Design, Implementation, and Case Study of a 
Function Level Unit Test Environment

Jim D. Creasman
VTAM™ Simulation, Process, and Quality
IBM Networking Systems Division
Research Triangle Park, NC 27709

Christopher J. Born
VTAM™ Development IIE
IBM Networking Systems Division
Research Triangle Park, NC 27709

Abstract

The continuing focus on reduction of software development cycle time, while maintaining ever increasing standards of quality, has caused renewed emphasis to be placed on software testing technology as innovative strategies are sought. Existing development processes have generally been refined to the point where little wasted time remains to be removed. This is particularly true in the development of older, well established software. A Function Level Unit Test Environment (FLUTE) provides a method for reorganizing the early stages of code development and test into a more efficient development model. The goal of this approach is to reduce the time spent on early testing, yet increase the value of such testing to the overall software quality. Following a statement of the relative issues, this paper defines what is meant by a Function Level Unit Test Environment and moves on to describe an actual implementation of this methodology within a large scale, systems software development project. A case study summarizes the results which were achieved.

1.0 Introduction

In recent years there has been an intense push towards reducing the cycle time of software development, while simultaneously increasing product quality (IBM’s Market-Driven Quality campaign holds defect elimination and cycle time/productivity among it’s highest software development goals [3]). The seeming disparity of these two pressures is nowhere more apparent than in the development of large scale, systems-level software. Given the complexity and foundation size of such systems, traditional approaches have generally relied on a highly sequential plan for the coding and early test phases of each functional increment. The established goal of low level testing is to insure that each new or changed line of code is executed, leaving later phases of test to certify or “test in” the quality of the integrated design.

The limitations of this process model are quickly realized if an attempt is made at shortening the cycle time significantly. Each functional increment can be thought of as a linear sequence of events where the success of each event depends on that of the succeeding events. The application of this thought process to the development plan produces a strategy that is highly serialized and prevents most attempts at parallel development without extensive scaffolding. That is, the function’s components cannot be easily uncoupled and redefined to allow asynchronous development to proceed. When coupled, the early development stages cannot be shortened. On the other hand, time cannot be taken from the later test phases without potential risk to the software’s quality.

Moreover, it is not uncommon in these environments to find that the entire product is executed just to perform low level testing — even though only a small fraction of the code executed is of interest to the developer/tester. Not only does this waste system resources, but it is an added encumbrance to the developer.

The authors define a “Function Level Unit Test Environment (FLUTE),” explain how it applies to solving or eliminating the above problems, and present a case study of its application. While the implementation is confined to a specific system programming environment, the ideas and topics presented are applicable to any large scale programming project.

2.0 Goals and strategy

The term, unit test, is used throughout to refer to the initial phase of test, also called module test. It is generally performed by the person responsible for writing the code and, as stated previously, its purpose is to execute each new or changed line of code. Function test (or function verification test) is a more formal stage of testing that follows unit test. This occurs once a significant share of the code exits unit test, and an executable function is available. This phase of test is often performed by a separate group specifically dedicated to this task [1].

The purpose of a Function Level Unit Test Environment is to provide an environment that increases the value of doing a unit (or module) test by executing function test cases during the unit test time frame. Furthermore, the challenge is to do this without extending the time taken to do a thorough unit test. The results are improved software quality through a more effective unit test, and less time spent on development and unit test.

The first step towards achieving these results is to expand the definition of unit test as ensuring "that all logic
paths are covered [1] to also include testing the *function* of the code as it relates to the whole product. In fact, in this new definition the emphasis is placed on testing function, with code coverage being a corollary of a good function test. To accomplish this during the unit test time frame requires the following:

- **Isolation of a test component.** It should not be assumed that the entire product is available at the time unit test is to occur. Instead, we distinguish a collection of modules that are functionally related, called a *component*. From this point onward, this body of code is regarded as if it were the "product". If this entity can be developed and tested in complete isolation from the other components, then enormous savings are possible from working on each such component in parallel. The desired uncoupling of events within the previous development process is achieved.

- **Formalization of the unit test process.** One of the benefits of later test phases is the formal manner in which they are approached. Test cases are planned, documented, and archived for future reference. Generally, some degree of automation exists for repeated execution of regression sets of test cases. When applied to unit test this becomes a powerful means for documenting the progress made, and thoroughly exercising the code that will be passed on to later test phases.

While this would seem both practical and straightforward to implement, the volatile nature of this stage of code development must be taken into account. The tools or process for documenting and executing the test cases should be straightforward to learn and use, of proven and perceived value, and be flexible enough to meet the shifting demands.

The key benefits realized when the above objectives are met include:

- **Increased productivity.** Because the test component is isolated from the rest of the product, the tester is able to spend more time actually executing their code, and less time bringing up the full product, or other more elaborate test system involving sophisticated scaffolding. Given the frequency of this task, even a small savings per bring-up has a large cumulative effect. Also, the isolated environment allows the developer to take full advantage of the knowledge they have relative to their component, versus the whole product.

- **Portability.** Another major advantage from an isolated test component, is that the code being tested is less likely to be system dependent. For example, code that will ultimately run on a Multiple Virtual Storage (MVS) operating system, can often be tested in a simpler test environment run under the Conversational Monitor System (CMS) in a virtual machine.

- **Preservation and focus of effort.** Finally, the formalization of the unit test environment introduces a formal language in which test scripts are written. This provides the following benefits:
  - If well written, the test cases document the unit test phase and fully describe how much and to what extent the code was tested. This can become a very useful repository to later phases of test, or for later releases of the product.
  - Having a standard language in which test scripts are written is the first step towards automated regression testing within the unit test environment. At any level of test, routine regression testing is a solid approach to preventing "bad fixes" from causing new errors. Unit test is no exception to this rule.
  - Simply having a formal process for writing test scripts forces the tester to focus their attention towards exercising the function of the code and not code coverage only. As we shall see, this is an important part of a Function Level Unit Test Environment.

The paradigm shift we have discussed is vividly illustrated in the following two graphs. Figure 1 shows the traditional approach to developing a particular function which spans four components (A, B, C, and D). Each component depends on the preceding component's changes to be at or near completion before development and test can begin.

![Figure 1. Traditional development model for UT/FVT](image1)

In contrast to the traditional model, Figure 2 shows the shift that occurs in moving to a Function Level Unit
Test Environment. The components are labeled A', B', C', and D' to indicate that roughly the same component content is implied. In this model, rather than waiting for the last component to exit unit test, a common end date is sought for the development and unit test phase. Following this is a short phase, labeled IT (Integration Test), whose purpose is to merge the previously isolated components and validate the component interfaces (The IT phase is a comparatively small effort with respect to time and resources, and the graph is not drawn to scale). Finally, a formal function verification test is entered.

3.0 FLUTE terms and concepts

Before continuing it is helpful to understand several key concepts and terms which provide the foundation for understanding the actual implementation of the Function Level Unit Test Environment, which we shall hereafter refer to simply as FLUTE.

Background information: First, consider Figure 3, which illustrates a calling tree. The leaves of the tree represent the modules (ie., code sections) which make up the product or function. The branches define the order and context in which control passes between the modules. For example, module A.1 is allowed to pass control directly to modules B.1, B.2, and/or B.3. However, A.1 can pass control to C.2 only by going through B.1. We shall use the terms call and return to indicate the direction control is passed between a pair of modules (eg., A.1 could call B.1, and then B.1 might return control to A.1).

Within any given calling tree, there are numerous paths. A path through the tree represents a sequence of modules which are executed under a given set of input conditions. Note that a path may begin with any module in the tree and continue downward through several levels before returning, and may exit the tree prior to returning control to the module at the beginning of the path. Also, a path may loop on itself any number of times, indicating that a module is called more than once.

Defining the Test Component: Building on the idea of a calling tree, the test component is a subset (or "subtree") of this tree. Figure 4 illustrates a typical test component which is defined over the calling tree shown in Figure 3. Modules which are contained in the Test Component are referred to as members. For example, this test component has eight members -- A.1, B.1, B.2, B.3, C.2, C.3, D.2, and D.3 -- enclosed by the dotted line.

This dotted line has a special place in our discussion. It is called the test component boundary, and is the boundary between reality and fiction. All the modules inside this area are part of the code being tested (ie., they are real). Those modules outside the boundary are considered scaffolding (ie., logically they are not a part of the code being tested). Stated another way, the test component boundary is the division between the code and data of the Test Component, and that of modules outside the test component boundary.

Of particular interest are the points at which the boundary is crossed. Generally, this occurs for one of three reasons:

1. A member of the test component is called from, or makes a call to, a non-member.
2. A call is made to a test component member. However, the tester has specified that a forced exit is to occur when this routine is called, perhaps to validate intermediate results.
3. A member of the test component references data which is not contained within any of the test component members.

Those which occur for the first two reasons are called entry/exit points, and are referred to as either primary, or secondary.

Primary Entry/Exit Points (PEEPs) occur whenever a test component member is passed control from the test script. They are always the first module to receive control along a test path. Typically, they are also the last module to have control before terminating the test path, though this is not always true.

Secondary Entry/Exit Points (SEEPs) are caused by a test component member calling a non-member routine, or by the test script forcing an exit at a call to a member. In Figure 4, the calls to non-members C.1, D.1, and E.1 all cause SEEPs to occur. Whenever this happens, scaffold code must be available to provide the function of the called routine. The implementation of FLUTE provides
ways for this type of scaffolding to be written using the FLUTE language, which is generally much easier than writing a compiled-language replacement.

4.0 FLUTE: The working model

The implementation of a Function Level Unit Test Environment is not a single program, but a complete tools package (collectively called FLUTE) developed for supporting unit test of code written for execution within an IBM System/370™ environment. At the heart of the package is the FLUTE language and its interpreter.

FLUTE provides a means to quickly build, bring up, and maintain a test environment. Test scripts are written in the flexible, yet powerful FLUTE language. Regression testing is also easily realized because once a test script is written and executes successfully, it can be run repeatedly, thus ensuring that later code changes do not corrupt previously tested function.

Figure 5 presents a structural overview of FLUTE during execution of a test script. The main FLUTE component is shown as the FLUTE Host/Driver. The Driver is responsible for the parsing, interpretation, and execution of the user's test script, while the Host's role is to initialize the specific environment and start the FLUTE Driver program. Consequently, the Host represents only a small portion of the overall function, and exists mainly to increase the portability of the FLUTE Driver.

The test code is labeled Test Component Code in the diagram, and represents the user's code that is being exercised. The area shown as Scaffolding is actually created and managed by FLUTE, even though it logically resides within the Test Component area.

The other principle FLUTE component is the FLUTE Runtime Services. The services provided include storage management, dispatching between the Driver and the Test Component, and providing end diagnostics and error recover should any part fail.

4.1 The FLUTE language

We have alluded to both the power and flexibility of the test script language as being necessary to the success of FLUTE. An additional requirement is that the language be easy to learn and use. For these reasons, the basic look and feel of the language is modeled after REXX. REXX (Restructured Extended Executor) is IBM's system product interpretive language [5], and enjoys wide use among IBM's major System/370™, based environments (eg., VM and MVS). It's ease of use and familiarity make it a natural choice. Thus, the basic features of the FLUTE language are:

- It is interpreted versus compiled. This allows for quick development of test scripts and is perhaps the single most important feature of the language.
Variables are allowed within the test cases, as well as standard arithmetic, comparison, and string operations. These constructs provide a means to write flexible, intelligent scripts.

The language is token oriented. From a support standpoint, this means that new commands and instructions can be added quickly as their need is identified.

Since the test component consists of compiled or assembled code, the next logical extension to the FLUTE language includes instructions and services which are logically associated with a compiled language. These include the notion of defined storage areas, the use of constants, and structured or mapped data. Various FLUTE instructions and functions operate at this level to provide accurate simulation of the product environment.

The third, and final, extension of the language meets those requirements which are unique to the purpose of FLUTE. These instructions provide effective storage management, path description and execution, and verification and debugging of test results. The remainder of this section will focus primarily on this class of instructions since they set the language apart and serve to make it a unit test language.

### 4.2 Writing test scripts

In FLUTE, the purpose of a test script is to exercise a particular function of the test component as it will ultimately be performed within the product. Since the test component is executed in isolation of the rest of the product, one view of the test script is that it is really specialized scaffolding which allows testing of a function by simulating the component environment and providing the input stimulus to drive the desired path(s).

In its simplest form then, the outline of a test script is as follows:

1. Test script setup
   - Initialize script variables and global data.
   - Specify output content and destination.
2. Pre-execution setup
   - Create and initialize external data areas.
   - Set registers, parameters, and other runtime information.
3. Path execution
   - Initiate and monitor the execution of a path through the test component by passing control to the appropriate member(s).
4. Test verification
Either after the path has completed, or at selected points along the path, verification of the expected results takes place.

- Verify the path executes as expected.
- Verify the registers and other output are set as expected.
- Verify the contents of external data areas.

The first step deals mainly with tailoring the test script to suit the developer's requirements and individual taste. It is generally standard among components and is performed only once. Of particular interest to the topic of this paper are the remaining three steps.

Pre-execution setup: Again, viewing the test script as scaffolding, the purpose of this phase is to accurately simulate the input to the test component. This input consists of the data and parameters passed to the PEEP, and the contents of the registers at the time the PEEP is called. To this end, the FLUTE language supports instructions which allow data areas to be easily defined (and allocated if necessary) and then initialized or altered. Also, a special set of FLUTE variables map to the set of general purpose registers. The register variables are $g0, g1, ..., g15,$ each corresponding to one of the 16 general purpose registers in the IBM System/37D™ architecture.

The tester should view these variables as though they were the actual machine registers. That is, whenever the test component code is entered, their values are copied to the machine's registers, and at the moment the test component code is exited, the machine registers are copied into these variables. From the test script's perspective, they always reflect the actual register contents last used by the test component.

For example,
\[
1g1 = addr(RPH); \\
1g15 = 4;
\]

assigns the address of a data area, named RPH, and the value 4 to registers 1 and 15, respectively.

A number of instructions are used to setup and maintain the program data areas. While more complex than setting registers, the language is tailored to balance the necessary complexity against the goal of being straightforward to learn and use.

FLUTE requires that each data area a test script references has its own definition. This storage definition, as it is called, is a single point of information which completely describes a block of storage the test case is using. Each storage definition has a unique name, called the storage name, which allows it to be referenced by various other instructions in the test script.

The storage definition also has a definition type and a map description. The definition type determines the attributes of the storage area -- where the storage is located and how it is used. The map description fills in the rest of the information -- how the storage "looks" and how big it is.

A storage definition must be created prior to referencing the storage. The DEFINE instruction is used for this purpose. The basic syntax for DEFINE is simple:

```
DEFINE(name) type map;
```

yet there are many variations which occur, depending mainly upon the type of storage area being defined. FLUTE recognizes four main types of definitions:

**BASED ON**

This definition maps existing storage within another, previously defined definition.

**BASED PTR**

This definition type is similar to the BASED ON type. However, the location of the storage is determined using a test script variable, as opposed to the name of another definition.

**DYNAMIC**

A DYNAMIC definition indicates that the actual storage will be allocated later in the test script using the special CREATE instruction.

**LOCATED AT**

Located storage must reside at a fixed location in the machine that is known at the time the DEFINE is executed.

There are two ways to specify how the storage will be mapped, either explicitly, or with the USING operand. Explicitly mapped storage means the map description is contained directly in the DEFINE instruction. With USING, the name of a FLUTEMAP file is specified as the map description.

FLUTEMAPs are special files which create a window over an area of memory. These maps are used in the exactly the same way as other languages use declared structures to map an area of memory. They allow the test script to use the concept of fields as opposed to working with addresses, offsets, and lengths. Having these files separate means that the test scripts do not require compilation should offsets or other elements of a map change. Also, having this standard interface means the FLUTE Driver is independent of the language the test component is written in.

The final requirement of the pre-execution phase is to actually assign values to the data areas. The ASSIGN instruction is used for this purpose at any point after the target storage is available. A single ASSIGN instruction is used to assign values to one or more fields within the storage definition. The basic syntax is,

1 In the FLUTE language, all test script variables must begin with either ! or #.
ASSIGN(name)
    field(value)
    field(value)
    ...;

where name is the name of a storage definition, and field and value are a field name and desired value, respectively, within the storage definition's map description.

Path execution: Recall that the purpose of this phase is to execute a specific path through the test component. In the definition of a test component the notion of entry/exit points was discussed. Every path begins at an entry point (ie., the PEEP), and may pass through other entry/exit points (ie., SEEPs) prior to completing. The instruction used to manage and describe this flow of control is called EXECUTE.

The EXECUTE instruction has a number of operands, but in its simplest form provides the following services:

- A formal description of the expected path,
- The ability to pass parameters,
- A means to scaffold non-members that are called as a result of executing this path,
- The ability to exit the test component whenever a member is called, and then return to or bypass the called member, and
- The ability to end the path at any exit point.

Figure 6 shows two examples of how EXECUTE is used for path description. For each example, the test component is shown first, with the corresponding EXECUTE instruction below. In example (1), the assumption is made that A calls B, then A calls C, while in example (2), A calls B, then B calls C.

Note that the AT clause within each EXECUTE instruction specifies the entry point name where control is to return to the test script, allowing the test script to monitor or change intermediate results throughout the path (The ellipsis ( . . . ) within each AT clause indicates that a group of FLUTE instructions may appear here).

The test component is running in isolation of the product, and it is often the case that certain service routines are omitted from the test component load module (eg., storage management routines, trace routines, etc.). While the services provided by these modules are critical to the success of the function, they are not logically a part of the component being tested, and the decision is made to scaffold their services. Since they may be called frequently, it is impractical to expect an AT to be placed where each call would occur. FLUTE provides the ALIAS instruction to allow a replacement test script to be executed by FLUTE should a particular entry point ever be called.

Test verification: The complement to assigning and setting up the test component input is that of verifying that the test component has modified and set the registers and other data areas as required. As seen in the EXECUTE instruction this verification step can be performed as the path is being executed, or after it has completed.

Registers can still be verified by examining the contents of the register variables. However, FLUTE supports a specialized instruction, called VERIFY, to allow various types of output data to be checked against expected values. The basic syntax for VERIFY is similar to the ASSIGN instruction,

VERIFY(name) [DISPLAY(type)]
    key(value)
    key(value)
    ...;

with the addition of the DISPLAY operand. Also, VERIFY is slightly more general purpose than ASSIGN, allowing for either storage, registers, or a parameter list to be verified. The name determines the type of verification that is performed. REGISTERS or PLIST implies the VERIFY is a register or parameter list verification.
respectively. Otherwise, name is assumed to be the name of a storage definition.

The meaning of key depends on the setting of name. For storage verification, the key is the name of a field within the storage definition. Value is the expected value of the field, register, or slot in the parameter list. The display type determines how much data is displayed, and has four possible values. If storage verification is being performed, their meaning is as follows:

**ALL**
Displays all fields within the storage definition.

**ERROR**
Displays only those fields specified in the VERIFY whose expected values differ from the actual values.

**NONE**
Suppresses the display of output.

**ONLY**
( Default) Displays only the fields listed in the VERIFY.

Regardless of the display option used, a return code variable (!rc) is set to indicate the success (!rc = 0) or failure (!rc = 1) of the VERIFY instruction. The test script can examine the contents of this variable and decide the appropriate action to take. An example of the VERIFY instruction used to confirm storage is shown below. The name of the storage definition is 'QUE':

```
verify(QUE)
QUEWQCHN('MY QUE')
QUEID(22)
QUEUNCON('B'11')
QESYNCH('B'1')
QUEOFFST(X'22')
```

The output from this example would be similar to that shown in Figure 7. Note the use of the asterisk (*) in the first column to flag fields which have discrepancies between the actual and expected values. Also, the T (Type) column identifies the field type with both expected and actual values formatted according to the type of data they represent. This means that a field whose type is "U" (Unsigned integer) will be displayed as a positive decimal integer, while a field shown as "A" (Address) will be formatted as hexadecimal data. The type is determined at the time the storage definition is created, either directly from the definition, or from an associated FLUTEMAP file.

<table>
<thead>
<tr>
<th>F Offs</th>
<th>T Field Name</th>
<th>Expected Value</th>
<th>Actual Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>S QUEWQCHN</td>
<td>MY QUE</td>
<td>MY QUE</td>
</tr>
<tr>
<td>0010</td>
<td>U QUEID</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>* 0011</td>
<td>B QUEUNCON</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>* 0011</td>
<td>B QESYNCH</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>* 0016</td>
<td>A QUEOFFST</td>
<td>0022</td>
<td>0020</td>
</tr>
</tbody>
</table>

Figure 7. Storage verification output from the VERIFY instruction

### 4.3 Sample test script

We conclude the discussion of the FLUTE language with an example of a complete test script. The test component is purposefully simple to allow the key elements of the test script format (as presented in the previous section) to be easily visible. The test component consists of a single member, called EXPO, whose function is to raise an integer value to a power and return the result. EXPO is passed three parameters. The first two, $BASE and $POWER, are input parameters corresponding to the base and exponent of the operation. The last parameter ($RESULT) is the resulting output from the computation.

The test case is to confirm that EXPO correctly computes $17^3 = 4913$. In English, the test script outline is:

1. Define and allocate the required storage:
   a. $BASE as a signed four-byte integer.
   b. $POWER as an unsigned one-byte integer.
   c. $RESULT as a signed four-byte integer.

2. Initialize the input:
   a. Set $BASE equal to 17.
   b. Set $POWER equal to 3.

- Path Execution
  Simulate the call to EXPO, passing the required input addresses.

- Test Verification
  1. Verify the input fields have not been changed.
     a. Verify $BASE is still 17.
     b. Verify $POWER is still 3.
  2. Verify that $RESULT is 4913 ($17^3$).

The actual test script to accomplish this is shown below.

```
/* Pre-execution Setup: */
/* */
define($BASE) dynamic SIGNED(31);
create($BASE);
define($POWER) dynamic UNSIGNED(8);
create($POWER);
define($RESULT) dynamic SIGNED(31);
create($RESULT);
assign($BASE) $BASE(17);
assign($POWER) $POWER(3);
/* */
/* Test Execution: */
execute(EXPR) plist(addr($BASE), addr($POWER), addr($RESULT)) /*
end;
*/
/* */
/* Test Verification: */
/* */
verify($BASE) $BASE(17);
verify($POWER) $POWER(3);
verify($RESULT) $RESULT(4913);
exit;

Assuming the test is a success, we would expect to see the output shown in Figure 8.

<table>
<thead>
<tr>
<th>F Offs</th>
<th>$BASE</th>
<th>Expected Value</th>
<th>Actual Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>17</td>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F Offs</th>
<th>$POWER</th>
<th>Expected Value</th>
<th>Actual Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F Offs</th>
<th>$RESULT</th>
<th>Expected Value</th>
<th>Actual Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>4913</td>
<td>4913</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8. Sample test script output

4.4 Executing test scripts

As mentioned, the central part of the FLUTE package is the FLUTE language interpreter (ie., the Driver). However, there are a number of other peripheral tasks which must be performed in order to create and maintain the test environment. Primarily these include:

- Generating the special files (called FLUTEMAPs) which provide dynamic mapping of structured data within the test case.

For the environment described in the case study, an automated process was developed to refresh these maps on a nightly basis, as well as a direct interface to allow the tester to create a private copy of a map (Maps are generated from the same source as used in compilation of the test code).

- Creating the test component load module.

The key requirement for building the test component is that it be simple and straightforward. The isolation of the test component requires that we move from a centralized test build, to one “mini-build” per component. A simple program is used to read a file containing the list of object files to be included in the build. The program then links each object deck into the test component load module [4].

- Loading and starting the Driver (ie., the Host function).

Since part of the function of the Host program is to make the FLUTE Driver more portable, a number of different Hosts exist. However, the primary host used in the case study is a general purpose, CMS-based program which performs the following services:

1. Loads the test component load module,
2. Reads the load module to determine if any entry points are unresolved (ie., the module is not present), and if found, dynamically “zaps” the code to ensure that control is passed to the FLUTE Dispatcher if a non-member is called [2], and
3. Loads and starts the FLUTE Driver program.

As an additional service, this Host program also facilitates the integration of a source level debugger into the test environment.

5.0 The case study

The following case study involves the use of FLUTE during the unit test phase of four components. The components were a part of an existing large system program (greater than one million lines of code). Three of the four components consisted of new code, while the fourth was existing code with many modifications. The code sizes varied, two were greater than 10,000 lines, one greater than 5000 lines, while the existing component had modifications numbering more than 7000 lines.
Each component executes as an independent process and communicates using messages. As messages are queued to a component's input queue, the runtime dispatches the component. Data hiding by each component is emphasized. This results in more signals being sent between components but removes data contention problems and the solutions for those problems. Because of the way the components are designed, they lend themselves to being tested in a FLUTE environment.

5.1 Unit testing phase

As discussed earlier, with large software products, it is time consuming and costly (machine usage) to execute the complete system. Using FLUTE, each of the components were able to avoid executing the complete system during unit test. Each component was tested independent of the full product. Depending on when a function was scheduled to be available in the full product, components would test that function using FLUTE. As soon as the component's part in the function exited unit test, it was made available to the next stage of testing.

Efforts focused on achieving functional coverage of the component during unit test, with the result being that by the time all function paths within a component were tested, the coverage in each module of the component would be close to 100 percent. Figure 9 illustrates this with each bar representing the current regression set. Initially, adding test scripts has a dramatic effect on the total lines executed. As the number of completed test cases grows the graph levels off, approaching the optimal 100 percent mark. However, it is important to note that the relative size of each bar (representing the functional coverage) increases as more test scripts are added.

![Figure 9. Code coverage per test scripts executed](image)

5.2 Test requirements

Each component team was responsible for 100 percent code coverage during unit test for each module within the component. Using FLUTE, a line by line unit test could be achieved while at the same time productivity and quality were increased significantly. Instead of each test case being comprised of testing a single module and the lines within that module, FLUTE provided a method for testing many modules and the lines in those modules during each test case.

Developers used FLUTE to create and queue messages to the component, then execute the code with the message's input. Depending on the component, a typical FLUTE test case consists of the sending and receiving of a set of messages which simulate a specific function.

5.3 Regression testing

Avoiding the introduction of new problems and affecting previously working function is important for both improving quality (preventing errors) and improving productivity (immediate feedback on changes). By regression testing a component after code was updated, the developers received instant feedback on whether the updates negatively affected any previously working function. This was extremely beneficial because it prevents developers from making an update that fixes one problem but introduces others.

Each component was periodically exercised by previously created set of test cases using the latest developed code. These sets contained all of the past developed test cases. The number of test cases varied in size for each component. One component had over 600 test cases while another had 200 test cases.

The uncovering of unknown bugs was another benefit of regression testing. There were instances when, after an update was made, it would have been okay to make the update available to the next phase of testing. However, because of the regression test, new bugs were uncovered. So, instead of the single update to fix a single problem, a single update was made that would fix a group of problems.

5.4 Integration Test

After all components completed unit test, they were merged into the full product and entered the Integration Test phase. During this phase of testing, mainline paths through the components were executed. The initial exposed bugs were as expected -- improper interpretation of interfaces between components. These errors were quickly detected and corrected. Relative to the total function verification test phase, both time and resources spent were minimal. Only one person was used during this phase.
5.5 Functional Verification Testing

Once the interface errors were corrected, testing of the complete product moved into the formal Functional Verification Test phase, where it progressed similar to past releases. However, during this testing phase, any problems reported and fixed within a component were also checked by FLUTE. Any FLUTE test cases affected by the fix were updated. Once this was completed, the component would be regression tested using the past developed FLUTE test cases. Only if no problems were found by the regression test, was the fix added to the complete build.

5.6 Quantitative Results

The results of using FLUTE have been exceptional. The productivity and quality goals set at the start of development were exceeded by a significant amount. Productivity was measured based on the lines of code each developer coded and tested. Quality was based on the number of errors found in the next phase of testing. For each new development effort, the quality and productivity goals are increased by a certain percentage over that of the previous development effort. The original goals for both productivity and quality were a 15 percent improvement over the past effort. Using FLUTE, productivity increased by 30 percent and quality increased by 20 percent.

5.7 Subjective appraisals from developers

The measurement of the usefulness of a software tool can be represented with numbers on productivity and quality. FLUTE has been shown to increase both. The amount of enthusiasm that the end-user has towards a software tool is also a good indicator. The results of an unscientific poll indicate that FLUTE has a very high end-user rating. Many users said that they would use it again. A sample of users' comments include the following.

"[FLUTE] increased my productivity 40 percent."

"I couldn't have committed to the schedule if I didn't have FLUTE."

"[FLUTE] allowed me to work independently and without interruptions during my testing."

The rest of the comments were all similar to these.

Acknowledgments

We acknowledge Mike Arnold for his help in reviewing the technical content of this paper, and Sandy Coats for her assistance in gathering the case study statistics. Finally, though too numerous to name individually, we are most appreciative to all the developers and testers who use FLUTE, without whom the goals and aspirations of this project would not have been realized.

Cited references


General references

IBM System/370 Principles of Operation, GA22-7000, IBM Corporation.

About the authors

Jim D. Creasman IBM Networking Systems Division, P.O. Box 12195, Research Triangle Park, North Carolina 27709. Mr. Creasman is a staff programmer in the VTAM™ Simulation, Process, and Quality department. He is currently responsible for the development and support of FLUTE and other simulation tools. He received his B.S. in mathematics from Mars Hill College in 1983, and a M.S. in applied mathematics from North Carolina State University in 1987.

Christopher J. Born IBM Networking Systems Division, P.O. Box 12195, Research Triangle Park, North Carolina 27709. Mr. Born is a senior associate programmer in the VTAM™ Development IIE department. His current responsibilities include test and environment support for current and future releases of VTAM™. He received his B.S.E. in computer engineering from The University of Michigan in 1985, and a M.S. in computer science from Michigan State University in 1989.