Automatic generation of random self-checking test cases

by D. L. Bird
C. U. Munoz

A technique of automatically generating random software test cases is described. The nature of such test cases ensures that they will execute to completion, and their execution is predicted at the time of generation. Wherever possible the test cases are self-checking. At run-time their execution is compared with the predicted execution. Also described are implementations of the technique that have been used to test various IBM programs—PL/I language processors, sort/merge programs, and Graphical Data Display Manager alphanumeric and graphics support.

In this paper our main intention is to communicate to the reader the concept of test case generators—programs that create random test material for software. The methods we advocate create test cases that execute to completion without error and are self-checking. The test cases themselves will detect errors at execution time.

Although the concepts of test case generation apply equally to different areas of software, it is important to note that a separate and distinct generator must be coded for each item of software to be tested by this method. A test case generator is a specific test tool rather than a general one. The main sections of the paper describe three specific test case generators that have been used to test IBM program products. These descriptions illustrate the techniques and provide guidance for the readers who wish to implement their own test case generators.

Predecessors of our generators

Two predecessors of our test case generators were important and are briefly described. They are "syntax machines" and Seaman's work on compiler testing.

Syntax test case generators. Syntax test case generators have been produced by Hanford,[1] Purdom,[2] Celentano,[3] and Bazzichi and Spadafora.[4]

The technique of these generators is most often used for compiler testing. A formal definition of the source language is provided as input to the generator. This definition runs to several hundred lines for a language such as PL/I. It specifies the permitted forms of all statements and their arguments. It defines also the various compound statements, such as loops, groups, and blocks. Random test cases will then be produced that obey these syntax rules. It is also possible to request a minimal set of tests that exercise "all productions" of the formal definition.

Such test cases are satisfactory for testing the syntax phase of a compiler and may uncover some defects, or "bugs," in code generation. Their limitation is that the resultant generated code will not, in general, be executable. Even if a test case does execute without ending abnormally, it is impossible to tell if it has executed correctly. The only way to detect bugs in the code generation phase, therefore, is to manually check the code produced—an immensely tedious and difficult job.

Seaman's PL/I test case generator. In October 1974, R. P. Seaman[5] described his work in testing high-level language compilers. His method created random test programs that exercised various features of the PL/I language. He ensured that the programs were executable (by initializing all variables, choosing valid subscripts, etc.), but did not go so far as to predict the execution of the randomly generated statements.

His method of verification depended on the existence at the time of two parallel compilers (the PL/I Optimizing and Checkout Compilers). Each test case would be compiled and executed twice. The resultant values of all the variables after the optimizing run would be compared with those after the checkout run. Any discrepancy would indicate a bug in one of the two compilers.

The method proved very successful, and Seaman noted, "The present method of writing test cases (hand-coding) is very much a hit and miss affair. There are far too many combinations of language and implementation features for an exhaustive test, so many arbitrary decisions have to be made. Currently, these random decisions are made when the test cases are written and never subsequently changed. As a result, there are permanent gaps in the testing coverage. We need to automate the writing of test cases so that these random arbitrary decisions are not fixed and frozen in the test cases forever. If this can be achieved, the effect will be a greater coverage of the compiler by the testing process."[5]

In the years following Seaman's work a team of three programmers, including one of the authors of this paper, did design and write a PL/I test case generator that produced executable, self-checking test cases. These test cases covered a large subset of the PL/I language and did not rely on the presence of a second compiler for verification. This PL/I generator is described in the next section.

PL/I Test Case Generator

The PL/I Test Case Generator was first used to test later releases of the PL/I Optimizing and Checkout Compilers. The test cases produced were syntactically correct and guaranteed to execute to completion without any errors or program checks. Another important aspect was that they were completely self-checking. If a test case produced no error diagnostics, it was known to have executed correctly and could be discarded. A test case several hundred lines in length could be generated in a few seconds.[6]

The heart of a test case generator is a loop.

The composition of a compiler test case. Generated compiler test cases are composed of three different types of statements:

- *Structural statements.* These form the basic framework of the test case. They include the PROCEDURE and END statements and the subroutine that sends out the error diagnostics.
- *Random statements.* These form the bulk of the test case. The type of statement and all its options and arguments are chosen randomly to span the language covered by the generator.
- *Self-checking statements.* These statements are intermingled with the random statements to verify at run-time that the test case executes in a correct manner.

A simple example illustrating this composition is the following:

```
PL00001:PROC OPTIONS (MAIN);        /* MAIN PROCEDURE (STRUCTURAL) */
DCL FIX01 FIXED BIN(31) INIT(3);    /* DECLARATION    (STRUCTURAL) */
DCL FIX02 FIXED BIN(31) INIT(3);    /* DECLARATION        (RANDOM) */
FIX01=2+FIX02; /* DEBUG: 5 */       /* ASSIGNMENT         (RANDOM) */
IF FIX01¬=5 THEN CALL ERROR(1);     /* VALIDATION  (SELF-CHECKING) */
END PL00001;                        /* END           (STRUCTURAL) */
```

The relative frequency of each of the possible source statements (and options) is controlled by a set of

weights. These weights (which are declared as variables inside the generator) each have a default value that may be respecified at test case generation time. If one particular type of statement has a high weight, it will appear densely in the generated test case. Should a statement have a zero weight, it will not appear at all.

At the end of each section of random statements, some checking code is added to the test case.

In this way, test cases may be focused on known weak areas of a compiler. Also, any language features not yet supported by a compiler under development may be weighted out until such support is available.

The structure of the test case generator. The heart of a test case generator is a loop that selects, randomly, the next source statement to be added to the test case. It then calls a generator subroutine responsible for creating a random example of that source statement. Each of these subroutines has three duties:

- It must create the text of the statement.
- It must predict the eventual execution of the statement.
- It must ensure self-checking of the statement.

The first of these duties is relatively easy to achieve and was the basis of previous test tools known as "syntax machines." The remaining two duties are harder to attain but provide the two key advantages of our method of test case generation: guaranteed executability and automatic self-checking.

In general there is one generator subroutine for each of the source statements in the language under test. There are also service routines such as a random-number generator, several expression generators, and variable-selection subroutines.

To predict the execution of each statement, the generator maintains a constantly updated dictionary of all the variables so far declared in the test case being generated. This dictionary holds information on the data type, array bounds, and, most important, the *current value* of each variable. These current values are continually changing as the test case is created statement by statement.

An example of statement generation. As an example of the steps involved in generating a single statement, we will consider the following (random) assignment statement:

```
FIX10(3) = FIX11*2 - FIX12(1,3);                    /* DEBUG:   49 */
```

These are the steps:

1. The random-number generator is called to choose which type of statement will be added to the test case. An "assignment" statement is chosen.
2. The subroutine responsible for generating assignments is called. It (randomly) decides that the assignment should be of the "fixed binary" type.
3. The subroutine next calls a service routine to choose the target variable. The service routine makes a random choice from all fixed-binary variables so far declared in the test case being created. It returns the text of the target variable (FIX10(3)) and a pointer to the position in the dictionary where the data of the variable are held.
4. The "assignment" subroutine now calls the binary expression subroutine to create the text of the right-hand side of the assignment and to predict its value at execution time.
5. The expression subroutine, which is coded recursively so that expressions of any complexity may be produced, constructs the source of a binary expression (FIX11*2 – FIX12(1,3)), using only constants or variables that are known to be initialized at this moment in the test case. It returns both the text and the predicted value of the expression, say, 49.
6. The assignment subroutine concatenates the text of the target variable, an equal sign, the text of the source expression, a semicolon, and finally, a debugging comment that gives the predicted value. It then calls a further service routine to add this statement to the test case under construction.
7. It stores the predicted value (49) into the dictionary entry for the target variable. Any further

statements that refer to the variable will use this latest value in their predictions.

8. Finally, the assignment subroutine passes control back to the main loop, ready for the selection of the next statement.

Note that the expression generator rejects any expressions that would cause an interrupt, such as zero-divide, at execution time. It also ensures that uninitialized variables are not used. In this way executability is ensured.

At the end of each section of random statements, some checking code is added to the test case. This code checks that the value of each modified variable

The execution of some types of random statement may affect the subsequent flow of execution in the test case.

is in accordance with the prediction made at test case generation time. The above assignment statement, for example, would trigger a self-checking statement of this type:

```
IF FIX10(3) ¬= 49 THEN CALL ERROR(126);
```

Every time a call to the error subroutine is added to the test case, the argument to be passed is incremented by one. So if the diagnostic "ERROR AT 126" occurs at run-time, it will be because the variable FIX10(3) did not have the predicted value of 49.

Thus, the assignment subroutine has fulfilled its three objectives. It has created the text of the assignment statement. It has predicted its eventual execution. It has also ensured that the correct execution of the statement will be verified.

Generation of statements that control program flow. The execution of some types of random statement may affect the subsequent flow of execution in the test case. In general, further random statements will be placed on the correct path of flow; a call to the error subroutine may be placed on the incorrect path.

Consider this example of a typical generated IF statement:

```
IF FIX22 > FIX11 • 9 • FIX03(7)
  THEN
    CALL ERROR(218);
  ELSE
    FIX12 = 5;
```

The steps involved in its creation are similar to those mentioned in the previous section. A relational expression is created, and its value (true or false) is predicted. Again we see the three elements—text, prediction of execution, and self-checking.

Calls and procedures. A problem that has not yet been mentioned is that of generating code that may be executed more than once at run-time. The generator does not incorporate a post-processing interpreter and must therefore be able to predict the execution of each statement at the time of its generation. Steps have to be taken to ensure that statements executed more than once (such as those inside do-loops) execute identically on each occasion. The same is true for the statements inside procedures that are called from more than one place in the test case.

For example, if the generated statement FIX01 = FIX02 + 2; is to be executed more than once, the value of FIX02 must be the same each time (otherwise the execution of subsequent statements using FIX01 would be unpredictable). The generator therefore makes a restriction on internal procedures. If a procedure is to be called from more than one place in the test case, its statements must refer only to variables declared inside the procedure.

For procedures called from only one place in the test case, the prediction of the statements immediately following a CALL statement may depend on the values of the variables on exit from the called procedure. Therefore, the called procedure must be generated immediately following the generation of the CALL. Since in general the position of an internal procedure would not be that immediately following the CALL, a text-moving mechanism is required. This mechanism is described later.

Do-loops. As mentioned previously, each random statement within an iterative do-loop must execute identically each time. To achieve such execution, the generator divides all variables declared in the

test case into two categories, those of "left bias" and those of "right bias." This categorization is randomly decided for each variable and array element at the time of its declaration. Within an iterative do-loop the following restrictions apply:

- Only right-biased variables may appear on the right-hand side of an assignment.
- Only left-biased variables may have their values altered.

These restrictions ensure that each statement will always execute identically. Apart from these restrictions, completely general statements are produced inside do-loops. Nesting of such loops is permitted to an implementation-defined level (seven, at present).

The correct execution of loops is verified by including self-checking statements inside the loops. These statements increment a variable that will later be checked. Other special-case statements may be inserted in the loop, such as an assignment to an array element whose index is a simple function of the loop variable. Again, the array would later be checked by the test case to verify correct execution.

Very little is gained by permitting large numbers of iterations in a test case. The number of iterations is therefore normally kept low. Keeping the number low does not represent any restriction on the syntactic form of the DO statement itself. The do-loop parameters will be created by calls to the appropriate expression generator. The expressions chosen will be adjusted by a constant, so that the number of iterations will be set as required.

The following is a typical generated DO group:

```
DO PTR01->FIX02 = -45 TO (483-DIM(BIT04,4)*12)-520 BY (FIX21-5);
  /* START = -45, INCREMENT = -3, END = -51, COUNT = 3 */
  ... various random statements ...
  FIX07 = FIX07 + 1;  /* INCREMENT LOOP CHECK VARIABLE */
END;
```

The predicted value for FIX07 at the end of the loop will be checked later, thereby verifying the correct execution of the DO statement.

Similar methods are used for do-loops with WHILE or UNTIL clauses.

The generator output routine. In several situations it would be inconvenient to place generated statements straight into the test case under construction. For example, it was mentioned that the text of an internal procedure must be generated immediately after the CALL to it (since the prediction of the statements following the CALL might be affected by variables that had changed value inside the called procedure). But in some cases the procedure cannot physically follow the CALL. For example, the CALL might be on the first branch of an IF statement, with an ELSE clause to follow.

In several situations it would be inconvenient to place generated statements straight into the test case under construction.

Similarly, the need for a declaration (typically for a self-checking variable) might become apparent while the generator is in the depths of constructing some complex statement.

To allow for subsequent reordering of the text, each statement is stored initially into an array. Associated with each element of the array is a flag, specifying what post-positioning, if any, is required. When all the statements have been generated, a post-processing subroutine reorders the text and writes it out to disk.

The same scheme is used to bring all the declarations in a block to the start of the block. This procedure aids debugging and improves the physical appearance of the test case. Another subroutine formats the test case, indenting nested DO, IF, and SELECT statements.

The generator weighting scheme. The method of weighting used is common to all our test case generators. Wherever the generator must choose between several alternatives, each of the alternatives has a numerical weight (or biasing factor) associated with it. These weights are declared as variables within the generator, and their values affect the relative frequencies with which various language features appear in the test cases.

For example, suppose that the generator has to choose between three possible options for a Record I/O READ statement. These options may have associated weights of 3, 1, and 2. A generator service routine is passed the three weights and chooses one

The generator has also successfully tackled complex language areas.

of the options by calling a random-number generator to select a number between 1 and 6 (= 3 + 1 + 2). If 1, 2, or 3 is chosen, option 1 will appear in the test case. If 4 is chosen, option 2 will appear; and if 5 or 6 is chosen, option 3 will appear. So, the relative frequency of option 1 being selected against option 2 is 3:1, in accordance with the weights.

Whenever an option is just two-way (should the character declaration incorporate the VARYING option, for example), it is simpler to use a single percentage weight. If this is set to 20, about 20 percent of the character strings in the test case will have the VARYING attribute.

It is for the implementer of the test case generator to decide which random decisions should have a weight accessible to the generator user, and which should be hard-wired as constants.

In addition to the weights, various options may be set. These options include the proposed length of the test case, the maximum permitted nesting of do-loops, and the complexity of expressions required. Again, these options all have default values. They may be overridden at generation time if required.

Extent of language covered by the PL/I generator. Self-checking test case generation has been extended to cover most of the data types, storage classes, statements, and built-in functions in the PL/I language. It is not the intention of this paper to give details of the implementation of all parts of the generator. The methods used to test compiler support of VSAM record I/O can be hinted at by giving some typical generated statements:

```
ON KEY(VSAMKS) GOTO VSM017;
WRITE FILE(VSAMKS) KEYFROM('APPLE') FROM(GREC(3));
CALL ERROR(93);       /* WRITE OUT OF SEQUENCE RAISES KEY CONDITION */
VSM017: REVERT KEY(VSAMKS);
```

The generator adds statements to the test case that will send data back and forth between the data set and the test case. Throughout the generation process the generator maintains a prediction of the current contents of the VSAM data set. On this occasion, the generator predicts that the WRITE statement will raise the KEY error condition. It therefore inserts extra code to trap the error. Should the error on-unit not be entered, diagnostic 93 will inform the test case runner of the error.

Wherever possible, the generator adds helpful comments to the generated statements to aid debugging:

```
READ FILE(PATH1) INTO (GREC(1)); /* NEXT KEY='ROSE', MAIN KEY='LEMON'*/
STR02.FIX08(2)=534 + FIX05*27 - LENGTH(BIT07);        /* DEBUG: 612 */
```

In the first statement, the predicted VSAM keys of a sequential read are given. In the second, the predicted assignment value is shown.

The generator has also successfully tackled complex language areas such as the REFER structure attribute. Even if adding generator support for a new language area seems likely to be a complex venture, it is usually the case that conventional testing of the new area would be even more difficult. And once a language feature is added to the generator, there is the huge bonus that it will automatically appear in conjunction with all the other supported language features. Problems due to interactions may be detected.

Types of testing not handled by the generator method. Since the test cases produced by the generator are syntactically correct, they are of little use in testing compile-time diagnosis of faulty input. Test cases for this area must therefore be handwritten. Similarly, run-time diagnostics testing is usually performed with handwritten test cases, partly because there is little virtue in repeating such tests, partly because in general they cannot be made self-checking.

It is inconvenient for the generator to produce some special cases. For example, it might be worthwhile to test whether or not a loop would iterate successfully 10 000 times. It would not be sensible to introduce such loops in the middle of randomly generated test cases, though, since they might then

take hours to execute. A handwritten test case would be used instead.

Finally, some system-related functions are difficult to support, including tasking, reentrancy, fetching, and the linking of external procedures. It will often prove convenient, though, to use generated test cases as the basis for testing these functions.

Summary of PL/I generator. The Optimizing and Checkout Compilers had been in the field for several years before test case generation was used with them. The full benefit of the method was therefore not felt. Nevertheless, generation has been used to locate several bugs in the existing code and to successfully test features added in subsequent releases such as VSAM support, the SELECT statement, DO UNTIL, DO REPEAT, and LEAVE.

Compiler testing is one of the most suitable areas for test case generation, since it is relatively easy to achieve the goal of self-checking. The need for testing by generator is also high, since large amounts of test material are required that would be expensive and time-consuming to produce by hand.

Alphanumerics and graphics test case generators

The Graphical Data Display Manager (GDDM) is an IBM program product that supports the display of text and graphics on terminals and printers. The first three releases of this product have been successfully tested by test case generators.

Three main types of processing are provided by the GDDM package: alphanumeric, general graphics, and business graphics. Each has its own distinct application program interface (API), and each was tested with a separate test case generator.

Alphanumeric processing consists of both output and input. The generated test cases involved were therefore able to incorporate a high degree of self-checking, using a scheme of data echoing that will be described later.

Graphics processing is output only. The end product—a picture on a screen or printer page—can be checked only by the human eye. However, various aids were devised to assist the checking of output from generated test cases. Despite initial doubts from some quarters as to whether test case generation would be effective in this area, the method proved extremely successful. Many hundreds of valid problems were located, and subsequent defect rates have been low.

It was also found that vast quantities of test material were required to locate the bugs. For a given release, testing would start as soon as any new function became available. Using the previously described weighting scheme, test cases would be produced containing all the GDDM function of the previous release plus the new function just added to the code by development. As development proceeded, more functions would be switched on in the generator. Eventually it would produce test cases covering the whole of the function for the new release. Under this method of testing, bugs were found at a uniform rate for some five or six months.

To create sufficient test material manually would have been impractically time-consuming and expensive. Using test case generators, just one programmer performed the functional verification test of some 60 000 lines of code.

The GDDM alphanumeric test case generator. The alphanumeric function to be tested is basically the definition of the alphanumeric fields that make up a screen layout and the transmission of data to and from these fields.

Since it is not possible to make "output" test cases self-checking, the output produced may need to be compared against the generated test case source. During early testing of new code, it may be necessary to check, for example, that field 22 is yellow and reverse-video and that it does start in column 13 and end in column 40. To ease this comparison, the generator predicts the appearance of each screenful of output and adds this prediction to the source of the test case (see example that relates to Figure 1). The generated test case also floods the entire screen with color before executing a block of alphanumeric statements. Subsequent declarations of alphanumeric fields cause holes to appear in this background, enabling the test case operator to "see" unassigned fields that would otherwise be invisible.

The transmission of alphanumeric data is self-checked, using the following echoing scheme. Output of I/O alphanumeric fields consists of a number of identical characters with one or more question marks included, for example QQQQQ?QQQ. The test case operator is expected to overtype the question marks with the other transmitted character (Q in this case). When a subsequent GET is performed on

the field, the generated test case will check that the ? has been correctedly modified to a Q.

This combination of "echoing" the sent data and requesting a field modification by the operator acts as an indirect check that the output was performed correctly. Although the fields are declared in random positions and are of random sizes, the operator knows that he must merely overtype the ? characters and point the light pen at those fields containing special light-pen prompts. The test case can therefore be executed without any knowledge of the test case source.

In the early days of testing new function, a few detailed comparisons might be made between the screen output and the corresponding test case source. Later on, many more runs would be made without any reference to the source unless an apparent error occurred.

As in the PL/I generator, there are separate subroutines for each of the alphanumeric API calls (i.e., each of the statements in the GDDM language). As before, each subroutine is responsible for creating the text of the statement, predicting its execution, and (as far as is possible) assisting in its checking.

Example of an alphanumeric statement subroutine. A description of the steps involved in generating the GDDM "ASDFLD" (field-declare) statement is now given to indicate to the reader how such a generator should be coded. Note that wherever the word "choose" is used, a call to the random-number generator will occur.

1. Choose a valid field number.
2. Erase the generator's internal memory of this field number if it was already in use. The previously occupied terminal screen positions will then be available for subsequently declared fields.
3. Choose a row and column position for the field within the current page dimensions.
4. Choose a depth and width for the field that does not carry the field outside the current page.
5. Call a generator subroutine to check that the proposed field would not illegally overlap any existing fields. If it would, return to 3. (Five attempts are made before generation of the statement is abandoned.)
6. Store information on field placement and field attributes into the dictionary of the generator for the current page.
7. Produce text of the statement, e.g.,

```
/*     FIELD-NUMBER  ROW   COLUMN   DEPTH   WIDTH   FIELD-TYPE   */
CALL ASDFLD(13,       18,    40,      6,      2,        1);
```

and add it to the test case being generated.

The subroutine has created the text of the ASDFLD statement and predicted its execution. It has also ensured that such execution would not cause an error (such as overlapping another field). Verification of execution will occur either via a visual check by the test case operator or indirectly if the field takes part in subsequent I/O operations.

An example of generated output to test alphanumeric support. Part of a test case produced by the GDDM alphanumeric test case generator follows as an example of generated output.

```
/*******************/
/* CREATE PAGE 10 */
/*******************/
CALL FSPCRT(10,27,67,3);                                  /* DEFINE PAGE */
CALL GSFLD(1,1,26,67);                              /* DEFINE GRAPHICS FIELD */
CALL FRAME;                          /*FILL IN BACKGROUND TO AID CHECKING*/

/*****************************************/
/* INITIAL FIELD DECLARATIONS FOR BLOCK */
/*****************************************/
CALL ASDFLD( 75,16,22, 7, 9,4);
CALL ASDFLD( 52, 7, 4, 1,80,0);                        /* WRAP-AROUND FIELD */
CALL ASDFLD( 97,17,38, 2,11,1);
/*************************************/
/* END OF INITIAL FIELD DECLARATIONS */
/*************************************/

CALL ASCPUT( 97,22,( 6)'2'||'?'||(15)'2');                  /* NUMERIC I/O */
CALL ASCPUT( 75,61,(61)'!');                          /* DEFERRED LP FIELD */
CALL ASCCOL( 75,14,' 53661 257356');  /* CHARACTER COLOR ATTRIBUTES */
CALL ASCSS( 97, 4,' A A');                 /* CHARACTER S-SET ATTRIBUTES */
CALL ASCPUT( 52, 4,( 1)'('||'?'||( 2)'(');              /* ALPHANUMERIC I/O */
CALL ASFCUR( 97, 1, 1);                               /* SET CURSOR POSITION */
CALL ASQCOL( 75,14,FIELD_CHECK);                 /* QUERY COLOR ATTRIBUTES */
IF SUBSTR(FIELD_CHECK,1,14)¬=' 53661 257356' THEN CALL ERROR( 42);
CALL ASDFLD( 75,10, 4, 7,13,4);                  /* REPLACE PREVIOUS FIELD */

      etc.

CALL ASCPUT( 29,63,(48)'8'||'?'||(14)'8');                  /* NUMERIC I/O */
CALL ASCPUT( 95, 6,'?'||( 5)'0');                           /* NUMERIC I/O */
 PAGE;

/*************************************************************/
/*                                                           */
/*              SCREEN    PREDICTION                         */
/*                                                           */
/*************************************************************/
/*
|.....8888888888888888888888888888888888|||
|88888888888888888?88888888888888+++++++++++++++++++++++++++++.....||||
|.....8.....||||
|.....8.....||||
|..... &++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++||||
|++++++.....8.....||||
|.....8.....||||
|.....8.....||||
|...?!!!!!!!!!!!!...8.....||||
|...?!!!!!!!!!!!!...8.....||||
|...?!!!!!!!!!!!!...8.....||||
|...?!!!!!!!!!!!!...8.....||||
|...?!!!+++++++++...8.....||||
|...?++++++++++++...8.....||||
|...?++++++++++++.....K.....||||
|.....K..22222?2222.....||||
|.....K..22222222222.....||||
|.....K.....||||
|.....K.....||||
|.....K.....||||
|..+.....++++++++++++.....||||
```

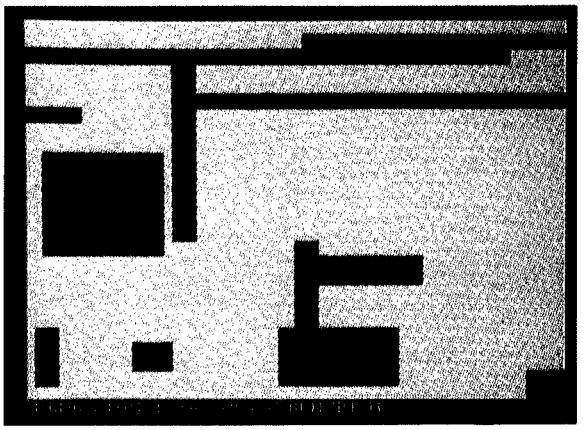
```
|..+...........?OO..............+++++++++++++.....................||||
|..+...........OOO..............+++++++++++++.....................||||
|..+.............................+++++++++++++.................1?++||||
|.....................................................................++++||||
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
*/

CALL ASDFLD(99,27,2,1,57,2);                    /* FIELD FOR LOGO */
CALL ASCPUT(99,44,'OUTPUT 11 FROM TESTCASE G175343 TO DEVICE 17');
CALL ASREAD(TYPE,MOD,COUNT);
/*****************************************************/
/* QUERY MODIFIED FIELDS AND CHECK RETURNED VALUES */
/*****************************************************/
CALL ASQMOD( 5,F_ID,TLENG,ILENG);
/*************************************************************/
/* GDDM-REL3 RETURNS FIELDS IN F_ID ORDER, LP FIELDS FIRST */
/*************************************************************/
IF F_ID( 1)¬= 75 THEN CALL ERROR( 47);          /*LIGHTPEN FIELD  1*/
IF TLENG( 1)¬=    91 THEN CALL ERROR( 48);
IF ILENG( 1)¬=    56 THEN CALL ERROR( 49);
IF F_ID( 2)¬= 29 THEN CALL ERROR( 50);           /*ALPHA FIELD  2*/
IF TLENG( 2)¬=    91 THEN CALL ERROR( 51);
IF ILENG( 2)¬=    63 THEN CALL ERROR( 52);
IF F_ID( 3)¬= 52 THEN CALL ERROR( 53);           /*ALPHA FIELD  3*/
IF TLENG( 3)¬=     8 THEN CALL ERROR( 54);
IF ILENG( 3)¬=     2 THEN CALL ERROR( 55);
IF F_ID( 4)¬= 95 THEN CALL ERROR( 56);           /*ALPHA FIELD  4*/
IF TLENG( 4)¬=     6 THEN CALL ERROR( 57);
IF ILENG( 4)¬=     6 THEN CALL ERROR( 58);
IF F_ID( 5)¬= 97 THEN CALL ERROR( 59);           /*ALPHA FIELD  5*/
IF TLENG( 5)¬=    22 THEN CALL ERROR( 60);
IF ILENG( 5)¬=    22 THEN CALL ERROR( 61);

     etc. (further checking code will follow)
```

The output from this section of generated code is shown in Figure 1.

The GDDM graphics test case generator. The main problem to be overcome with graphics testing is verification of the random output on the color terminal. Some bugs appear as program checks, loops, or incorrect diagnostics, but many appear as faulty output at the terminal. It would be immensely tedious and time-consuming to have to compare each line or curve on the screen with the statement in the test case that produced it, in the hope of finding a mismatch and thus a bug. The following five techniques have therefore been used to enable easy verification of the output from the random test cases.

Sending markers to the predicted primitive end points. Every time the generator creates a primitive statement (for example, a line or an arc), it calculates the predicted end points of the primitive. Before adding a WRITE statement to the test case, the generator creates a loop of MARKER primitives to send colored markers to each of the predicted primitive end points. A complex marker consisting of a yellow diamond and a black plus sign is used so that it will show distinctly against any shaded background.

Figure 1 Output from the section of generated alphanumerics test case; the operator will overtype the ? character, causing those fields to be echoed back to the test case for self-checking



The eventual output from the test case will therefore be a merging of the random primitives and the markers that predict their end points. When the test case is executed, it will not be necessary to verify the position of each primitive against the text of the random statement that produced it. The tester need only confirm that

1. Every primitive has a special marker at each end.
2. There are no unattached special markers on the screen.

This method allows the correctness of the output to be verified at a glance.

Interspersing query-current-position statements. After creating a primitive, the generator may decide to add a GDDM GSQCP (query-current-position) statement followed by a checking statement. This check will confirm that GDDM's calculation of the current position is the same as that previously predicted by the test case generator. It does not guarantee that the primitives will be drawn to the correct position, since the drawing has not yet taken place.

Backing grid as aid to detailed checking. When it is desired to perform a detailed check on the output produced by GDDM, the user may tell the generator to use "grid" mode. The created test case will then use a simple window (coordinate system) of $X = 0$

$\rightarrow$ 100 and $Y = 0 \rightarrow 100$, rather than a randomly chosen one, and a backing grid will be sent to the terminal by the test case to enable easy verification of the position of the primitives.

Restricting all primitives to the viewport. When generating a test case, the user may specify whether or not all primitives should be restricted to the viewport. It is usual to run the generator in "restricted" mode, and several GDDM drawing errors were detected by observing that primitives had strayed outside the viewport.

The method used by the generator to ensure that arcs remain inside the viewport is to simulate drawing the arc by calculating the current position every few degrees of the proposed sweep. If the current position moves outside the viewport, the sweep is backtracked and then used to replace the previously chosen sweep. With MOVEs and LINEs, all that is required is to select the TO-point inside the viewport.

The generator may be run in "unrestricted" mode to test whatever "clipping" functions may be supported by the product under test.

Automatic commenting of test cases. As with our other test case generators, helpful comments are added to the source of the test case. Here the purpose of the added comments is to aid a detailed verification of the graphics output. For example, an ARC primitive might appear as

```
CALL GSARC(  36,  59,-56.4);      /*ARC-END AT +53.12E+00, +9.92E+00 */
```

and if a detailed check were being performed, the tester could check whether the arc did indeed end at the predicted (53.12, 9.92). Another example is

```
CALL GSAREA(1);                   /*TURQUOISE AREA,SHADING PATTERN=11*
```

The generator keeps track of which attributes the test case has set at each point and can therefore save the tester the trouble of scanning back to find the last setting of an attribute.

The following excerpt from a generated graphics test case produces the output shown in Figure 2.

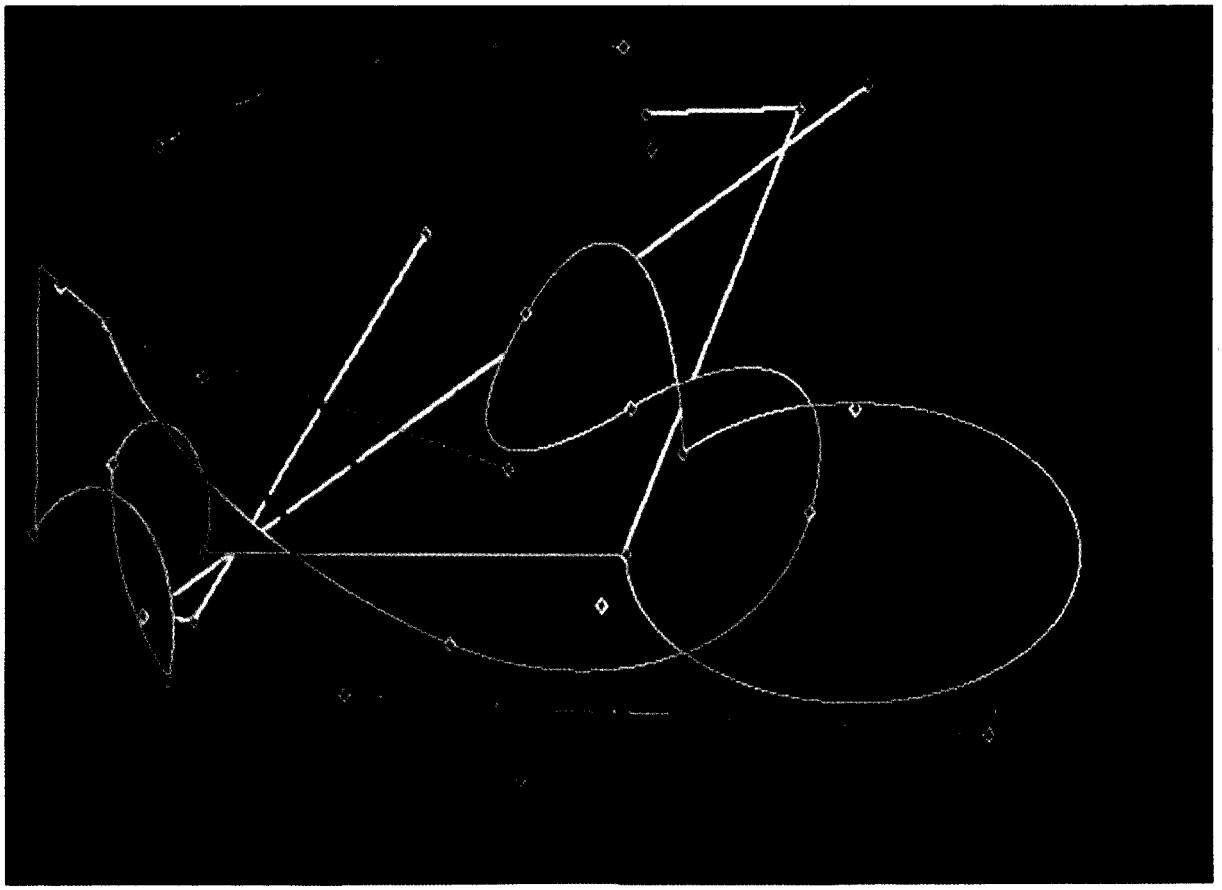
```
CALL DSUSE(1,11);                              /*USE PRIMARY DEVICE 11 */
CALL SS_INIT_LOAD;                                /* LOAD SYMBOL SETS */
CALL FSPCRT(29,0,0,2);                               /* DEFINE PAGE */
CALL GSFLD( 1, 1,31,52);                      /* DEFINE GRAPHICS FIELD */
CALL GSCLP(1);                                     /* SET CLIPPING ON */
CALL GSPS( 1.00000E+00, 6.42999E-01);         /* DEFINE PICTURE SPACE */
/* NO GSVIEW STATEMENT,WHOLE PICTURE SPACE WILL BE USED */
/* NO GSWIN STATEMENT, (0:100,0:100) IS DEFAULT */
```

```
CALL GSSEG(0);                                        /*  UNNAMED SEGMENT*
CALL GRID;                  /* SUBROUTINE CALL TO SEND GRID TO SCREEN */
CALL GSMOVE(    15,    54);              /* FROM(      0.0,      0.0) *
 /******************/
 /* BLOCK       1 */
 /******************/
CALL GSCM(1);                                 /* CHARACTER MODE = HARDWARE */
CALL GSCOL(2);                                          /* NEW COLOR IS RED */
CALL GSARC(    45,    96,- 19);      /*ARC-END AT +29.6E-01,+66.0E+00*/
CALL GSMOVE(    71,    50);              /* FROM(      2.9,      66.0) *
CALL GSARC(    38,    18,- 58);      /*ARC-END AT +82.6E+00,+69.7E-01*/
CALL GSQLW(BIN_AR(1));                                  /* QUERY LINEWIDTH */
IF BIN_AR(1)¬=  1 THEN CALL ERROR(  2);              /* WRONG LINEWIDTH */


CALL GSLINE(    27,    12);              /* FROM(     82.6,      6.9) */
CALL GSMOVE(    51,    97);              /* FROM(     27.0,     12.0) */
CALL GSARC(    44,    51,  53);      /*ARC-END AT +11.4E+00,+84.2E+00*/
CALL GSCOL(5);                                /* NEW COLOR IS TURQUOISE */
CALL GSELPS(-15.0E+00,+12.2E+01,-273, 5.33881E+01, 8.37031E+01);
/*CENTRE AT:    68,    68.SWEEP= 4.36938E+02 TO  4.05938E+02 */

    etc.

CALL GSPAT(78);                          /* SET PATTERN FROM 64-COLOR SET */
CALL GSCOL(7);                                        /*WHITE FOR 64 COLORS*/
 /*****************/
 /* AREA STARTING */
 /*****************/
CALL GSAREA(1);
   /*SHADING IS FROM 64 COLOUR SET,PATTERN IS NO.  78,DRAWN BOUNDARY*/
DCL PFLT_X03( 6) FLOAT DEC(6) INIT(    11,    61,    73,    30,    55,
   56);
DCL PFLT_Y03( 6) FLOAT DEC(6) INIT(    35,     3,    70,    30,    95,
   44);
CALL GSPFLT(6,PFLT_X03,PFLT_Y03);        /* FROM(      7.0,     61.0) */
CALL GSARC(    71,    31,-319);      /*ARC-END AT +51.1E+00,+30.9E+00*/

   etc.
 /*********************************************/
 /*  SEND OUT MARKERS AT PREDICTED ENDPOINTS  */
 /*********************************************/
DCL XY01( 60) FLOAT DEC(6) INIT(
      15.0,     54.0,      2.9,     66.0,     71.0,     50.0,
      82.6,      6.9,     27.0,     12.0,     51.0,     97.0,
      11.4,     84.2,     53.3,     83.7,     52.9,     88.4,
      66.0,     89.0,     49.0,     24.0,     34.0,     73.0,
      14.0,     22.0,      9.8,     22.7,     72.0,     92.0,
      13.0,     85.0,     42.0,      1.0,      7.0,     61.0,
      36.0,     19.0,     67.0,     36.5,     51.5,     50.0,
      42.5,     62.5,     56.0,     44.0,     51.1,     30.9,
      14.9,     31.0,      7.3,     42.6,     11.9,     14.1,
       0.5,     33.6,      1.0,     69.0,     41.0,     42.0);
CALL GSCOL(6);                                  /* MARKERS IN YELLOW */
DO I=3,2;                        /* YELLOW DIAMONDS ROUND BLACK PLUS-SIGNS */
CALL GSMS(I);                          /* MARKER FOR PREDICTED END-POINTS */
DO J=1 TO 30;                           /* LOOP THROUGH SAVED POSITIONS */
CALL GSMARK(XY01(2*J-1),XY01(2*J));
END;                                                          /* J-LOOP */
CALL GSCOL(8);                                /* SET COLOR TO BLACK */
END;                                                          /* I-LOOP */


CALL ASDFLD(99,32,2,1,44,2);        /* DEFINE ALPHA FIELD FOR PROMPT */
CALL ASCPUT(99,44,'CLIPPED 1 FROM TESTCASE G164636 TO DEVICE 11');
/*****************************************/
/*   GDDM GRAPHICS WILL NOW BE OUTPUT    */
/*****************************************/
CALL ASREAD(TYPE,MOD,COUNT);
```

GDDM business graphics generator. Business graphics test cases are, by their nature, output-only. Since it is not possible to make "output" test cases self-checking, the output produced must in theory be compared to the matching test case source. Performing a detailed comparison, particularly that of plots against data, is necessarily a time-consuming process.

With an automatically generated test case, this comparison is neither easier nor more difficult than with a handwritten test case. The test case generator offers no benefits in this area. Its main advantage is that the source of, say, a 1000-line test case can be automatically generated in seconds rather than needing a few days to be coded manually.

Figure 2 Output from randomly generated graphics test case; the diamond-shaped markers indicate the predicted end point of each random line or curve



The experience gained from the first three releases of GDDM business graphics showed that most bugs display themselves not by a histogram bar being a quarter of an inch too low, but in some more immediately obvious fashion. For example, defects might be indicated by program checks, erroneous error messages, bars stretching off the top of the chart, legends not matching components, no data appearing, or any obvious shading errors. Although some generated test case runs were made that involved a detailed comparison of the source with the output, many more runs, indeed nearly all runs, were made without any reference to the test case source unless the output looked suspicious.

A typical test case might be 1200 lines long and put out 15 charts. There might be, say, four suspicious aspects to the output, and the relating test case source would be inspected. On investigation, some strange combination of options and data might justify some of the suspected output, and maybe one or two valid problems would be found.

Brief description of business graphics generator internals. Each time in going around the main loop, the generator selects a chart type and then calls the high-level procedure responsible for producing the statements that cause such a chart to be built. This procedure will first call several subroutines, each responsible for one or more chart layout statements. These subroutines decide randomly whether or not to produce each statement, according to the associated weight of the statement. Should a statement be selected, any options or operands are also chosen by reference to weights and the random-number generator.

Figure 3 Output from a randomly generated business graphics test case

Next, the procedure chooses the basic data specification. For example, the number of bars, number of components, and data ranges may have to be selected.

Next, several subroutines are called that are responsible for axis-preparation statements. The statements already generated may now be jumbled into a random order. The penultimate step is to generate the declarations to hold the actual chart data; various techniques are used here. For example, surface chart data may be generated using the formula

$$Y_{nc} = Y_{\text{low}} + K_c * (X_n - X_{\text{low}}) ** E_c$$

where

$Y_{nc} = n$th dependent data value in cth component
$X_n = n$th independent data value
K_c and E_c = constants chosen so the dependent data will be inside the prescribed range

Finally, the chart-plotting statement itself is produced. As with the generators previously described, the test cases are guaranteed to execute without error. For example, the generator will ensure that the total data for a pie chart do not exceed 100 percent and that negative values are not used on a logarithmic axis.

The following excerpt from a generated business graphics test case produces the output shown in Figure 3.

```
/*********************************************************/
/*                                                       */
/*       GDDM HISTOGRAM WILL NOW BE PRODUCED             */
/*                                                       */
/*********************************************************/
DCL TEXT_ATT001(2) INIT(2/*COLOR*/,0/*CHAR MODE*/);
DCL PAT002(8) INIT(2,3,6,8,3,1,1,8);              /* DATA FOR CHPAT */
DCL AX_ATT003( 3) INIT(3/*COLR*/,3/*LT*/,1/*LW*/);     /* AXIS ATTRS */
DCL KEY004 CHAR(32) INIT('FRANCE  BELGIUM SPAIN   ITALY   ');
                                        /*DATA FOR CHKEY STATEMENT*/


   /***************************************/
   /* CHART LAYOUT                        */
   /***************************************/
CALL CHHATT(2,TEXT_ATT001);          /* MODIFY HEADER TEXT ATTRIBUTES */
CALL CHPAT(8,PAT002);                  /*RESPECIFY PATTERN TABLE*/
CALL CHHEAD(43,'*HISTOGRAM FROM GENERATED TESTCASE G145501*');
CALL CHHMAR(7,3);                     /*DEFINE HORIZONTAL MARGINS */


   /***************************************/
   /* AXIS PREPARATION                    */
   /***************************************/
CALL CHSET('ABREV');                    /* 3-LETTER ABBREVIATIONS */
CALL CHXDAY(6);                         /* USE DAY LABELS */
/* NO 'CHXRNG' STATEMENT , X-AXIS WILL USE AUTOSCALE */
CALL CHXTIC(     11,4);                   /* X-AXIS SCALE MARKS */
CALL CHXSCL(1.0E-01);                   /* X-AXIS SCALING FACTOR */
CALL CHXSET('XTICK');                   /* CROSS TICKS ON X-AXIS */
CALL CHXSET('LABMIDD');
CALL CHXSET('NOFO');
/* NO 'CHYRNG' STATEMENT , Y-AXIS WILL USE AUTOSCALE */
CALL CHYTTL(30,'*  PRIMARY Y-AXIS TITLE      *');    /* Y-AXIS TITLE */
CALL CHYTIC(2000000,2);                   /* Y-AXIS SCALE MARKS */
CALL CHYSET('ATCENTRE');              /*VERTICAL AXIS TITLE POSITION*/
CALL CHYSET('LABADJACENT');
CALL CHDATT( 3,AX_ATT003);              /* MODIFY DATUM ATTRIBUTES */
CALL CHYDTM(8569067);                     /*Y DATUM LINE,STATE 1*/


   /***************************************/
   /* MISCELLANEOUS                       */
   /***************************************/
CALL CHKEY(4,8,KEY004);                      /*DEFINE CHART KEYS*/
CALL CHSET('KBOX');                        /*LEGEND WILL BE BOXED*/
CALL CHKEYP('H','B','C');                 /*DEFINE LEGEND POSITION*/


   /***************************************/
   /* DATA PREPARATION                    */
   /***************************************/
DCL XLO005(10) INIT(     65,     69,     76,     79,     86,     93,
  98,    106,    116,    119);/*RANGE-STARTS*/
DCL XHI005(10) INIT(     69,     76,     79,     86,     93,     98,
 106,    116,    119,    133);/*RANGE-ENDS*/
/* SPECIAL CASE , TOUCHING BARS */


DCL DAT006(40) FLOAT DEC(6) INIT( /*ABSOLUTE DATA*/
/*COMP 1:*/   2055461,   2615040,   3574260,   3733298,   4355491,
   5041735,   5747890,   6552162,   7411141,   8156578,
/*COMP 2:*/   1705850,   1748224,   1938725,   1986289,   2219337,
   2567247,   3030064,   3692706,   4566554,   5468934,
/*COMP 3:*/   1701621,   1719223,   1825382,   1855148,   2011877,
   2268949,   2639343,   3208919,   4011031,   4885378,
/*COMP 4:*/   1700449,   1707663,   1765853,   1784078,   1887291,
   2073248,   2363399,   2842584,   3563192,   4392173);
   /***************************************/
   /* HISTOGRAM PLOT                      */
   /***************************************/
CALL CHHIST(4,10,XLO005,XHI005,DAT006);              /*PLOT HISTOGRAM*/
CALL CHYDTM(1064425);                      /*Y DATUM LINE,STATE 2*/
CALL ASREAD(TYPE,MOD,COUNT);             /*CAUSES OUTPUT TRANSMISSION*/
```

Sort/merge test case generators

The sort/merge test case generator is, essentially, two separate generators. The first is a syntax generator which produces the sort/merge control statements or definition. The second is a sort/merge data generator which produces the input and expected output files for the sort/merge program under test. The sort/merge program is executed using the control statements from the syntax generator and

the input files from the data generator. The resultant output is then compared against the predicted output file from the data generator, thereby providing the self-checking mechanism. This sequence is shown in Figure 4.

The technique of generating the sort/merge control statements is similar to those used by the other generators and will not be described here. The syntax generator phase must also set up the data structure that describes the sort/merge application represented by the control statements that are generated.

Sort/merge data generation. To describe the sort/merge data generation techniques, we start with a simple sort data generation example. We then consider how different file structures, key orderings, and record formats are handled.

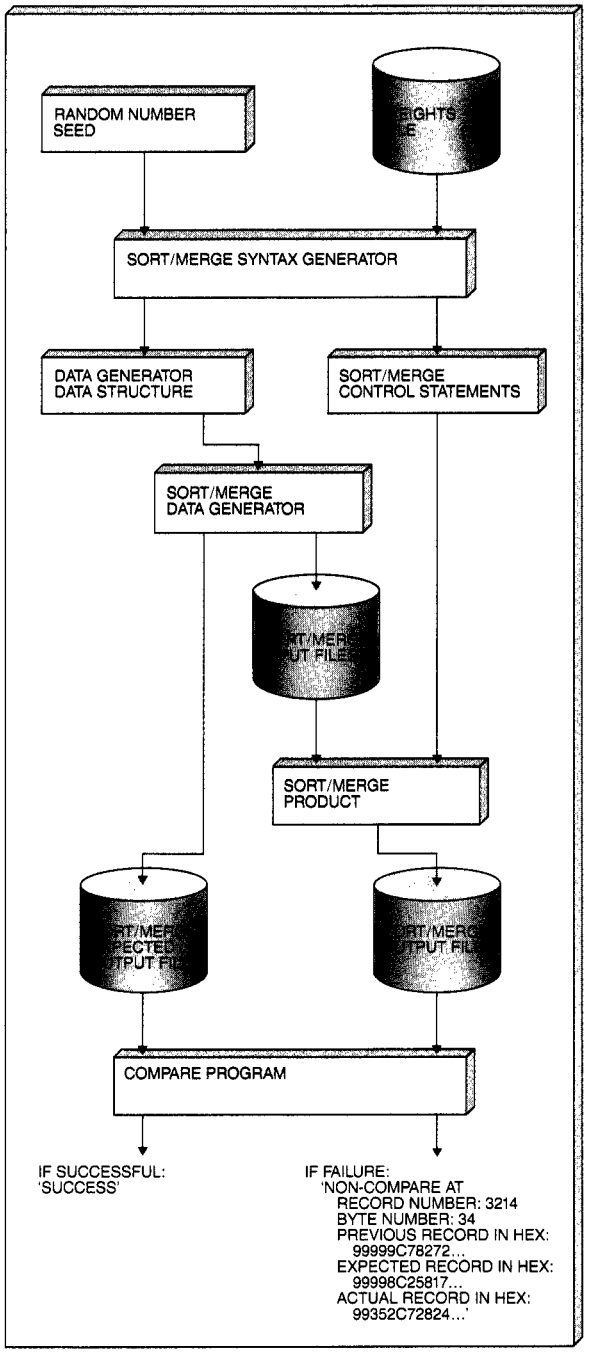
Simple sort. The simple sort data generation example is shown in Figure 5. The sort/merge data generator creates the expected output file for the test case by producing records with the integers in the key field (Field 1) and random numbers in the scrambling field (Field 2). The expected output file produced by the data generator is then sorted on the scrambling field to produce the input file for the sort application to be tested.

The first improvement we can make to this example is to eliminate the need to sort the expected output to obtain the input, as shown in Figure 6. We perform this elimination by writing the expected output file to an intermediate relative (random access) file sequentially first. Then we read the records, which were written sequentially, back in a random order and write them to the sequential input file.

Note that in Figure 6 the records in the input file are in the same order as in Figure 5. We no longer need to sort the expected output file to obtain the input file for the sort application.

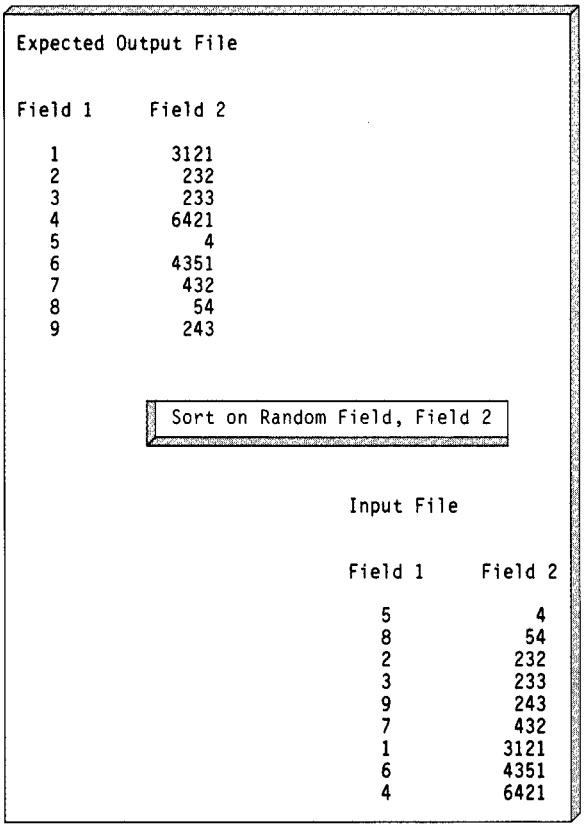
Multiple input files. Another situation that arises in sort applications, and almost always in merge applications, is multiple input files. This situation is shown in Figure 7. For sort applications, multiple input files are supported by reading in the relative file randomly and writing the records to the input files in the order in which they were read from the random file. To perform this operation, the number of records in each of the input files must be known or arbitrarily set. Note that in Figure 7 the records

Figure 4 Sort/merge test case generation environment



are read in randomly in the same order as that shown in Figure 6. When the records are processed by the sort program, they will be read in the same

Figure 5 Simple sort



order as if they had all been in a single input file. This procedure allows the presequencing of the sort application input to be controlled.

For merge applications, the records written to the relative file are read back in the order in which they were written. As these records are read back in, they are written to a randomly chosen input file of the merge application to be tested (Figure 8). Note that for each of the input files, the records in the input file are in order.

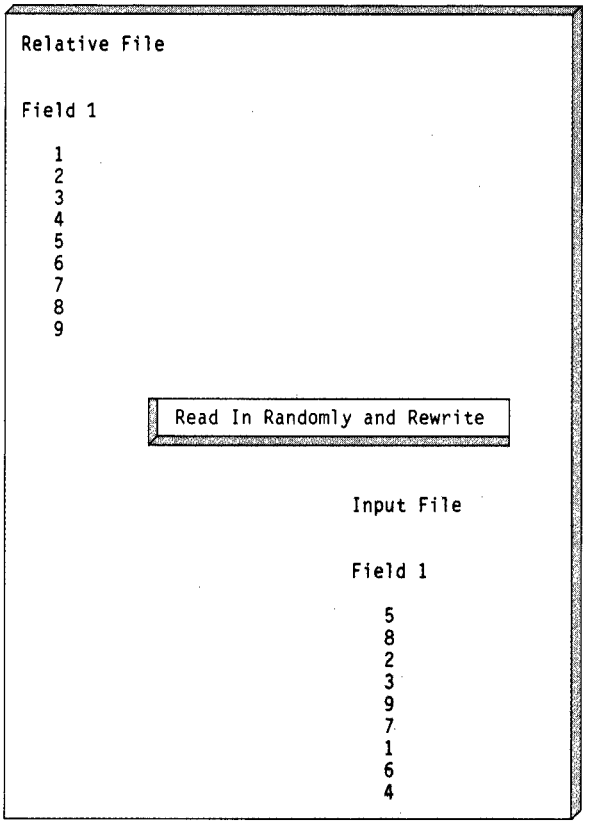
The IBM DPPX Sort/Merge product and IBM DOS Sort/Merge product support subsetting of the input. To accomplish the subsetting, the DPPX product uses the SELECT subcommand and the DOS product uses the INCLUDE and OMIT statements. The function provided is the exclusion of some of the input records from the sort or merge application processing. Thus, some records from the input data sets are not found in the output data set.

The sort/merge data generator supports the testing of this function by adding records, which are not included in the processing, to the input files during the rewriting of the relative file to the actual input files. These records are omitted from the predicted output file, against which the final comparison is made.

Support for ordering keys. Our simple sort example contained only one field of integers to be ordered. The general case is the ordering on fields of many different data types: character or EBCDIC, zoned decimal, packed decimal, fixed point, etc. The ordering key may be formed with any number of these fields in any order. The fields may be of any length and may be ordered either ascending or descending. The fields may be in a user-specified alternate collating sequence and may have differing offsets in different record types, as discussed later.

We will use an example to show how the support for ordering keys is implemented. The example, in

Figure 6 Eliminating the need to sort the expected output



OS/VS Sort/Merge control statements, follows:

```
SORT FIELDS=(1,3,CH,D,6,3,ZD,A,11,3,PD,D)
```

The major key is a character or EBCDIC data type with a length of three characters and in descending order. The intermediate key is zoned decimal with a length of three digits and in ascending order. The minor key is packed decimal with a length of three bytes (five digits) in descending order.

The sort/merge data generator first builds a table to control the generation of the ordering field data. The table contains information to be used for incrementing the key (Figure 9).

The data generator proceeds with the generation by creating the lowest sequencing value. Then the data generator selects a field to be incremented, based on the table entry for "increment." This step is followed by a selection of a byte or digit within the field to be incremented. And last, the amount of the increment is randomly produced.

Figure 10 continues with the example to demonstrate this process. Record 1 has the low order sequence value. Record 2 increments the third byte of the character field by "E0." Record 3 increments the first digit of the zoned decimal field by 9. Record 4 increments the third digit of the packed decimal field by 7. Record 5 increments the first digit of the zoned decimal field by 9, causing a change of sign within the zoned decimal field. Record 6 increments the third digit of the packed decimal field by 4, causing the third digit to "borrow" from the second digit, therefore incrementing the second digit by 1. Record 7 increments the first digit of the zoned decimal field by 2, with the result that the zoned decimal field changes signs and there is a "carry" out of the zoned decimal field to the character field.

This example illustrates the technique for supporting ordering keys of all varieties in the sort/merge data generator. As we have seen, the technique is to simply perform arithmetic on the key with incrementing and carrying within fields and across fields. A similar technique is used for user-specified alternate collating sequences.

Different record formats. The IBM DPPX Sort/Merge product supports the sorting of records that do not have their key fields in the same locations in

Figure 7 Multiple input files

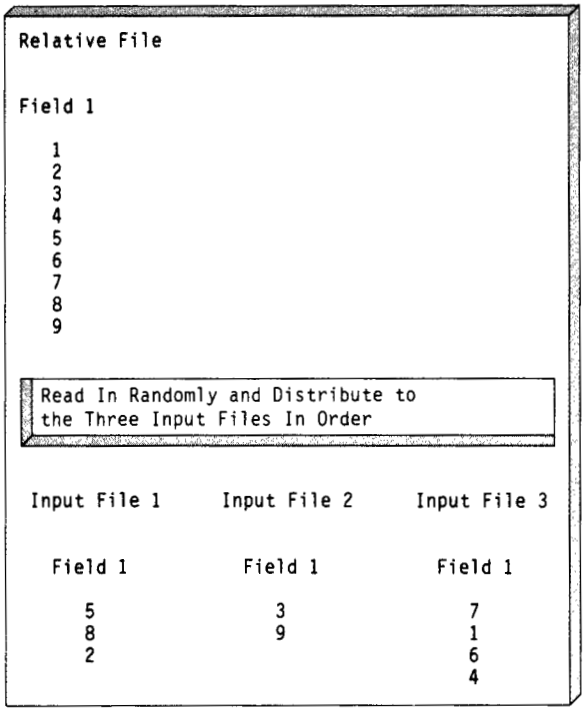


Figure 8 Merge application

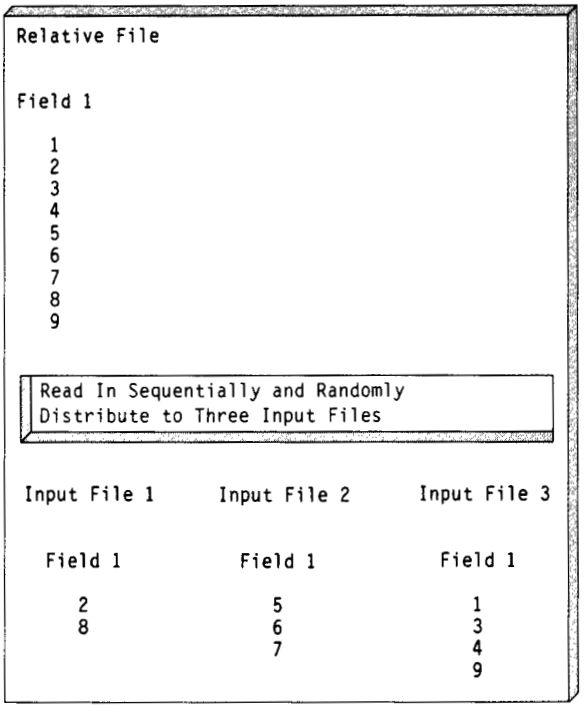


Figure 9 Controlling the generation of ordering field data

Field Number	Data Type	Increment	Start Byte	Amount	Asc/Desc
1	CH	Yes	3	255	Desc
2	ZD	Yes	1	9	Asc
3	PD	Yes	1	9	Desc

each record. Other fields are used to identify the record types that differ. An example of DPPX Sort/Merge control statements for such a case is given in Figure 11.

The example defines a sort application in which input records with an "A" in offset 1 will have their sort field at offset 5, input records with a "B" in offset 1 will have their sort field at offset 15, etc. The data generator chooses the record type of a record after it has determined the key value for the record. It then sets the record-type-identifying field in the record followed by entering the key fields into the record at their appropriate offsets.

The techniques we have described encompass virtually all the permissible sort/merge operations.

Concluding remarks

Test case generation has several benefits compared to the presently available alternatives. Our experience has been that the cost of coding and debugging a test case generator is considerably less than that of creating sufficient handwritten test cases for the same function. A generator also represents a better investment, since it can be enhanced to test future releases of the product under test.

The range of alternatives and combinations in the testing coverage for a particular feature is higher than it would be in any fixed amount of handwritten test cases. Once a new feature is included in the generator, it has the potential of appearing with all the other features already implemented.

Finally, the design and coding of a generator is likely to prove considerably more interesting than writing conventional test cases. The job satisfaction of the testers is therefore enhanced.

Software products are becoming ever more complex, and the amount of test material they require increases accordingly. It is highly desirable to find automated processes to create such material. It is equally important that the test cases produced should be easy to run and verify.

In this paper we have briefly described several implementations of our test case generator principle. All have been used successfully to test significant IBM program products. They have proved themselves to be both efficient and effective—small numbers of test personnel have located large numbers of defects. As we have seen, the method has been applied to areas such as graphics, which were initially thought unsuitable for the technique. Many more areas of testing should benefit equally.

Acknowledgments

Many contributions were made to the PL/I Test Case Generator by T. Clowes and R. Weir. Management support for test case generation work at the

Figure 10 Generated ordering field data

	Character Field	Zoned Decimal Field	Packed Decimal Field
Record 1	FFFFFF	F9F9D9	99999C
Record 2	FFFF1F	F9F9D9	99999C
Record 3	FFFF1F	F0F9D9	99999C
Record 4	FFFF1F	F0F9D9	99299C
Record 5	FFFF1F	F8F0C1	99299C
Record 6	FFFF1F	F8F0C1	98899C
Record 7	FFFF1E	F9F9D8	98899C
Record 8	FFFEDE	F9F9D8	98899C
Record 9	FFFEDE	F9F9D8	98891C

Figure 11 Different record formats

```
SORT
  SOURCE RESOURCE(INPUT) TYPE(RSDS) LRLEN(80)
    FIELD NAME(RECTYPE) LENGTH(1) OFFSET(1) FORMAT(CHAR)
    SELECT FIELD(RECTYPE) COND(EQ) CHAR(A)
      FIELD NAME(SORTME) LENGTH(4) OFFSET(5) FORMAT(NUM)
    SELECT FIELD(RECTYPE) COND(EQ) CHAR(B)
      FIELD NAME(SORTME) LENGTH(4) OFFSET(15) FORMAT(NUM)
    SELECT FIELD(RECTYPE) COND(EQ) CHAR(C)
      FIELD NAME(SORTME) LENGTH(4) OFFSET(25) FORMAT(NUM)
    SELECT FIELD(RECTYPE) COND(EQ) CHAR(D)
      FIELD NAME(SORTME) LENGTH(4) OFFSET(35) FORMAT(NUM)
  TARGET RESOURCE(OUTPUT) TYPE(RSDS)
    ORDER FIELD(SORTME)
END
```

IBM Santa Teresa Laboratory is currently being provided by P. Lue, M. Beasley, and L. Kaleda; previously support was given by S. McAulay and R. Dayton. The authors are grateful to D. Gasich and H. Stinton, and to the referees of the *IBM Systems Journal* for their comments on early versions of this paper.

Cited references

1. K. V. Hanford, "Automatic generation of test cases," *IBM Systems Journal* **9,** No. 4, 242–257 (1970).
2. P. Purdom, "A sentence generator for testing parsers," *BIT* **12,** 366–375 (1972).
3. A. Celentano, "Compiler testing using a sentence generator," *Software Practice and Experience* **10,** 897–913 (1980).
4. F. Bazzichi and I. Spadafora, "An automatic generator for compiler testing," *IEEE Transactions on Software Engineering* **FE-8,** No. 4, 343–353 (July 1982).
5. R. P. Seaman, "Testing high level language compilers," *IEEE Computer System and Technology Conference* (October 1974), pp. 6–14.
6. D. L. Bird, T. Clowes, and R. E. Weir, "Method of Operating a Computer to Produce Test Case Programs," U.K. Patent Number: 1,479,122 (July 6, 1977), p. 2.

General references

D. L. Bird, "Internal representation of arrays in automatically generated test case programs," *IBM Technical Disclosure Bulletin* **19,** No. 3, 1112–1113 (August 1976).

D. L. Bird, T. Clowes, D. G. Jacobs, and R. E. Weir, *A Test Program Generator for Testing Compilers of PL/I-Like Languages*, IBM United Kingdom Laboratories Limited, Winchester, Hampshire, United Kingdom (February 1977).

D. L. Bird, "Method of Operating a Computer to Produce Test Case Programs," U. K. Patent Number: 1,510,240 (May 17, 1978).

D. L. Bird, "Generating Graphics Test Case Programs," European Patent Application 52684. (Application also made for USA patent.)

D. L. Bird, "Test case generation for multiple device support," *IBM Technical Disclosure Bulletin* **25,** No. 7B, 3765–3767 (December 1982).

C. U. Munoz, *Test Case Generator Tutorial*, Technical Report 03.122, IBM Corporation, Santa Teresa Laboratory, P.O. Box 50020, San Jose, CA 95150 (November 1980).

David L. Bird *IBM United Kingdom Laboratories Limited, Hursley Park, Winchester, Hampshire SO21 2JN, England.* Mr. Bird joined the IBM Hursley Laboratory in 1968. After two years of development work on the PL/I Checkout Compiler, he moved to the area of software testing. Since then he has tested various language compilers, VTAM, CICS/VS, and the GDDM alphanumerics and graphics package. In 1973, working with two other programmers, he designed and coded the first generator of executable, self-checking test cases (for PL/I). Several generators for other software products have followed, and he is currently using generators to test various aspects of GDDM graphics. He is author or joint author of three patents on test case generation and has also written four books on contract bridge. Mr. Bird received his M.A. degree in mathematics from Gonville and Caius College, Cambridge, England.

Carlos Urias Munoz *IBM General Products Division, Santa Teresa Laboratory, P.O. Box 50020, San Jose, California 95150.* Mr. Munoz is currently a staff programmer. He joined IBM in 1974 as a junior programmer working in the build and release group for IMS and CICS. During 1976, he was a member of the team responsible for transferring development of the PL/I products from the Hursley Laboratory to San Jose. During this assignment he first became familiar with the test case generation technology and then used this technology while testing DPPX sort/merge in 1978 and 1979. Mr. Munoz then joined Programming Assurance of the General Products Division (GPD) as a technical evaluator for the PL/I products. In 1981 he initiated a technology project to extend the function of the PL/I Test Case Generator and to experiment with various testing approaches utilizing the scope and productivity of the tool. He was a joint recipient of an excellence award in 1981 for the sort/merge work, and in 1983 he was presented with a GPD achievement award. Mr. Munoz received a B.A. in mathematics from the University of California, Los Angeles, in 1974.

Reprint Order No. G321-5193.