Invited Paper

Software Testing

HIROSHI KAWATA*, HIROSHI YOSHIDA*, MUNEAKI NAGAI* and HIROSHI SAIJO*

This paper briefly surveys the current status of software testing and provides an introduction to the methodologies and techniques employed by Fujitsu. The survey will examine software testing from the following viewpoints: (1) designing test cases from external specifications, (2) automatic test program generation, (3) fault injection testing, and (4) application system testing.

1. Introduction

The purpose of testing is to ensure software quality through the identification of programming errors or "bugs" and the validation of proper program operation. It is essential that software be subjected to a variety of tests, at various phases in the development cycle, under different operating conditions and with different data. These tests are all necessary and should complement each other. Yet no matter hw ambitious the test plan for a program is, it can never cover all possible cases. Instead, it is necessary to design and select test cases that will provide the highest level of confidence in the quality of the software. Thus, test case design and selection become extremely important. In the first part of this paper, we discuss test case design in general and describe our method of designing test cases based on external specifications.

Although test case design is a very important part of the testing effort, creation ot test data and confirmation of test results are also very important. Moreover, they require significatn amounts of time and manpower, and can be very tedious. Various tools for automatic test data generation and/or test result confirmation are in use today or in the advanced stages of research and development. In the second part of this paper, we will comment on the current status in this area from our point of view and then provide an outline of our test program generator, which is based on attribute grammar.

A system software product such as an operating system is equipped with an error recovery mechanism to localize the effects of errors and permit the system to continue operation when they occur. Recently, in some papers, there has been discussion of "fault injection testing," which is a technique intended to test these error recovery mechanisms. In the third part of this paper, we will discuss our methods for testing fault-tolerant computer systems. We will also describe a methodology for evaluating the degree of fault tolerance of a system by using a Mean Time Between Failure (MTBF) metric.

In system testing, all components and products that constitute a system are assembled, the operational environment is established, and testing is done according to the development objectives. At this stage, configuration management and/or modification management become more important than in previous stages, as of course do the testing activities themselves, which include functional integrity testing, stress testing, performance testing, and regression testing. In the case of general-purpose system software products such as operating systems, simple system testing is not sufficient before delivery to a specific customer, and application system testing is also required, which we will describe through an example.

2. Designing Test Cases

In functional testing, test cases are designed and selected on the basis of external specifications such as functional specifications. Typically, test cases are designed as follows:

1. Divide the functions to be tested into small units that can be handled easily.
2. For each unit of function resulting from Step 1, categorize and analyze the test factors (test factors include such items as input conditions, environmental conditions, and results).
3. Using boundary-value analysis and equivalent-value partitioning, analyze the statuses each test factor can take.
4. Find the possible combinations of statuses for each test factor.
5. Select test cases from the combinations made in Step 4.

Equivalent-value partitioning and boundary-value analysis are effective methods for reducing the total number of statuses of various test factors. Cause-effect

*Fujitsu Limited, 140 Miyamoto, Numazu-shi, Shizuoka 410-3, Japan.

graphing and state transition diagrams are often used to study combinations of test factor statuses. Tools exist that can generate test cases from cause-effect graphs automatically [4].

In a complex and large-scale system, however, it becomes extremely diffucult to connect a cause and its effect. Instead of using cause-effect graphs, we use a test factor analysis table that primarily analyzes causes [5, 6]. Figure 1 shows an example of a test factor analysis table.

Since the design of test cases using a factor analysis table is done mainly by studying the cause (such as the input conditions and environmental conditions), it is relatively simple in comparision with cause-effect graphing. Nevertheless, as the number of test cases increases, and thus the size of the set from which test cases must be chosen, the following problems surface:

1. There are no clear criteria for selecting test cases. When the number of test cases is large, the test case candidates are selected subjectively rather than according to an objective standard. This gives rise to the next problem.
2. It is difficult to evaluate the coverage of test cases selected.

To overcome these problems, we and our colleagues [6] have developed a method for selecting test cases by applying the "design of experiments" method [7]. By applying an orthogonal array of the "design of experiments" method, one can find the smallest set of test cases that satisfies the following condition: "between any two factors, there must be the same number of combinations of statuses for each factor," if the number of statuses (represented as p) for each factor is the same and is a power of a prime number. Table 1 depicts the orthogonal array for p=2. It demonstrates that to satisfy the above condition when p=2, four test cases are required for test factors of two or three, and eight test cases for test factors of four to seven.

For the above table to be used, the number of statuses for all factors must be equal. In designing test cases, however, the number of statuses of each test factor take different values from each other in many cases. It is possible to overcome this problem through the use of a Boolean graph, as depicted in Fig. 2. This graph combines factors and their statuses by logical operations so that an orthogonal array can be applied to this graph.

In this Boolean graph, the statuses of a factor are connected by 'logical or' and combination conditions between two factors are connected by 'logical and.' The input to 'logical or' is set at 2 so that application of an orthogonal array for p=2 at a 'logical or' node (that is, one of nodes 21, 22, 23, and 31 in Fig. 2) allows test cases to be selected.

When there are many factors and statuses, the number of test cases may still be large. Further reduction may be required. In such situations, we can relax the condition to "between any two factors, there must be at least one combination of all possible combinations of their statuses." As can be seen in Table 2, this reduces the number of test cases.

(a) External specification

LABEL	INSTRU-CTIONS	OPERAND

	FID	TYPE={OUT I DSP}
		PGM={AIM I ACS I BATCH}
		COPYLIB={YES I NO}

(b) Factor analysis table

FACTOR		TYPE A	PGM B	COPYLIB C
STATUS	1	OUT	AIM	YES
	2	DSP	ACS	NO
	3		BATCH	

Fig. 1 Example of an external specification and a corresponding factor analysis table.

Table 1 Orthogonal array for p=2.

FACTOR / TEST CASE	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
T2	1	0	1	0	0	1	1	0	1	0	1	0	1	0
T3	0	1	1	0	1	0	1	0	0	1	1	0	0	1
T4	1	1	0	0	1	1	0	0	1	0	0	0	1	1
T5	0	0	0	1	1	1	1	0	0	0	0	1	1	1
T6	1	0	1	1	1	0	0	0	1	0	1	1	0	1
T7	0	1	1	1	0	1	0	0	0	1	1	1	1	0
T8	1	1	0	1	0	0	1	0	1	1	0	1	0	0
T9	0	0	0	0	0	0	0	1	1	1	1	1	1	1
T10	1	0	1	0	0	1	1	1	0	1	0	1	0	1

statuses of each factor is represented as 0 or 1

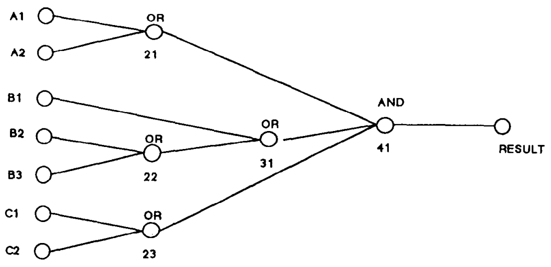


Fig. 2 A Bodean graph for Fig. 1.

Evaluation of Our Methodology

Our methodology simply introduces one objective criterion for the number of test cases, based on the

Table 2 Reduced combination table derived from Table 1.

FACTOR / TEST CASE	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
T2	1	0	1	1	0	1	1	0	1	1	0	1	1	0
T3	0	1	1	0	1	1	0	1	1	0	1	1	0	1
T4	1	1	0	1	1	0	1	1	0	1	1	0	1	1
T5	0	0	1	1	1	0	0	0	1	1	1	0	0	0
T6	1	1	0	0	0	1	1	1	0	0	0	1	1	1
T7	0	1	1	1	0	1	1	0	0	0	1	0	0	1
T8	1	0	0	0	1	0	0	1	1	1	0	1	1	0
T9	0	1	0	0	1	0	1	1	0	0	1	0	1	0

number of combinations of test factor statuses. While there is no theoretical basis for believing that the application of this criterion will yield sufficient coverage, we can empirically state that it has been satisfactory to date. Compared with previous techniques for selecting test cases, which relied on personal knowledge and experience, this methodology usually requires a wider selection of test cases to ensure its coverage, and results in an improvement of the bug detection capability. Through the use of a test case design tool that generates test cases from a test factor analysis table, the manpower needed to design test cases is also decreased significantly [5, 6]. However, there are cases in which we must test all combinations. In such situations, our methodology is naturally not effective.

3. Automatic Generation of Test Programs

A method for automatic generation of test programs typically defines a test grammar in a formal way, such as BNF, and generates test programs from this description. It is a relatively simple task to randomly generate test programs according to syntax description of the language. However, the genration of practical executable programs requires solutions to several problems.

The first problem in generating test programs is that of representing contextual dependencies in the language. Duncan [8] has resolved this problem by using attribute grammar and has developed a test program generator using a parser generator technique. From our experience, however, use of a general parser generator will require the description of all the information regarding attributes and the passing of attributes. It also results in a large amount of description.

The second problem is in the generation of test programs with selfchecking code. If the confirmation of the testing results requires a large amount of manpower, the benefit of automatic generation of test programs will be reduced. D. L. Bied and C. U. Munoz [9] describe automatic generation of test programs that are executable and as self-checkable as possible. They discuss the results of its application to a PL/I language processor, a sort/merge and graphical data display manager, although it is not based on a formal language definition. D. C. Ince [5] pointed out that attribute grammar would be very effective in solving the problems of both data generation and the confirmation of execution results.

The last problem is functional coverage. To assure adequate coverage we must be able to generate executable programs with complex structures such as loops. The reports published so far seem to describe relatively simple cases [9, 11, 12].

We here describe our test program genrator, TPGEN, based on attribute grammar. It generates high-quality test programs mainly for programming language processors by using the following features [14]:

1. Generated test programs are executable and have self-checking code for validating execution results.
2. Generated test programs are guaranteed to have specific testing coverage for functionality.
3. TPGEM simulates execution of the generated test programs and if abnormal events such as zero divides or infinite loops are detected, it back-tracks to a specified position and selects and alternative production rule to avoid such abnormal events. Introduction of this mechanism has succeeded in generating a wide variety of programs that include complex loop structures.
4. TPGEN provides a control facility for the order of applying production rules and makes it possible to manipulate language rules such as "variables used in the execution portion must coincide with those in the declaration portion."

Outline of TPGEN's Generation Principle

TPGEN uses attribute grammar as the basic method for defining the syntax and semantics of the target language. Definition of semantics by attribute grammar is done by giving attributes to each grammar symbol (terminal symbols and nonterminal symbols), and by describing how to manipulate each attribute. In TPGEN, attributes are introduced in order to describe how symbol tables, program control sequences, tables that model the status of external files, and so on are changed when the generated source text is executed. These symbol tables, control sequences, and so on are attributes pre-defined by TPGEN and visible to its users.

Attribute grammar is a popular method for defining programming languages. TPGEN has adopted this for the following reasons:

1. Adding and/or updating the language definition is easy, since attribute passing between production rules is function-like (it has no side-effects) in attribute grammar. Such characteristics are well adapted to our quality assurance environment, where it is necessary to modify the definition or add a new set of definition to our programming languages in a fairly short time according to the contents of the added language features.
2. When back-tracking is required because of abnor-

```
<<test-case>>   -> <stmt>         ;     ································································(1)
<stmt>          -> <assign-stmt>  |     ································································(2-1)
                     <if-stmt>          |     ··························································(2.2)
                     <goto-stmt>    ;    ·······························································(2-3)
<assign-stmt>   -> <<var>> "=" <<int-const>>  (insert checking routine) | ································(3-1)
                     <<var>> "=" <expr>  ; ·····················································(3-2)
<if-stmt>       -> "IF"     <b-expr> "THEN" <assign-stmt>
                     "ELSE" <assign-stmt> "ENDIF" (insert checking routine)  ;    ························(4)
<goto-stmt>     -> "GOTO" <stmt-no>  (insert checking routine)  <stmt-no>
                   <stmt-no> ; ·································································(5)
<expr>          -> <primary> "+" <primary> | ···················································(6-1)
                     <primary> "-" <primary>  | ·················································(6-2)
<b-expr>        -> <primary> "GT" <primary> | ··················································(7-1)
                     <primary> "LT." <primary> | ················································(7-2)
                     <primary> ".EQ." <primary> ;        ········································(7-3)
<primary>       -> <<ref-var>> | ·······························································(8-1)
                     <<int-const>> ; ···························································(8-2)
                   notes: symbols enclosed by <<and>> and are special non-terminal symbols pre-defined by TPGEN:
                   <<var>> is replaced by one of the symbols already declared.
                   <<ref-var>> is replaced by one of the symbols already declared and already has some value.
                   <<int-const>> is replaced by a randomly selected integer constant.
                   <<test-case>> will be explained later.
```

Fig. 3 Simplified example of language definition (syntax only).

mal execution, TPGEN must restart generating the environment of the test program, such as the symbol tables. Since in attribute grammar all the information is stored as attributes in each node of a derived tree, it is easy to back-track to the specified point, resume the generation environment, and select alternative rules.

We will now explain how TPGEN generates test programs from the language definition. Figure 3 shows a simplified definition of a small subset of FORTRAN (see [14] for more details). In the following, symbols enclosed in arrowheads are non-terminal, and symbols enclosed in double quotes are terminal symbols.

```
      VAR1 = 100
C*****************************************************
C     TEST CASE 1 : TEST OF IF STMT
C*****************************************************
      IF VAR1 .GT. 200 THEN                                          (a)
        VAR1 = VAR1 - 100                                            (a)
      ELSE                                                           (a)
        VAR2 = 50                                                    (a)
          CALL  INTCHK(1, VAR2, 50, 'ASSIGN STMT INVALID.')  ........(b)
        VAR1 = VAR1 + 100                                            (a)
      ENDIF                                                          (a)
      CALL  INTCHK(1, VAR1, 200, 'IF STMT INVALID.')      ...........(c)
C*****************************************************
C     TEST CASE 2 : TEST OF GOTO STMT
C*****************************************************
```

Fig. 4 Example of generated text.

If we select the production rules in Fig. 3 in the order (1), (2-2), (4), (7-1), (8-1), (8-2), (3-1), (6-1), . . . , then part (a) of the source text of Fig. 4 will be generated. Parts (b) and (c) of Fig. 4 are generated after part (a) has been simulated, based on the description of the 'insert checking routine.' A detailed description of the 'insert checking routine' is given in [14].

The procedure adopted by TPGEN for generating executable test programs with self-checking code is as follows:

1. TPGEN selects production rules in a specified way (randomly or by considering functional coverage) and generates a source text corresponding to one test case. By a test case, we mean a source text that is a sequence of terminal symbols derived from starting symbol 《test-case》 and is executable by itself. Usually, we define a language so that one test case corresponds to one statement (a simple statement or a compound statement) with related statements such as initialization statements and declaration statements.

2. TPGEN simulates how each status (the contents of symbol table and so on) is changed by the execution of the generated text, based on the semantic definition of selected production rules. In order to confirm that the same status change will be given by the execution of the object code compiled by the target language processor, TPGEN inserts subroutine calls at specified positions to confirm the values of variables.

3. A source text for one test case is generated by the above procedure. A test program usually consists of several test cases, each with several hundred lines of codes, except when the program size itself is a major factor in test cases.

Self-checking Code

Although TPGEN understands status changes for all variables, users of TPGEN must specify the position at which self-checking code should be inserted, because TPGEN does not understand the structure of the target language. For example, if the 'THEN' clause of an 'IF' statement consists of one assignment statement, insertion of self-checking code inside the 'THEN' clause may require, in some languages, grouping of these statements. The current TPGEN does not understand the target language to that extent.

Automatic checking of execution results is very important in the inspection and testing of our software products. We had been using chekcing routines for many years even before the introduction of TPGEN. We have checking routines for each type of variable and for each target language. Checking routines themselves are coded in each target language. They receive, as parameters, a variable, its expected value, and other information that is used to identify the source text in case of an error. A test program generated by TPGEN is executable once it is link-edited with the above checking routines.

TPGEN inserts self-checking code for specified variables based on the semantic definition of the language. Validation of the contents of external files is done by validation of variables when they retrieve a record from that file. Users may specify the insertion of checking routines in such a way as to "insert checking routines for all the variables whose statuses have been changed since the last time their status change was confirmed." In this case, TPGEN generates code to check status changes for all the necessary variables. Insertion of checking routines is not restricted to the end of each trest case (see Fig. 4).

Test programs generated by TPGEN thus have self-checking code, and if a test program is executed correctly, the program is discarded and only the information concerning what kind of functional test was done is stored in our data base.

Evaluation of TPGEN

TPGEN has been in use for more than two years in our Quality Assurance Department for several language processors, including FORTRAN, C, LISP, PROLOG, Al-oriented shell, sort-merge, and COBOL-embedded SQL. At the time of their functional enhancement, these products were inspected partially, using TPGEN with more than 2000 production rules and several hundred KLOC of generated programs.

TPGEN has been effective for generating test programs that are executed normally. In the case of FORTRAN, about 80% of the normal functional testing can be done by using TPGEN. TPGEN is not effective for functional testing of special functions such as the Γ function, special files such as VSAM, and so on. In the case of SQL, most of the functional testing for Data Manipulation Language (DML) could actually be done by using TPGEN, but TPGEN is not effective for Data Definition Language (DDL). In the case of DDL, information concerning testing environments and operational procedures must be provided in order to make TPGEN generate test programs with self-checking code. It is not possible or realistic to describe this information with our simple model, in which only symbol tables, program control sequences, and so on can be simulated.

4. Fault Injection Testing

System software such as operating system software is usually designed in such a way that the impact on the overall system of its own errors is minimized. System software typically also incorporates built-in mechanisms that permit processing to continue while the areas damaged as a result of an error are isolated. Fault injection tests these error recovery mechanisms by deliberately introducing or injecting software faults and hardware faults into the system.

One fault injection technique is to overwrite in memory a portion of data or program [15, 16]. Chillarege [16] reports experiments using this technique on IBM's MVS/XA operating system, where 34% of system program errors have been traced to memory overwirting. Another technique for fault injection is to abruptly and unexpectedly terminate a unit of work. This can be accomplished by using a CANCEL command at an operator's console, for example.

Fault injection can also be achieved by using hardware. Recent reports describe attaching a multi-pin probe to LSI pins or exposing an LSI circuit board to heavy ion radiation [17, 18]. Gunneflo [18] reports that many error variations can be generated by using radiation. An I/O error simulator that we call the Hardware Trouble Smiulator (HTS) has been employed for many years as a means of simulating hardware faults by using software.

Testing Fault-tolerant Operating Systems

Fault injection is especially important in the evaluation and verification of fault-tolerant systems. However, the methods mentioned above have some shortcomings when used for testing commercial fault-tolerant operating systems.

1. If faults are entirely randomly generated, there is a high probability that faults will be injected into events with a high frequency of occurrence, such as I/O waiting. However, there is no guarantee that the coverage of such randomly created faults will be sufficient.
2. The nondeterministic fault injection method has difficulty in generating the same fault repeatedly and in investigating the error recovery process when the recovery process fails.
3. It is not evident that appropriate errors occur as a result of indirect fault injection such as memory overwriting.

We will now describe the methodology we employ to test our fault-tolerant system [23]. We evaluate fault tolerance by estimating the MTBF on the basis of test results. Our fault injection testing generates an appropriate mixture of hardware and software faults based on the distribution of the occurrence of such faults in user environments. We use two techniques to inject hardware faults:

1. A hardwar emechanism for simulating problems

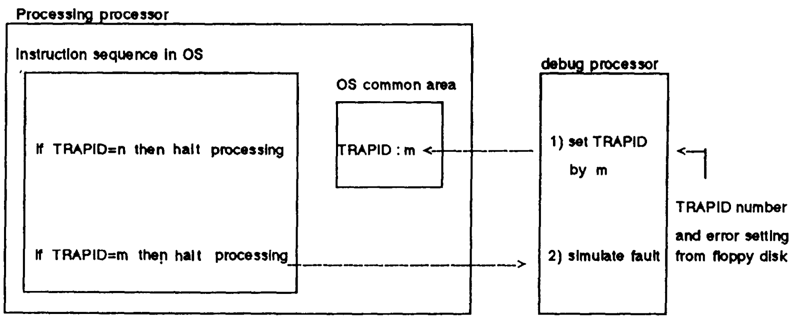

Processing processor
Instruction sequence in OS
If TRAPID=n then halt processing
If TRAPID=m then halt processing
OS common area
TRAPID : m
debug processor
1) set TRAPID by m
2) simulate fault
TRAPID number and error setting from floppy disk

Fig. 5 Fault injection mechanism of OS.

2. A software tool (HTS) for simulating I/O errors.

The software fault injection mechanism relies on an integrated, built-in fault-simulating mechanism in the operating system itself. This mechanism is depicted in Fig. 5.

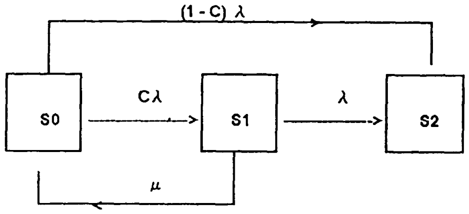
Our fault-tolerant system has a debugging processor with a debugging program that can overwrite the contents of memory and restart the processor. Using this feature, we can assign a number to the specific field named TRAPIS within the operating system's common area that can be referred to from all the programs in the operating system.

We imbed a sequence of instructions in the operating system checkpoints. Each sequence has a number assigned to if. If the number is equal to the TRAPID field, the instruction sequence halts processing. We assign a unique number to each instruction sequence in the entire system, and their locations are chosen so as to test effective cases, such as before and after disk data updating, that require different types of error recovery processing.

When the instruction sequence is executed, it detects whether its own number is equal to the contents of the TRAPID field and, if so, and execution control is transferred from the operating system to the debugger. The debugger will then simulate faults by overwriting memory or registers, or by transferring control to one of several specific routines that will simulate faults such as program checks, infinite waits, or infinite loops. This fault simulation is automatically performed according to the number of the TRAPID field and the error type specification, obtained from the floppy disk attached to the debugging processor. Thus, fault injection testing of the operating system can be accomplished by inputting a large number of randomly arranged pairs of TRAPID numbers and error type specifications. Moreover, it is easy to regenerate the same test sequence by using the testing information stored in the floppy disk.

Estimation of MTBF

We adopted an estimation model to assess the fault tolerance of the system. This fault tolerance comes



S0: on-line module is active
S1: on-line module is down and spare module is active
S2: both modules are down (system down)
λ: failure rate
μ: repair rate
C: coverage

Fig. 6 Markov model for standby spare system.

from the estimation of the MTBF after measurement of the coverage, which is the fault-proof rate of intentionally injected faults generated by the above fault injection testing. Our method may be explained as follows.

The reliability of a fault-tolerant system is normally calculated by analyzing the system as a stochastic process [19]. This is usually done according to the Markov model, which assumes a failure (rate (λ) and a repalr rate (μ) as constants. We introduced a Markov model of a standby spare system, including a coverage concept, as shown in Fig. 6.

This model can be solved analytically and the MTBF can be derived as follows:

$$\mathrm{MTBF}=((1+C)\lambda+\mu)/(\lambda(\lambda+(1-C)\mu))$$

When the system is in a stable condition, the above equation can be approximated by assuming that $\lambda \gg \mu$, as follows:

$$\mathrm{MTBF}=1/(1-C)^{*}1/\lambda$$

This means that the MTBF of the standby spare system is $1/(1-C)$ times greater than that of the single system ($1/\lambda$). Now we can estimate the MTBF as follows:

1. Find the actual coverage, C, for injected faults

2. Measure the occurrence frequency and time interval of faults that occur accidentally during testing, and put those values into a fault occurrence frequency transition model such as a non-homogeneous Poisson process (NHPP) model, and estimate the MTBF ($1/\lambda$) as a single system at the end of testing.

3. Estimate the MTBF value by the above equation.

Evaluation of Our Methodology

We are now applying this approach to our fault-tolerant operating system, and its evaluation has not yet been completed. At present, we can say that the testing procedure and its use of this methodology has been simplified and highly automated. Measurement of the coverage is very effective for the evaluation and improvement of our development and testing activities themselves. But this methodology still involves some problems and difficulties.

1. It is not known to what extent our fault injection mechanism and the work load effectively simulate faults that might occur naturally

2. It is not easy to make a work load that transfers control to some specific checkpoints.

5. Application System Testing

System software products such as operating systems for general-purpose computers are developed for the general market, not for individual customers. On the other hand, application systems such as banking systems, insurance systems, and manufacturing systems are constructed by developing customer-specific application software on the general-purpose system software products. In the case of large-scale application systems, a customer-specific application software system usually exceeds one million lines of code.

Application system testing confirms that a computer's system can meet the customer's functional, performance, and operational requirements. Such application system testing is performed in the following three steps:

1. System testing using a pseudo-application program. This verifies that general-purpose system software products meet customer-specific requirements in the environment in which they are intended to operate.

2. Application system testing including customer-specific application software development. The key point is to meet the customer's business requirements, which include the operational condition, performance requirement, and error recovery requirement. The Decision on selecting and/or applying general-purpose software products and general application packages is very important and strongly influences the productivity and maintainability of the application software developed.

3. Acceptance testing. This is done by the customer together with Fujitsu, in the actual operational environment with all the related facilities, including many terminals.

The purposes of these three aspects of application system testing are of course different and involve different personnel, but they also have many similarities. We shall now discuss system testing using pseudo-application programs by taking an example of a financial application system.

System Testing Using a Pseudo-Application Program

Generally speaking, a large-scale system includes scores of products, and it is confirmed that the functions, performance, and operationality of these products will, as a whole, conform to the customer's requirements on the basis of their operational environments and system configuration. This testing is usually done in parallel with the application software development, and to achieve this, a pseudo-application program is developed, after analyzing and modelling how the application program uses general-purpose system products. This will minimize the number of faults be detected at the acceptance testing, which is done after the completion of the application software.

This testing is rarely possible without the patient extraction of test cases, crcation of a testing procedure document, and running of prepared test jobs by a test team whose members are assembled from interrelated sections. For example, a test team may be com-posed of twenty members, extract more than 1000 test cases and write testing procedure documents of over 100 pages. Whether the test is successful or not depends on the quality of test cases extracted and the testing procedure documents. So many products are included in a large system that a wide range of knowledge and experience is required for this work. It is therefore very important to enrich communications among many organizations [20]. As an example, this testing may consist of the following four tests.

Installation testing: calls for the preparation of hardware and software products according to manuals that ultimately are delivered to our customers. In this pahse of testing, installability of software is established. In addition, the system is confugured so that operational, performance and error recovery testing may be performed. The key point here is to construct a representative test system that corresponds to the customer's configuration and environment. A sample financial system configuration is shown below.

	Test system	Real system
No. of clusters (host machines)	5	6
No. of DASDs	235 (420GB)	352(630GB)
No. of terminals	34	1700
Amount of application software		
Operational software	15 KLOC	300 KLOC
Business software	30 KLOC	1200 KLOC

Operational testing: Verifies that routine customer operations, from initial system start-up to system shutdown, can be performed as expected. Normal operational testing tests a customer's complete operational cycle from morning start-up and system activation through customer application execution and system deactivation and shutdown. Operational testing at abnormal times varies widely, depending on the customer's operation policy. For example, backend processing must continue even if the number of active machines is reduced to one, where as front-end processing may stop in this case, but pre-determined terminals must continue operation.

Performance testing: verifies through measurement that the customer's performance requirements are satisfied. Examples include confirming that the system can process one million transactions per hour, and validating that hot standby switching can be performed in less than 3 seconds. As it is not realistic to install the thousands of terminals and lines that make up today's customer environments, we have developed MTS, a multi-terminal simulator. MTS can be used to simulate terminals, lines, and terminal operators.

Error recovery testing: verifies that, in the event of an error, damaged portions are cut off, the recovery mechanism functions well, and the customer's operations continue as expected. For instance, destruction of a database on a DASD is simulated by generating I/O errors using a hardware trouble simulator (HTS), and data base corruption on the system storage unit (SSU) is simulated by a clip technique that generates SSU data transmission errors. Terminal errors and line errors can be tested by cutting power or disconnecting lines.

6. Summary

In this paper we have focused on some of the topics in dynamic testing. In testing, it is of course important to try to find as many bugs as possible. Another key point is how, once bugs are found, they are treated. We believe that causal analysis aimed at identifying the root cause of errors is an essential element of software testing—one that will yield significant improvements in software quality [21, 22].

References

1. Myers, G. J. The Art of Software Testing, John Willey & Sons, Inc. 1979.
2. Goodenough, J. B. and Gerhart, S. L. Toward a Theory of Test Data Selection, *IEEE Trans. Softw. Eng.*, SE-1, 2 (June 1975), 156-173.
3. Howden, W. E. Reliability of the Path Analysis Testing Strategy, *IEEE Trans Softw. Eng.*, SE-2, 3 (Sept. 1976), 209-215.
4. Chusho, T. Functional Testing and Structural Testing, *Japanese Perspectives in Software Engineering*, Addison-Wesley Publishing company (1989), 155-185.
5. Tatsumi, K. Conceptual Support for Test Case Design, *Proceedings of COMPSAC87* (1987), 285-290.
6. Shimokawa, H. and Sato, S. Test Cases Design Based on Design of Experiment, *The Fourth Symposium on Quality Control in Software Production (in Japanese)*, Federation of Japanese Science and Technology (1984), 1-8.
7.Taguchi, G. Design of Experiments, "Maruzen Publishing Co. (1976).
8. Duncan, A. G. and Hutchison, J. S. Using Attributed Grammar to Test Designs and Implementation, *Proceedings of the 5th ICSE* (1981), 170-178.
9. Bied, D. L. and Munoz, C. U. Automatic Generation of Random Self-Checking Test Cases, *IBM Systems Journal*, **22**, 3 (1983), 229-245.
10. Ince, D. C. The Automatic Generation of Test Data, *The Computer Journal*, **30**, 1 (1987), 63-69.
11. Maurer, P. M. Generating Test Data with Enhanced Context-Free Grammars, *IEEE Software* (July 1990), 50-55.
12. Bazzichi, F. and Spadafora, I. An Automatic Generator for Compiler Testing, *IEEE Trans. Softw. Eng.*, SE-**8**, 4 (July 1982), 343-353.
13. Seaman, R. P. Testing a High Level Language Compiler, *IEEE Comp. Syst. and Tech. Conf.* (Oct. 1974), 6-14.
14. Saijo, H. and Kawata, H. A Practical Test Program Generator Based on Attribute Grammar, Internal Document, Fujitsu Limited (1990).
15. Chillarege, R. Understanding Large Systems Failures—A Fault Injection Experiment, *Digest of papers, 19th Annual International Symposium on Fault Tolerant Computing* (1989), 356-363.
16. Barton, J. H. et al. Fault Injection Experiments Using FIAT, *IEEE Trans. Computer*, **39**, 4 (1990), 575-582.
17. Ariat, J. Crouzwt, Y. and Laprie, J. C. Fault Injection for Dependability Validation of Fault-Tolerant Computing Systems, *Digest of Papers, 19th Annual International Symposium on Fault Tolerant Computing* (1989), 348-355.
18. Gunneflo, U., Karlsson, J. and Torin, J. Evaluation of Error Schemes Using Fault Injection by Heavy-Ion Radiation, *Digest of papers, 19th Annual International Symposium on Fault Tolerant Computing* (1989), 340-347.
19. Tohma, Y. et al. Theory on Fault Tolerant Systems, The Institute of Electronics, Information and Communication Engineers (Japanese), 1990.
20. House, D. E. and Newman, W. F. Testing Large Software Products, *ACM SIGSOFT Software Engineering Notes*, **14**, 2 (Apr. 1989), 71-78.
21. Hino, K. Analysis and Prevention of Software Errors as a QC Activity, Engineering (Japanese) (Jan. 1985), 6-10.
22. Mays, R. G. et al. Experiences with Defect Prevention, *IBM Systems Journal*, **29**, 1 (1990), 4-32.
23. Yoshida, H., Suzuki, H. and Okazaki, K. Fault Tolerance Methodology of the SXO Operating System for Continuos Operation, The 1991 pacific RIM International Symposium on Fault Tolerance Systems (1991), 182-187.

(Received December 26, 1990)