A Selective Software Testing Method Based on Priorities Assigned to Functional Modules

Masayuki Hirayama[†‡], Tetsuya Yamamoto[†], Jiro Okayasu[†]
[†] R&D Center System Engineering Lab., TOSHIBA Corporation, Japan
masayuki.hirayama@toshiba.co.jp

Osamu Mizuno[‡], Tohru Kikuno[‡]
[‡] Graduate School of Engineering Science, Osaka University, Japan
{o-mizuno, kikuno}@ics.es.osaka-u.ac.jp

Abstract

As software systems have been introduced to many advanced applications, the size of software systems increases so much. Simultaneously, the lifetime of software systems becomes very small and thus their development is required to accomplish within relatively short period. In this paper, we propose a new selective software testing method that aims to attain the requirement of short period development. The proposed method consists of 3 steps: Assign priorities to functional modules (Step 1), Derive a test specification (Step 2), and Construct a test plan (Step 3) according to the priorities. In Step 1, for development of functional modules, we select both product and process properties to calculate priorities. Then, in Step 2, we generate detailed test items for each module according to its priority. Finally, in Step 3, we manage test resources including time and developer's skill to attain the requirement. As a result of experimental application, we can show superiority of the proposed testing method to the conventional testing method.

Keywords: *Software testing, Testing priority, Design of testing specification*

1. Introduction

According to the growth of the use of software, the size of software tends to increase. Consequently, the amount of testing required also increases, since large software contains many functions, operations and behaviors, all of which should be tested[1, 10]. However, the period available for software development is usually limited. As a result, in the worst case, testing of some items is omitted. The same situation applies to software embedded in products containing microcomputers. In particular, in regard to information systems whose functional specifications have ballooned in recent years, the complexity of the systems has been increasing so that it is very difficult to test them.

At present, no effective solutions are available for assuring reliability of software within a limited period of time. Conventionally, discussion of software system testing has focused on techniques to increase coverage of source code and functions[3, 4, 6, 11]. But recently, in the development of large systems, if an attempt is made to extract as many test items as possible so as to achieve coverage of all functions of the target system, the number of test items will become unmanageably huge. There is simply insufficient time available to test all these test items. Therefore, software system testing focused on coverage has become inappropriate. Considering the recent trends of software development, a highly efficient and highly reliable testing method capable of detecting a high ratio of system faults is required. Also required is a testing method capable of detecting and eliminating faults that would have serious impacts on the software system[7].

In order to satisfy these requirements, this paper discusses a selective testing method. Concerning a selective testing approach, Musa proposed the test item selection approach, too[8]. His approach focused on the operational situation for mainly business systems in user side, and performed tests with referring to the operational patterns. Musa's approach did not touch with products' specification and functions' features in detail, and also did not consider the features of development process focusing on the embedded software. These product's feature and process feature for embedded software are as important as the software operational profiles for business software. There are other methods to generate test items using operational profile[2]. On the other hand, Elbaum *et al*. analyzed effects of priorized testing[9]. They proposed the metrics for assessing the rate of fault detection of prioritized test cases and their prioritization technique are mainly used for determining the execution order of test suites. They do not refer to how to give priority for each function in the target soft-

ware.

In this method, testing priorities for functions of the target software are assigned from various viewpoints such as product's feature and development process view. Based on the priorities assigned to functions, effective testing can be performed. In this paper, we outline the selective testing method in which the test items are selected and tested based on the priorities assigned to functions in embedded software. This paper also presents an experimental example of this method and discusses the effectiveness of this method.

2. Selective Testing Method

2.1. Outline

Generally, in conventional testing, software reliability is assured by testing all possible test items of the target software. In some cases, a great amount of effort is expended on testing items that have little influence to system reliability, since the conventional technique accords the same priority to all test items. So, in such cases a long period is required for software testing.

On the contrary, in a case of selective testing methods, functions in the target software are prioritized. Functions with high priorities are tested in detail and functions with low priorities are tested less intensively. Moreover, the order of testing is also controlled. That is, in order to use the period of system testing effectively, the selective testing method aims to acquire the highest reliability in the shortest period of testing.

However, an effective technique for prioritizing testing functions has not been proposed until now. In this research, we propose a systematic technique for prioritizing functions, generating test items based on priorities, and controlling the order of testing. Figure 1 shows an outline of our proposed method. As Figure 1 shows, our selective method consists of three phases: function priority assignment, test item generation, and test control.

2.2. Function priority assignment

Test items for functions in the target system are prioritized by referring to use case analysis results. Concerning the assignment of priority, the viewpoints and metrics for test priority evaluation are prepared. A system function list with weighted priority is created based on the prioritization, and it is used in the system test phase.

2.3. Test item generation

Referring to the priority evaluation for each function, test items for each function are designed. In this test item generation, the quality and quantity of required test items are

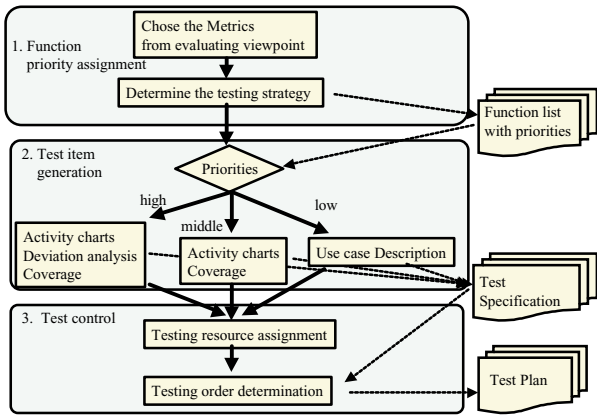


Figure 1. Outline of proposed method

controlled according to the priorities of the functions to be tested. For a function with high priority, use case description is used and deviation analysis from a normal case is performed, and finally detailed test items are extracted[5]. On the other hand, for a function with low priority, only the general operation indicated in the software specifications is checked. As a result of this work, test items are classified according to their priority.

2.4. Test control

Considering the test period and resources available, the execution sequence or order of test items generated is controlled and the person in charge of the system test is designated. In order to guarantee achievement of high reliability within the limited test period, it is necessary to detect the more important faults at an early stage. For this reason, in control of test work, an execution sequence is determined according to the priority of the test items and their ease of testing. Moreover, highly skilled people are assigned to high priority functions. This information is collected in a test plan document and serves as the input for the actual test work.

3. Technique for Test Priority Assignment

3.1. Outline

Generally, faults do not exist uniformly in software. Moreover, the influence on the user or the system varies among faults. An efficient testing technique focused on testing the portion involving many serious faults is proposed here. In the proposed technique, in order to realize a test based on the above-mentioned concept, functions in the tar-

get system are evaluated from two or more viewpoints, and are assigned priority for testing.

Test priority assignment is realized by the following four steps.

Step-1 Extraction of the testing function

Functions for testing are listed, referring to the specifications of the target software.

Step-2 Setting of the evaluation viewpoint for testing priority

The viewpoint for function prioritizing is discussed and determined.

Step-3 Setting of the metrics weight

Metrics for prioritizing are determined for the viewpoints. Also, weighting of metrics is discussed and determined. This means, in the testing of a system, the tester decides which metrics are assigned a high weight.

Step-4 Prioritizing of the testing function

Based on the weighted metrics, the test priorities for functions are determined. According to priorities, different instruction for testing is assigned to each test item.

3.2. Evaluation viewpoints for priority assignment

Regarding software faults, it is necessary to take into consideration three factors: fault injecting, fault detection, and fault influence. Also, in assigning priority for test items, these factors should be considered. Functional size and complexity of developed software have great influences on fault injecting. And the skill of the developer of the target system and the development process employed in the software design phase or implementation phase also have great influences on fault injecting.

On the other hand, fault detection has been largely concerned with use cases and the use frequency of functions. Concerning the influences of faults, it is necessary to consider what kind of influences faults have on users. Therefore, in prioritizing test items, fault injecting, fault detection, and fault influence are taken into consideration.

Regarding the scope of evaluation for functions, "S1: Product property" and "S2: Process property" are considered. Figure 2 shows the scopes, views and metrics for prioritizing function.

(a) Product property

Product property contains evaluation metrics for a software product. Product property is determined in advance for all functions to be realized as a system or software.

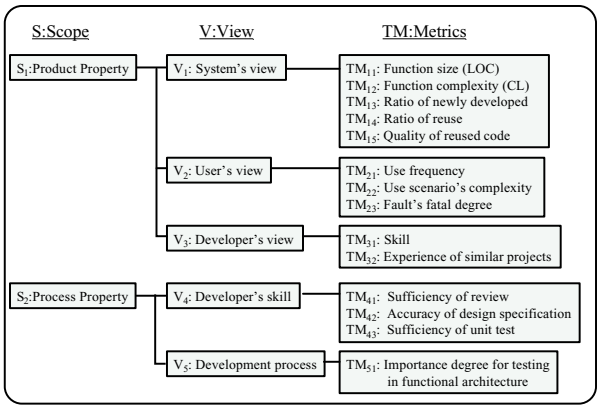


Figure 2. Evaluation viewpoints and metrics

V_1**: System's viewpoint** In particular, a system viewpoint is important for evaluation of fault injecting. This viewpoint takes into account the size and complexity of software. Also taken into account are the ratio of software reuse, the ratio of the newly development portion of the software and the quality of reused code.

V_2**: User's viewpoint** The user viewpoint is closely related to fault detection. This viewpoint takes into account the complexity of the user's use scenario (operation), the use frequency, the degree of the seriousness of faults, etc. For example, in the case of cellular phone software, functions that are important for younger users are different from those are important for other users. Thus, for the user viewpoint, it is important to take into account the variation of the actor and the use scenario in the UML use case description.

V_3**: Developer's viewpoint** Usually, the developer has sufficient knowledge of the structure of the developed function or of the relations among functions. Also, the developer has sufficient knowledge of the faults in previous products of a similar nature and a good understanding of the functions that should be tested. From the viewpoint of the developer, which functions should be tested is taken into account.

(b) Process property

The process property has a great influence on fault injecting. Regarding the process property, the following points concerning software functions are evaluated.

- Skill of the developer in charge
- Development process – Was the development process performed properly?

V_4**: Developers' skill** Many software faults are attributable to the inadequacy of the developer's skill. If two software developers independently of each other tackle the same software development task, the one whose skill level is lower will cause more faults.

V_5**: Development process** Even if two developers have the same skill level, the appropriateness of the respective development processes employed by the developers has an important bearing on the number and seriousness of the faults. Experience indicates that if an inappropriate development process is employed, the number and seriousness of the faults increases.

3.3. Evaluation metrics

Functions are quantitatively evaluated from the above-mentioned viewpoints, and testing priorities are assigned to the functions. Thus, evaluation metrics (TM_{ij}) for measuring and evaluating testing priority are prepared for the above-mentioned viewpoints (V_i).

For example, regarding the system viewpoint, TM_{11}: the size of functional structure (LOC), TM_{12}: complexity (CL: cyclomatic number), TM_{13}: ratio of newly developed software, TM_{14}: ratio of reuse, etc. are adopted as evaluation metrics.

The value of each evaluation metrics (TM_{ij}) is from 1 to 10. This value is given as a relative evaluation among the target testing functions.

For example, TM_{13} is valued according to the ratio of newly developed lines of code for each function. If the ratio of newly developed lines of code is from 0% to 10% and 10% to 20%, TM_{13} is determined to 1 and 2, respectively. For another example, as shown in Fig. 3, TM_{11} is assigned from 1 to 10 according to each function's size in software. Assume that the size of the largest function is 1000LOC and the size of the smallest function is 100LOC. For functions with 100 – 190LOC and 910 – 1000LOC, TM_{11} is determined to 1 and 10, respectively. According to this definition, for a function with 560LOC, TM_{11} is determined to 6.

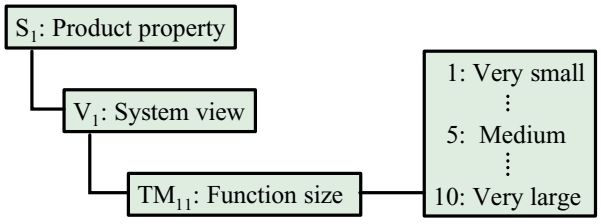


Figure 3. An example of evaluation

3.4. Decision on test priority value

It is necessary to consider the various viewpoints described above in order to assign priorities to testing functions. However, the function that an end user considers important does not always coincide with the function on which the developer focuses in software testing. Also, the function that an end user considers important does not coincide with the function considered to be important according to the process viewpoint. Therefore, it is necessary to make an overall judgment of the evaluation results based on two or more viewpoints, rather than based on a single viewpoint. In actual testing, we have to determine which viewpoints and metrics are important. This strategy is achieved by weighting of metrics.

For example, regarding a function K, if the value of evaluation metrics TM_{ij} is X_{ij}, and the weight of this metrics (test strategy coefficient) is W_{ij}, a test priority is calculated by the following formula. (W_{ij} is a weighting factor with a value of from 0 to 1.0.)

$$\text{Test priority}(K) = \sum W_{ij} \times X_{ij}$$

For example, in a test, in the case that a strategy focused on the user viewpoint is adopted, the weighting factors for user's use frequency (TM_{21}) and the fatal degree of a detected fault (TM_{23}) are about 1.0. Also, the weight coefficient of evaluation metrics for the developer's viewpoint (V_3) should be 0.2 or lower. Moreover, regarding the viewpoints and metrics (Fig. 2) for evaluation, it is possible to use only their subsets. For example, in the case that a process property is not taken into account, all the coefficients of evaluation metrics for a process property are 0.

Let us consider the example of the document storage function in word processing software. The following three evaluation metrics are considered.

- TM_{12}: function complexity
- TM_{21}: use frequency
- TM_{23}: fatal degree of fault

Each value is set to $TM_{12} = 4$, $TM_{21} = 6$, and $TM_{23} = 8$. A weighting factor is set to W_{12}= 0.2, $W_{21} = 1.0$, and $W_{23} = 1.0$. The test priority of this document storage function is calculated by the following formulas: $0.2 \times 4 + 1.0 \times 6 + 1.0 \times 8 = 14.8$.

4. Application Experiment

4.1. Purpose of experiment

Here, an experiment to evaluate the effectiveness of the selective test technique is introduced. In the experiment,

the proposed technique was applied to the test process of a "software development tool" development. In the experiment, as shown in Fig. 4, two independent test teams, A and B, were prepared for the purpose of comparison.

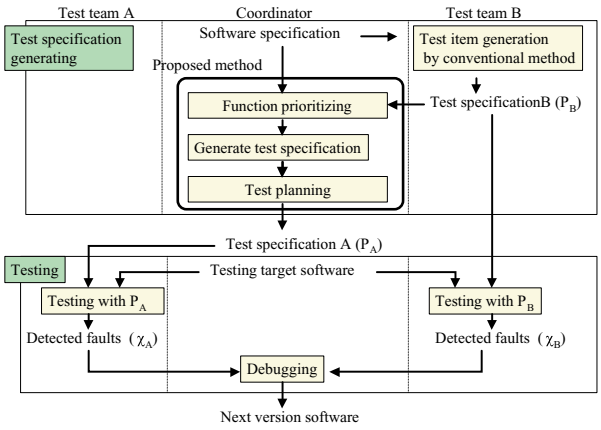


Figure 4. Outline of experiment

One team A performed the selective test and the other team B performed the conventional test, and the fault detection performances were compared. In the conventional test, all test items were tested in the order of the test specifications. In Fig. 4, χ_A and χ_B represent sets of detected faults by team A and team B, respectively. Then, P_A and P_B represent test specifications for teams A and B, respectively.

From the results of this experiment, we confirmed the following facts. In the case of the selective test technique, many important faults were detected as a result of assignment of priorities to target testing functions. On the other hand, in the case of the conventional test, the rate of detection of important faults was low compared with that in the case of the selective test technique.

4.2. Target software for testing

The target software of this experiment was a tool that supports a unit test. The main functions of this tool are selection of a test target module, automatic generation of test data, and automatic generation of a stub driver. The main features are as follows.

Language: C Language.

Size of software: 30KLOC.

Ratio of newly developed software: all newly developed

4.3. Design of experiment

4.3.1 Test team

Two independent teams, A and B, tested the functions of the target software using the selective test technique and the conventional test technique, respectively. In this experiment, the software-development experience of the members of the two teams and their skills are the same level, and there is little difference in their capabilities. Also, software design and implementation was done by a different team from those performing the testing. Furthermore, apart from these teams, the person who prepared the software specifications coordinated the entire experiment. The organization of the experiment is shown in Fig.4.

4.3.2 Test specification preparation

First, based on the specifications of software, a member of team B prepared test specification P_B in the conventional manner. Next, after the test experiment coordinator checked test specification B, the functions of the target software were prioritized. Then, the test experiment coordinator prepared test specification P_A for the selective test based on the result of prioritization. In test specification A, the priority was specified in three phases (high, medium, low) for each functional item. Test team A was instructed to execute the test using test specification A. Test team B was instructed to execute the test using test specification B.

4.3.3 Implementation of the tests

In the test phase, test teams A and B tested independently. The test period consisted of 3 cycles. Figure 5 outlines the overall procedure of the experiment. The period of time for each cycle was about one week. The faults detected by each team within each cycle were fed back to the developer upon completion of each cycle, and correction and debugging of the detected faults were done. Upon completion of this work, the next cycle was begun. Then, a regression test was performed for the new version of the software, using the same test specification as in the preceding cycle.

4.4. Prioritization of test items

In this experiment, we focused on evaluation of the user viewpoint (V_2) of product property. In the experiment, TM_{21}: the use frequency of each function, TM_{22}: the use scenario's complexity, and TM_{23}: the fatal degree of fault were adopted as evaluation metrics. The experiment coordinator determined the value of evaluation metrics for functions, taking into consideration the outline, the characteristics, etc. of each function from the specifications. Also, as a test strategy, equal weight was given to all evaluation

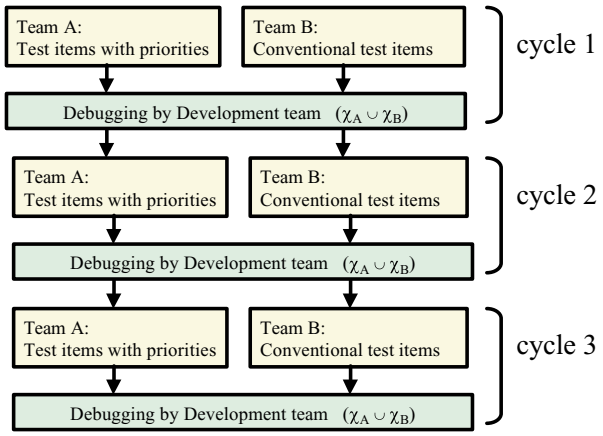


Figure 5. Detail of test execution

metrics. Therefore, the test priority of a function can be calculated by the following formula: $0.33 \times TM_{21} + 0.33 \times TM_{22} + 0.33 \times TM_{23}$.

4.5. Classifications of test items

Figure 6 shows a list of functions sorted in the order of the priority after evaluation of each function. In this experiment, we classified all functions as following way: (1) We firstly determined "High priority" functions by choosing 50% of all functions from the top of the list in Fig. 6. In this case, 118 functions are classified as high priority[1]. (2) We then determine "Medium priority" functions by choosing 25% of functions from the top of the rest of functions. (3) Finally, the rest of functions are classified as "Low priority."

In this experiment, since the target software was under actual development, we had to be careful on the reliability of software. So we enlarged the ratio of high priority functions. The numbers of test items for the priority levels were as follows: 118 high priority test items, 51 medium priority test items, and 56 low priority test items.

A part of the test specification A prepared for team A is shown in Fig. 7(a). As shown in this figure, the test specification A provides concrete instructions for test items of high priority functions, such as where importance should be placed in testing or test variations. On the other hand, regarding test items of low priority functions, the test specification A specifies that tests with low intensity are sufficient. As shown in Fig. 7(b), the test specification B for team B does not provide any particular instructions for any test item.

[1]Since including functions with the same score, the number of functions are slightly different with 50% of all functions.

5. Experimental Result

5.1. Fault detection in each cycle

Table 1 shows the number of detected faults of each cycle in the experiment. As mentioned before, χ_A and χ_B represent sets of faults detected by test teams A and B, respectively.

Table 1. Number of detected faults for each cycle

(a) Cycle-1

Cycle-1	Priority			Total
	High	Medium	Low	
χ_A	11	3	5	19
χ_B	8	2	7	17
$\chi_A \cap \chi_B$	5	2	4	11
$\chi_A \cup \chi_B$	14	3	8	25

(b) Cycle-2

	Priority			Total
	High	Medium	Low	
χ_A	8	1	5	16
χ_B	4	2	5	11
$\chi_A \cap \chi_B$	0	0	3	3
$\chi_A \cup \chi_B$	12	3	7	22

(c) Cycle-3

	Priority			Total
	High	Medium	Low	
χ_A	6	0	2	8
χ_B	0	1	1	2
$\chi_A \cap \chi_B$	0	0	1	1
$\chi_A \cup \chi_B$	6	1	2	9

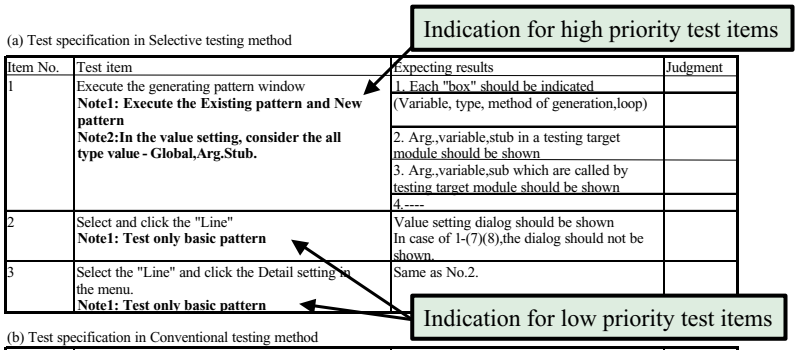
(1) Cycle-1

A total of 14 faults related to high priority functions were detected by the selective testing team A and the conventional testing team B. Team A detected about 80 percent of these. The number of faults detected by both the team A and the team B was 5 ($= |\chi_A \cap \chi_B|$). The number of faults related to high priority functions detected by the conventional testing team B but not by the selective testing team A was 3 ($= |\chi_B - (\chi_A \cap \chi_B)|$). As a result of detailed investigation of these three items, we confirmed that no clear instruction was provided for these three items, although detailed instructions should have been provided in the test specification because their priority is high. Also, a total of 8 faults related to low priority functions were detected, approximately 90 percent($= 7/8$) of which were detected by conventional testing.

Test strategy		
fatal degree	use frequency	complexity
0.33	0.33	0.33

Function ID	Functions		system property			score
	function Category	Operation	fatal degree	use frequency	complexity	
9	project window	refresh latest data	8	6	9	7.67
2	main	open the project	9	5	10	7.67
42	test case window	show the all items	6	10	7	7.67
1	main	create a new project	10	2	9	7
0	initial activity	perform the initial activity	10	5	3	6
34	value setting dialog	edit the setting value	6	7	5	6
23	pattern generation window	show the all items	6	5	7	6
30	value setting dialog	random	5	7	6	6
31	value setting dialog	limited value	7	4	7	6
63	value setting dialog	show the setting value	6	8	4	6
7	project window	operate the tree	5	10	2	5.67
8	project window	add the project source	7	2	8	5.67
45	test case window	select a test case	4	10	3	5.67
58	test execution dialog	execute test	7	6	3	5.33
28	pattern generation window	save the generated pattern	9	3	3	5

Figure 6. Functions to be tested with priority



(a) Test specification in Selective testing method

Item No.	Test item	Expecting results	Judgment
1	Execute the generating pattern window **Note1: Execute the Existing pattern and New pattern** **Note2:In the value setting, consider the all type value - Global,Arg.Stub.**	1. Each "box" should be indicated (Variable, type, method of generation,loop)	
		2. Arg.,variable,stub in a testing target module should be shown	
		3. Arg.,variable,sub which are called by testing target module should be shown	
		4.----	
2	Select and click the "Line" **Note1: Test only basic pattern**	Value setting dialog should be shown In case of 1-(7)(8),the dialog should not be shown.	
3	Select the "Line" and click the Detail setting in the menu. **Note1: Test only basic pattern**	Same as No.2.	

(b) Test specification in Conventional testing method

Item No.	Test item	Expecting results	Judgment
1	Execute the generating pattern window	1. Each "box" should be indicated (Variable, type, method of generation,loop)	
		2. Arg.,variable,stub in a testing target module should be shown	
		3. Arg.,variable,sub which are called by testing target module should be shown	
		4.----	
2	Select and click the "Line"	Value setting dialog should be shown In case of 1-(7)(8),the dialog should not be shown.	
3	Select the "Line" and click the Detail setting in the menu.	Same as No.2.	

Figure 7. Test specifications to be delivered

(2) Cycle-2

A total of 22 faults were detected, almost the same number as were detected in cycle-1. As a result of analysis of these 22 faults, it was confirmed that they can be classified into the following three categories:

a) Faults related to new functions (software) that had not been implemented at the time of cycle-1.

b) New faults resulting from fault correction in cycle-1.

c) Errors in correction of faults in cycle-1.

Of these faults, six are considered to be type b) or c). In cycle-2, of 12 faults related to high priority functions, 75 percent (= 8/12) were detected by selective testing. Twenty-five percent of faults(= 4/12) were detected by conventional testing.

(3) Cycle-3

Following the removal of many of the faults as a result of cycle-1 and cycle-2, the final authentication test was performed. The total number of faults detected in cycle-3 was nine, of which 67 percent(= 6/9) were related to high priority functions, and all of them were detected by selective testing. About half of the detected faults were errors in correction of faults in cycle-2.

5.2. Evaluation of detected faults

Table 2 shows the number of detected faults in the three cycles. In Table 2, faults that were detected in two or more cycles are counted only once. Of 118 test items for high priority functions (as mentioned in subsection 4.5), 22 faults were detected by selective testing and 11 faults were detected by conventional testing. The total number of faults

related to high priority functions was 26. Eighty-five percent (= 22/26) of these were detected by selective testing. On the other hand, only 42 percent(= 11/26) of them were detected by conventional testing.

Regarding 51 test items related to medium priority functions (as mentioned in subsection 4.5), each test method detected four faults. The total number of faults related to medium priority functions was six when any overlapping was eliminated.

Regarding 56 test items related to low priority functions (as mentioned in subsection 4.5), a total of 13 faults were detected. Of these, seven (54%) were detected by selective testing and 12 (92%) were detected by conventional testing.

Table 2. Number of test items that detected faults

Priority	Total	Selective method		Conventional method	
High	26	22	85%	11	42%
Medium	6	4	67%	4	67%
Low	13	7	54%	12	92%
Total	45	33	73%	27	60%

5.3. Test period

Table 3 shows the period of time required for testing in our experiment. Regarding the period of time required, no significant difference was observed between the conventional method and the selective method.

Table 3. Time needed for experiment

Cycle	Selective method (hours)	Conventional method (hours)
Cycle-1	30	33
Cycle-2	25	23
Cycle-3	13.5	22

6. Evaluation

6.1. Influence of instruction for test items

For selective testing, detailed instructions were provided for test items related to high priority functions to enable control of the quality of testing. As a result of the experiment, it was confirmed that regarding test items related to high priority functions, the number of faults detected by selective testing was about twice that detected by conventional testing. Regarding test items related to medium priority functions or low priority functions, no instructions were provided in selective testing. Consequently, unlike in the case of detection of faults related to high priority functions, there is little difference in the rates of fault detection between selective testing and conventional testing for low and medium priority functions.

Thus, by selective testing, it is possible to efficiently detect faults that are related to the crucial portion of a test target system. In selective testing, test items related to high priority functions are tested in various ways, and test items related to low priority functions are tested in terms of a simple pattern of testing.

On the contrary, with conventional testing in which test items are tested in the order of a specification document, there is no guarantee whether important functions or portions of a target system are sufficiently tested, and as a result, there is a possibility that important faults are not detected.

6.2. Efficient use of the test period

Tables 1 to 3 indicate that in selective testing time is used efficiently by placing importance on test items related to important functions, and faults are detected efficiently. In particular, in cycle-3 it is assumed that in the case of the selective testing method, faults related to high priority functions are intensively detected in a shorter time than in the case of the conventional method. In the case of conventional testing method, time is allocated to each test item equally regardless of the degree of importance of test items, and therefore, it is inefficient.

Based on the results of the experiment, we concluded that the degree of detection of faults in testing could be greatly improved by prioritizing functions from the viewpoint of the product property. We think that it is possible to enhance fault detection efficiency and software quality by taking process property also into account in the prioritizing step.

6.3. Required effort

In the experiment, additional effort was required mainly in the prioritizing activities for each test item. Though this effort was relatively smaller than the other effort required in the test phase. Effort for prioritizing the functions are about 2 or 3 hours in total. On the other hand, the overall test phase activities were required about 3 weeks. So the additional effort for adapting selective testing method can be think to be almost permissible margin.

7. Conclusion and Future Work

This paper reported a selective testing method for efficient system testing. In the selective testing method, properties of functions of the test target system are analyzed, priority is assigned to each function, and a test plan is drawn up based on these priorities. The main features of the selective testing method are as follows.

1. This technique consists of three phases: priority assignment for functions to be tested, design of test items, and testing control.

2. Prioritization of functions of the test target system is done by referring to the analysis results from the product property viewpoint and the process property viewpoint.

3. For prioritization of functions, the sum of weighted evaluation metrics values is adopted.

We experimentally applied this proposed method to a system test of actual software in order to evaluate the effectiveness of the proposed method. In the experiment, functions were prioritized based on the evaluation from the product property viewpoint. It was confirmed that by taking priorities into consideration, it is possible to allocate sufficient time and effort for testing of important functions, and the rate of detection of faults related to important functions can be improved.

In future work, in regard to test strategies, we intend to investigate prioritization of functions from the process viewpoint and to identify the most suitable way of weighting the metrics.

References

[1] B. Beizer. *Black-box Testing: techniques for functional testing of software and systems*. John Wiley & Sons, NY, 1995.

[2] K. Y. Cai. *Software defect and operational profile modeling*. Kluwer Academic Publishers, 1998.

[3] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 26(9):83–88, 1996.

[4] P. D. Coward. A review of software testing. *Information and Software Technology*, 30(3):189–198, 1988.

[5] M. Hirayama, T. Yamamoto, T. Kishimoto, O. Mizuno, and T. Kikuno. Generating test items for checking illegal behavior in software testing. In *Proc. of 9th Asian Test Symposium (ATS2000)*, pages 235–240, 2000.

[6] Y. K. Malaiya. Antirandom testing: Getting the most out of black-box testing. In *Proc. 6th International Symposium on Software Reliability Engineering*, pages 86–95, 1995.

[7] D. M. Marks. *Testing Very Big Systems*. McGraw-Hill, 1992.

[8] J. D. Musa. Software-reliability-engineered testing. *IEEE Computer*, 29(11):61–68, 1996.

[9] A. M. S. Elbaum and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proc. of 23rd International Conference on Software Engineering*, pages 329–338, 2001.

[10] I. Sommerville. *Software Engineering, 4th edition*. Addison-Wesley, MA, 1992.

[11] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *Proc. of 17th International Conference on Software Engineering*, pages 41–50, 1995.