOG - - - - - SESSION # 176 ------------------
```

In Figure 10, the DIALOG TRACE entry (at 14:54) identifies the application, function, and screen when a trace was initiated and indicates (at 14:70) when one of those values changed. A function trace entry appears at both the beginning and end of a dialog service call and shows the parameters with which that service was invoked. See lines headed TBCREATE BEGIN and TBCREATE END at 14:56 and 14:58, respectively. They indicate that the user dialog called ISPF to create a table named T1 through the ISPEXEC interface and that the table had two keys, variables EMPLNUM and EMPLNAME. Variable trace entries span two lines and show the variable name, pool, value, type of reference, and service causing the reference. For example, one can infer from the variable trace entry at 14:62–14:64 that variable EMPLNUM was being traced by the user. At this time, his dialog called the TABDD service to add a row to the table. That service needed to get the value of variable EMPLNUM, thus causing the entry "876914 " in this trace log.

Tables

ISPF tables provide a convenient way to save application data either permanently or temporarily. A set of table services makes this function available to the application. The tables option of dialog test gives the tester quick interactive access to many table services for examining or manipulating tables during a test.

Using the options identified on the table panel shown in Figure 11, the tester can easily do the following:

- Enter test data into an ISPF table to use as input for the dialog being tested.
- Correct errors in a table during a test so that a test can continue without interruption.
- Substitute for an unavailable part of a dialog that would have manipulated a table to provide input for the test.

Figure 11 Test tables panel

```
------------------------------  TABLES  ---------------------------------------
OPTION ===> 1_

   1  Display row                        5  Display structure
   2  Delete row                         6  Display status
   3  Modify row
   4  Add after row

TABLE NAME    ===> testtab                CURRENT ROW:

ROW IDENTIFICATION:
  BY ROW NUMBER ===> *                   (* = current row)

  BY VARIABLE     VALUE                  (Search for row if row number blank)
     empname      Maurer
     --------
     --------
     --------
     --------
     --------
```

- Display the status and structure of a table to verify its proper creation by the dialog.

Before selecting this option, a table can be opened or created, as appropriate, by invoking the TBOPEN or TBCREATE service on the DIALOG SERVICES option. In other cases, the dialog being tested would already have created or opened the table.

The status and structure options are useful for determining the existence, accessibility, and characteristics of a table. The row options facilitate examination and modification of the contents of the table. The tester is given much flexibility in identifying the table row of interest. He may enter an absolute row number, or request the current, top, or bottom row, or provide specific variable names and values to search for in the row (as shown in Figure 11). Subsequently, table display panels are used to present individual rows of the table to the tester, with each row of the display showing one variable from the table being examined or updated. Columns on the display are used to indicate the type of variable and its current value.

Concluding remarks

The dialog test functions of ISPF/PDF have been designed to exploit the capabilities of a full-screen terminal in assisting the programmer to debug an ISPF application. As an extension to the program development system, this support provides consistency and familiarity to the programmer, and reduces the training needed.

The uniqueness of the PDF test functions lies in their use of the full screen of the terminal. The display gives presentations of all information entered so far,

and easy ways to modify and add to it, through a minimum number of interactions. For interactive applications, the tester has available the key test functions found in most debugging tools—breakpoints, traces, and variable display and modification—as well as assistance for ISPF-specific items such as panels and tables. This function is furnished at the symbolic level of the application and is designed to be adaptable to a wide variety of testing needs.

Inexperienced programmers have been successful and productive, even enthusiastic, in their debugging of new dialogs using these options.

Early feedback from internal IBM users indicates satisfaction with this approach. Inexperienced programmers have been successful and productive, even enthusiastic, in their debugging of new dialogs using these options. They have learned how to use the test functions from the panels themselves and the tutorials, because no publications were available at that time and the developers were unable to teach them. The real test of our work will come with wider use by more kinds of testers and many different kinds of ISPF applications. We look forward to seeing the further requirements that such use might generate, and to seeing other debugging tools also move to a full-screen approach.

Acknowledgments

The author gratefully acknowledges the editorial assistance of Samuel B. Lee and the contributions made by all who worked on this project, especially the lead designer, Stanley A. Miller.

Cited references

1. P. E. Schniedler, Jr., "IBM markets a center to improve productivity," and W. P. Martorelli, "Conn. Center Speeds Applications," *Information Systems News*, No. 1 (February 8, 1982).
2. P. H. Joslin, "System productivity facility," *IBM Systems Journal* **20,** No. 4, 388–406 (1981).
3. W. A. Delaney, "Predicting the costs of computer programs," *Data Processing Magazine* **8,** No. 10, 32–34 (October 1966).
4. F. J. Rubey, "A comparative evaluation of PL/I," *Datamation* **14,** No. 12, 20–25 (December 1968).
5. *Interactive System Productivity Facility, General Information*, GC34-2078-0; available through IBM branch offices.
6. *Interactive System Productivity Facility, Dialog Management Services*, SC34-2088-0; available through IBM branch offices.
7. *Interactive System Productivity Facility, Program Development Facility for MVS, Reference*, SC34-2089-0; available through IBM branch offices.

Meg E. Maurer *IBM Information Programming Services, P.O. Box 390, Poughkeepsie, New York 12602.* Ms. Maurer first joined IBM in 1968 at the Poughkeepsie Programming Center on the OS/360 project. Her assignments were in control program design and early feasibility studies of virtual memory and its application to operating systems. From 1972 to 1977 she was employed outside IBM successively as an applications project leader, data base administrator, and manager. She rejoined IBM in 1977 to work on prototype software problem determination tools. Since 1980, she has been a manager of the ISPF/PDF development project, with responsibility for dialog test function. Ms. Maurer received a B.S. degree in psychology at Fordham University in 1968. She has also done work toward an M.B.A. degree at the University of Connecticut at Stamford and at Marist College in Poughkeepsie, New York.

Reprint Order No. G321-5194.